Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Bounded Arithmetic and Randomized Computation

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Correlatrice:
Dott.ssa Melissa Antonelli

Presentata da:
Davide Davoli

Sessione 1
Anno Accademico 2021/2022

Acknowelgements

I would like to express my sincere gratitude to my esteemed supervisor Prof. Ugo Dal Lago whom has guided me through this research with patience, kindness and enthusiasm. Together with Ugo, I would also like to express my gratefulness to my co-supervisor Dr. Melissa Antonelli, for her great willingness to help me complete this work. Together with them, I am also thankful to Dr. Paolo Pistone and Prof. Isabel Oitavem for the cordiality and for their assistance at every stage of the research project. A wholehearted praise goes to my father Gianni for the immense trust he has in me and to my mother, Patrizia, who could not see neither the beginning nor the end of many things I did, but yet inspires many of them. Finally, my sincere gratitude goes to my loved ones for supporting me every time and filling my days with glee.

Contents

Introduction

At a broad level, Logic relies on proofs, which are assessments of an entailment between some premises and a consequence. Proofs are, ultimately, syntactical objects relating some formulæ — the *premises* — to another formula — called *consequence* — according to some system of rules, called proof system.

Similarly, Computer Science is founded on the notion of program. Again, programs are syntactical objects, relating some input data with an output. Even programs, for being accepted by the compiler, must follow a system of rules called type system.

The correspondence between programs and proofs — and at the higher level the correspondence between types and formulæ — grounds the so-called Curry-Howard isomorphism [11, 20]. This relation brings fruitful contributions to both the fields of Computer Science and logic. In Computer Science, for example, it is possible to shape computational formalisms upon well-behaved logics. These formalisms, for instance, are particularly suited for program analysis because of their solid logical properties. On the side of Logic, this correspondence makes it possible to represent proofs by means of computer programs, reducing the act of deducing a proof to the act of coding a well typed function.

Another consequence of this correspondence is the possibility to approach computational complexity by means of peculiar logical theories, called bounded srithmetics. These are basically first-order logical theories generated by four elements: a logical language $\mathcal{L}$, its semantics $[\![\cdot]\!]$, a set of axioms A and a proof system $\vdash$, which are capable to capture well known classes of program — often complexity classes — relating them with collections of forumlæ. This relation is grounded on the notion of provability: each derivation rule has a well known corresponding computation step. Thus, a program — i.e. a sequence of computational steps — can be represented by means of a proof, which is a sequence of deduction rules. For this reason, within a bounded arithmetics it si possible to identify certain *provable formulæ* which correspond to computable functions. Tailoring this theory — or, more often, its language $\mathcal{L}$ and the set of its axioms A — it is possible to identify, within a bounded arithmetic, classes of formulæ which correspond to well-known complexity classes.

In his PhD thesis [6], S. Buss managed to develop a bounded arithmetic which characterizes the complexity class **FP**. Precisely, Buss proved that every polynomial-time computable function corresponds to a function which is Σ_1^b-definable in the corresponding bounded theory S_2^1 [6]. This result is very insightful, but no similar works have been proposed in the realm of probabilistic computation, yet.

In contrast with deterministic computation, the probabilistic framework allows the computational models — canonically *Probabilistic Turing Machine* — to take purely random choices in each step of its computation. Thus, in this setting, reduction becomes a stochastic process, and the semantics of a program can no more be reduced to a function mapping inputs to outputs. Instead, it becomes a function which associates to any input a distribution of probability over its possible outputs.

The probabilistic framework has some relevant advantages with respect to the deterministic one: in particular, random choices allow to approximate solutions to problems with smallest average case time consumption and arbitrarily low error. This is evident if we examine the complexity class **BPP**, which contains all the decision problems having a probabilistic poly-time algorithm solving them with arbitrarily low error, and thus is intended to capture *feasibility* in probabilistic computation. Indeed, there are problems which are in **BPP**, but which are yet not known to be in **P**. For instance, the probabilistic algorithm for solving the Polynomial Identity Test is not known to have any solution in **P**, but it is known to have one in **BPP**. As for **NP**, there are open problems concerning **BPP** and **P**. The most important is probably **P** = **BPP**, indeed many problems in **BPP** have been recently discovered being in **P** such as, for example, the Primality Test.

Since November 2021 I am involved in a joint research project with Prof. Ugo Dal Lago, Dr. Paolo Pistone and Dr. Melissa Antonelli from University of Bologna and with Prof. Isabel Oitavem from NOVA University of Lisbon, Portugal. The contribution of our work consists precisely in extending Buss' work to the study of probabilistic complexity classes. The basic idea is to generalize S^1_2's language and semantics to a quantitative setting. Concretely, the first step consists in relating bounded formulæ with some effective model for probabilistic computation.

For this purpose, we introduced three novel classes of functions and prove them equivalent:

1. The class of *polynomial-time oracle recursive functions* ($\mathcal{POR}$), which is a Cobham style function algebra [9], with the capability to query an *oracle-function* $\omega : \{\mathtt{0},\mathtt{1}\}^* \longrightarrow \{\mathtt{0},\mathtt{1}\}$ to *possibly random* bits during the evaluation.

2. The class of functions which are Σ^b_1-representable in RS^1_2, where RS^1_2 are di axioms of our *randomized bounded theory.* This theory is expressed in a "probabilistic word language", which is a canonical first-order word language with equality inspired from [15], augmented by a special unary "probabilistic" predicate $\mathtt{Flip}(\cdot)$ [1].

3. The class of **SFP**-*functions*, which is the class of functions computable by polynomial-time Stream Machines. These are almost ordinary $k+1$-taped Turing machines; the only difference is that one of their tapes, called the *oracle tape*, is used as a left-to-right read-only tape and contains an infinite sequence of random bits. These machines differ from standard probabilistic Turing machines [24, 16], as their access to randomness is close to that of $\mathcal{POR}$'s functions: their computation depends on a function $\eta : \mathbb{N} \longrightarrow \{\mathtt{0},\mathtt{1}\}$ which describes the configuration of the *oracle tape.*

Our main result consists in proving that the class of functions which are Σ^b_1-representable in RS^1_2 is precisely the class of polynomial-time computable ones which, in turn, coincides with the class of **SFP**-functions. Then, starting from this equivalence, it seems possible to characterize probabilistic classes, such as **BPP** or **ZPP**, using formulæ of the bounded theory RS^1_2 together with non-standard quantifiers. For instance, functions corresponding to problems in **BPP** could be characterized leveraging a measure-sensitive quantifier **C** in the style of MQPA, [2, 1].

In this work, I will describe the main advances we have made to this end, primarily focusing on my personal contributions which are the reductions from $\mathcal{POR}$ to **SFP**, and vice-versa, the equivalence between **SFP** and **PPT**, A logical characterization of some probabilistic complexity classes.

The thesis is structured as follows: in Chapter 1, I recall describe the scientific context of this work, recalling some basic probability notions together, together with Buss' Bounded Arithmetic, then in Chapter 2, the RS^1_2 is defined together with the $\mathcal{POR}$ function algebra and a representability result between the two classes. Later, I will focus on my personal contributions to this research: in Chapter 3, I investigate the equivalence between $\mathcal{POR}$ and the class of functions which are computable in polynomial time by a Probabilistic Turing Machine, i.e. **PPT**. Thus, I generalize the main result of Chapter 2 Showing that the RS_2^1 representable functions are exactly the **PPT** ones. In Chapter 4, I show that the equivalence between $\mathcal{POR}$ and **PPT** has, as corollary, the equivalence between a Cobham-style function algebra — thereby introduced and called $\mathcal{PTF}$ — and the complexity class **FP**. Finally, Chapter 5 contains the characterizations of some standard probabilistic complexity classes by means of a semantical condition expressed in a word language $\mathcal{L}^{\mathsf{MQ}}$ inspired by $\mathcal{L}$ and MQPA.

Chapter 1

Preliminaries

Before getting into the main concern of this thesis — i.e. the development of a bounded arithmetic for randomized computation — we would like to briefly recall some some standard notions which are nevertheless fundamental for our work. To this aim, within the following sections, we will fix the notation for standard Turing Machines, recall some basic results of Probability and Measure Theory, describe some basic aspects of the probabilistic computational paradigm and, finally, we will recall the main results of S. Buss' bounded arithmetic. However, these sections each one independent from the others and none of them contains peculiar details affecting the following part of this work. For these reasons, the reader that feels confident on these subject can directly jump to the next chapter, in which we define our Randomized Bounded Arithmetic and prove it equivalent to the thereby introduced $\mathcal{POR}$ function algebra.

1.1 Turing Machines

In this section, we define Turing Machines. This is aimed to fix the notation of all the similar computational paradigms which will be introduced in what follows.

Definition 1 (Turing Machine)**.** A *Turing machine* is a quadruple $M := \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, where:

- $\mathcal{Q}$ is a finite set of states ranged over by q_i and similar meta-variables.
- $q_0 \in \mathcal{Q}$ is an initial state.
- Σ is a finite set of characters ranged over by c_i *et simila*.
- $\delta : \hat{\Sigma} \times \mathcal{Q} \longrightarrow \hat{\Sigma} \times \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ is a transition function describing the new configuration reached by the machine,

where L and R are two fixed and distinct symbols, e.g. `0` and `1`, $\hat{\Sigma} = \Sigma \cup \{\circledast\}$ and $\circledast$ represents the *blank character*, such that $\circledast \notin \Sigma$.

As for other flavors of Turing machines of this work, without loss of generality, we will take in exam single-taped machines with a binary alphabet only. The *configuration* of a ordinary TM is a tuple which keeps track of the current state and some strings representing the state of the machine tape(s).

Definition 2 (Configuration of a TM)**.** The *configuration of a TM* is a triple $\langle \sigma, q, \tau \rangle$, where:

- $\sigma \in \hat{\Sigma}^*$ is the portion of the work tape on the left of the head;

- $q \in A$ is the current state of the machine;
- $\tau \in \hat{\Sigma}^*$ is the portion of the work tape on the right of the head.

The shift of the tape is naturally defined by pre-fixing and post-fixing characters to strings The dynamic behavior of a TM is defined recurring to a transition function.

Definition 3 (TM Transition Function)**.** Given a TM, $M = \langle \mathcal{Q}, q, \Sigma, \delta\rangle$, we define the *partial transition function* $\vdash \delta : \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \longrightarrow \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^*$ between two configurations as:

$$\begin{aligned} \langle \sigma, q, c\tau\rangle &\vdash_\delta \langle \sigma c', q', \tau\rangle && \text{if } \delta(q,c) = \langle q', c', R\rangle \\ \langle \sigma c_0, q, c\tau\rangle &\vdash_\delta \langle \sigma, q', c_0c_1'\tau\rangle && \text{if } \delta(q,c_1) = \langle q', c_1', L\rangle. \end{aligned}$$

The transitive closure of $\vdash$ yields the reachability function between configurations of the machine, as defined below:

Definition 4 (TM Reachability Function)**.** Given a TM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$, $\{\triangleright^n_M\}_n$ is the smallest family of relations such that:

$$\langle \sigma, q, \tau\rangle \triangleright^0_M \langle \sigma, q, \tau\rangle$$
$$\big(\langle \sigma, q, \tau\rangle \triangleright^n_M \langle \sigma', q', \tau'\rangle\big) \wedge \big(\langle \sigma', q', \tau'\rangle \vdash_\delta \langle \sigma'', q'', \tau''\rangle\big) \to \big(\langle \sigma, q, \tau\rangle \triangleright^{n+1}_M \langle \sigma'', q'', \tau''\rangle\big).$$

Even in this case, without loss of generality, we assume that TMs do not use final states: computation is regarded as concluded whenever the transition function is undefined on the current configuration.

Proposition 1. *For any TM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$ and $n \in \mathbb{N}$, $\triangleright^n_M$ is a* partial *function.*

Proof. Trivial, by the fact that $\vdash$ is a function and because the composition of two functions is a function, too. □

Notation 1 (Final Configuration)**.** Given a TM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$, and a configuration $\langle \sigma, q, \tau\rangle$, we write $\langle \sigma, q, \tau\rangle \not\vdash_\delta$ when there are no σ', q', τ' such that $\langle \sigma, q, \tau\rangle \vdash_\delta \langle \sigma', q', \tau'\rangle$.

Finally, let us introduce the notion of function computable by TMs.

Definition 5 (TM Computation)**.** Given a TM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$ and a function $g : \mathbb{S} \longrightarrow \mathbb{S}$, we say that M *computes* f, written $f_M = g$ if and only if for every string $\sigma \in \mathbb{S}$, there are $n \in \mathbb{N}$ and a $\tau \in \mathbb{S}, q' \in \mathcal{Q}$, such that:

$$\langle \boldsymbol{\epsilon}, q_0, \sigma\rangle \ \triangleright^n_M \ \langle \gamma, q', \tau\rangle \not\vdash_\delta,$$

with $f(\sigma)$ being the longest suffix of γ not including $\circledast$.

Definition 6 (Poly-Time Turing Machine)**.** A *poly-time stream Turing machine* is an TM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$ such that:

$$\exists p \in \mathsf{POLY}.\forall \sigma \in \mathbb{S}, \eta \in \mathbb{B}^{\mathbb{N}}.\exists n \leq p(|\sigma|)\big(\langle \boldsymbol{\epsilon}, q_0, \sigma, \eta\rangle \triangleright^n_M \langle \gamma, q', \tau, \psi\rangle \not\vdash_\delta .$$

We define the class of functions which are computable by poly-time TMs.

Definition 7 (The Class **FP**)**.**

$$\mathbf{FP} := \{f \in \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \mid f = f_M \text{ for some polynomial TM, } M\}$$

1.2 On Measure Theory and Probability Thery

This section is aimed to introduce some basic notions and notations of Measure and Probability theory with the aim to the introduction of those basic technical notions which will be necessary to follow the following discussions. The reader who is interested in a deeper and complete introduction to Probability Theory can consult Feller's "An Introduction to Probability Theory and its Applications" [12] or Billingsley's "Probability and Measure" [5]. In particular, Billingsley's work has been taken as a reference in many parts of this work.

Probabilities studies how it is possible to *measure* the degree of certainty of some events. Let Ω be an arbitrary set of events. In Probability Theory, the algebra of sets describing probabilistic events and their composition is called a "field".

Definition 8 (Field)**.** A filed $\mathcal{F}$ is a pair $\langle \Omega, F\rangle$ such that $F \subseteq \mathcal{P}(\Omega)$ and:

- $\Omega \in F$;
- $X \in F \to \overline{X} \in F$;
- $X \in F \wedge Y \in F \to X \cap Y \in F$.

Due to the presence of complementation, observe that the third condition in the definition of a field could be restated recurring to intersection rather than set union, obtaining an equivalent definition. This result can be trivially shown by induction on the proof that a set belongs to a specific field. Moreover, we would like to point out that the elements of a field can only be finite or co-finite with respect to Ω.

If we relax this finitary condition, we end up defining a different algebraic structure, called σ-field, or σ-algebra.

Definition 9 (σ-field)**.** A σ-field Σ is a set $\Sigma \subseteq P(\Omega)$ such that:

- $\Omega \in F$;
- $X \in F \to \overline{X} \in F$;
- $X_1, X_2, \ldots \in \mathbb{N}. \to \bigcup_{i\in\mathbb{N}} X_i \in F$.

If Σ is a σ-field, then the pair $\langle \Omega, \Sigma\rangle$ are called *measurable spaces.* This epithet is due to the possibility to use these sets to define measure functions $\mu : \Sigma \longrightarrow \mathbb{R}$, peculiar functions associating a real number to each set.

The elements of a σ-field intuitively represent all the possible outcomes of a random experiment and, within a probabilistic context, are called *events.* Thus, assignment of a degree of certainty to each possible random event can be accomplished by defining a function which assigns to each element of a σ-field a value, representing the certainty of that event. However a function, in ordered to be a *measure function*, must verify some constraints upon its values.

Definition 10 (Measure Function)**.** If $\langle \Omega, \Sigma\rangle$ is a measure space, then a function $\mu : \Sigma \longrightarrow \mathbb{R}$ is a *measure function* if and only if:

- $\forall X \in \Sigma.\mu(X) \geq 0$;
- $\mu(\emptyset) = 0$;
- For all the countable collections $\{X_i\}_{i\in\mathbb{N}}$, if all the X_i are pairwise disjoint, then:
$$\mu\left(\bigcup_{i\in\mathbb{N}} X_i\right) = \sum_{i\in\mathbb{N}} \mu(X_i).$$

However, there is a class of measure functions with stronger constrains which are those actually employed Probability Theory to measure the degree of certainty of probabilistic events.

Definition 11 (Probability Measure)**.** If $\mathcal{F} := \langle \Omega, F \rangle$ is a field and $P : \Sigma \longrightarrow \mathbb{R}$, then P is a probability measure if and only if:

- $\forall X \in \Sigma . 0 \leq \mu(X) \leq 1$;
- $\mu(\emptyset) = 0 \wedge \mu(\Omega) = 1$;
- For all the countable collections $\{X_i\}_{i \in \mathbb{N}}$, if all the X_i are pairwise disjoint, then:

$$\mu\left(\bigcup_{i \in \mathbb{N}} X_i\right) = \sum_{i \in \mathbb{N}} \mu(X_i).$$

Remark 1. *All the probability measures are measures, while the converse does not hold.*

These are all the ingredients we need to define a probability space:

Definition 12 (Probability Space)**.** If $\langle \Omega, \Sigma \rangle$ is a measurable space and P, then the triple $\langle \Omega, \Sigma, P \rangle$ is called *Probability Space.*

Events by themselves are the pure outcome of a probabilistic experiment. For this reason it is often useful to define functions associating each possible event to a generical interpretation. These functions are called "random variables".

Definition 13 (Random Variable)**.** If $\langle \Omega, \Sigma, P \rangle$ is a probability space and $K \neq \emptyset$ is a set, a function $Z : \Omega \longrightarrow K$ is a random variable.

As for ordinary events, it is possible to associate a degree of certainty to each possible value assumed by a random variable. This is simply done recurring to the measure of their pre-image.

Definition 14 (Probability of a Random Variable)**.** If $\langle \Omega, \Sigma, P \rangle$ is a probability space and $Z : \Omega \longrightarrow K$ is a random variable, then we define the probability distribution of the variable Z as the function $Pr : K \longrightarrow [0, 1]$ defined as follows:

$$Pr[Z = k] = P(Z^{-1}(\{k\})).$$

Fixed a set K, we call $\mathbb{D}(K)$ the set of all the probability distributions over K.

These few notions are everything required to walk through the probabilistic concerns of this work. For a better understanding of these notion, we take in exam a school-book like example of probability theory:

Example 1. Let us build a model for the experiment of rolling a dice, in order to determine the probability of getting an outcome which is greater or equal than 4. Then, the set Ω of all the possible outcomes of the experiment can be defined as $\Omega := \{1, 2, 3, 4, 5, 6\}$. Basing on Ω, we can define a probability space as follows:

- $\Sigma := \mathcal{P}(\Omega)$;
- $P := X \mapsto \frac{|X|}{|\Omega|}$.

The triple $\langle \Omega, \Sigma, P \rangle$ is a probability space: indeed Σ is a finite σ algebra on Ω and P is a probability measure. Finally, our experiment can be modeled by means of the random variable $Z : \Omega \longrightarrow \{0, 1\}$ defined as follows:

$$Z(x) := \begin{cases} 1 & \text{if } x \geq 4 \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we can quantify $Pr[Z = 1]$ as follows:

$$Pr[Z = 1] = P(Z^{-1}(1)) = P(\{4, 5, 6\}) = \frac{|\{4, 5, 6\}|}{|\Omega|} = \frac{3}{6} = \frac{1}{2}.$$

1.3 Randomized Computation

Standard deterministic computation is defined upon TMs. This computational model features all the capabilities of a standard calculus formalism, indeed, as meant by Alan Turing, computation can be represented as a process of rewriting some expressions following predefined rules [23]. However, although the TM computational model is sufficiently expressive to shape mathematical computations, it is still not capable to capable to model *purely random events*.

Even though the existence of randomness is itself debatable, there are real world experiments — such as a coin toss, for instance — whose outcome is sufficiently uncertain to be considered random for practical purposes. Thus, it makes sense to consider computational models which, during the computation, may take purely random choices. Random algorithm are nowadays pervasive in many fields of Computer Science, such as Cryptography, Numerical Analysis and Machine Learning.

Intuitively, the output of a random algorithm *can be* uncertain. Thus, if a random algorithm is employed to solve a problem, the solution it proposes *can be* uncertain — and, in many cases, it is. For an algorithm, producing uncertain or approximated solutions is not an advantage, especially if this feature is considered by itself. However, sometimes probabilistic algorithms can solve computationally hard problems with simple algorithms, low error and lower time complexity than any non-random solution. In the Introduction, we have already mentioned some of these cases: the Polynomial Identity Testing Problem (PIT) and the Primality Test. While PIT is known to be in **BPP**, there is no evidence of this problem being in **P**; contrarily the Primality Test problem has been shown being in both **BPP** and **P**, but the time consumption of the probabilistic Miller Rabin Test on is sensibly lower than the AKS algorithm which is up to now, the fastest non-random algorithm [26]. In complexity theory, the trade-off between approximation and time consumption leads to the definition of complexity classes which capture a new notion of feasibility based on Random Computation.

Within this section, we introduce the definition a of a computational model for randomized computation and we define some complexity classes which model a probabilistic notion of feasibility, discussing their inter-relations and properties.

Definition 15 (Probabilistic Turing Machines [4])**.** A *probabilistic Turing machine* is a Turing machine with two transition functions, namely δ_0, δ_1. Given an input x, the PTM chooses at each step with probability $\frac{1}{2}$ to apply the transition function δ_0 and with the same probability to apply δ_1. This choice is independent from all the previous ones.

To model the semantics of a Probabilistic Turing Machines (PTM, for short), we cannot recur to functions from strings to strings, because the output of the computation is the outcome of a stochastic process, so it results in a probability distribution. Indeed, called $\mathbb{S}$ the set of binary strings, the semantics of a PTM is a function

$$f : \mathbb{S} \longrightarrow \mathbb{D}(\mathbb{S}).$$

Within the class of PTM-computable functions, we can identify a thinner class imposing a polynomial time bound: the class of **PPT** functions.

Definition 16 (Class **PPT**)**.** The *class* **PPT** is the class of random functions from $\mathbb{S}$ to $\mathbb{D}(\mathbb{S})$ which are computable by a PTM in at most a polynomial number of steps.

Notice that when defining **PPT** we could make a slightly different requirement on the time complexity of those functions: indeed, we could require their *expected* time consumption to be at most polynomial. This dichotomy is pervasive in probabilistic complexity: usually, algorithm with *worst case* time bounds are called *Monte Carlo* algorithm, while algorithm with *average case* time bounds are usually called *Las Vegas* algorithms.

Characterizing feasible decisional problem by means of probabilistic algorithms, we are interested in two properties of these procedures: of course we re interested polynomial time complexity — which can be captured by the **PPT** class — but we also want to impose limitations to the amount of error these algorithms can make. An example of how it is possible to capture both these property is the *Bounded error Probabilistic Poly-time* complexity class **BPP**.

Definition 17 (Class **BPP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **BPP** if and only if, said f_L the characteristic function of the language, there is a **PPT** function f such that:

$$\forall x \in \mathbb{S}.Pr[f(x) = f_L(x)] \geq \frac{2}{3}.$$

One can argue that $\frac{1}{3}$ of wrong detections can not be considered *low error*. However, we must also take in account that, since it is possible to evaluate the same probabilistic function multiple times, it has been shown that the amount of error of a **BPP** function can be reduced arbitrarily preserving the program's poly-time consumption, [4, Lemma 7.9]. The class **BPP** is usually referred to as a *double sided error* class. Indeed the probability of error is equal for all the strings which are in the language and those who are not. One may also be interested in studying randomized algorithms which do not allow false positive or false negatives or even both of them. These three possibility represent three different nuances of feasibility within the probabilistic framework. Any of them is captured by a different complexity class.

Definition 18 (Class **RP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **RP** if and only if there is a **PPT** function f such that:

$$\forall \sigma \in L.Pr[f(\sigma) = \mathtt{1}] \geq \frac{2}{3}$$
$$\forall \sigma \notin L.Pr[f(\sigma) = \mathtt{0}] = 1.$$

It is almost trivial to see that **RP** is included within **BPP**: take an **RP** function f deciding L: the probability of error of f is always smaller than $\frac{1}{3}$, which is the definition of **BPP**. Thus, $L \in$ **BPP**.

Usually, the complexity classes which are not trivially known to be closed under complementation — **NP** and **RP** are among those classes — cause the introduction of the class containing all the complementations of their problems as a class as on its own. This would be useless for classes which are closed under complementation, because that novel class would be identical to the original one.

The complementary of **RP** is co-**RP** and is the class of languages L such that there is a **PPT** function f_L accepting the members of L with no probabilistic error at all and refusing the strings which not belonging to L with probability of error smaller than $\frac{1}{3}$. This is another example of *one-sided* probabilistic error. The definition of this class can be formally stated as follows:

Definition 19 (Class co-**RP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in co-**RP** if and only if there is a **PPT** function f such that:

$$\forall \sigma \in L. Pr[f(\sigma) = \mathtt{1}] = 1$$
$$\forall \sigma \notin L. Pr[f(\sigma) = \mathtt{0}] \geq \frac{2}{3}.$$

Even in this case, one can observe that co-**RP** $\subseteq$ **BPP**. Finally one can be interested in stressing even more the error requirements allowing no error at all. The complexity class thus obtained is called **ZPP** and contains all those languages which are known to be solvable with no probabilistic error with polynomial time complexity, which is quite peculiar, especially considering that the random choices of the algorithm, for these programs do not cause any probabilistic outcome. This even thinner complexity class is called **ZPP**, which stands for *"Zero Probabilistic Poly-time error"*.

Definition 20 (Class **ZPP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **ZPP** if and only if, said f_L the characteristic function of L, there is a **PPT** function f such that:

$$\forall \sigma \in \mathbb{S}. \forall \mathtt{b} \in \{\mathtt{0}, \mathtt{1}\} f(\sigma) = k \rightarrow f_L(\sigma) = 1$$
$$\forall \sigma \in \mathbb{S}. Pr[f(\sigma) \notin \{\mathtt{0}, \mathtt{1}\}] < \frac{1}{3}.$$

In this definition, we require that the machine has no margin of error on all its answer, however we admit the possibility that it in some cases it may output a value different from `0` and `1`: this kind of answer should be interpreted as "I do not know". The definition of **ZPP** given above is not the standard one, but is an alternative characterization which requires the algorithm to be a *Monte Carlo* algorithm rather than a *Las Vegas* one. This because our Randomized Bounded Arithmetic characterizes the **PPT** functions, which are defined on top of *Monte Carlo* algorithms, so this definition will allow a more natural characterization of such class in Chapter 5.

It is possible to see that this class is included within **BPP**, it is a consequence of **ZPP** = **RP** $\cap$ co-**RP**, which we prove in Theorem 1. This result also entails that **ZPP** is closed under complementation.

Theorem 1. **ZPP** = **RP** $\cap$ *co*-**RP**.

Proof. We first show that **ZPP** $\subseteq$ **RP**$\wedge$**ZPP** $\subseteq$ co-**RP**. We examine only the second proposition: **ZPP** $\subseteq$ co-**RP**. Let f_L be the decision function for a language $L \in$ **ZPP**, take the following decision function for L:

$$g(x) := \begin{cases} \mathtt{1} & \text{if } f(x) \neq \mathtt{0} \wedge f(x) \neq \mathtt{1} \\ \mathtt{0} & \text{otherwise.} \end{cases}$$

Suppose that $\sigma \in L$, then $f(\sigma) = \mathtt{1} \vee (f(\sigma) \neq \mathtt{0} \wedge f(\sigma) \neq \mathtt{1})$, so $Pr[g(\sigma) = \mathtt{1}] = 1$. Otherwise, assume that $\sigma \notin L$ $f(\sigma) = \mathtt{0} \vee (f(\sigma) \neq \mathtt{0} \wedge f(\sigma) \neq \mathtt{1})$. In the first case, $g(\sigma) = \mathtt{0}$, while in the second case $g(\sigma) = \mathtt{1}$ causing a wrong detection, but by definition of **ZPP**, we know that it can happen with probability smaller than $\frac{1}{3}$. So $g \in$ co-**RP**. The proof that **ZPP** $\subseteq$ **RP** is analogous. Now we show that **RP** $\cap$ co-**RP** $\subseteq$ **ZPP**. Suppose that f_1 is a **PPT** function such that $f_1(\sigma) = \mathtt{1}$ if $\sigma \in L$, and for $\sigma \notin L$ $f_1(\sigma) = \mathtt{1}$ with probability smaller than $\frac{1}{3}$. Similarly,

$$\begin{tikzcd}
 & \mathbf{RP} \arrow[dr, "\subseteq" description] & \\
\mathbf{ZPP} = \mathbf{RP} \cap \text{co-}\mathbf{RP} \arrow[ur, "\subseteq" description] \arrow[dr, "\subseteq" description] & & \mathbf{BPP} \\
 & \text{co-}\mathbf{RP} \arrow[ur, "\subseteq" description] &
\end{tikzcd}$$

Figure 1.1: Inclusion schema between complexity classes.

suppose that f_2 is a **PPT** function such that $f_2(\sigma) = \mathtt{0}$ if $\sigma \notin L$, and for $\sigma \in L$ $f_1(\sigma) = \mathtt{0}$ with probability smaller than $\frac{1}{3}$. Then consider the **PPT** function g defined as follows:

$$g(x) := \begin{cases} f_1(x) & \text{if } f_1(x) = \mathtt{0} \\ f_2(x) & \text{if } f_2(x) = \mathtt{1} \\ \mathtt{11} & \text{otherwise.} \end{cases}$$

Suppose that $\sigma \in L$, then $g(\sigma) = \mathtt{11}$ only if $f_2(\sigma) = \mathtt{0}$, which happens with probability smaller than $\frac{1}{3}$. Similarly, if $\sigma \notin L$, then $f(x) = \mathtt{11}$ if and only if $f_1(\sigma) = \mathtt{0}$, which happens with probability smaller than $\frac{1}{3}$. $\square$

Considerations on Probabilistic Complexity Classes In the previous part of this section, we have shown some relations among Probabilistic Complexity Classes. In particular we have shown how **BPP**, **RP**, co-**RP** and **ZPP** are one included in the other as described by Figure 1.1. Moreover, Theorem 1, is a quite surprising result: the corresponding question for non determinism — i.e. $\mathbf{P} = \mathbf{NP} \cap \text{co-}\mathbf{NP}$ — is still open.

Another relation between probabilistic complexity classes and the non-probabilistic ones is that $\mathbf{BPP} \subseteq \Sigma_2^p \cap \Delta_2^p$, [17]. Thus if $\mathbf{P} = \mathbf{NP}$, then we would get $\mathbf{P} = \mathbf{BPP}$. This motivates even more the interest in investigating these classes.

Despite these interesting inter-relations between non-probabilistic and probabilistic complexity classes, we would like to point out that the latter are inherently different from the former: they are considered *semantic complexity classes*, instead of *syntactic complexity classes*.

Intuitively, all of those classes having a procedure c which decides whether the function computed by a given algorithm a passed as argument belongs or not to that class are considered syntactic complexity classes. For example, **P** is a *syntactic* complexity class, because it suffices to take c defined as follows:

$$c(a) := \begin{cases} \mathtt{1} & \text{if } a \text{ is the encoding of a Timed Turing Machine with polynomial time bound} \\ \mathtt{0} & \text{otherwise.} \end{cases}$$

Indeed, without lack of generality, one can consider every Timed Turing Machine as the implementations of a *decision* algorithm. A similar argument establishes, for example, that **NP** is a *syntactic* complexity class as well. Conversely, classes which are not known to have this property are called *semantic complexity classes.* For instance, take in exam **BPP**: it is not sufficient to check whether the algorithm a is the encoding of a Timed Probabilistic Turing Machine with polynomial time bound. Indeed, one must also assess that the probabilistic error is bounder for every input. Unfortunately, testing whether a given PTM has this property is not believed decidable, [4]. Together with **BPP**, even **RP**, co-**RP** and **ZPP** are considered *semantic complexity classes.*

These considerations suggest that the analysis of probabilistic complexity classes is intrinsically harder than the analysis of syntactical complexity classes, but that due to the relations between probabilistic complexity classes and open problems of Computer Science, the research in this field is surely worth its effort.

1.4 Bounded Arithmetic

A Bounded Arithmetic is a logical theory on a first-order language with identity, a preorder predicate symbol — within this section we employ the symbol $\leq$ —, bounded quantifiers (Definition 21) and some arithmetical functions.

Definition 21 (Bounded Quantifier, [6])**.** A *bounded quantifier* is a quantifier of the form $Qx \leq t$ with t a term not involving x. A *sharply bounded quantifier* is one of the form $Qx \leq |t|$. $\forall x$ and $\exists x$ are unbounded quantifiers. A *bounded formula* is one with no unbounded quantifiers.

Our interest towards bounded arithmetic is due to their capability to ground logical characterizations of well known complexity classes. This is usually done adopting appropriate interpretations for term functions and adopting a constructive proof system. This allows to identify classes of formulæ which correspond to well known complexity classes.

Within this section we will introduce a standard bounded arithmetic due to S. Buss [6], with the aim to pave the way for the development of a novel bounded arithmetic for randomized computation in Chapters 2 and 3.

1.4.1 The Language of Buss' Arithmetic

The language $\mathcal{L}_{\mathsf{Buss}}$ is the first-order language with identity whose terms are described by the grammar of Definition 22, and whose formulæ are given in Definition 23.

Definition 22 (Terms)**.** Let $x, y, \dots$ denote variables. Terms are defined by the following grammar:

$$t, s ::= x \;\big|\; 0 \;\big|\; S(t) \;\big|\; t + s \;\big|\; t \cdot s \;\big|\; \lfloor \frac{1}{2} t \rfloor \;\big|\; |x| \;\big|\; \#.$$

Definition 23 (Formulæ)**.** Let $x, y, \dots$ denote variables and $t, s, \dots$ terms. Formulæ are defined by the following grammar:

$$F, G ::= t = s \;\big|\; t \leq s \;\big|\; \neg F \;\big|\; F \wedge G \;\big|\; F \vee G \;\big|\; F \rightarrow G \;\big|\; \exists x.F \;\big|\; \forall x.F.$$

The domain of $\mathcal{L}_{\mathsf{Buss}}$ terms is $\mathbb{N}$, and term functions are interpreted as follows:

- $S(x)$ is the successor function.
- $+, \cdot$ are respectively sum and product.
- $\lfloor \frac{1}{2} x \rfloor$ computes the integer part of the number obtained dividing x by 2.

- $x\#y : 2^{|x|+|y|}$.
- $|x| := \lceil \log_2(x+1) \rceil$. Notice that $|x|$ computes the size of the encoding of x in base 2.

Together with this logic, Buss defines a syntactical hierarchy of bounded formulæ which is aimed to characterize the whole Polynomial Hierarchy [27] by means of *provable, bounded* formulæ. However, within this short introduction, we are not interested in studying relativized complexity classes so we will only define the set of Σ_1^b formulæ, which are meant to characterize **P** throughout the notion of Σ_1^b representability. Indeed, the Randomized Bounded Arithmetic presented in Chapters 2 and 3 is aimed to capture the **PPT** functions or their sub-classes, without dealing with relativized complexity classes. Thus, it is defined on the same notions used by Buss to characterize the class **FP**, extending them to probabilistic computation.

Definition 24 (Σ_1^b-Formula)**.** A Σ_0^b-formula is a subword quantified formula, i.e. a formula belonging to the smallest class of $\mathcal{L}_{\mathsf{Buss}}$ containing atomic formulæ and closed under Boolean operations and subword quantifications. A Σ_1^b*-formula* in $\mathcal{L}_{\mathsf{Buss}}$ is a formula of the form $\exists x \leq |t(\vec{z})|.F(\vec{z}, x)$, where F is a subword quantified formula. For simplicity, we will call Σ_1^b the class containing all and only the Σ_1^b-formulæ.

1.4.2 The S_2^1 Theory

An insightful result of Buss' work is the characterization of complexity classes by mans of the notion of *provability* under a constructive deduction system. This is an insightful result: indeed, under a *constructive* proof system any proof of an existential formula must contain, or imply the existence of, algorithms for finding the object which is proved to exist, [7]. For instance, if $\forall x.\exists y.A(x, y)$ is provable, then there must be an algorithm exhibiting y given x. As we state above, we are mainly interested in the characterization of polynomial time complexity, for this reason, we will only show the arithmetical description given for the class **P**. The theory introduced by Buss to this aim is called S_2^1 and is defined as follows:

Definition 25 (S_2^1 Theory)**.** S_2^1 is the first-order theory with language $\mathcal{L}_{\mathsf{Buss}}$ containing:

- A set of basic equational axioms defining the function symbols of $\mathcal{L}_{\mathsf{Buss}}$ with respect to $\leq$ and to the identity predicate [6].
- The polynomial induction schema PIND:

  $$A(0) \land \forall x.(A(\lfloor \frac{1}{2}x \rfloor) \to A(x)) \to \forall x.A(x).$$

  for every $A \in \Sigma_1^b$.

The notion of Σ_1^b-representability which captures **FP** is given as follows:

Definition 26 (Σ_1^b Representability)**.** Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be a function. It is Σ_1^b definable if and only if:

- $\forall \vec{n} \in \mathbb{N}^k.A(\vec{n}, f(\vec{n}))$ holds.
- $S_2^1 \vdash \forall \vec{x}.\exists! y.A(\vec{x}, y)$.

Theorem 2. *A function f is in* **P** *if and only if it is Σ_1^b definable.*

The proof of this result requires a significant amount of technical work, so we will not show it within this section, however it can be found in Buss' PhD thesis [6]. Moreover, that proof is structurally similar to one showing that our notion of Σ_1^b representability captures exactly the class of **PPT** functions. Indeed, in Chapter 4, we prove that part of the Buss' proof can be given as a corollary of some our results.

Before getting into the discussion of our Randomized Bounded Arithmetic, we would like to point out some of the intuitions behind Theorem 2. Because the careful reader will certainly observe that similar guidelines have been taken in account in Chapter 2 defining our Randomized Bounded Arithmetic.

- The definition of $\mathcal{L}_{\mathsf{Buss}}$'s terms is aimed to impose a polynomial grow rate to the size of quantified terms. As a consequence, referring to the definition of Σ_1^b representability, we know that the size of all the terms within that formula are polynomially bound in the size of $\vec{x}$.
- The PIND induction system differs form the standard induction system on natural numbers because of the stronger induction hypothesis. This is done with the aim to impose a polynomial bound to the number of induction steps within a proof. Indeed, due to the *constructiveness* of this axiom system, any derivation of a S_2^1 formula corresponds to the definition of a function. Thus, due to the Curry Howard isomorphism, bounding the number of inductive steps within a constructive proof, is equivalent to bounding the number of recursive calls in the represented function.

Finally, we would like to point out that Buss' S_2^1 Theory is not the only bounded arithmetic characterizing **P**. Indeed, an interesting variation on Buss' work is Ferreira's PTCF [14]. This theory is equivalent to Buss' S_2^1, but it is defined on a different set of axioms. Whilst Buss' Bounded arithmetics is modeled on natural numbers, Ferreira's PTCA is modeled directly on binary strings. This, in our opinion, has some benefits on the overall theory: it requires a smaller set of function, and thus less axioms, it has a more natural notion of term size and sharp bounds. Finally, it emphasizes the attention on the size of the terms rather than on the natural number they represent, making it easier to deal with time-complexities. These are some of the reasons why we decided to ground our Randomized Bounded Arithmetic on strings rather than natural numbers.

Chapter 2

A Randomized Bounded Arithmetic

In this chapter, we introduce a novel bounded arithmetic for randomized computation strongly inspired by Ferreira's PTCA, [13]. As for PTCA, the terms of our arithmetic describe binary strings by means of two simple operations: concatenations and repetitions. The only syntactical difference between our arithmetic and Ferreira's is the presence of a predicate `Flip`, which we will discuss later. However, the main difference between our bounded arithmetic and Ferreira's or Buss' ones lies within its semantics: we defined a novel quantitative semantics which associates to each formula a measurable set of functions, which can be used to associate a degree of probability to each formula our bounded arithmetic. This Randomized Bounded Arithmetic (RBA) is based on a new first-order "probabilistic language" called $\mathcal{L}$, and is aimed to capture the class of **PPT** functions by means of a slightly modified notion of Σ^b_i-representability and the theory RS^1_2: a variation on Buss' S^1_2 [6]. To this aim, we define the class of *polynomial-time oracle recursive functions* $\mathcal{POR}$ and a notion of arithmetical representability which extends Buss' and Ferreira's. Finally, we show that $\mathcal{POR}$ functions are exactly the Σ^b_1-representable ones. However, we will not examine the details of the proofs, but we will only outline their high-level structure. This because the main focus of this work is on my personal contributions to this research, while the proofs of the aforementioned results have been mainly developed by my collaborators Melissa Antonelli, Ugo Dal Lago, Isabel Oitavem and Paolo Pistone. For the detailed proofs of the results within this section, we invite the reader to consult [3], an unpublished set of notes, describing exhaustively and extensively the details of this research. This chapter is structured as follows:

1. In Section 2.1, we define the first-order language $\mathcal{L}$ together with its semantics and the $\mathcal{POR}$ class of functions.
2. In Section 2.2, we outline the proof that all functions in $\mathcal{POR}$ are Σ^b_1-representable in RS^1_2. The proof is by induction on the structure of $\mathcal{POR}$ and is inspired by the encoding machinery from [6, 13].
3. In Section 2.3 we outline the proof that all functions which are Σ^b_1-representable in RS^1_2 are in $\mathcal{POR}$ by way of realizability techniques similar to Cook and Urquhart's one [10].

These results — namely Theorem 3 and 3 — prove that Σ^b_1 representable function of RS^1_2 are exactly the $\mathcal{POR}$ functions, as stated by Theorem 3.

Theorem 3 (Σ^b_1-Representability). *Within $\mathcal{L}$, the Σ^b_1-representable functions of RS^1_2 are exactly the $\mathcal{POR}$ functions.*

In Chapter 3, this result will be extended to the set of **PPT** functions.

Theorem 4 (Σ^b_1-Representability of **PPT** Functions)**.** *It holds that:*

$$\forall G \in \Sigma^b_1.RS^1_2 \vdash \forall x \exists ! y.G(x,y) \to \exists f_G \in \mathbf{PPT}.\forall x,y \in \mathbb{S}.\mu\left(\llbracket G(x,y) \rrbracket\right) = Pr[f_G(x) = y]$$

and that:

$$\forall f \in \mathbf{PPT}.\exists G_f \in \Sigma^b_1..RS^1_2 \vdash \forall x \exists ! y.G(x,y) \land \forall x,y \in \mathbb{S}.\mu\left(\llbracket G(x,y) \rrbracket\right) = Pr[f_G(x) = y].$$

2.1 The $\mathcal{POR}$ Function Algebra and RS^1_2 Theory

In this section, we collect the definitions of the $\mathcal{POR}$ (Section 2.1.2) function algebra and the system of axioms RS^1_2, (Section 2.1.3).[1] Before doing so, for clarity's sake, we define the notational conventions we are adopting in this whole work. This is done in Section 2.1.1.

2.1.1 Notational preliminaries

Definition 27 (Sets)**.**

- We call $\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}$ the set of bits.
- $\mathbb{S} = \mathbb{B}^*$ is the set of binary strings of finite length.
- $\mathbb{O} = \mathbb{B}^{\mathbb{S}}$ indicates the set of functions from $\mathbb{N}$ to $\mathbb{B}$.

Notice that given $\omega \in \mathbb{B}$ and $x \in \mathbb{S}$, $\omega(x)$ denotes *one* specific bit $b \in \mathbb{B}$, the so-called x-th bit of ω. Meta-variables $\omega', \omega'', \dots$ are used to denote the elements $\mathbb{O}$.

For every $x, y \in \mathbb{S}$, we will use $x \subseteq y$ to express that x is an *initial* or *prefix substring* of y. Within $\mathbb{S}$, we define two binary operations (determining a ring) which allow us to define strings on top of other strings.

Definition 28 (String concatenation)**.** Given two strings $x, y \in \mathbb{S}$, we denote with $x \frown y$ (which will always be abbreviated as simply xy) the concatenation of x and y.

Definition 29 (Binary Product)**.** Given two strings $x, y \in \mathbb{S}$, we define their binary product, denoted $x \times y$, the string obtained by concatenating x with itself for $|y|$-times. Formally:

$$\begin{aligned} x \times \boldsymbol{\epsilon} &:= \boldsymbol{\epsilon} \\ x \times y\mathtt{b} &:= (x \times y) \frown x. \end{aligned}$$

2.1.2 The Class $\mathcal{POR}$

The $\mathcal{POR}$ class of function is strongly inspired by Ferreira's PTCA [13], which itself is a Cobham style function algebra [9]. The main difference between $\mathcal{POR}$ and Ferreira's PTCA is that $\mathcal{POR}$ functions carry an additional functional argument $\omega : \mathbb{S} \longrightarrow \mathbb{B}$ which is intended to be used as a source of random bits. To do so, a novel function is added to the set of base functions, i.e. the query function $Q(x, \omega) = \omega(x)$.

Definition 30 (The Class $\mathcal{POR}$)**.** The *class* $\mathcal{POR}$ is the smallest class of functions from $\mathbb{S}^n \times \mathbb{O}$ to $\mathbb{S}$, containing:

[1]We would like to precise that for brevity we call RS^1_2 both the system of axioms and the theory they induce. However, to be proper, we should refer the theory using a different name.

- The *empty* function $E(x, \omega) = \boldsymbol{\epsilon}$;
- The *projection* functions $P^n_i(x_1, \dots, x_n, \omega) = x_i$, for $n \in \mathbb{N}$ and $1 \leq i \leq n$;
- The *word-successor* $S_{\mathtt{b}}(x, \omega) = x\mathtt{b}$, for every $\mathtt{b} \in \mathbb{B}$;
- The *conditional* function

$$C(\boldsymbol{\epsilon}, y, z_{\mathtt{0}}, z_{\mathtt{1}}, \omega) = y$$
$$C(x\mathtt{b}, y, z_{\mathtt{0}}, z_{\mathtt{1}}, \omega) = z_{\mathtt{b}},$$

  where $\mathtt{b} \in \mathbb{B}$.
- The *query* function $Q(x, \omega) = \omega(x)$;

and closed under:

- *Composition*, such that f is defined from $g, h_1, \dots, h_k$ as

$$f(\vec{x}, \omega) = g(h_1(\vec{x}, \omega), \dots, h_k(\vec{x}, \omega), \omega).$$

- *Bounded recursion on notation*, such that f is defined from g, h_0, h_1 as

$$f(\vec{x}, \boldsymbol{\epsilon}, \omega) = g(\vec{x}, \omega);$$
$$f(\vec{x}, y\mathtt{0}, \omega) = h_0\big(\vec{x}, y, f(\vec{x}, y, \omega), \omega\big)|_{t(\vec{x},y)};$$
$$f(\vec{x}, y\mathtt{1}, \omega) = h_1\big(\vec{x}, y, f(\vec{x}, y, \omega), \omega\big)|_{t(\vec{x},y)},$$

  where t is defined from $\boldsymbol{\epsilon}, \mathtt{0}, \mathtt{1}, \frown, \times$ by explicit definition, which means that t can be obtained by a finite term using composing the functions $\frown$ and $\times$ on the constants $\boldsymbol{\epsilon}$, $\mathtt{0}$, $\mathtt{1}$ and the variables $\vec{x}$ and y.[2]

2.1.3 The Theory RS^1_2

The theory RS^1_2 is a bounded arithmetic inspired by Ferreira's [13]. This first-order theory relies on the following components:

- A first-order language $\mathcal{L}$ with equality whose terms represent strings, introduced in Section 2.1.3.1.
- A quantitative semantics $[\![\cdot]\!]$ associating a measurable set to each formula, which is described in Section 2.1.3.2.
- A set of axioms, and a derivation system, discussed in Section 2.1.3.3.

2.1.3.1 The Language $\mathcal{L}$.

The language $\mathcal{L}$ is the first-order language with equality defined in [15], augmented by a predicate symbol $\mathtt{Flip}(\cdot)$, as described below:

Definition 31 (Terms)**.** Let $x, y, \dots$ denote variables. Terms are defined by the following grammar:

$$t, s ::= x \;\big|\; \epsilon \;\big|\; \mathtt{0} \;\big|\; \mathtt{1} \;\big|\; t \frown s \;\big|\; t \times s.$$

[2]This language is identical to the one of $\mathcal{L}$ terms, Definition 31.

Notation 2. The symbol $\frown$ is usually omitted: $t \frown s$ is abbreviated as ts.

Definition 32 (Formulæ)**.** Let $x, y, \dots$ denote variables and $t, s, \dots$ terms. Formulæ are defined by the following grammar:

$$F, G ::= \texttt{Flip}(t) \;\Big|\; t = s \;\Big|\; t \subseteq s \;\Big|\; \neg F \;\Big|\; F \wedge G \;\Big|\; F \vee G \;\Big|\; F \to G \;\Big|\; \exists x.F \;\Big|\; \forall x.F.$$

As a syntactical facilitation, we define some notations which do not extend the language's expressive power, yet they enable us to write more concise and clear formulæ.

Notation 3 (Exponentiation)**.**

- For every term t of $\mathcal{L}$, the abbreviation $\texttt{1}^t$ stands for the term $\texttt{1} \times t$.
- Given two terms of $\mathcal{L}$ t, s, the formula $t \preceq s$ is syntactic sugar for $\texttt{1}^t \subseteq \texttt{1}^s$, meaning that the length of t is less than or equal to that of s.
- If t, r, s are strings, the abbreviation $t|_r = s$ denotes the following formula:

  $$(\texttt{1}^r \subseteq \texttt{1}^t \wedge s \subseteq t \wedge \texttt{1}^r = \texttt{1}^s) \vee (\texttt{1}^t \subseteq \texttt{1}^r \wedge s = t)$$

  saying that s is the *truncation* of t at the length of r.

As for Buss' arithmetic, even our arithmetic relies on bounded quantification. This is done with the aim of keeping the size of the terms within the formulæ under control and then, to keep the functions' time complexity within specific classes. To this aim, Buss introduces two different flavors of quantification, namely bounded quantification and sharply bounded quantification (Definition 21); within our framework, they respectively correspond to bounded quantification and subword quantification.

Notation 4 (Bounded Quantification)**.** In $\mathcal{L}$, *bounded quantification* is quantification in the form $\forall x \preceq t.F$, which abbreviates $\forall x.\texttt{1}^x \subseteq \texttt{1}^t \to F$, or $\exists x \preceq t.F$.

This form of quantification, similarly to Buss' bounded quantification $Qx \leq k$[3], ranges over $O(2^{|k|})$ values of x.

Notation 5 (Subword quantification)**.** *Subword quantification* is quantification in the form $\forall x \subseteq^* t$ and $\exists x \subseteq^* t$, such that $\forall x \subseteq^* t.F$ and $\exists x \subseteq^* t.F$ abbreviate (resp.) $\forall x.(\exists w \subseteq t.(wx \subseteq t) \to F$ and $\exists x.(\exists w \subseteq t.(wx \subseteq t) \wedge F)$.

As for Buss' sharp quantification, $\forall x \leq |k|$, allows the variable x to range over a set of values whose size is linear in k. For readability's sake, in the following, we also abbreviate the so-called *initial subword quantifications* $\forall x.x \subseteq t \to F$ as $\forall x \subseteq t.F$ and $\exists x.x \subseteq t \wedge F$ as $\exists x \subseteq t.F$.

2.1.3.2 Semantics for Formulæ in $\mathcal{L}$

Upon $\mathcal{L}$, we can define different semantics: for the aims of this work, we want our semantics to capture measure-related concerns about $\mathcal{L}$'s formulæ. To this end, we introduce the alternative, *quantitative* semantics for $\mathcal{L}$-terms and formulæ inspired by [1]. Terms are interpreted in a standard way, so mapping constants to canonical strings and functions to members of $\mathbb{S}^{\mathbb{S}\times\mathbb{S}}$. This is done in Definition 33 below. On the other hand, the semantics for formulæ is inherently quantitative, as any formula is associated with a (measurable) set, $[\![F]\!] \in \sigma(\mathscr{C})$. Intuitively, the

[3]With $k \in \mathbb{N}$ and $Q \in \{\forall, \exists\}$

quantitative semantics of $\mathcal{L}$ associates to a formula F the set of characteristic functions $\omega \in \sigma(\mathscr{C})$ such that, if these functions are employed as `Flip`'s interpretations for the *qualitative* semantics of $\mathcal{L}$, then F is valid. In order to convey this intuition, we first introduce the standard *qualitative* semantics then the *quantitative* one and, finally, in Remark 2 we then show their relation.

Definition 33 (Interpretation for Terms)**.** An environment $\xi : \mathcal{G} \mapsto \mathbb{S}$, where $\mathcal{G}$ is the set of term variables, is a mapping that assigns to each variable a string. Given a term t in $\mathcal{L}$ and an environment ξ, the *interpretation of t in ξ* is the string $[\![t]\!]_\xi \in \mathbb{S}$ inductively defined as follows:

$$
\begin{aligned}
[\![\epsilon]\!]_\xi &:= \boldsymbol{\epsilon} & \qquad [\![x]\!]_\xi &:= \xi(x) \in \mathbb{S}\\
[\![\mathtt{0}]\!]_\xi &:= \mathtt{0} & [\![t \frown s]\!]_\xi &:= [\![t]\!]_\xi [\![s]\!]_\xi\\
[\![\mathtt{1}]\!]_\xi &:= \mathtt{1} & [\![t \times s]\!]_\xi &:= [\![t]\!]_\xi \times [\![s]\!]_\xi.
\end{aligned}
$$

The standard, *qualitative* model for terms and formulæ of $\mathcal{L}$ consists in $\mathscr{W} = (\mathbb{S}, \frown, \times)$. In this case, logical operators are interpreted in the canonical way and $\mathtt{Flip}(\cdot)$ is treated as a standard, unary predicate of first-order logic, which is interpreted as a subset of $\mathbb{S}$.

Definition 34 (Qualitative Semantics for $\mathcal{L}$-Formulæ)**.** Given a formula F in $\mathcal{L}$ and an interpretation $\rho = (\xi, \omega^{\mathtt{FLIP}})$, where $\xi : \mathcal{G} \to \mathbb{S}$ and $\omega^{\mathtt{FLIP}} \in \mathbb{O}$, the *interpretation of F in ρ*, $[\![F]\!]_\rho$, is inductively defined as follows:

$$
\begin{aligned}
[\![\mathtt{Flip}(t)]\!]_\rho &:= \begin{cases} 1 & \text{if } \omega^{\mathtt{FLIP}}([\![t]\!]_\xi) = \mathtt{1}\\ 0 & \text{otherwise}\end{cases} &
[\![\neg G]\!]_\rho &:= 1 - [\![G]\!]_\rho\\
& & [\![G \wedge H]\!]_\rho &:= min\{[\![G]\!]_\rho, [\![H]\!]_\rho\}\\
[\![t = s]\!]_\rho &:= \begin{cases} 1 & \text{if } [\![t]\!]_\xi = [\![s]\!]_\xi\\ 0 & \text{otherwise}\end{cases} &
[\![G \vee H]\!]_\rho &:= max\{[\![G]\!]_\rho, [\![H]\!]_\rho\}\\
& & [\![G \to H]\!]_\rho &:= max\{(1 - [\![G]\!]_\rho), [\![H]\!]_\rho\}\\
[\![t \subseteq s]\!]_\rho &:= \begin{cases} 1 & \text{if } [\![t]\!]_\xi \subseteq [\![s]\!]_\xi\\ 0 & \text{otherwise}\end{cases} &
[\![\forall x.G]\!]_\rho &:= min\{[\![G]\!]_{(\xi\{x \leftarrow s\}, \omega^{\mathtt{FLIP}})} \mid s \in \mathbb{S}\}\\
& & [\![\exists x.G]\!]_\rho &:= max\{[\![G]\!]_{(\xi\{x \leftarrow s\}, \omega^{\mathtt{FLIP}})} \mid s \in \mathbb{S}\}.
\end{aligned}
$$

The model $\mathscr{W}$ can be extended to a probability space by considering as the underlying sample space $\mathbb{O}$. There are standard ways of building a well-defined σ-algebra and a probability space over $\mathbb{O}$.

Definition 35. For every countable set A, each K a finite subset of A and $H \subseteq \mathbb{B}^K$, each subset of $\mathbb{B}^A$ of the form:

$$\mathsf{C}(H) = \{\omega \in \mathbb{B}^A \mid \omega|_K \subseteq H\},$$

are called *cylinders* over A, [5].[4]

Let $\mathscr{C}$ denote the set of all cylinders and the σ-algebra generated by cylinders over $\mathbb{O}$. The smallest σ-algebra including $\mathscr{C}$, which is Borel's, is indicated as $\sigma(\mathscr{C})$. There is a natural way of defining a probability measure μ on $\mathscr{C}$, the canonical way is described in Definition 36.

Definition 36 (Cylinder measure, [5])**.** Be $S \subseteq \{\mathtt{0},\mathtt{1}\}^A$ for a countable A, K a finite subset of A and $H \subseteq \mathbb{B}^K$, (in this context $A = \mathbb{S}$, but later A we will tale in exam even the case $A = \mathbb{N}$) such that:

$$S = \{\omega \in \{\mathtt{0},\mathtt{1}\}^A \,|\, \omega|_K \in H\}$$

then, we define $\mu(S)$ as follows:

$$\mu(S) = \frac{|H|}{2^{|K|}}.$$

[4]To be precise, these objects are a slight variation of Billingsley's *cylinders*, which are subsets of $\{\mathtt{0},\mathtt{1}\}^{\mathbb{N}}$.

Without proving that Definition 36 is a measure over the σ-field $\sigma(\mathscr{C})$, we still cannot introduce any quantitative semantics for $\mathcal{L}$. These results are in Chapter 6: in Remark 21, we prove that μ is a function, while in Remark 22, we show that μ is a measure function on $\sigma(\mathscr{C})$.

Due to the existence of μ, the canonical model $\mathscr{W} = (\mathbb{S}, \frown, \times)$ can be generalized to $\mathscr{P} = (\mathbb{O}, \frown, \times, \sigma(\mathscr{C}), \mu_{\mathscr{C}})$. As anticipated, this new semantics has a concrete application based of probability experiments. Indeed, when interpreting sequences in $\mathbb{O}$ as the outcome of Bernoulli's process, the set of sequences such that the n-th coin flip's result is $\mathtt{1}$ (for any fixed $n \in \mathbb{N}$) is assigned measure $\frac{1}{2}$, meaning that each random bit is uniformly distributed and independent of the others. Namely:

$$[\![\mathtt{Flip}(t)]\!]_\xi := \mathsf{C}(\{[\![t]\!]_\xi \mapsto \mathtt{1}\}).$$

Definition 37 (Quantitative Semantics)**.** Given a formula F and an environment $\xi : \mathcal{G} \to \mathbb{S}$, where $\mathcal{G}$ is the set of term variables, the *interpretation of F in ξ*, $[\![F]\!]_\xi$, is the (measurable) set of sequences inductively defined as follows:

$$[\![\mathtt{Flip}(t)]\!]_\xi := \{\omega \mid \omega([\![t]\!]_\xi) = \mathtt{1}\}$$

$$[\![t = s]\!]_\xi := \begin{cases} \mathbb{O} & \text{if } [\![t]\!]_\xi = [\![s]\!]_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![t \subseteq s]\!]_\xi := \begin{cases} \mathbb{O} & \text{if } [\![t]\!]_\xi \subseteq [\![s]\!]_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned} [\![\neg G]\!]_\xi &:= \mathbb{O} \setminus [\![G]\!]_\xi \\ [\![G \vee H]\!]_\xi &:= [\![G]\!]_\xi \cup [\![H]\!]_\xi \\ [\![G \wedge H]\!]_\xi &:= [\![G]\!]_\xi \cap [\![H]\!]_\xi \\ [\![G \to H]\!]_\xi &:= (\mathbb{O} \setminus [\![G]\!]_\xi) \cup [\![H]\!]_\xi \\ [\![\exists x.G]\!]_\xi &:= \bigcup_{i \in \mathbb{S}} [\![G]\!]_{\xi\{x \leftarrow i\}} \\ [\![\forall x.G]\!]_\xi &:= \bigcap_{i \in \mathbb{S}} [\![G]\!]_{\xi\{x \leftarrow i\}}. \end{aligned}$$

The semantics is well-defined since the sets $[\![\mathtt{Flip}(t)]\!]_\xi$, $[\![t = s]\!]_\xi$ and $[\![t \subseteq s]\!]_\xi$ are measurable and measurability is preserved by all the logical operators.

Notation 6. For readability's sake, in the following part of this work, we may abbreviate the former interpretation as $[\![\cdot]\!]_\omega$ and the quantitative interpretation $[\![\cdot]\!]_\xi$ simply as $[\![\cdot]\!]$.

As we mentioned before, the *quantitative* and the *qualitative* semantics are strictly related one with each other:

Remark 2. *For each $\mathcal{L}$ formula F, environment ξ, function $\omega \in \mathbb{O}$ and each interpretation $\rho = (\omega, \xi)$, let $[\![\cdot]\!]_\xi$ the* quantitative *semantics for $\mathcal{L}$ described in Definition 37 and $[\![\cdot]\!]_\rho$ the* qualitative *semantics for $\mathcal{L}$ described in Definition 34, then it holds that:*

$$[\![F]\!]_{(\xi,\omega)} = 1 \leftrightarrow \omega \in [\![F]\!]_\xi.$$

Proof. The proof goes by induction on F's syntax. The only base case is `Flip`'s, which is a direct consequence of Definitions 37 and 34. All the other cases are trivial consequences of the induction hypothesis(es) and the definitions of the two semantics. □

2.1.3.3 The Theory RS^1_2.

As we anticipated in the previous sections, the theory RS^1_2 includes the axioms by [13] as expressed in $\mathcal{L}$ using any derivation system for classical first-order logic, such as, for example, sequent calculus or natural deduction. As for Buss' Bounded Arithmetic, our characterization relies on syntactically defined formulæ. In particular, the $\mathcal{POR}$ class will be captured by particular Σ^b_1 formulæ.

Definition 38 (Σ_1^b-Formula)**.** A Σ_0^b-formula is a subword quantified formula, i.e. a formula belonging to the smallest class of $\mathcal{L}$ containing atomic formulæ and closed under Boolean operations and subword quantifications. A Σ_1^b*-formula* in $\mathcal{L}$ is a formula of the form $\exists x.\big(x \preceq t(\vec{z}) \land F(\vec{z}, x)\big)$, where F is a subword quantified formula. For simplicity, we will call Σ_1^b the class containing all and only the Σ_1^b-formulæ.

Definition 39 (Theory RS_2^1)**.** The theory RS_2^1 is defined by axioms belonging to two classes:

- *Basic* axioms:
  1. $x\epsilon = x$;
  2. $x(y\mathtt{0}) = (xy)\mathtt{0}$;
  3. $x(y\mathtt{1}) = (xy)\mathtt{1}$;
  4. $x \times \epsilon = \epsilon$;
  5. $x \times y\mathtt{0} = (x \times y)x$;
  6. $x \times y\mathtt{1} = (x \times y)x$;
  7. $x \subseteq \epsilon \leftrightarrow x = \epsilon$;
  8. $x \subseteq y\mathtt{0} \leftrightarrow x \subseteq y \lor x = y\mathtt{0}$;
  9. $x \subseteq y\mathtt{1} \leftrightarrow x \subseteq y \lor x = y\mathtt{1}$;
  10. $x\mathtt{0} = y\mathtt{0} \rightarrow x = y$;
  11. $x\mathtt{1} = y\mathtt{1} \rightarrow x = y$;
  12. $x\mathtt{0} \neq y\mathtt{1}$;
  13. $x\mathtt{0} \neq \epsilon$;
  14. $x\mathtt{1} \neq \epsilon$.
- Axiom scheme for *induction on notation*:

$$B(\epsilon) \land \forall x.\big(B(x) \rightarrow B(x\mathtt{0}) \land B(x\mathtt{1})\big) \rightarrow \forall x.B(x),$$

  where B is a Σ_1^b-formula in $\mathcal{L}$.

And closed under the rules of standard first-order classical logic.

2.2 All Functions in $\mathcal{POR}$ are Σ_1^b-Representable in RS_2^1

In this section we show that each polynomial-time oracle function is Σ_1^b-representable in our theory RS_2^1. To this end, we extend the standard definition of Σ_1^b-representability so to fit our class of string functions and the probabilistic word language $\mathcal{L}$.

Definition 40 (Σ_1^b-Representability)**.** A function $f : \mathbb{S}^j \times \mathbb{O} \rightarrow \mathbb{S}$ is Σ_1^b*-representable* in RS_2^1 if and only if there is a Σ_1^b-formula $G(\vec{x}, y)$ in $\mathcal{L}$ such that:

1. $RS_2^1 \vdash \forall \vec{x}.\exists y.G(\vec{x}, y)$
2. $RS_2^1 \vdash \forall \vec{x}.\forall y.\forall z.\big(G(\vec{x}, y) \land G(\vec{x}, z) \rightarrow y = z\big)$
3. For all $n_1, \dots, n_j, m \in \mathbb{S}$, and $\omega \in \mathbb{O}$, $f(n_1, \dots, n_j, \omega) = m$ if and only if $\omega \in [\![G(n_1, \dots, n_j, m)]\!]$.

This definition extends Buss' representability condition, adding a constraint which links the formula's qualitative semantics to the additional functional parameter of the $\mathcal{POR}$ function algebra. This constraint — the third one — is the characterizing feature of this definition. Indeed, it captures the main peculiarity of probabilistic computation: evaluation is a stochastic process so the semantics of a program is a function mapping an input to a probability distribution which reflects all the possible outcomes of the computation. The main result of this section is Theorem 5. We do not show an entire and comprehensive proof of the result, because I did not directly work on it. However, we will show an exhaustive sketch of the proof given in [3]. The proof relies on a well-known result by Parikh.

Proposition 2 ("Parikh" [22]). *Let $F(\vec{x}, y)$ be a bounded formula in $\mathcal{L}$ such that $RS^1_2 \vdash (\forall \vec{x})(\exists y)F(\vec{x}, y)$. Then, there is a term t such that $RS^1_2 \vdash \forall \vec{x}.\exists y \preceq t(\vec{x}).F(\vec{x}, y)$.*

Actually, Parikh's theorem is usually presented in the context of Buss' bounded theories, stating that, if B is a first-order formula with variables interpreted over Natural Numbers, then if $S^i_2 \vdash \forall \vec{x}.\exists y.B$, there is a term $t(\vec{x})$ such that $S^i_2 \vdash \forall \vec{x}.\exists y \leq t(\vec{x}).B(\vec{x}, y)$, [6, 7]. However, due to [15], Buss' *syntactic* proof holds for Ferreira's Σ^b_1-NIA [14] as well. Finally, the same result holds for RS^1_2, because the set of its axioms does not contain any specific rule concerning $\texttt{Flip}(\cdot)$ and so is defined exactly upon the same axioms of Ferreira's Σ^b_1-NIA [13].

Theorem 5. *Every $f \in \mathcal{POR}$ is Σ^b_1-representable in RS^1_2.*

Proof Sketch. The proof is by induction on the structure of functions in $\mathcal{POR}$.

Base case. Each basic function is Σ^b_1-representable in RS^1_2.

- $f = E$ is Σ^b_1-represented in RS^1_2 by the formula:

$$G_E(x, y) := x = x \wedge y = \epsilon.$$

1. Given x, it suffices to take $y = 1$ as a witness of the existential quantifier.
2. Uniqueness is a consequence of identity's transitivity.
3. The quantitative claim comes is a consequence of Definition 37.

All the base case are similar to this one, apart from Q, which is shown below:

$f = Q$ is Σ^b_1-represented in RS^1_2 by the formula:

$$G_Q(x, y) := \big(\texttt{Flip}(x) \wedge y = \texttt{1}\big) \vee \big(\neg\texttt{Flip}(x) \wedge y = \texttt{0}\big).$$

Notice that this proof relies on the fact that every $f \in \mathcal{POR}$ is capable of invoking exactly *one* oracle.

1. Existence is proved by cases: if $RS^1_2 \vdash \texttt{Flip}(x)$, let $y = \texttt{1}$. By the reflexivity of identity $RS^1_2 \vdash \texttt{1} = \texttt{1}$ holds, so also $RS^1_2 \vdash \texttt{Flip}(x) \wedge \texttt{1} = \texttt{1}$. In this case we conclude that:

$$RS^1_2 \vdash \exists y.((\texttt{Flip}(x) \wedge y = \texttt{1}) \vee (\neg\texttt{Flip}(x) \wedge y = \texttt{0})).$$

If $RS^1_2 \vdash \neg\texttt{Flip}(x)$, the proof is analogous.

2. Uniqueness is established relying on the transitivity of identity.

3. Finally, we need to show that for every $n, m \in \mathbb{S}$ and $\omega^* \in \mathbb{O}$, $Q(n, \omega^*) = m$ if and only if $\omega^* \in [\![G_Q(n, m)]\!]$.
   Suppose $m = \mathtt{1}$. Then it holds that $Q(n, \omega^*) = \mathtt{1}$, which is equivalent to $\omega^*(n) = \mathtt{1}$, thus:

$$\begin{aligned}
\Big[\!\Big[\big(\mathtt{Flip}(n) \wedge m = \mathtt{1}\big) \vee \big(\neg\mathtt{Flip}(n) \wedge m = \mathtt{0}\big)\Big]\!\Big] &= [\![\mathtt{Flip}(n) \wedge m = \mathtt{1}]\!] \cup [\![\neg\mathtt{Flip}(n) \wedge m = \mathtt{0}]\!] \\
&= \big([\![\mathtt{Flip}(n)]\!] \cap [\![\mathtt{1} = \mathtt{1}]\!]\big) \cup \big([\![\neg\mathtt{Flip}(n)]\!] \cap [\![\mathtt{1} = \mathtt{0}]\!]\big) \\
&= \big([\![\mathtt{Flip}(n)]\!] \cap \mathbb{O}\big) \cup \big([\![\neg\mathtt{Flip}(n)]\!] \cap \emptyset\big) \\
&= [\![\mathtt{Flip}(n)]\!] \\
&= \{\omega \in \mathbb{O} \mid \omega(n) = \mathtt{1}\}.
\end{aligned}$$

   Clearly, $\omega^* \in \Big[\!\Big[\big(\mathtt{Flip}(n) \wedge m = \mathtt{1}\big) \vee \big(\neg\mathtt{Flip}(n) \wedge m = \mathtt{0}\big)\Big]\!\Big]$. The case $m = \mathtt{0}$ and the opposite direction are proved similarly.

Inductive case. The discussion of this cases is out of the scope of this work. However, The proofs of claims 1 and 2 follow the same approach described by Ferreira in [14, 13]. In particular this can be done because, from a *syntactical* point of view, RS^1_2 axioms and Ferreira's are exactly the same. This allows us to reuse Parikh's Proposition [22] (rewritten in Proposition 2) to fit the induction hypothesis into the Σ^b_1 representability condition of the claims. Indeed, the induction hypothesis does not pose any bound over y's size. But according to Proposition 2 this can be done. So, for instance, the case of composition, the Σ^b_1 formula is:

$$G(x, y) := \exists z_1 \preceq t_{h_1}(\vec{x}). \dots \exists z_k \preceq t_{h_k}(\vec{x}).\big(G_{h_1}(\vec{x}, z_1) \wedge \dots \wedge G_{h_k}(\vec{x}, z_k) \wedge G_g(z_1, \dots, z_k, y)\big).$$

where the existence of $h_1, \dots, h_k$ is a consequence of Parikh's Proposition.

□

2.3 From RS^1_2 Representability to $\mathcal{POR}$

In this section, we show that if a function is Σ^b_1-representable in RS^1_2, then it is in $\mathcal{POR}$, as stated by Theorem 6

Theorem 6. *Let $RS^1_2 \vdash \forall x.\exists y \preceq t.A(x, y)$, where A is a Σ^b_1-formula with only x, y free. For any function $f : \mathbb{S} \times \mathbb{O} \to \mathbb{S}$, if $\forall x.\exists y \preceq t.A(x, y)$ represents f so that:*

1. *$RS^1_2 \vdash \forall x.\exists! y.A(x, y)$*
2. *$[\![A(s_1, s_2)]\!] = \{\omega \mid f(s_1, \omega) = s_2\}$,*

then $f \in \mathcal{POR}$.

The proof is based on the work by Cook and Urquhart's for the system IPV$^\omega$ [10] and is structured as follows:

1. First, in Section 2.3.1, we define a basic equational theory $\mathcal{POR}^\lambda$ for a simply typed λ-calculus enriched with primitives corresponding to functions of $\mathcal{POR}$.
2. Then, in Section 2.3.2, we we show that for each Σ^b_1-representable function there is a $\mathcal{POR}^\lambda$ term which computes that function. To do so, we define a first-order *intuitionist* theory $I\mathcal{POR}^\lambda$, which extends $\mathcal{POR}^\lambda$ with usual predicate calculus. Then, we show that from any derivation of $\forall x.\exists y.A(x, y)$, where A is a Σ^b_0-formula, one can extract a λ-term $\mathbf{t}$ of $\mathcal{POR}^\lambda$ such that $\forall x.A(x, \mathbf{t}x)$ is provable in $I\mathcal{POR}^\lambda$. This allows us to conclude that every function which is Σ^b_1-representable in IRS^1_2 is in $\mathcal{POR}$. Finally, we extend this result to classical RS^1_2.

As for Section 2.2, we will not get into the technical details of the proof, as they can be consulted in [3].

2.3.1 The System $\mathcal{POR}^\lambda$

$\mathcal{POR}^\lambda$ is an equational theory for a simply typed λ-calculus augmented typed constants for $\mathcal{POR}$-like primitive functions.

2.3.1.1 The $\mathcal{POR}^\lambda$ Theory

$\mathcal{POR}^\lambda$ is an equational theory whose terms are ordinary simply-typed λ-terms. Together with ordinary λ-terms, the language of $\mathcal{POR}^\lambda$ contains constants corresponding to $\mathcal{POR}$-like primitives (Definition 41). Thus, the derivation system of $\mathcal{POR}^\lambda$ is induced by a set of equational-axioms defining the reduction rules of the base functions of $\mathcal{POR}^\lambda$, λ-terms and a set derivation rules describing the behavior of the identity predicate.

Definition 41 (Terms of $\mathcal{POR}^\lambda$)**.** *Terms of $\mathcal{POR}^\lambda$* are standard, simply typed λ-terms plus the constants from the signature below:

$$\begin{aligned}
0, 1, \epsilon &: \mathsf{s} \\
\cdot &: \mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s} \\
\mathsf{Tail} &: \mathsf{s} \Rightarrow \mathsf{s} \\
\mathsf{Trunc} &: \mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s} \\
\mathsf{Cond} &: \mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s} \\
\mathsf{Flipcoin} &: \mathsf{s} \Rightarrow \mathsf{s} \\
\mathsf{Rec} &: \mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{s} \Rightarrow \mathsf{s}.
\end{aligned}$$

In addition to the constants of Definition 41, we employ a set of axioms endowing specific equational behaviors to these symbols. These equations ground a $\mathcal{POR}$-like function algebra in which:

- $\mathsf{Tail}(x)$ computes the string obtained by deleting the rightmost of x; thus it can be defined by the axioms:

$$\begin{aligned}
\mathsf{Tail}(\boldsymbol{\epsilon}) &= \boldsymbol{\epsilon} \\
\mathsf{Tail}(xb) &= b.
\end{aligned}$$

- $\mathsf{Trunc}(x, y)$ outputs the *left* prefix x obtained truncating x at the length of y; this corresponds to the $\mathcal{L}$ (and $\mathcal{POR}$'s truncating operator $\cdot|.$).
- $\mathsf{Cond}(x, y, z, w)$ corresponds to $\mathcal{POR}$'s C: indeed, it computes the function that yields y when $x = \boldsymbol{\epsilon}$, z when $x = x'\mathsf{0}$, and w when $x = x'\mathsf{1}$;
- $\mathsf{Flipcoin}(x)$ is a random bit generator, corresponding to $\mathcal{POR}$'s Q.
- Rec is the operator for bounded recursion on notation.

Notice that with respect to $\mathcal{POR}$, we are not defining a composition term constant, this because λ-calculus is itself compositional. For readability's sake, we will use some syntactical conventions:

Notation 7. We will usually denote $x \cdot y$ simply as xy. Moreover, to enhance readability, let $\mathtt{T}$ be any constant $\mathsf{Tail}, \mathsf{Trunc}, \mathsf{Cond}, \mathsf{Flipcoin}, \mathsf{Rec}$ of arity n, we indicate $\mathtt{T}\mathsf{u}_1 \ldots \mathsf{u}_n$ as $\mathtt{T}(\mathsf{u}_1, \ldots, \mathsf{u}_n)$.

$\mathcal{POR}^\lambda$ is reminiscent of PV$^\omega$ by Cook and Urquhart [10]; however an important variation is the constant $\mathsf{Flipcoin}$, deonting a function which randomly generates either $\mathtt{0}$ or $\mathtt{1}$ depending on its input string. As for RS_2^1, our deduction system does not have $\mathsf{Flipcoin}$ specific rules, because the behavior of this primitive will be only given by means of a set of axioms T_ω, depending on a specific $\omega \in \mathbb{O}$, such that:

$$\mathsf{Flipcoin}(t) \in T_\omega \Leftrightarrow \omega(t) = \mathtt{1}$$

Definition 42 ($\mathsf{Flipcoin}$-Defining Axioms)**.** Given a function $\omega \in \mathbb{O}$, we denote with T_ω the set of flip-defining axioms:

$$\begin{aligned} \mathsf{Flipcoin}(\mathtt{0}) &= \omega(\mathtt{0}) \\ \mathsf{Flipcoin}(\mathtt{1}) &= \omega(\mathtt{1}) \\ \mathsf{Flipcoin}(\mathtt{00}) &= \omega(\mathtt{00}) \\ \ldots \quad &= \ldots . \end{aligned}$$

2.3.1.2 Relating $\mathcal{POR}$ and $\mathcal{POR}^\lambda$

In this section, we assess that all the functions within $\mathcal{POR}$ are representable by a term $\mathbf{t} \in \mathcal{POR}^\lambda$. This is done in Theorem 7 by induction on the syntactical syntax of the function. To do so, we define an encoding of strings within $\mathcal{POR}^\lambda$ (Definition 43) and notion of representability of $\mathcal{POR}$ functions in $\mathcal{POR}^\lambda$ (Definition 44).

Definition 43. For any string $s \in \mathbb{S}$, let $\overline{\overline{s}} : \mathsf{s}$ denote the term of $\mathcal{POR}^\lambda$ corresponding to it, i.e.:

$$\begin{aligned} \overline{\overline{\boldsymbol{\epsilon}}} &:= \epsilon \\ \overline{\overline{s\mathtt{0}}} &:= \overline{\overline{s}}\mathtt{0} \\ \overline{\overline{s\mathtt{1}}} &:= \overline{\overline{s}}\mathtt{1}. \end{aligned}$$

Definition 44 (Provable Representability)**.** For every $f : \mathbb{S}^j \times \mathbb{O} \longrightarrow \mathbb{S}$, we say that a term $\mathbf{t} : \mathsf{s} \Rightarrow \ldots \Rightarrow \mathsf{s}$ of $\mathcal{POR}^\lambda$ *provably represents* f if and only if for all strings $s_1, \ldots, s_j, s \in \mathbb{S}$, and $\omega \in \mathbb{O}$,

$$f(s_1, \ldots, s_n, \omega) = s \quad \Leftrightarrow \quad T_\omega \vdash_{\mathcal{POR}^\lambda} \mathbf{t}\overline{\overline{s_1}} \ldots \overline{\overline{s_j}} = \overline{\overline{s}}.$$

with T_ω being a $\mathsf{Flipcoin}$-defining set of axioms, as described in Definition 42

The representability of base functions is almost trivial. Take, for example, the most peculiar of $\mathcal{POR}^\lambda$ base constants: $\mathsf{Flipcoin}$. Its representability is shown in the example below.

Example 2. The term $\mathsf{Flipcoin} : \mathsf{s} \Rightarrow \mathsf{s}$ provably represents the query function $Q(x, \omega) = \omega(x)$ of $\mathcal{POR}$, since for any $s \in \mathbb{S}$ and $\omega \in \mathbb{O}$,

$$\mathsf{Flipcoin}(\overline{\overline{s}}) = \overline{\overline{\omega(s)}} \vdash_{\mathcal{POR}^\lambda} \mathsf{Flipcoin}(\overline{\overline{s}}) = \overline{\overline{Q(s, \omega)}}.$$

This observation can be generalized to cover all the base cases and even the inductive ones, which are composition and bounded recursion on notation. This is stated by the first claim of Theorem 7. The second claim guarantees the inverse direction. An exhaustive proof of this result is out of the scope of this work. For that reason, we will only take in exam some exemplificative cases of the proof. For further details, see [3].

Theorem 7.

1. *Any function $f \in \mathcal{POR}$ is provably represented by a term $\mathbf{t} \in \mathcal{POR}^\lambda$.*
2. *For any term $\mathbf{t} \in \mathcal{POR}^\lambda$, there is a function $f \in \mathcal{POR}$ such that f is provably represented by $\mathbf{t}$.*

Corollary 1. *For any function $f : \mathbb{S}^j \times \mathbb{O} \to \mathbb{S}$, $f \in \mathcal{POR}$ if and only if f is provably represented by some term $\mathbf{t} : \mathsf{s} \Rightarrow \ldots \Rightarrow \mathsf{s} \in \mathcal{POR}^\lambda$.*

2.3.2 Realizability of IRS_2^1

In this section, we show that for each IRS_2^1-representable function it is possible to extract term $\mathbf{t}$ of $\mathcal{POR}^\lambda$. The theory IRS_2^1 is obtained from the same axioms of $\mathcal{L}$, using an intuitionist derivation system instead of a classical logic proof system. To prove this result, we extend $\mathcal{POR}^\lambda$'s language to include first-order quantifiers and propositional connectives, even $\mathcal{POR}^\lambda$'s derivation system is extended with an induction system and an intuitionist predicate calculus. The theory thus obtained is called $\mathcal{IPOR}^\lambda$. The correspondence between $\mathcal{IPOR}^\lambda$ and IRS_2^1 relies on a proof of realizability [7, 10]: by induction on the proof that $IRS_2^1 \vdash F$ we show that we can build an $\mathcal{IPOR}^\lambda$ term $\mathbf{t}$ realizing F, similarly, by induction on the fact that $\mathbf{t}$ realizes F, we can show that F provable under IRS_2^1's axioms. As a consequence of the Definition of realizability given in [3], we get that, if $IRS_2^1 \vdash \forall x.\exists! y F(x,y)$, then the term $\mathbf{t}$ realizing that formula is such that $F(x, \mathbf{t}x)$. This proves the following Theorem.

Theorem 8. *Let $\forall x.\exists y.A(x,y)$ be a closed theorem of $\mathcal{IPOR}^\lambda$, where A is a Σ_1^b-formula. Then, there exists a closed term $\mathbf{t} : \mathsf{s} \Rightarrow \mathsf{s}$ of $\mathcal{POR}^\lambda$ such that:*

$$\vdash_{\mathcal{IPOR}^\lambda} \forall x.A(x, \mathbf{t}x).$$

Now, we have all the ingredients to prove that if a function is Σ_1^b-representable in IRS_2^1, in the sense of Definition 38, then it is in $\mathcal{POR}$.

Corollary 2. *For any function $f : \mathbb{O} \times \mathbb{S} \to \mathbb{S}$, if there is a closed formula Σ_1^b-formula $A(x,y)$ in $\mathcal{L}$ such that:*

1. $IRS_2^1 \vdash \forall x.\exists! y.A(x,y)$
2. $[\![A(\overline{\overline{s_1}}, \overline{\overline{s_2}})]\!] = \{\omega \mid f(\omega, s_1) = s_2\}$,

then $f \in \mathcal{POR}$.

Proof. Since $\vdash_{IRS_2^1} \forall x.\exists! y.A(x,y)$, also $\vdash_{\mathcal{IPOR}} \forall x.\exists! y.A(x,y)$. From $\vdash_{\mathcal{IPOR}^\lambda} \forall x.\exists y.A(x,y)$ we deduce $\vdash_{\mathcal{IPOR}^\lambda} \forall x.A(x, \mathsf{g}x)$ for some closed term $\mathsf{g} : \mathsf{s} \Rightarrow \mathsf{s}$ of $\mathcal{POR}^\lambda$, by Theorem 8 and, by Theorem 7, there is a function $g \in \mathcal{POR}$ such that for any $\omega \in \mathbb{O}, s_1, s_2 \in \mathbb{S}$,

$$T_\omega \vdash_{\mathcal{IPOR}^\lambda} A(\overline{\overline{s_1}}, \overline{\overline{s_2}}) \;\Leftrightarrow\; g(s_1, \omega) = s_2.$$

From this we conclude:

$$\begin{aligned} g(s_1,\omega) = s_2 \;&\Leftrightarrow\; T_\omega \vdash_{\mathcal{IPOR}^\lambda} A(\overline{\overline{s_1}}, \overline{\overline{s_2}}) \\ &\Leftrightarrow\; \omega \in [\![A(\overline{\overline{s_1}}, \overline{\overline{s_2}})]\!] \\ &\Leftrightarrow\; f(s_1, \omega) = s_2. \end{aligned}$$

So, $f = g$ and, thus, $f \in \mathcal{POR}$. □

To conclude this the main claim of this chapter, we show that any function which is Σ^b_1-representable in RS^1_2 is also Σ^b_1-representable in $I\mathcal{POR}^\lambda$. Once this result has been accomplished, Corollary 2 will allow us to obtain the result we are looking for, asserting that all the functions which are also Σ^b_1-representable in IRS^1_2 can be represented $I\mathcal{POR}^\lambda$, too. This representability result is the claim of Proposition 3. Then, it is possible to show that the realizability interpretation extends to such $I\mathcal{POR}^\lambda$ + EM, so that for any of its closed theorems $\forall x.\exists y \preceq \mathbf{t}.A(x,y)$, with A in Σ^b_1, there is a closed term $\mathbf{t} : \mathsf{s} \Rightarrow \mathsf{s}$ of $\mathcal{POR}^\lambda$ such that $\vdash_{I\mathcal{POR}} \forall x.A(x, \mathbf{t}x)$. In doing so, we first introduce the theory $I\mathcal{POR}^\lambda$ + EM, where EM is the Excluded Middle rule:

$$\vdash_{I\mathcal{POR}^\lambda+EM} A \vee \neg A.$$

Thus, we prove that a representability theorem in the style of Theorem 8 relating this class to $I\mathcal{POR}^\lambda$, this is done in Proposition 3.

Proposition 3. *Let $\forall x.\exists y \preceq \mathbf{t}.A(x,y)$ be a closed theorem of $I\mathcal{POR}^\lambda$ + EM, where A is a Σ^b_1-formula. Then, there exists a closed term $\mathbf{t} : \mathsf{s} \Rightarrow \mathsf{s}$ of $\mathcal{POR}^\lambda$ such that:*

$$\vdash_{I\mathcal{POR}^\lambda} \forall x.A(x, \mathbf{t}x).$$

Even in this case, a comprehensive and exhaustive proof is out of the scope of this chapter. The reader who is interested in further details can consult [3].

Then, we can observe than, since $I\mathcal{POR}^\lambda$ + EM contains IRS^1_2 plus the Excluded Middle law, then all the theorems of RS^1_2 are in $I\mathcal{POR}^\lambda$ + EM, too. From this we conclude our proof arguing as for Corollary 2.

Corollary 3. *Let $RS^1_2 \vdash \forall x.\exists y \preceq t.A(x,y)$, where A is a Σ^b_1-formula with only x, y free. For any function $f : \mathbb{S} \times \mathbb{O} \to \mathbb{S}$, if $\forall x.\exists y \preceq t.A(x,y)$ represents f so that:*

1. *$RS^1_2 \vdash \forall x.\exists! y.A(x,y)$*
2. *$[\![A(\overline{\overline{s_1}}, \overline{\overline{s_2}})]\!] = \{\omega \mid f(s_1, \omega) = s_2\}$,*

then $f \in \mathcal{POR}$.

As we mentioned in the introduction of this chapter, Corollary 3, together with Theorem 5, completes the proof of Theorem 3.

Chapter 3

On the equivalence between PPT and $\mathcal{POR}$

In the Chapter 2, we have shown that there is a strong correspondence between the class of functions $\mathcal{POR}$ and the Σ_1^b formulæ of $\mathcal{L}$, precisely that the set of the Σ_1^b representable formulæ of $\mathcal{L}$ are exactly the $\mathcal{POR}$ functions, (Theorem 3).

For this reason, we are now interested in how $\mathcal{POR}$ relates to some other probabilistic classes of functions. Indeed, if we prove that there is correspondence between $\mathcal{POR}$ and some standard class of functions, such as **PPT** (Definition 16), we could be able to define an arithmetical characterization of probabilistic complexity classes such as **BPP** in the style of Theorem 3. To this aim, we show that as it happens for Cobham's algebra [9] and **FP** functions, the $\mathcal{POR}$ function algebra corresponds to the **PPT** class. Moreover, since **BPP** is defined on top of **PPT**, this paves the way for determining a logical characterization of this specific complexity class, as done in Chapter 5.

Informal definitions of PTMs and **PPT** functions have already been given in Chapter 1, but to healp the reader in walking through this section, we will briefly recall those concepts.

Definition 15 (Probabilistic Turing Machines [4])**.** A *probabilistic Turing machine* is a Turing machine with two transition functions, namely δ_0, δ_1. Given an input x, the PTM chooses at each step with probability $\frac{1}{2}$ to apply the transition function δ_0 and with the same probability to apply δ_1. This choice is independent from all the previous ones.

Definition 16 (Class **PPT**)**.** The *class* **PPT** is the class of random functions from $\mathbb{S}$ to $\mathbb{D}(\mathbb{S})$ which are computable by a PTM in at most a polynomial number of steps.

However, formal definitions of PTMs and **PPT** will be given in Section 3.4, respectively in Definitions 101 and 109. Referring to Definitions 15 and 16, we can state more precisely the goal of this chapter: we show that for each $f \in \mathbf{PPT}$, there is a corresponding function $g \in \mathcal{POR}$, such that the probability that $f(x) = y$ is equal to the measure of the set of $\omega \in \mathbb{O}$ with $g(x, \omega) = y$. We show that also the converse claim holds, formally:

Conjecture 1.

i) For each $f \in \mathbf{PPT}$, there is $g \in \mathcal{POR}$ such that:

$$\forall x, y. \Pr\big[f(x) = y\big] = \mu\big(\{\omega \in \mathbb{O} \mid g(x, \omega) = y\}\big).$$

$$\begin{tikzcd}
\mathcal{POR} \arrow[dr, "3.3.2"] & & & & \mathbf{SFP} \arrow[llll, "\text{Section 3.2}"'] \arrow[r, leftrightarrow, "\text{Section 3.4}"] & \mathbf{PPT} \\
& \mathsf{SIFP}_{\mathsf{RA}} \arrow[dr, "3.3.3"'] & & \mathbf{SFP}_{\mathbf{OD}} \arrow[ur, "3.3.5"] & & \\
& & \mathsf{SIFP}_{\mathsf{LA}} \arrow[ur, "3.3.4"'] & & &
\end{tikzcd}$$

Figure 3.1: Schema of the Reductions from $\mathcal{POR}$ from **PPT** and vice-versa

*ii) For each $g \in \mathcal{POR}$, there is $f \in$ **PPT** such that:*

$$\mu\big(\omega \in \mathbb{O} \mid g(x,\omega) = y\}\big) = \Pr\big[f(x) = y\big].$$

In order to prove Conjecture 1, we proceed as follows:

1. We define an intermediate formalism between $\mathcal{POR}$ and **PPT** whose random choices are made *explicit* by means of a stream $\eta : \mathbb{N} \longrightarrow \{\mathtt{0},\mathtt{1}\}$ containing an infinite sequence of bit determining the outcome of all the possible random choices. Such formalism is **SFP**. It will be introduced in Section 3.1, but we prove the correspondence between **PPT** and **SFP** only at the end, namely in Section 3.4. This delay should not be too much of a problem because the definition of **SFP** is almost the same of **PPT** so, in our opinion, their equivalence is almost trivial.

2. On top of the definition of **SFP**, we define its encoding in $\mathcal{POR}$ and prove it correct. This is carried out in Section 3.2.

3. Finally, we prove that each function in $\mathcal{POR}$ can be encoded in **SFP**, too. This reduction is presented in Section 3.3 and steps through three intermediate formalisms aimed to separate the different technical concerns of this proof.

The overall picture of the reduction is described by Figure 3.1.

3.1 The Class SFP

In the perspective of Conjecture 1, Probabilistic Turing machines are not suitable computational models. But they are anyway needed to prove our claim, because the definition of the **PPT** class relies on these machines. Moreover, **BPP** and other probabilistic complexity classes are defined on top of the definition of PTMs, so a characterization of these classes must pass through the characterization of **PPT** functions. In Chapter 1, we have already defined that computational model and the class of **PPT** functions (Definitions 101 and 109).
As we stated above, in the perspective of Conjecture 1, PTMs are not a suitable computational model. Indeed, the source of randomness in a PTM is "implicit", as coming from a *uniform distribution of probability*, and this affects the machine output, lifting it from a plain string, to a distributions of strings. On the other hand, in $\mathcal{POR}$, randomness is enucleated by the oracle function, Q returning strings (not probability distributions).

To bridge this gap we introduce the *class* **SFP**, which is based on a better-fitting computational model, namely *stream Turing machines* (STM, for short). In STMs, the source of randomness consists in an "explicit", read-only random tape. More precisely, these machines can be defined as ordinary TMs with $k+1$ tapes, one of which is designated as a read-only *oracle*, and stores an infinite stream of characters to be read from left to right. These machines are a valuable intermediate model, linking $\mathcal{POR}$ and **PPT**. On the one hand, they use the oracle tape as an *explicit* source or randomness, that can be represented as a function $\eta : \mathbb{N} \longrightarrow \mathbb{B}$, whose role is similar to ω's one in $\mathcal{POR}$. On the other hand, STMs share many features with multi-tape TMs.[1] Thus, many well-known results about the latter model still hold.

Definition 45 (Stream Turing Machine)**.** A *stream Turing machine* is a quadruple $M := \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$, where:

- $\mathcal{Q}$ is a finite set of states ranged over by q_i and similar meta-variables.
- $q_0 \in \mathcal{Q}$ is an initial state.
- Σ is a finite set of characters ranged over by c_i *et simila.*
- $\delta : \hat{\Sigma} \times \mathcal{Q} \times \hat{\Sigma} \times \mathbb{B} \longrightarrow \hat{\Sigma} \times \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ is a transition function describing the new configuration reached by the machine.

L and R are two fixed and distinct symbols, e.g. `0` and `1`, $\hat{\Sigma} = \Sigma \cup \{\circledast\}$ and $\circledast$ represents the *blank character*, such that $\circledast \notin \Sigma$.

Without loss of generality, we can assume $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, and define the canonical STM as follows:

Definition 46 (Canonical STM)**.** A *canonical stream machine* is an STM, such that $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.

The *configuration* of a STM is a tuple which keeps track of the current state and some other objects representing the state of the machine tape(s). The portion of the oracle tape which has not been queried yet is represented by means of a function $\eta : \mathbb{N} \longrightarrow \mathbb{B}$.

Definition 47 (Configuration of STM)**.** The *configuration of an STM* is a quadruple $\langle \sigma, q, \tau, \eta\rangle$, where:

- $\sigma \in \hat{\Sigma}^*$ is the portion of the work tape on the left of the head;
- $q \in A$ is the current state of the machine;
- $\tau \in \hat{\Sigma}^*$ is the portion of the work tape on the right of the head;
- $\eta \in \mathbb{B}^{\mathbb{N}}$ is the portion of the oracle tape that has not been read yet.

At each step, the machine queries a new value on the oracle tape. Shifting on the work tape is naturally defined by pre-fixing and post-fixing characters to strings, so formalized by a shifting operation between the function η and a string σ.

Definition 48 (Shifting Operation)**.** Given $n \in \mathbb{N}$, $\sigma \in \mathbb{S}$, and $\eta : \mathbb{N} \longrightarrow \mathbb{B}$, we define the *shifting of η by σ*, $\sigma\eta(\cdot)$, by induction on the structure of σ:

$$(\boldsymbol{\epsilon}\eta)(n) := \eta(n)$$
$$(\mathtt{b}\tau)\eta(n) := \begin{cases} \mathtt{b} & \text{if } n = 0 \\ (\tau\eta)(n-1) & \text{otherwise.} \end{cases}$$

The dynamics of STMs is defined as predictable by extending the notion of standard transition function in the natural way.

[1]Conversely, we can see a TM as special STM, which basically "ignores" the oracle tape.

Definition 49 (STM Transition Function)**.** Given an STM, $M = \langle \mathcal{Q}, q, \Sigma, \delta \rangle$, we define the *partial transition function* $\vdash_\delta \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \mathbb{B}^{\mathbb{N}} \longrightarrow \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \mathbb{B}^{\mathbb{N}}$ between two configurations as:

$$\begin{aligned} \langle \sigma, q, c\tau, \mathtt{b}\eta \rangle &\Vdash_\delta \langle \sigma c', q', \tau, \eta \rangle && \text{if } \delta(q, c, \mathtt{b}) = \langle q', c', R \rangle \\ \langle \sigma c_0, q, c\tau, \mathtt{b}\eta \rangle &\Vdash_\delta \langle \sigma, q', c_0 c_1' \tau, \eta \rangle && \text{if } \delta(q, c_1, \mathtt{b}) = \langle q', c_1', L \rangle \end{aligned}$$

with $\mathtt{b} \in \{\mathtt{0}, \mathtt{1}\}$.

The configuration reached by the machine after n steps of computation is obtained by composing n times its reachability function, defined below.

Definition 50 (STM Reachability Function)**.** Given a STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, we indicate with $\{|\triangleright^n_M\}_n$ the smallest family of relations such that:

$$\langle \sigma, q, \tau, \eta \rangle |\triangleright^0_M \langle \sigma, q, \tau, \eta \rangle$$
$$\big(\langle \sigma, q, \tau, \eta \rangle |\triangleright^n_M \langle \sigma', q', \tau', \eta' \rangle\big) \land \big(\langle \sigma', q', \tau', \eta' \rangle \Vdash_\delta \langle \sigma'', q'', \tau'', \eta'' \rangle\big) \to \big(\langle \sigma, q, \tau, \eta \rangle |\triangleright^{n+1}_M \langle \sigma'', q'', \tau'', \eta'' \rangle\big).$$

Without loss of generality, we assume STMs not to be defined based on final states: computation is regarded as concluded whenever the current configuration does not define the transition function.[2]

Proposition 4. *For any STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$ and $n \in \mathbb{N}$, $|\triangleright^n_M$ is a* partial *function.*

Notation 8 (Final Configuration)**.** Given an STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, and a configuration $\langle \sigma, q, \tau, \eta \rangle$, we write $\langle \sigma, q, \tau, \eta \rangle \not\Vdash_\delta$ when there are no $\sigma', q', \tau', \eta'$ such that $\langle \sigma, q, \tau, \eta \rangle \Vdash_\delta \langle \sigma', q', \tau', \eta' \rangle$.

Finally, let us introduce the notion of function computable by (poly-time) STMs.

Definition 51 (STM Computation)**.** Given an STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, $\eta : \mathbb{N} \longrightarrow \mathbb{B}$ and a function $g : \mathbb{N} \longrightarrow \mathbb{B}$, we say that M *computes* g, written $f_M = g$ if and only if for every string $\sigma \in \mathbb{S}$, and oracle tape $\eta \in \mathbb{B}^{\mathbb{N}}$, there are a number $n \in \mathbb{N}$, $\tau \in \mathbb{S}, q' \in \mathcal{Q}$, and a function $\psi : \mathbb{N} \longrightarrow \mathbb{B}$ such that:

$$\langle \epsilon, q_0, \sigma, \eta \rangle \ |\triangleright^n_M \ \langle \gamma, q', \tau, \psi \rangle \not\vdash_\delta,$$

with $f_M(\sigma, \eta)$ being the longest suffix of γ not including $\circledast$.

Definition 52 (poly-time Stream Machine)**.** A *poly-time stream machine* is an STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$ such that:

$$\exists p \in \mathsf{POLY}. \forall \sigma \in \mathbb{S}, \eta \in \mathbb{B}^{\mathbb{N}}. \exists n \leq p(|\sigma|) \big(\langle \epsilon, q_0, \sigma, \eta \rangle |\triangleright^n_M \langle \gamma, q', \tau, \psi \rangle \not\vdash_\delta .$$

We call **SFP** the class of functions which are computable by poly-time STMs.

Definition 53 (The Class **SFP**)**.**

$$\mathbf{SFP} := \{ f \in \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \longrightarrow \mathbb{S} \mid f = f_M \text{ for some canonical poly-time STM } M \}.$$

Remark 3. *All definitions given so far concern single-input, single-tape machines only. We could naturally generalize them as multi-input and multi-tape machines in the standard way.*

[2]Indeed, in all these cases, we could imagine to add a final state q_F and a transition to that state for each blocking configuration.

Finally, the notion of initial prefix of an oracle tape is also crucial: we will show that a polynomially long prefix of the oracle tape is sufficient to determine the value of the function.

Definition 54 (Prefix of Oracle Tape)**.** For each function $\eta : \mathbb{N} \longrightarrow \mathbb{B}$, $\sigma \in \mathbb{S}$ and $n \in \mathbb{N}$, we define η_n as:

$$\eta_n = \sigma \quad \Leftrightarrow \quad \forall i < n.\eta(i) = \sigma(i).$$

Before going on, notice that $\mathcal{POR}$ and **SFP** are two inherently different sets:

$$\mathcal{POR} \subseteq \bigcup_{i \in \mathbb{N}} \mathbb{S}^i \times \mathbb{B}^{\mathbb{S}} \longrightarrow \mathbb{S}$$

$$\mathbf{SFP} \subseteq \bigcup_{i \in \mathbb{N}} \mathbb{S}^i \times \mathbb{B}^{\mathbb{N}} \longrightarrow \mathbb{S}.$$

Thus, some effort is required to relate these two classes and, actually, the correspondence obtained in this context is weaker than the one between Σ^b_1-formulæ representable in RS^1_2 and $\mathcal{POR}$, from Sections 2.2 and 2.3. Differently from formulæ of $\mathcal{L}$ and functions in $\mathcal{POR}$, which have access to their source of randomness in almost the same way, we cannot define a precise identity between **SFP** and $\mathcal{POR}$. We could relate sets of (oracle) functions, *which are not the same*, by considering their measure, and obtain a weaker (but strong enough) result, providing encodings between the two classes, which preserve measures of the random sources associated to the given input and output.

Theorem 9.

*i) For each $f \in$ **SFP**, there is a $g \in \mathcal{POR}$ such that for every $x, y \in \mathbb{S}$:*

$$\mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}\big) = \mu\big(\{\omega \in \mathbb{O} \mid g(x, \omega) = y\}\big).$$

*ii) For each $g \in \mathcal{POR}$, there is an $f \in$ **SFP** such that for every $x, y \in \mathbb{S}$:*

$$\mu\big(\{\omega \in \mathbb{O} \mid g(x, \omega) = y\}\big) = \mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}\big).$$

In the two following Sections, we will address the two reductions separately, namely: in Section 3.2, we will show that **SFP** can be reduced to $\mathcal{POR}$ and in Section 3.3 we will prove the converse reduction. Finally, in Section 3.4, we will establish that a similar relation holds between **SFP** and **PPT**, too.

3.2 From SFP to $\mathcal{POR}$

In this section, we address the first of the two claims in Theorem 9:

Lemma 1. *For any $f \in$ **SFP**, there is $g \in \mathcal{POR}$ such that for every $x, y \in \mathbb{S}$,*

$$\mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}\big) = \mu\big(\{\omega \in \mathbb{O} \mid g(x, \omega) = y\}\big).$$

When reducing **SFP** to $\mathcal{POR}$, we can either build a direct encoding or not. A direct encoding would require an on-line management of the probabilistic choices made by the $\mathcal{POR}$ and the **SFP** functions. This would make the proof cumbersome and probably opaque. Intuitively, we would need to simulate any function computed by an STM with the oracle tape associated with η by means of a $\mathcal{POR}$-function g, querying a fresh *coordinate* of its oracle ω at each simulate step, in turn emulating the corresponding value written on η. From a technical viewpoint, providing such apparently-natural correspondence between η and ω (i.e. linking functions in $\mathbb{O}$ and in $\mathbb{B}^{\mathbb{N}}$) is not trivial. For this reason, we prefer to separate the probabilistic concerns from the computational ones, ending up with an arguably easier and clearer proof, paying the cost of introducing two other intermediate formalisms. The first new formalism, called "*Finite* Stream Turing Machines", is defined as a variation of the STM formalism in which the oracle tape contains a finite-length string, instead of an infinite stream of characters. On top of the "*Finite* Stream Turing Machines", we define a class of functions which can be computed in polynomial time with respect to the length of the first input. This class of functions, called $\mathcal{PTF}$, is strongly related to **SFP**. Finally, we will identify a subset of $\mathcal{POR}$, namely $\mathcal{POR}^-$: the largest subset of $\mathcal{POR}$ which can be defined without the function Q. Showing that all the $\mathcal{PTF}$ functions have a correspondent function in $\mathcal{POR}^-$ allows us to decouple the purely behavioral concerns of the reduction to the measure-theoretic ones. Indeed, we address the technical problems concerning the actual simulation of the $\mathcal{PTF}$ machine within our function algebra $\mathcal{POR}^-$ in a first result. Then, by the trivial inclusion of $\mathcal{POR}^-$ in $\mathcal{POR}$, we obtain that any machine defining a $\mathcal{PTF}$ function is in $\mathcal{POR}$ as well. Thus, *in another result* we establish that the measures of the sets in Lemma 1 are identical.

Before giving the details of the proof of Lemma 1, we would like to outline its main steps, in order to facilitate the reader in walking through this section. Given an arbitrary $f \in \mathbf{SFP}$ with time bound $p \in \mathsf{POLY}$, we define a function $h : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S}$ such that:

$$f(x, \eta) = h\big(x, \eta_{p(|x|)}\big)$$

where h is "something" very close to an ordinary *poly-time* function[3]; in particular, $f \in \mathcal{PTF}$. To prove this, we pass through the corresponding class of machines, as done in Section 3.2.2. Then, we define a function $h' : \mathbb{S} \times \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ such that, for any $x, y \in \mathbb{S}$ and $\omega \in \mathbb{B}^{\mathbb{S}}$:

$$h'(x, y, \omega) = h(x, y)$$

and we prove that $h' \in \mathcal{POR}^-$. Finally, in Section 3.2.4, we define a function $e : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S} \in \mathcal{POR}$ to mimic the prefix extractor of Definition 54, namely we define e such that its output have *the same distribution* of all possible η's prefixes, but taking a functional argument with different signature. To do so we need deal with a bijection between $\mathbb{S}$ and $\mathbb{N}$, ensuring that for each $\eta \in \mathbb{B}^{\mathbb{N}}$ there is an $\omega \in \mathbb{B}^{\mathbb{S}}$ such that any prefix of η is an output of $e(y, \omega)$ for a specific y. Proving $e \in \mathcal{POR}$, since $\mathcal{POR}$ is closed under composition and $\mathcal{POR}^- \subseteq \mathcal{POR}$, we get the claim. In particular:

$$g(x, \omega) := h'(x, e(x, \omega), \omega).$$

We will structure the proof as follows:

- In Section 3.2.1, we will give the formal definitions of $\mathcal{POR}^-$ and $\mathcal{PTF}$.
- In Section 3.2.2, we will show that for each $f \in \mathbf{SFP}$ with time bound $p \in \mathsf{POLY}$ there is a $\mathcal{PTF}$ $h : \mathbb{S} \longrightarrow \mathbb{S}$ such that:

$$f(x, \eta) = h\big(x, \eta_{p(|x|)}\big).$$

- In Section 3.2.3, we show that $\mathcal{POR}^-$ is complete with respect to $\mathcal{PTF}$ functions. [4] This entails that there is a function $h' \in \mathcal{POR}$ such that:

$$\forall x, y, \omega . h(x, y) = h'(x, y, \omega).$$

[3]The main difference lies in the fact that ordinary poly-time function should take one input only.

[4]And, in particular with to poly-time functions.

- Then, in Section 3.2.4, we show that $e \in \mathcal{POR}$.
- In Section 3.2.5, we join all the results in order to prove the claim.

3.2.1 Preliminary Notions

The proof of Lemma 1, relies on the introduction of some auxiliary notions. In particular, we need all the necessary instruments to show that the function h introduced in the proof of Lemma 1 is actually poly-time. To do so, we define:

- The class of *Poly-Time Finite Stream Turing Machine* computable functions $\mathcal{PTF}$, defined in Section 3.2.1.1.
- A subset of $\mathcal{POR}$ which is expressive enough to capture the class $\mathcal{PTF}$. This function algebra is defined in Section 3.2.1.2.

3.2.1.1 (Poly-time) Finite Stream Turing Machine

The Finite Stream Turing Machine (FSTM, for short) are a blending between STMs, introduced in Section 3.1 and ordinary Turing Machines. Basically, they are ordinary Stream Machines, but with the difference that the second tape can contain a finite stream of values.[5]

Definition 55 (Finite Stream Turing Machine)**.** A *Finite Stream Turing Machine* is a quadruple, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, where:

- $\mathcal{Q}$ is a finite set of states ranged over by the meta-variables q_i.
- $q_0 \in \mathcal{Q}$ is the initial state.
- Σ is a finite set of characters ranged over by the c_i meta-variables.
- $\delta : \hat{\Sigma} \times \mathcal{Q} \times \hat{\Sigma} \times \hat{\Sigma} \longrightarrow \hat{\Sigma} \times \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ is a transition function describing the new configuration reached by the machine.

L and R are two distinct symbols, $\hat{\Sigma} = \Sigma \cup \{\circledast\}$ and $\circledast$ represents the blank character such that $\circledast \notin \Sigma$.

A *canonical Finite Stream Turing Machine* is a FSTM, such that $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, $L = \mathtt{0}$, and $R = \mathtt{1}$. Configurations are defined in the standard way.

Definition 56 (FSTM Configuration)**.** The *configuration of a FSTM* is a 4-tuple $\langle \sigma, q, \tau, \xi \rangle$, where

- $\sigma \in \hat{\Sigma}^*$ is the portion of the work tape to the left of the head;
- $q \in \mathcal{Q}$ is the current state of the machine;
- $\tau \in \hat{\Sigma}^*$ is the portion of the work tape to the right of the head;
- $\sigma \in \hat{\Sigma}^*$ is the portion of the secondary tape to the right of the head.

Now, it is possible to define the TM's transition function.

Definition 57 (FSTM Transition Function)**.** Given an FSTM, $M = \langle \mathcal{Q}, q, \Sigma, \delta \rangle$, we define the *partial transition function* $\vdash_\delta \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \hat{\Sigma}^* \longrightarrow \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^*$ between two configurations as:

$$\begin{aligned} \langle \sigma, q, c\tau, d\xi \rangle &\vdash_\delta \langle \sigma c', q', \tau, \xi \rangle && \text{if } \delta(q, c, d) = \langle q', c', R \rangle \\ \langle \sigma c_0, q, c\tau, d\xi \rangle &\vdash_\delta \langle \sigma, q', c_0 c_1' \tau, \xi \rangle && \text{if } \delta(q, c_1, d) = \langle q', c_1', L \rangle. \end{aligned}$$

[5]Some definitions are omitted but can be found in Chapter 6, Section 6.2.2

As for STMs, the configuration reached by the machine M after n steps of computation is obtained by composing n times its reachability function, denoted by $\triangleright^n_M$.

Definition 58 (FSTM Reachability Function)**.** Given a FSTM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, we indicate with $\{\triangleright^n_M\}_n$ the smallest family of relations such that:

$$\langle \sigma, q, \tau, \xi \rangle \triangleright^0_M \langle \sigma, q, \tau, \xi \rangle$$
$$\big(\langle \sigma, q, \tau, \xi \rangle \triangleright^n_M \langle \sigma', q', \tau', \xi' \rangle\big) \wedge \big(\langle \sigma', q', \tau', \xi'' \rangle \vdash_\delta \langle \sigma'', q'', \tau'', \xi'' \rangle\big) \rightarrow \big(\langle \sigma, q, \tau, \xi \rangle \triangleright^{n+1}_M \langle \sigma'', q'', \tau'', \xi'' \rangle\big).$$

As for TMs and STMs, we assume FSTMs not to use final states.

Proposition 5. *For any FSTM, $M\langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$ and $n \in \mathbb{N}$, $\triangleright^n_M$ is a function.*

Notation 9 (Final Configuration)**.** Given a FSTM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, and a configuration $\langle \sigma, q, \tau, \xi \rangle$, we write $\langle \sigma, q, \tau, \xi \rangle \not\vdash_\delta$ when there are no σ', q', τ', ξ' such that $\langle \sigma, q, \tau, \xi \rangle \vdash_\delta \langle \sigma', q', \tau', \xi' \rangle$.

Definition 59 (Poly-time Finite Stream Turing Machine)**.** A *poly-time Finite Stream Turing machine* is a FSTM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$ such that:

$$\exists p \in \mathsf{POLY}.\forall \sigma, \tau \in \mathbb{S}.\exists n \leq p(|\sigma|)\big(\langle \boldsymbol{\epsilon}, q_0, \sigma, \tau \rangle \triangleright^n_M \langle \gamma, q', \sigma', \tau' \rangle \not\vdash_\delta .$$

Definition 60 (FSTM Computation)**.** Given a FSTM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$ and a function $g : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S}$, we say that f_M *computes* g, written $f_M = g$, if and only if for every $\sigma, \tau \in \mathbb{S}$, there are a natural number $n \in \mathbb{N}$, $\xi \in \mathbb{S}$ and $q' \in \mathcal{Q}$, such that:

$$\langle \boldsymbol{\epsilon}, q, \sigma, \tau \rangle \triangleright^n_M \langle \gamma, q, \sigma', \tau' \rangle$$

with $f(\sigma, \tau)$ being the longest suffix of γ not including $\circledast$.

We are now able to define the class $\mathcal{PTF}$, namely the class of functions which are computable by poly-time FSTMs.

Definition 61 (The Class $\mathcal{PTF}$)**.**

$$\mathcal{PTF} := \{f \in \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S} \mid f = f_M \text{for some poly-time FSTM } M\}.$$

The class $\mathcal{PTF}$, basically, is a restriction to a finite stream of random bits of the class **SFP** class.

3.2.1.2 The Class $\mathcal{POR}^-$

Furthermore, we present the class $\mathcal{POR}^-$ which is defined as $\mathcal{POR}$ except for the absence of the query function Q. The reasons why this class is interesting are manifold:

- First of all, because we show that it is sound and complete with respect to Ferreira's PTCA [14] (Remarks 7 and 8). This entails that, showing the completeness of $\mathcal{POR}^-$ respect to $\mathcal{PTF}$ functions and the completeness of $\mathcal{PTF}$ functions with respect to *ordinary poly-time functions* (Lemma 23) yields, as a corollary, the proof that Ferreira's PTCA contains the class of poly-time computable functions.
- Obviously we have $\mathcal{POR}^- \subseteq \mathcal{POR}$, and that the output of a function in $\mathcal{POR}^-$ does not depend on its oracle ω. For this reason, proving the reducibility of $\mathcal{PTF}$ to $\mathcal{POR}^-$, we will separate the machine-related part of the reduction, which concerns the actual implementation of an FSTM machine in $\mathcal{POR}$ from the measure-theoretic aspects of Theorem 9.

The completeness of $\mathcal{POR}^-$ with respect to $\mathcal{PTF}$ functions is proved in Section 3.2.3.

Definition 62 (The Class $\mathcal{POR}^-$)**.** The *class* $\mathcal{POR}^-$ is the smallest class of functions $\mathbb{S}^n \times \mathbb{O} \longrightarrow \mathbb{S}$ containing:

- The empty function $E(x, \omega) = \boldsymbol{\epsilon}$;
- The projection function $P^n_i(x_1, \dots, x_n, \omega) = x_i$;
- The word-successor $S_{\mathtt{b}}(x, \omega) = x\mathtt{b}$, for every $\mathtt{b} \in \mathbb{B}$
- The conditional function
$$C(\boldsymbol{\epsilon}, y, z_{\mathtt{0}}, z_{\mathtt{1}}, \omega) = y$$
$$C(x\mathtt{b}, y, z_{\mathtt{0}}, z_{\mathtt{1}}, \omega) = z_{\mathtt{b}},$$
  where $\mathtt{b} \in \mathbb{B}$;

and closed under:

- Composition, such that f is defined from $g, h_1, \dots, h_k$ as:
$$f(\vec{x}) = g\big(h_1(\vec{x}, \omega), \dots, h_k(\vec{x}, \omega), \omega\big);$$
- Bounded recursion, such that f is defined from g, h_1, h_2 as:
$$f(\vec{x}, \boldsymbol{\epsilon}, \omega) := g(\vec{x}, \omega);$$
$$f(\vec{x}, y\mathtt{0}, \omega) := h_1\big(\vec{x}, y, f(\vec{x}, y, \omega), \omega\big)|_{t(\vec{x},y)};$$
$$f(\vec{x}, y\mathtt{1}, \omega) := h_2\big(\vec{x}, y, f(\vec{x}, y, \omega), \omega\big)|_{t(\vec{x},y)};$$
  where t is defined from $\boldsymbol{\epsilon}, \mathtt{0}, \mathtt{1}, \frown, \times$ by explicit definition.

3.2.2 From SFP to $\mathcal{PTF}$

First of all, we show that for any $f \in \mathbf{SFP}$, the corresponding $\mathcal{PTF}$ function $h : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S}$ can be constructed. The core idea consists in showing a procedure to transform a STM into an equivalent FSTM. Indeed, according to Definition 53, there is a poly-time STM M, such that $f = f_M$. Thus, we want to construct the corresponding *poly-time* FSTM N, which, taken for second argument a sufficiently long polynomial prefix of η, behaves exactly like M. In particular, N is constructed basing on a two-taped FSTM, whose transition function is exactly M's. For each computation, N performs a number of steps which is *exactly the same number of those performed* by M. So also this standard FSTM must be *poly-time.*

Lemma 2. *For each $f \in$ **SFP** with time-bound $p \in \mathsf{POLY}$, there is an $h \in \mathcal{PTF}$ such that for any $\eta \in \mathbb{B}^{\mathbb{N}}$ and $x, y \in \mathbb{S}$,*
$$f(x, \eta) = h(x, \eta_{p(|x|)}).$$

Proof. Assume that $f \in \mathbf{SFP}$. By Definition 53, there is a poly-time STM, $M = \langle \mathcal{Q}, q_0, \Sigma, \delta \rangle$, such that $f = f_M$. Let us define an FSTM N which, given a polynomially long prefix of η, behaves like M. The Definition of N is identical to the definition of M. Formally, for any $k \in \mathbb{N}$ and some $\sigma, \tau, y' \in \mathbb{S}$,
$$\langle \boldsymbol{\epsilon}, q_0', x, y \rangle \triangleright^k_{\delta'} \langle \sigma, q, \tau, y' \rangle \quad \Leftrightarrow \quad \langle \boldsymbol{\epsilon}, q_0', x, y\eta \rangle |\!\triangleright^k_{\delta} \langle \sigma, q, \tau, y'\eta \rangle.$$

Moreover, N requires a number of steps which is exactly equal to the number of steps required by M, and thus is in $\mathcal{PTF}$, too. We conclude the proof defining $h = f_N$. $\square$

3.2.3 From $\mathcal{PTF}$ to $\mathcal{POR}^-$

In this section we will show that all those functions which are in $\mathcal{PTF}$ can be represented in $\mathcal{POR}^-$, too. This can be formalized as follows:

Lemma 3. *For any $f \in \mathcal{PTF}$ and $x \in \mathbb{S}$, there is $g \in \mathcal{POR}^-$ such that $\forall x, y, \omega. f(x, y) = g(x, y, \omega)$.*

Proof Sketch. Any configuration of an FSTM can be encoded in a string, thus we can pass the encoding of the initial machine's configuration to a function which emulates the execution of the machine which computes f (called M_f) for a polynomial number of steps. When M_f reaches a final configuration, it is sufficient to extract from the final configuration the longest portion of the primary tape which is on the left and free form $\circledast$ characters. $\square$

Formally proving Lemma 3 requires a lot of work which, for the sake of a clearer presentation can be found in Section 6.1 of Chapter 6. In this part we only show that $\mathcal{POR}^-$ enjoys three important properties which entail Lemma 3:

- It is possible to encode FSTMs, together with configurations and their transition functions simply using strings. Moreover, there is a function *apply* $\in \mathcal{POR}^-$ which, receiving as input the encoding of a FSTM configuration c and the encoding of a FSTM transition function δ, computes the encoding configuration obtained applying the function $\vdash_\delta$ on c. Intuitively, the *apply* function simulates the outcome of a computation step onto a FSTM configuration.
- For each $f \in \mathcal{POR}^-$ and $x, y \in \mathbb{S}$ if there is a term $t(x)$ in $\mathcal{L}$ bounding the size of $f(x, \omega)$ for each ω, then there is a function which applies $|y|$ times f on its own output.
- There is a function *dectape* which extracts the machine's output from any encoded final configuration of an FSTM machine.

3.2.3.1 The Function *apply*

To define *apply*, we show that given the encoding of a finite function[6], can be *interpreted* by means of a $\mathcal{POR}^-$ function, the *total function simulator* called *sim*.[7]

The *sim* function is defined by induction on the number of elements in the encoding of the simulated function (the number of pairs in the function's graph). This value is obtained by means of the function $\pi_0(y, \omega)$ which, as shown in Remark 20, returns the number of element in a collection, assuming that such collection is represented on top of lists, as in this case. Said this value n, for each $1 \leq i \leq n$, the function *sim'* extracts the i-th projection from the function's graph and compares its first element with the queried value. If these values are identical, then it returns the second projection of the pair, otherwise it proceeds recursively.

Definition 63 (Total Function Simulator)**.** We define the *total function simulator* $sim(\cdot, \cdot, \omega)$ as follows:

$$\begin{aligned}
sim'(y, x, \boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon};\\
sim'(y, x, z\mathtt{b}, \omega) &:= if\big(\pi_2(\pi(y, z\mathtt{b}, \omega), \omega), sim'(y, x, z, \omega), eq(\pi_1(\pi(y, z\mathtt{b}, \omega), \omega), x, \omega), \omega\big);
\end{aligned}$$

$$sim(x, y, \omega) := sim'\big(y, x, \pi_0(y, \omega), \omega\big).$$

[6]For details, see Corollary 14 in Chapter 6

[7]We call it *total* because, since $\mathcal{POR}^-$ is total, it defines a default value to be returned when the simulated function is not defined on the queried input.

The formal proof of the correctness of this function is in Chapter 6, Lemma 29 and the implementative details of the functions and the encoding employed can be found in Section 6.1 of Chapter 6.

As a consequence of $sim \in \mathcal{POR}$, it holds that there is a function *apply* in $\mathcal{POR}$ which, given a transition function δ, simulates the function $\vdash_\delta$ on the encodings of machine's configuration. The formal proof of this property (Lemma 4), together with the definition of the function *apply* are quite cumbersome. For this reason, they are given in Chapter 6, Section 6.2.3. The correctness of *apply* is stated as follows:

Lemma 4 (Correctness of *apply*)**.** *The function apply is such that for any FSTM M with transition function δ, said x_δ and $h(c)$ respectively, the given encodings of δ and the encoding of a configuration c,*[8]

$$\forall \omega \in \mathbb{O}. \vdash_\delta (c) = d \rightarrow apply(x_\delta, h(c), \omega) = h(d);$$
$$\forall \omega \in \mathbb{O}. (c \not\vdash_\delta) \rightarrow apply(x_\delta, h(c), \omega) = h(c).$$

Here, assuming that Lemma 4 holds, we show how it is possible to employ the function *apply* to prove Lemma 3.

3.2.3.2 Power Function

In this section, we show that for each function in $\mathcal{POR}^-$, if the size of its outputs is bounded by a term in $\mathcal{L}$, then its n-th power is in $\mathcal{POR}^-$. Thanks to this result, we will be able to compute the polynomial transitive closure of *apply*, in order to compute the final configuration of any FSTM, given its δ function and its input.

To this end, we define the function schema $sa_{\cdot,\cdot}$, which allows us to compute the transitive closure of the function *apply*. To do so, we must show that the growth of the terms computed by *apply* is under control. This is due to the fact that the only iterative mechanism of $\mathcal{POR}^-$ is bounded recursion on notation, which requires that, at each step, the function outputs are bounded in size. So, the self-application schema must pass through bounded recursion on notation and thus, it requires size bounds. For this reason, we show that the term-growth of the size growth of the output of the function *apply* is at most constant. This is done in Lemma 6

Lemma 5. *For each $f : \mathbb{S}^{k+1} \times \mathbb{O} \longrightarrow \mathbb{S} \in \mathcal{POR}$, if there is a term $t \in \mathcal{L}$ such that $\forall x, \vec{z}, \omega. f(x, \vec{z}, \omega)|_t = f(x, \vec{z}, \omega)$ then there is also a function $sa_{f,t} : \mathbb{S}^{k+2} \times \mathbb{O} \longrightarrow \mathbb{S}$ such that:*

$$\forall n \in \mathbb{N}. \forall x \in \mathbb{S}, \omega \in \mathbb{O}. sa_{f,t}(x, \underline{n}_{\mathbb{N}}, \vec{z}, \omega) = \underbrace{f(f(f(x, \vec{z}, \omega), \vec{z}, \omega), \ldots)}_{n\ times}.$$

For sake of readability, we prove this result and define the $sa_{\cdot,\cdot}$ function schema in Chapter 6, the details are in Section 6.2.3.

Lemma 6 (*apply* Size Growth)**.** *For all $x_c \in \mathbb{S}$ being the encoding of an FSTM configuration and for each encoding of a transition function δ in a string $x_\delta \in \mathbb{S}$ and for each $\omega \in \mathbb{O}$, there is a $k \in \mathbb{N}$ such that $apply(x_c, x_\delta, \omega)|_{x_c 1^k}$.*

The specific value of the size bound expressed in the lemma above is not particularly meaningful, and it is mainly due to the encodings we decided to adopt, while it is important that fixed a FSTM machine we can always find such k. Even in this case, the technical proof of this result is in Chapter 6 (Section 6.2.3).

[8] These encodings exist as a consequence of Corollaries 14 and 15. In what follows, the functions thereby described will be considered the canonical encoding of FSTM transition functions and configurations within $\mathcal{POR}$.

3.2.3.3 Representing Transition in $\mathcal{POR}^-$

In this section, we show that it is possible to define a function *dectape* $\in \mathcal{POR}^-$ which takes in input the encoding of a tape and $\omega \in \mathbb{O}$, and returns the longest suffix of the tape on the left of the head without any occurrence of $\circledast$. This requires us to do some technical work. First, we show that some auxiliary functions are in $\mathcal{POR}^-$; these functions are: the difference function *diff* returning the difference between two numbers, the list-projector function π_n taking (the encoding of) a list and a number n as its input and returning the n-th element of the list, and the right-remover function *rrs*. Moreover, it also depends on a couple of control-related functions the string identity predicate $eq : \mathbb{S}^2 \times \mathbb{O} \longrightarrow \mathbb{S}$ and the control structure $if : \mathbb{S}^3 \times \mathbb{O} \longrightarrow \mathbb{S}$, thus we show that even those two functions are in $\mathcal{POR}^-$.[9] While decoding the final configuration of a canonical TM, we need to extract the longest sequence of bits on the immediate left of the head. To do so, we introduce an auxiliary function $\rho : \mathbb{S}^2 \times \mathbb{O} \longrightarrow \mathbb{S}$ that is supposed to take the encoding of a tape as its input and return the y-th right character of the tape. Formally,

$$\rho(x, y, \omega) = \pi\big(x, \mathit{diff}(\pi_0(x, \omega), y, \omega), \omega\big).$$

Then, we define a *decoding* function so that, at each step, checks whether the character obtained from ρ is the encoding of $\circledast$ of Definition 128, i.e. `111`. If so, it returns $\boldsymbol{\epsilon}$. , This control is done recurring to eq.

Definition 64 (*dectape* Function)**.** Let $dec : \mathbb{S} \times \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ be an auxiliary function defined as follows:

$$\begin{aligned} \mathit{dectape}'(x, \boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon}; \\ \mathit{dectape}'(x, y\mathtt{b}, \omega) &:= \mathit{if}\big(\mathit{dectape}'(x, y, \omega)\rho(x, y\mathtt{b}, \omega), \boldsymbol{\epsilon}, \neg eq(\rho(x, y\mathtt{b}, \omega), \omega), \mathtt{111}, \omega)\big)|_x. \end{aligned}$$

with

$$\rho(x, y, \omega) := \pi\big(x, \mathit{diff}(\pi_0(x, \omega), y, \omega), \omega\big).$$

We define the function *dectape*: $\mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ as follows:

$$\mathit{dectape}(x, \omega) := \mathit{dectape}'\big(x, rrs(\pi_0(x, \omega), \omega), \omega\big).$$

The function *dectape* returns a string which is the longest suffix not including $\circledast$ of the tape encoded with $\underline{\cdot}_{\mathbb{T}}$ and stored in x. By Definition 51, this is precisely the value computed by the machine. A formal proof of this statement is given in Chapter 6, the details are in Lemma 30

Lemma 7. *The function dectape $\in \mathcal{POR}^-$ is such that if $\underline{\cdot}_{\mathbb{T}}$ is the encoding for tapes of Definition 128, then for any $\sigma \in \{\mathtt{0}, \mathtt{1}, \circledast\}^*$, $\omega \in \mathbb{O}$,*

$$\mathit{dectape}(\underline{\sigma}_{\mathbb{T}}, \omega) = \tau$$

and τ is the longest suffix of σ without $\circledast$.

Let us also define a function *size* which allows us to compute the encoding of the size of a string throughout $\underline{\cdot}_{\mathbb{N}}$.

Definition 65. The function $size : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ is defined as follows:

$$\begin{aligned} \mathit{size}(\boldsymbol{\epsilon}, \omega) &:= \mathtt{1}; \\ \mathit{size}(x\mathtt{b}, \omega) &:= \mathit{size}(x, \omega)\mathtt{1}|_{x\mathtt{11}}. \end{aligned}$$

The correctness of the function *size*, with respect to the encoding we are adopting — namely $n \mapsto \mathtt{1}^{n+1}$ — can be shown by induction.

[9] Formal definitions are presented in Chapter 6, Section 6.1.2.

3.2.3.4 Composing the Pieces

It is now possible to show that for every $\mathcal{PTF}$-function there is a corresponding function in $\mathcal{POR}^-$. The conclusion is the *composition* of three main ingredients:

- Machine steps can be simulated by means of the function *apply*.
- The function *apply* can be self-applied a polynomial numbers of times.
- We can extract the value computed by the machine by its final configuration.

These results allow us to give a proof to Lemma 3.

Lemma 3. *For any $f \in \mathcal{PTF}$ and $x \in \mathbb{S}$, there is $g \in \mathcal{POR}^-$ such that $\forall x, y, \omega. f(x, y) = g(x, y, \omega)$.*

Proof. By definition of $\mathcal{PTF}$, there is an FSTM computing f on a machine M with a polynomial time-bound, p, and a transition function δ. Let us consider a function g defined as follows:

$$g(x, y, \omega) = \mathit{dectape}(\pi_1(sa_{\mathit{apply}, t_M}(x_\delta, \langle \underline{\circledast}_{\mathbb{T}}, \underline{0}_{\mathbb{N}}, \underline{x}_{\mathbb{T}}, \underline{y}_{\mathbb{T}} \rangle^4_{\mathbb{L}}, \underline{p}(\mathit{size}(x, \omega), \omega), \omega), \omega), \omega)$$

where $\underline{p}$ is the $\mathcal{POR}^-$-function which computes the encoding of the value of the polynomial p.[10] This function is correct as the self-application of *apply* for $p(|x|)$ times returns in the machine configuration, as a consequence of Lemma 4. Finally, *dectape* extracts the longest suffix free from blank characters which is on the left of the head in the encoding of the tape reached at the end of the computation. This is a consequence of the correctness of the projector π_1 with respect to the encoding of lists (for details, see Chapter 6, Section 6.1) and of the correctness of *dectape* (Lemma 30). As required by Definition 51, this is exactly $f(x, y)$. ☐

Before proceeding with the main result of this section, we would like to point out that Lemma 3 entails, as a consequence, that the polynomial time computable functions are exactly the functions computable by a canonical Cobham style function algebra. This results is placed in a chapter by its own: Chapter 4. As another consequence of Lemma 3, we show the result we were aiming to: each function $f \in \mathbf{SFP}$ can be simulated by a function in $g \in \mathcal{POR}^-$, using as an additional input a polynomial prefix of f's oracle.

Corollary 4. *For each $f \in \mathbf{SFP}$ and polynomial time-bound $p \in \mathsf{POLY}$, there is a function $g \in \mathcal{POR}^-$ such that for any $\eta : \mathbb{N} \longrightarrow \mathbb{B}$, $\omega : \mathbb{N} \longrightarrow \mathbb{B}$ and $x \in \mathbb{S}$,*

$$f(x, \eta) = g\big(x, \eta_{p(|x|)}, \omega\big).$$

Proof. Assume $f \in \mathbf{SFP}$ and $y = \eta_{p(|x|)}$. By Lemma 2, there is a function $h \in \mathcal{PTF}$ such that, for any $\eta : \mathbb{N} \longrightarrow \mathbb{B}$ and $x \in \mathbb{S}$,

$$f(x, \eta) = h\big(x, \eta_{p(|x|)}\big).$$

Moreover, due to Lemma 3, there is also a $g \in \mathcal{POR}^-$ such that for every $x, y \in \mathbb{S}, \omega \in \mathbb{O}$,

$$g(x, y, \omega) = h\big(x, y\big).$$

Then, the desired function is g. ☐

[10] This function is in $\mathcal{POR}^-$, because we have shown that this class contains all the polynomials, details are in Corollary 16.

3.2.4 Extractor Function in $\mathcal{POR}$

Finally, we construct the extractor function $e(x, \omega)$ we introduced discussing the proof of Lemma 1 within $\mathcal{POR}$. Indeed, by means of e we can sample enough random bits from ω to fill randomly the oracle-tape of the FSTM machine we are emulating. Intuitively, this function is very simple: extracts $|x| + 1$ bits from ω and concatenates them in its output. The function e, in order to sample form ω a string uniformly at random, passes through a bijection $dy : \mathbb{N} \longrightarrow \mathbb{S}$, called *dyadic representation* of a natural number. Thus, the e function can simply enumerate its numerals $\underline{0}_{\mathbb{N}}, \underline{1}_{\mathbb{N}}, \underline{2}_{\mathbb{N}}, \ldots$, and sample the corresponding coordinates of η obtained throughout dy. Intuitively, for each natural number $n \in \mathbb{N}$, if n_2 is the encoding of n in base 2, the dyadic representation of m is $(m + 1)_2$ without is leftmost bit. The intuition is at the basis of our definition of an "extractor" function in $\mathcal{POR}$.

First, we define a function $bin : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ such that, if x has length k, then for every $\omega \in \mathbb{O}$, $bin(x, \omega)$ is the binary encoding of k.

Definition 66. Let us define an auxiliary function $binsucc : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$,

$$\begin{aligned} binsucc(\boldsymbol{\epsilon}, \omega) &:= \mathtt{1}; \\ binsucc(x\mathtt{0}, \omega) &:= x\mathtt{1}|_{x\mathtt{00}}; \\ binsucc(x\mathtt{1}, \omega) &:= binsucc(x, \omega)\mathtt{0}|_{x\mathtt{00}}. \end{aligned}$$

Then, $bin : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ is defined as:

$$\begin{aligned} bin(\boldsymbol{\epsilon}, \omega) &:= \mathtt{0}; \\ bin(x\mathtt{b}, \omega) &:= binsucc\big(bin(x, \omega), \omega\big)|_{x\mathtt{b}}. \end{aligned}$$

On top of the definition of the binary encoding of a number, we define the dyadic encoding of it:

Definition 67. We call $dy : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ the function in $\mathcal{POR}^-$ such that $\forall n \in \mathbb{N}, \omega \in \mathbb{O}. dy(\underline{n}_{\mathbb{N}}, \omega)$ is the dyadic encoding of n. Namely:

$$dy(x, \omega) := lrs(bin(x, \omega), \omega)$$

where lrs is the string manipulator defined in 133 which removes the leftmost bit from a string, if it exists, otherwise it returns $\boldsymbol{\epsilon}$.

In Chapter 6 (Lemma 31) we prove that for any ω, the function $dy(\underline{n}_{\mathbb{N}}, \omega)$ is a bijection between a a natural number n and a string — its dyadic representation. This result will be crucial to prove Lemma 1, because it relates the set $\mathbb{B}^{\mathbb{N}}$ to $\mathbb{B}^{\mathbb{S}}$ throughout a bijection. We then define a function $e \in \mathcal{POR}$, which takes a string and an oracle as its input and returns a finite string obtained picking $|x|$ bytes from it, choosing exactly the coordinates of ω corresponding to the dyadic encoding of the first $|x|$ natural numbers.

Definition 68. Let $e : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ be defined as follows:

$$\begin{aligned} e(\boldsymbol{\epsilon}, \omega) &= \boldsymbol{\epsilon}; \\ e(x\mathtt{b}, \omega) &= e(x, \omega)Q\big(dy(x, \omega), \omega\big)|_{x\mathtt{b}}. \end{aligned}$$

Definition 69. We define $\sim_{dy}$ as the smallest relation in $\mathbb{B}^{\mathbb{S}} \times \mathbb{B}^{\mathbb{N}}$ such that:

$$\eta \sim_{dy} \omega \leftrightarrow \forall n \in \mathbb{N}. \eta(n) = \omega(dy(\underline{n}_{\mathbb{N}}, \omega)).$$

This relation has some good properties:

Lemma 8. *It holds that:*

- $\forall \eta \in \mathbb{B}^{\mathbb{N}}.\exists! \omega \in \mathbb{B}^{\mathbb{S}}.\eta \sim_{dy} \omega$;
- $\forall \omega \in \mathbb{B}^{\mathbb{S}}.\exists! \eta \in \mathbb{B}^{\mathbb{N}}.\eta \sim_{dy} \omega$.

Proof. The proofs of the two claims are very similar. For this reason, we will take in exam only the first claim. By the fact that dy is a bijection with respect to its first argument and is constant with respect to the second, we obtain the existence of an ω which is in relation with η. Now suppose that there are ω_1, ω_2 both in relation with η being different. It holds then that $\exists \sigma \in \mathbb{S}$ such that $\omega_1(\sigma) \neq \omega_2(\sigma)$. Then, since dy is in $\mathcal{POR}^-$, the value of its last argument does not affect the value of its output, moreover $dy(\underline{\cdot}_{\mathbb{N}}, \omega)$ it is a bijection so there is an $n \in \mathbb{N}$, such that $dy(\underline{n}_{\mathbb{N}}, \omega) = \sigma$, so we get $\eta(n) = \omega_1(\sigma) \neq \omega_2(\sigma) = \eta(n)$, which is a contradiction. □

Corollary 5. *The relation $\sim_{dy}$ is a bijection.*

Proof. Consequence of Lemma 8. □

3.2.5 Concluding the Proof

Lemma 1. *For any $f \in$ **SFP**, there is $g \in \mathcal{POR}$ such that for every $x, y \in \mathbb{S}$,*

$$\mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}\big) = \mu\big(\{\omega \in \mathbb{O} \mid g(x, \omega) = y\}\big).$$

Proof. From Corollary 4, we know that there is a function $f' \in \mathcal{POR}^-$, and a $p \in \mathsf{POLY}$ such that:

$$\forall x, y \in \mathbb{S}.\forall \eta.\forall \omega.y = \eta_{p(x)} \to f(x, \eta) = f'(x, y, \omega). \qquad (*)$$

So, by the fact that $\mathcal{POR}^- \subseteq \mathcal{POR}$, $f' \in \mathcal{POR}$, too. Fixed an $\overline{\eta} \in \{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}$, its image with respect to $\sim_{dy}$ is in

$$\{\omega \in \mathbb{O} \mid f'(x, e(p'(size(x, \omega), \omega), \omega), \omega) = y\}.$$

Indeed, by Lemma 33, it holds that $\overline{\eta}_{p(x)} = e(p(size(x, \omega), \omega)$, where p' is the $\mathcal{POR}^-$ function computing the polynomial p and $size$ is the $\mathcal{POR}^-$ function computing the encoding of the natural number which corresponds to the size of its first input. By $(*)$ we have the claim. It also holds that, fixed an $\overline{\omega} \in \{\omega \in \mathbb{O} \mid f'(x, e(p'(size(x, \omega), \omega), \omega), \omega) = y\}$, then its pre-image with respect to $\sim_{dy}$ is in $\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}$. The proof is analogous to the one we showed above. Now, since $\sim_{dy}$ is a bijection between the two sets:

$$\mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x, \eta) = y\}\big) = \mu\big(\{\omega \in \mathbb{O} \mid f'(x, e(p(size(x, \omega), \omega), \omega), \omega) = y\}\big)$$

which concludes the proof. □

3.3 From $\mathcal{POR}$-functions to SFP

In this section, we show the proof of the second part of Theorem 9, namely that:

Lemma 9. *For each $g \in \mathcal{POR}$, there is an $f \in$ **SFP** such that for every $x, y \in \mathbb{S}$:*

$$\mu\big(\{\omega \in \mathbb{O} \mid g(x,\omega) = y\}\big) = \mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x,\eta) = y\}\big).$$

The main obstacle towards a proof of this result has to do with the different ways adopted by $\mathcal{POR}$ and **SFP** to access randomness. In particular, the $\mathcal{POR}$ formalism includes the function $Q(x,\omega) := \omega(x)$. Due to compositionality, the coordinate x used to access the oracle can be any output of any $\mathcal{POR}$ function. For this reason, this function can access exactly $2^{|x|}$ values of ω. From now on, we will call this modality *"random access to the oracle"*. Conversely, the Stream Machine's formalism accesses randomness in a linear way: at each transition, a new cell of the oracle η is consumed and the value at that coordinate is used to determine the next configuration reached by the machine. This means that an **SFP** machine can explore at most a polynomial number of coordinates of its oracle η; this is what we call *"linear access to the oracle"*.

This difference makes the encoding from $\mathcal{POR}$ to **SFP** non-trivial. For instance, take $Q(x,\omega)$: there is not a natural counterpart of Q in **SFP**, simply because the poly-time STMs can access only a polynomial number of cells of a tape, regardless from their input. As a consequence any encoding from $\mathcal{POR}$ to the STMs' formalism cannot preserve the random access to the oracle within a polynomial number of steps. For this reason, our reduction employs a mechanism which allows to simulate *random access to the oracle* in a formalism which does not. We achieved this result implementing a sort of *associative table*, which records each queried coordinate and the random answer given the first time this particular coordinate was queried. To do so, and to show it formally, we introduce an intermediate imperative and probabilistic formalism, the *String's Imperative Flipping Paradigm* SIFP, and we spell it out into two variants: $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$. The first is complete with respect to $\mathcal{POR}$ and is characterized by a fully random access to the oracle ω, whilst the latter supports a linear and *on-demand* access to randomness, which means that the programmer has a partial control on the queried coordinate. In particular, $\mathsf{SIFP}_{\mathsf{LA}}$ programs use oracle functions $\eta : \mathbb{N} \longrightarrow \mathbb{B}$ which are explored consecutively, starting from the coordinate 0. Once reduced $\mathcal{POR}$ to $\mathsf{SIFP}_{\mathsf{LA}}$, we will introduce an *on-demand* variant of **SFP**, namely $\mathbf{SFP_{OD}}$. This variation on **SFP** is based on a version of the STM paradigm whose machines do not necessarily shift on the right the head on the oracle tape at each step of the computation. This is done employing a transition function supporting two different kinds of step: shifting steps and non-shifting steps. Thus, $\mathbf{SFP_{OD}}$ is used as a sort of bridge between $\mathsf{SIFP}_{\mathsf{LA}}$ and **SFP** because it supports *on-demand* access to randomness, as for $\mathsf{SIFP}_{\mathsf{LA}}$, but is defined over a computational model very similar to STMs. This allows us to decouple the aspects related to the access to the source of randomness from the paradigm-related concerns: indeed, we will shift from the imperative paradigm $\mathsf{SIFP}_{\mathsf{LA}}$ to the On Demand STMs grounding $\mathbf{SFP_{OD}}$ without changing the modality of access to the oracle; then, we will change the modality of access to the oracle in two homogeneous computational models, i.e. the On Demand STMs and standard STMs.

As sated above, the introduction of these intermediate formalisms is aimed at solving all the different issues of the complete reduction in distinct steps. As an alternative, it would have been possible to encode directly $\mathcal{POR}$ in the STM formalism, but this would have caused the proof to become unnecessarily complex. Conversely, our approach allows to address one specific problem for each formalism, namely:

- We address the problem of introducing a cost model for $\mathcal{POR}$ when we reduce it to $\mathsf{SIFP}_{\mathsf{RA}}$.

- We show the equivalence between random access to the oracle and linear access when we encode $\mathsf{SIFP}_{\mathsf{RA}}$ into $\mathsf{SIFP}_{\mathsf{LA}}$.

The other steps of our reduction are not too complex because:

- We can employ a canonical reduction from an imperative formalism to a Turing machine when we reduce $\mathsf{SIFP}_{\mathsf{LA}}$ to multi-tape STMs
- We can adapt some well-known equivalence to reduce the number of tapes of a STM with a polynomial overhead.

This section is organized as follows:

- In Section 3.3.1, we define the imperative programming languages of the SIFP family, together with their small step and big step semantics.
- In Section 3.3.2, we prove that the functions computed by polynomial $\mathsf{SIFP}_{\mathsf{RA}}$ programs are exactly the $\mathcal{POR}$ functions.
- In Section 3.3.3, we show that the classes $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$ are equi-expressive.
- In Section 3.3.4, we address the reduction from $\mathsf{SIFP}_{\mathsf{LA}}$ to $\mathbf{SFP}_{\mathbf{OD}}$, which is a variation on **SFP** supporting *on-demand* access to the oracle.
- In Section 3.3.5, we show that $\mathbf{SFP}_{\mathbf{OD}}$ can be reduced to **SFP**.
- Finally, in Section 3.3.6, we put together all these partial results for showing that the second part of Conjecture 1 holds.

3.3.1 The SIFP language

In this section we define the family of imperative programming languages that we call SIFP, [11] together with their syntaxes and operational semantics. We do so to employ these languages in the reduction from $\mathcal{POR}$ to **SFP**. Indeed, this would allow us to give a school-book style reduction from an almost standard imperative paradigm — i.e. $\mathsf{SIFP}_{\mathsf{LA}}$ — to a TM-like paradigm — i.e. $\mathbf{SFP}_{\mathbf{OD}}$ — and then to show that $\mathcal{POR}$ can be reduced to $\mathsf{SIFP}_{\mathsf{LA}}$ and $\mathbf{SFP}_{\mathbf{OD}}$ can be reduced to **SFP**. This is, at least in our opinion, much easier than reducing directly $\mathcal{POR}$ into **SFP**.

The SIFP formalism is defined following the approach adopted by Winskel in [28] for the definition of IMP. Indeed, SIFP is an imperative programming language with assignments and a while construct; however, for coherence with respect to $\mathcal{POR}$, its expression are strings instead of natural numbers, as for IMP. Finally, we add a construct — either $\mathsf{Flip}(e)$ or $\mathsf{RandBit}()$ — to model random choices. Formally, The SIFP programming language can be instantiated in two different ways:

- $\mathsf{SIFP}_{\mathsf{RA}}$, which uses a primitive $\mathsf{Flip}(e)$ to store in a specific register called R, a random bit whose value is equal to $\omega(\sigma)$, where σ is the result of the evaluation of e in the current environment.
- $\mathsf{SIFP}_{\mathsf{LA}}$, which instead of $\mathsf{Flip}()$ uses the primitive $\mathsf{RandBit}()$ generating a random bit — and storing it in a specific register — without requiring any input value.

[11] Acronym of *String's Imperative Flipping Paradigm*

These two sub-languages are defined with two operational semantics each-one: the standard *big-step* semantic and the *small-step* operational semantic. In particular, the *big-step* semantic is the standard semantic model of SIFP, while the *small-step* semantic is introduced for two main reasons:

- It defines a cost model for the program: basically the time consumption of a program is the number of transition it requires to reach the ending state;
- It is a technical tool for the reduction from $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$, which allows us to prove a simulation result between the two classes by induction on the number of steps required by the computation.

The common features of all the $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$ are:

- The use of different families of registers (mainly for mnemonic purposes):
  - The family of registers $\{X_i\}_{i\in\mathbb{N}}$, which contain the program's input at the beginning of the computation.
  - A register R which contains the value of the computation and which determines, at the end of the computation, the value computed by the program.
  - The family of registers $\{S_i\}_{i\in\mathbb{N}}$, which are used for supporting programs' renaming $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$.
  - The family of registers $\{Y_i\}_{i\in\mathbb{N}}$. These registers employed, for instance, as backup registers during the reduction form $\mathcal{POR}$ to **SFP**.
  - The general purpose registers Q, Z, T, B.

  In what follows we will denote with *Id* a generic register.
- String expressions, which correspond to $\mathcal{L}$ term's interpretation, with the difference that here we do not allow binary operators to have two fully binary sub-expressions.
- An assignment statement.
- A looping statement.

Finally, for facilitating the reduction from $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$, we introduce a final statement, called **halt** to be placed at the end of a program. This, once again, is done with the aim of simplifying the reduction from $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$. We start by defining the syntax of SIFP's' correct programs. Later we introduce the definition of four different semantics for four different languages.

Definition 70 (Correct programs of SIFP)**.** The language of SIFP programs is $\mathcal{L}(\mathsf{Stm})$, i.e. the set of strings produced by the non-terminal symbol Stm defined by:

$$\begin{aligned}
\mathsf{Id} &::= X_i \mid Y_i \mid S_i \mid R \mid Q \mid Z \mid T \qquad i\in\mathbb{N}\\
\mathsf{Exp} &::= \epsilon \mid \mathsf{Exp.0} \mid \mathsf{Exp.1} \mid \mathsf{Id} \mid \mathsf{Exp} \sqsubseteq \mathsf{Id} \mid \mathsf{Exp} \land \mathsf{Id} \mid \neg\mathsf{Exp}\\
\mathsf{Stm}_{\mathsf{RA}} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm}_{\mathsf{RA}};\mathsf{Stm}_{\mathsf{RA}} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\}_{\mathsf{RA}} \mid \mathsf{Flip}(\mathsf{Exp})\\
\mathsf{Stm}_{\mathsf{LA}} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm}_{\mathsf{LA}};\mathsf{Stm}_{\mathsf{LA}} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\}_{\mathsf{LA}} \mid \mathsf{RandBit}()\\
\mathsf{Stm}'_{\mathsf{RA}} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm}_{\mathsf{RA}};\mathsf{Stm}'_{\mathsf{RA}} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\}_{\mathsf{RA}} \mid \mathsf{Flip}(\mathsf{Exp}) \mid \mathbf{halt};\\
\mathsf{Stm}'_{\mathsf{LA}} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm}_{\mathsf{LA}};\mathsf{Stm}'_{\mathsf{LA}} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\}_{\mathsf{LA}} \mid \mathsf{RandBit}() \mid \mathbf{halt};\\
\mathsf{Stm} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm};\mathsf{Stm} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\} \mid \mathsf{Flip}(\mathsf{Exp}) \mid \mathsf{RandBit}() \mid \mathsf{Flip}(\mathsf{Exp})\\
\mathsf{Stm}' &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm};\mathsf{Stm} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\} \mid \mathsf{Flip}(\mathsf{Exp}) \mid \mathsf{RandBit}() \mid \mathsf{Flip}(\mathsf{Exp}) \mid \mathbf{halt};
\end{aligned}$$

Notation 10. We suppose $\cdot;\cdot$ to be *right associative.*

On top of this grammar, we define the notion of $\mathsf{SIFP_{RA}}$ program, too.

Definition 71 ($\mathsf{SIFP_{RA}}$)**.** The language of the $\mathsf{SIFP_{RA}}$ programs is $\mathcal{L}(\mathsf{Stm_{RA}})$, i.e. the set of strings produced by the non-terminal symbol $\mathsf{Stm_{RA}}$ described in Definition 70.

Definition 72 ($\mathsf{SIFP_{LA}}$)**.** The language of the $\mathsf{SIFP_{LA}}$ programs is $\mathcal{L}(\mathsf{Stm_{LA}})$, i.e. the set of strings produced by the non-terminal symbol $\mathsf{Stm_{LA}}$ described in Definition 70.

Definition 73 (Store)**.** A store is a function $\Sigma : \mathsf{Id} \rightharpoonup \{\mathtt{0}, \mathtt{1}\}^*$.

Definition 74 (Empty store)**.** An *empty* store is a store which is total and constant on ϵ. We represent such object as $[]$.

The reason why we want a store to be total rather than undefined on some registers is because it allows us to access the content of all registers even if those have not been assigned to any expression, without defining error management functionalities for these languages. This, for instance, is useful when dealing with back-ups of register values.

As for [28], the semantics of programs can be given as a *function* between a a pair in the form ⟨Program, Store⟩ to another store. Where the store on the left of the tuple records the values of the registers at the beginning of the computation and the one on the right of the tuple contains the values of the registers at the end. The modifications on the store are done updating the values within its registers.

Definition 75 (Store updating)**.** We define the updating of a store Σ with a mapping from $y \in \mathsf{Id}$ to $\tau \in \{\mathtt{0}, \mathtt{1}\}^*$ as:

$$\Sigma[y \leftarrow \tau](x) := \begin{cases} \tau & \text{if } x = y \\ \Sigma(x) & \text{otherwise.} \end{cases}$$

The notion of *Store* allows us to define the semantics of a $\mathsf{SIFP_{RA}}$ program as a function which maps a pair $\langle \Sigma, \omega \rangle$ to another store. The semantics of the expression is the same for all the languages of the SIFP family; while, as we stated above, the semantics of the programs changes from $\mathsf{SIFP_{RA}}$ to $\mathsf{SIFP_{LA}}$. This is why we are defining a *Semantics of* SIFP *expressions* and an *Operational semantics of* $\mathsf{SIFP_{RA}}$ instead of an *Operational semantics of* SIFP.

Definition 76 (Semantics of SIFP expressions)**.** The semantics of an expression $E \in \mathcal{L}(\mathsf{Exp})$ is the smallest relation $\rightharpoonup : \mathcal{L}(\mathsf{Exp}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^*) \times \mathbb{O} \times \{\mathtt{0}, \mathtt{1}\}^*$ closed under the following rules:

$$\frac{}{\langle \epsilon, \Sigma \rangle \rightharpoonup \epsilon} \qquad \frac{\langle e, \Sigma \rangle \rightharpoonup \sigma}{\langle e.\mathtt{0}, \Sigma \rangle \rightharpoonup \sigma \frown \mathtt{0}} \qquad \frac{\langle e, \Sigma \rangle \rightharpoonup \sigma}{\langle e.\mathtt{1}, \Sigma \rangle \rightharpoonup \sigma \frown \mathtt{1}}$$

$$\frac{\langle e, \Sigma \rangle \rightharpoonup \sigma \qquad \Sigma(Id) = \tau \qquad \sigma \subseteq \tau}{\langle e \sqsubseteq Id, \Sigma \rangle \rightharpoonup \mathtt{1}} \qquad \frac{\langle e, \Sigma \rangle \rightharpoonup \sigma \qquad \Sigma(Id) = \tau \qquad \sigma \not\subseteq \tau}{\langle e \sqsubseteq Id, \Sigma \rangle \rightharpoonup \mathtt{0}}$$

$$\frac{\Sigma(Id) = \sigma}{\langle Id, \Sigma \rangle \rightharpoonup \sigma} \qquad \frac{Id \notin dom(\Sigma)}{\langle Id, \Sigma \rangle \rightharpoonup \epsilon}$$

$$\frac{\langle e, \Sigma \rangle \rightharpoonup \mathtt{0}}{\langle \neg e, \Sigma \rangle \rightharpoonup \mathtt{1}} \qquad \frac{\langle e, \Sigma \rangle \rightharpoonup \sigma \qquad \sigma \neq \mathtt{0}}{\langle \neg e, \Sigma \rangle \rightharpoonup \mathtt{0}}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \mathtt{1} \qquad \Sigma(Id) = \mathtt{1}}{\langle e \wedge Id, \Sigma\rangle \rightharpoonup \mathtt{1}} \qquad \frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \Sigma(Id) = \tau \qquad \sigma \neq \mathtt{1} \wedge \tau \neq \mathtt{1}}{\langle e \wedge Id, \Sigma\rangle \rightharpoonup \mathtt{0}}$$

Before introducing the semantics of programs, it is important to show that the relation introduced in Definition 76 is a function; this result is crucial for proving that the semantics of SIFP programs defined in the following sections are not just simple relations, but even functions. The proof of this statement is in Chapter 6, namely Lemma 34

3.3.1.1 Big step semantics

The *big-step* operational semantics of $\mathsf{SIFP}_{\mathsf{RA}}$ is defined following [28].

Definition 77 (Big Step Operational Semantics of $\mathsf{SIFP}_{\mathsf{RA}}$)**.** The semantics of a program $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ is the smallest relation $\triangleright \subseteq \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times \mathbb{O} \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*)$ closed under the following rules:

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma}{\langle Id \leftarrow e, \Sigma, \omega\rangle \triangleright \Sigma[Id \leftarrow \sigma]} \qquad \frac{\langle s, \Sigma, \omega\rangle \triangleright \Sigma' \qquad \langle t, \Sigma', \omega\rangle \triangleright \Sigma''}{\langle s; t, \Sigma, \omega\rangle \triangleright \Sigma''}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \mathtt{1} \qquad \langle s, \Sigma, \omega\rangle \triangleright \Sigma' \qquad \langle \mathbf{while}(e)\{s\}, \Sigma', \omega\rangle \triangleright \Sigma''}{\langle \mathbf{while}(e)\{s\}, \Sigma, \omega\rangle \triangleright \Sigma''} \qquad \frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \sigma \neq \mathtt{1}}{\langle \mathbf{while}(e)\{s\}, \Sigma, \omega\rangle \triangleright \Sigma}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \omega(\sigma) = b}{\langle \mathsf{Flip}(e), \Sigma, \omega\rangle \triangleright \Sigma[R \leftarrow b]}$$

The operational semantic of $\mathsf{SIFP}_{\mathsf{LA}}$ is almost identical to the operational semantic of $\mathsf{SIFP}_{\mathsf{RA}}$, apart from the fact that it is defined using oracle functions $\eta : \mathbb{N} \longrightarrow \mathbb{B}$ instead of functions $\omega : \mathbb{S} \longrightarrow \mathbb{B}$.

Definition 78 (Big Step Operational Semantics of $\mathsf{SIFP}_{\mathsf{LA}}$)**.** The semantics of a program $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{LA}})$ is the smallest relation $\triangleright \subseteq \big(\mathcal{L}(\mathsf{Stm}_{\mathsf{LA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times \mathbb{B}^{\mathbb{N}}\big) \times \big((\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times \mathbb{B}^{\mathbb{N}}\big)$ closed under the following rules:

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma}{\langle Id \leftarrow e, \Sigma, \eta\rangle \triangleright \langle \Sigma[Id \leftarrow \sigma], \eta\rangle}$$

$$\frac{\langle s, \Sigma, \eta\rangle \triangleright \langle \Sigma', \eta'\rangle \qquad \langle t, \Sigma', \eta\rangle \triangleright \langle \Sigma'', \eta''\rangle}{\langle s; t, \Sigma, \eta\rangle \triangleright \langle \Sigma'', \eta''\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \mathtt{1} \qquad \langle s, \Sigma, \eta\rangle \triangleright \langle \Sigma', \eta'\rangle \qquad \langle \mathbf{while}(e)\{s\}, \Sigma', \eta\rangle \triangleright \langle \Sigma'', \eta''\rangle}{\langle \mathbf{while}(e)\{s\}, \Sigma, \eta\rangle \triangleright \langle \Sigma'', \eta''\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \sigma \neq \mathtt{1}}{\langle \mathbf{while}(e)\{s\}, \Sigma, \eta\rangle \triangleright \langle \Sigma, \eta\rangle} \qquad \frac{}{\langle \mathsf{RandBit}(), \Sigma, \mathtt{b}\eta\rangle \triangleright \langle \Sigma[R \leftarrow \mathtt{b}], \eta\rangle}$$

Relying on the definition of the operational semantics for $\mathsf{SIFP}_{\mathsf{LA}}$ and $\mathsf{SIFP}_{\mathsf{RA}}$, it is possible to associate to each program the function it computes, simply placing the inputs in the registers of the family $X_{i\in\mathbb{N}}$ at the beginning of the computation and taking the output from the register R at the end of the computation. For sake of completeness, before employing the function $\triangleright$ to that aim, we must show that the operational semantics of both $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$ are not just simple relations, but even functions. The proof of this result is in Section 6.2.5 of Chapter 6, namely Lemmas 35 and 36.

Definition 79 (Function evaluated by a $\mathsf{SIFP}_{\mathsf{RA,LA}}$ program)**.** We say that the function evaluated by a correct $\mathsf{SIFP}_{\mathsf{RA}}$ or $\mathsf{SIFP}_{\mathsf{LA}}$ program P is $[\![\cdot]\!] : \mathcal{L}(Stm_{\mathsf{RA,LA}}) \longrightarrow (\mathbb{S}^n \times \mathbb{O} \longrightarrow \mathbb{S})$, defined as below[12]:

$$[\![P]\!] := \lambda x_1, \dots, x_n, \omega. \triangleright (\langle P, [][X_1 \leftarrow x_1], \dots, [X_n \leftarrow x_n], \omega\rangle)(R).$$

3.3.1.2 Small Step Semantics

The reduction of $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$ can be given in different ways. We found useful to formulate the result as a weak simulation result between a $\mathsf{SIFP}_{\mathsf{RA}}$ program and its $\mathsf{SIFP}_{\mathsf{LA}}$ implementation. To do so, we define two single *small-step* semantics for $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$ and show that they are equivalent to the *big-step* semantics in Definitions 77 and 78. In order to define these *small-step* semantics, we employ the **halt**-extended versions of the $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$ languages. This construct — which, semantically, causes the program to stop — simplifies some definitions and proofs if placed at the end of a program. Moreover, the *small-step* semantics are enriched by additional data structures:

- an ordinary binary string values in the case of $\mathsf{SIFP}_{\mathsf{LA}}$'s *small-step* semantics;
- a map from strings to Boolean values for $\mathsf{SIFP}_{\mathsf{LA}}$'s *small-step* semantics.

These structures are used to keep track of the accesses made to the oracle function — $\omega \in \mathbb{B}^{\mathbb{S}}$ for $\mathsf{SIFP}_{\mathsf{RA}}$ and $\eta \in \mathbb{B}^{\mathbb{N}}$ for $\mathsf{SIFP}_{\mathsf{LA}}$ — by the program.

Definition 80 ($\mathbb{SB}$-Map)**.** A $\mathbb{SB}$-Map is the is the smallest set containing the strings described by the following grammar:

$$M ::= (\sigma, \mathtt{b}) :: M \mid \varepsilon$$

with $\sigma \in \mathbb{S}$, $\mathtt{b} \in \mathbb{B}$ and ε being a generic symbol. We call $adt(M)$ the set of all the possible $\mathbb{SB}$-maps.

For sake of readability, we will represent both these binary strings and $\mathbb{SB}$-Maps uniformly, i.e. by means of the meta-variable Ψ. This is also aimed to highlight that these structures play the same role in both the relations. We can formally define the *small-step* operational semantics of $\mathsf{SIFP}_{\mathsf{LA}}$ as follows:

Definition 81 (Small Step semantics of $\mathsf{SIFP}_{\mathsf{LA}}$)**.** The step semantics of a program $P \in \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}})$ is the smallest relation

$$\rightsquigarrow_{\mathsf{LA}} \in \mathcal{P}\left(\left(\mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times \{\mathtt{0},\mathtt{1}\}^*\right) \times \left(\mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times \{\mathtt{0},\mathtt{1}\}^*\right)\right)$$

closed under the following rules:

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma}{\langle Id \leftarrow e; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle P, \Sigma[Id \leftarrow \sigma], \Psi\rangle} \qquad \frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \sigma \neq \mathtt{1}}{\langle \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle P, \Sigma, \Psi\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \mathtt{1}}{\langle \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle s; \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle}$$

[12]Instead of the infixed notation for $\triangleright$, we will use its prefixed notation. So, the notation express the store associated to the P, Σ and ω by $\triangleright$. Moreover, notice that we employed the same function symbol $\triangleright$ to denote two distinct functions: the *big-step* operational semantics of $\mathsf{SIFP}_{\mathsf{RA}}$ programs and the *big-step* operational semantics of $\mathsf{SIFP}_{\mathsf{LA}}$ programs

$$\frac{}{\langle \mathsf{RandBit}(); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle Q, \Sigma[R \leftarrow \mathtt{1}], \Psi \mathtt{0}\rangle}$$

$$\frac{}{\langle \mathsf{RandBit}(); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle Q, \Sigma[R \leftarrow \mathtt{0}], \Psi \mathtt{1}\rangle}$$

Before defining the *small-step* semantics for $\mathsf{SIFP}_{\mathsf{RA}}$, we would like to briefly discuss the *small-step* semantics of $\mathsf{SIFP}_{\mathsf{LA}}$ defined above. The semantics of the deterministic statements is canonical, indeed:

- We treat the $\cdot;\cdot$ as the action prefixing operator.
- We manage assignments basically recording on the store the effects of the statement.
- We have two rules for the **while**$()\{\}$ statement:
  - If the guard is true, the **while**$(e)\{s\}$ statement is interpreted executing its body s and then the whole statement again.
  - If the guard e is not true, its semantics skips to the next statement.

All these rules, are non-random so they do not modify the list Ψ. Differently, the two rules for the $\mathsf{RandBit}()$ statement can generate two different configurations depending on the value that all the oracles of the set on the right have on their n-th coordinate: if such value is $\mathtt{1}$ we record it in R, similarly if such value is $\mathtt{0}$. In a similar fashion, we define the step semantics for $\mathsf{SIFP}_{\mathsf{RA}}$:

Definition 82 (Small Step semantics of $\mathsf{SIFP}_{\mathsf{RA}}$)**.** The step semantics of a program $P \in \mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}})$ is the smallest relation

$$\rightsquigarrow_{\mathsf{RA}} \in \mathcal{P}\left(\left(\mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times adt(M)\right) \times \left(\mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0},\mathtt{1}\}^*) \times adt(M)\right)\right)$$

closed under the following rules:

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma}{\langle Id \leftarrow e; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle P, \Sigma[Id \leftarrow \sigma], \Psi\rangle} \qquad \frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \sigma \neq \mathtt{1}}{\langle \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle P, \Sigma, \Psi\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \mathtt{1}}{\langle \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle s; \mathbf{while}(e)\{s\}; P, \Sigma, \Psi\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \forall b \in \{\mathtt{0},\mathtt{1}\}.(\sigma, b) \notin \Psi \qquad \mathtt{b} \in \{\mathtt{0},\mathtt{1}\}}{\langle \mathsf{Flip}(e); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma[R \leftarrow \mathtt{b}], (\sigma, \mathtt{b}) :: \Psi\rangle}$$

$$\frac{\langle e, \Sigma\rangle \rightharpoonup \sigma \qquad \exists b \in \{\mathtt{0},\mathtt{1}\}.(\sigma, b) \in \Psi \qquad \mathtt{b} \in \{\mathtt{0},\mathtt{1}\}}{\langle \mathsf{Flip}(e); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma[R \leftarrow \mathtt{b}], \Psi\rangle}$$

The closure of the relation $\rightsquigarrow_{\cdot}$ for $\cdot \in \{\mathsf{LA}, \mathsf{RA}\}$ under the transitive property produces respectively the families of relations $\rightsquigarrow^n_{\mathsf{LA}}$ and $\rightsquigarrow^n_{\mathsf{RA}}$. These new relations allow us to extend the single step transition to the number of steps necessary for the reductions.

Definition 83 (Indexed Transitive Closure of $\rightsquigarrow_{\mathsf{LA}}$ and $\rightsquigarrow_{\mathsf{RA}}$)**.** For $\cdot \in \{\mathbf{LA}, \mathbf{RA}\}$, we define the family of relations $\{\rightsquigarrow^n_{\cdot}\}_{n\in\mathbb{N}}$ as the set of functions closed and minimal with respect to the following rules:

$$\langle P, \Sigma, \Psi\rangle \rightsquigarrow^0_{\cdot} \langle P, \Sigma, \Psi\rangle$$
$$\left(\langle P, \Sigma, \Psi\rangle \rightsquigarrow^n_{\cdot} \langle P', \Sigma', \Psi'\rangle \wedge \langle P', \Sigma', \Psi'\rangle \rightsquigarrow_{\cdot} \langle P'', \Sigma'', \Psi''\rangle\right) \to \left(\langle P, \Sigma, \Psi\rangle \rightsquigarrow^{n+1}_{\cdot} \langle P'', \Sigma'', \Psi''\rangle\right).$$

This single-step semantics can be used to define the cost model we will see in Section 3.3.2 to prove the polynomial time complexity of the translation of $\mathcal{POR}$ in $\mathsf{SIFP}_{\mathsf{RA}}$.

Definition 84 (Cost Model For $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$)**.** We say that a program $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{LA}}) \cup \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ has time complexity $f : \mathbb{N} \longrightarrow \mathbb{N}$ if and only if for every $x \in \mathbb{S}$, there is a $k \in \mathbb{N}$, a store $\Sigma' : Id \rightharpoonup \mathbb{S}$, and a Ψ either in $adt(L)$ or $adt(M)$ such that if

$$\langle P; \mathbf{halt};, [][X \leftarrow x], \mathbb{O}\rangle \rightsquigarrow^k_{\mathsf{RA}} \langle \mathbf{halt};, \Sigma', \Psi\rangle,$$

then $k \leq f(|x|)$.

Finally, we define the transitive closure of $\rightsquigarrow_{\cdot}$ for $\cdot \in \{\mathbf{LA}, \mathbf{RA}\}$.

Definition 85 (Step Semantics Transitive Closure)**.**

$$\rightsquigarrow^*_{\mathsf{LA}} := \bigcup_{n\in\mathbb{N}} \rightsquigarrow^n_{\mathsf{LA}};$$
$$\rightsquigarrow^*_{\mathsf{RA}} := \bigcup_{n\in\mathbb{N}} \rightsquigarrow^n_{\mathsf{RA}}.$$

The oracle-related informationstored in the data structure Φ is aimed at associating to each possible reduction the set of oracle functions which lead the corresponding *big-step* reduction to produce the same result. These two relations are defined as follows.

Definition 86 (Oracle-Map Coherency Relation for $\mathsf{SIFP}_{\mathsf{RA}}$)**.** For every $\omega \in \mathbb{O}$ and $\Psi \in adt(M)$, we write $\omega \succ \Psi$ if and only if

$$\forall (\sigma, \mathtt{b}) \in \Psi . \omega(\sigma) = \mathtt{b}.$$

Similarly, for $\mathsf{SIFP}_{\mathsf{LA}}$:

Definition 87 (Oracle-String Coherency Relation for $\mathsf{SIFP}_{\mathsf{LA}}$)**.** For every $\eta \in \mathbb{B}^{\mathbb{N}}$ and $\Psi \in \{\mathtt{0}, \mathtt{1}\}^*$, we write $\eta \succ \Psi$ if and only if Ψ is a prefix of η.

This relation, in addition to expressing the coherency between an oracle function and a collection of constraints, allows us to associate naturally a measure to strings and to $\mathbb{SB}$-maps. This measure corresponds to the measure of the functions which are coherent with it.

Definition 88 (String Measure)**.** For each $\Psi \in Ss$, we define $\mu(\Psi)$ as follows:

$$\mu(\Psi) = \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\}).$$

Definition 89 ($\mathbb{SB}$-map Measure)**.** For each $\Psi \in Ss$, we define $\mu(\Psi)$ as follows:

$$\mu(\Psi) = \mu(\{\omega \in \mathbb{B}^{\mathbb{S}} | \omega \succ \Psi\}).$$

Notice that, as for program semantics, we are using the same symbol $\succ$ to denote two different relations, this emphasizes the strong link between the two relations.

3.3.1.3 Lexical facilitations

Before getting into the reduction involving $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$, we introduce some notational facilitations in order to increase the proof's readability.

Notation 11. Writing a SIFP program, we use the notation $E \sqsubset F$ as a shorthand (syntactic sugar) for $\neg(F \sqsubseteq E)$.

The notation above may be handled with care: it is a simplification. Indeed it does not hold that if a string σ is not a weak prefix of another string τ it is a strong prefix of τ, but when we employ the shorthand $E \sqsubset F$ writing our reduction, we always work under the invariant that E is a prefix of F. We can show that, in those cases, the semantics of the expression behaves as expected.

Remark 4. $E \subseteq F \to (E \subset F) \leftrightarrow \neg(F \subseteq E)$.

Proof. Suppose $E \subseteq F$. Suppose that $\neg(F \subseteq E)$, this entails $\neg(F = E)$ which is equivalent to $\neg(E = F)$, together with $E \subseteq F$, we get $E \subset F$. Suppose $E \subset F$ and $F \subseteq E$, it hoslds that $F \subseteq E \leftrightarrow F \subset E \vee F = E$; in the first case we have an absurd because $\subset$ is not reflexive, while in the second case we obtain $E \subset E$ which is absurd and we conclude the proof. ☐

Notation 12. Writing a SIFP program, we the notation

```
**if**$(c)\{$
Stm;
$\}$
```

for representing:

```
$B \leftarrow \epsilon.1$;
  **while**$(c \wedge B)\{$
  Stm;
  $B \leftarrow \epsilon.0$
  $\}$
```

It is indeed true that:

- The statement Stm is executed if and only if c holds.
- The statement Stm is executed only once.

Notation 13 (pseudo-procedure)**.** A pseudo-procedure is a syntactic sugar for the SIFP's language, which consists in a *pseudo-procedure's name*, a *body* a list of *formal parameters.* A call to such expression must be interpreted as the inlining of the pseudo-procedure's body in the place of the call in which the names of the formal parameters by the actual parameters and all the *free* names of the body are substituted by fresh names of the family $\{S_k\}_{k\in\mathbb{N}}$.

Pseudo-procedures allow us to factorize pieces of code and represent programs in a concise way, but they have nothing to do with actual procedures and calls. Indeed, as we have shown, the SIFP formalism does not contain any notion of function and callable object.

Before getting into the actual reduction, we need to face some preliminar work.

3.3.2 From $\mathcal{POR}$ to the $\mathsf{SIFP}_{\mathsf{RA}}$ Language

In this section we show that the $\mathcal{POR}$ functions can be encoded by means of *poly-time* $\mathsf{SIFP}_{\mathsf{RA}}$ programs. In particular, the encoding $\mathcal{POR}$'s bounded recursion in SIFP is not completely straightforward: these are defined upon three functions $h_0, h_1, g \in \mathcal{POR}$ and a term $t \in \mathcal{L}$, which is used as a size bound to the terms obtained from h_0 and h_1. Basically, this bound guarantees that all the $\mathcal{POR}$ functions are polynomially bounded (in time and size, according to Lemma 10). For this reason, we need to show that we can express the size bound $t \in \mathcal{L}$ by means of an expression of SIFP, i.e. a production of $\mathcal{L}(\mathsf{Exp})$. The polynomial complexity of the $\mathcal{POR}$'s encoding in $\mathsf{SIFP}_{\mathsf{RA}}$ relies on two other results:

- $\mathcal{POR}$ terms have polynomial size in the size of their variables, as proved in Lemma 10.
- $\mathcal{L}$ terms have polynomial size in the size of their variables, as proved in Lemma 11.

For this reason, to show that all the $\mathcal{POR}$ functions can be represented by means of *poly-time* $\mathsf{SIFP}_{\mathsf{RA}}$ programs, we must preliminarily show the aforementioned results. For sake of readability, in the current section, some of the proof of these resulsts are moved to Chapter 6, Section 6.2.5.

Lemma 10. *The size of a term in $\mathcal{L}$ is poynomial in the size of its variables.*

This result can be leveraged to prove the inductive case of the analogous result for $\mathcal{POR}$ terms.

Lemma 11. *For each $f \in \mathcal{POR}$, the following holds:*

$$\forall x_1, \dots, x_n . \forall \omega . \exists p \in \mathsf{POLY} . |f(x_1, \dots, x_n, \omega)| \leq p(|x_1|, \dots, |x_n|).$$

Thanks to these results, we can step to showing that each $\mathcal{L}$ term can be represented in $\mathsf{SIFP}_{\mathsf{RA}}$ (Lemma 14). As we mentioned before, this result is necessary to show that the bounded recursion schema can be encoded in $\mathsf{SIFP}_{\mathsf{RA}}$. However, before showing the proof of Lemma 14, we need to prove some intermediate results, which will simplify the proof. In particular, for sake of readability, we define the pseudo-procedures *copyb*, which copies the $|Z|$-th bit of S at the end of R, given that Z contains the $|Z|$-th prefix of S, and prove its correctness and complexity. Moreover, the definition of this simple program will help us in showing to the reader the schema we employ to prove that a certain program is — or is part of — the encoding of some $\mathcal{POR}$ functions in $\mathsf{SIFP}_{\mathsf{RA}}$ or $\mathsf{SIFP}_{\mathsf{LA}}$:

- We define the encoding.
- We prove the correctness of such encoding with respect to some invariant properties.
- We show that the complexity of such program is polynomial in time.

Complexity results are proved referring to the cost model described in Definition 84.

Definition 90 (*copyb* pseudo-procedure)**.** The *copyb* pseudo-procedure is defined as:

```
$copyb(Z, S, R) :=$ if$(Z.\mathsf{0} \sqsubseteq S)\{$
        $Z \leftarrow Z.\mathsf{0};$
        $R \leftarrow R.\mathsf{0};$
    $\}$
    if$(Z.\mathsf{1} \sqsubseteq S)\{$
        $Z \leftarrow Z.\mathsf{1};$
        $R \leftarrow R.\mathsf{1};$
    $\}$
```

Lemma 12 (Complexity of *copyb*)**.** *The pseudo-procedure copyb requires a number of steps which is a polynomial in the sizes of its arguments with respect to Definition 84.*

Proof. The two **if**( ){ }s are described in Remark 12, and they cause no iteration. Moreover, the two statements are mutually exclusive, so this pseudo-procedure requires at most 5 steps. □

Lemma 13 (Correctness of *copyb*)**.** *After an execution of copyb:*

- *If the first argument is a strong prefix of the second, the size of the first argument (Z) increases by one, and is still a prefix of the second argument (S).*
- *Otherwise, the values stored in the first two registers don't change.*
- *Each bit which is stored at the end of Z is stored at the end of R.*

Proof. Suppose that the value stored in Z is a strong prefix of the value which is stored in S. Clearly it is true that $Z.\mathsf{0} \sqsubseteq S \vee Z.\mathsf{1} \sqsubseteq S$. In both cases, *copyb* increases the length of the portion of Z which is a prefix of S. If Z is not a prefix of S none of the two `ifs` is executed. The last conclusion comes from the observation that each assignment to Z is followed by a similar assignment to R. □

Lemma 14 (Term Representation in $\mathsf{SIFP_{RA}}$)**.** *All the terms of $\mathcal{L}$ can be represented in $\mathsf{SIFP_{RA}}$. Formally: $\forall t \in \mathcal{L}.\exists \mathfrak{M} \in \mathcal{L}(\mathsf{Stm})[\![\mathfrak{M}_t]\!](\sigma_1, \dots, \sigma_n) = t(\sigma_1, \dots, \sigma_n)$.*

Proof. We proceed by induction on the syntax of t. The correctness of such implementation is given by the following invariant properties:

- The result of the computation is stored in R.
- The inputs are stored in the registers of the group X.
- The function $\mathfrak{M}$ does not write the values it accesses as input.

$\mathfrak{M}$ is defined as follows:

- $\mathfrak{M}_\epsilon := R \leftarrow \epsilon$
- $\mathfrak{M}_\mathsf{0} := R \leftarrow \boldsymbol{\epsilon}.\mathsf{0}$
- $\mathfrak{M}_\mathsf{1} := R \leftarrow \boldsymbol{\epsilon}.\mathsf{1}$
- $\mathfrak{M}_{Id} := R \leftarrow Id$.

This pseudo-procedure *copyb* turns out to be useful in both the encodings of $\frown$ and $\times$. For the $\frown$ operator, we proceed with the following encoding:

```
$\mathfrak{M}_{t\frown s} :=\mathfrak{M}_s$
        $S \leftarrow R;$
        $\mathfrak{M}_t$
        $Z \leftarrow \epsilon;$
        **while**$(Z \sqsubset S)\{$
          $copyb(Z, S, R)$
          $\}$
```

The correctness of $\mathfrak{M}_{t\frown s}$ is a consequence of the correctness of *copyb*. We encode the $\times$ function as follows:

```
$\mathfrak{M}_{t\times s} :=\mathfrak{M}_t$
        $T \leftarrow R;$
        $\mathfrak{M}_s$
        $S \leftarrow R;$
```

```
$Z \leftarrow \epsilon$;
$R \leftarrow \epsilon$;
$Q \leftarrow \epsilon$;
while($Z \sqsubset S$){
  if($Z.\mathsf{0} \sqsubseteq S$){
    $Z \leftarrow Z.\mathsf{0}$;
    while($Q \sqsubset T$){
      $copyb(Q, T, R)$
      }
    $Q \leftarrow \epsilon$;
    }
  if($Z.\mathsf{1} \sqsubseteq S$){
    $Z \leftarrow Z.\mathsf{1}$;
    while($Q \sqsubset T$){
      $copyb(Q, T, R)$
      }
    $Q \leftarrow \epsilon$;
    }
  }
```

This program is correct because of the IH and the correctness of *copyb*, which has been proved in Lemma 13: the procedure, basically, matches s, by writing in the Z register its prefixes. At each cycle a new character is added to the Z register and the content of T (t according to the IH) is added in R; this process is repeated until it is equal to S. □

For each term of $\mathcal{L}$ t, it holds that $\mathfrak{M}_t$ is poly-time.[13] Up to now we have shown that the terms of $\mathcal{L}$ can be represented by means of polynomial SIFP expressions: as we mentioned above, this is fundamental for showing that the bounded recursion schema can be implemented in $\mathsf{SIFP_{RA}}$. All the preliminar work for the main result of this section has been accomplished, so we can state it formally.

Lemma 15 (Implementation of $\mathcal{POR}$ in $\mathsf{SIFP_{RA}}$)**.** *For every function $f \in \mathcal{POR}$, there is a $\mathsf{SIFP_{RA}}$ program P such that: $forall x_1, \dots x_n.[\![P]\!](x_1, \dots, x_n, \omega) = f(x_1, \dots, x_n, \omega)$. Moreover, if $f \in \mathcal{POR}^-$, then $\mathfrak{L}_f$ does not contain any $\mathsf{Flip}(e)$ statement.*

The reader may observe that the statement contains two claims: one concerning the correctness of the translation itself and another concerning *its shape*. The second result may appear superfluous. Indeed, it is unnecessary with respect to the reduction from $\mathcal{POR}$ to $\mathsf{SIFP_{RA}}$, but it will be used later — precisely in Corollary 7, where we show that the functions $\mathcal{POR}$ described in Section 6.1 are in $\mathsf{SIFP_{LA}}$, too. Lemma 15, employs the pseudo-procedure *trunc* to represent the $\cdot|.$ operator of $\mathcal{L}$ in $\mathsf{SIFP_{RA}}$. This function symbol is employed in the inductive case of the bounded recursion: it is placed after each recursive call in the induction schema, and truncates the output of the function to a polynomially long term.[14]

[13]This is shown in Chapter 6, Lemma 37.

[14]Again, for sake of readability, we place the definition of the *trunc* pseudo-procedure in Chapter 6 together with some results showing formally that the pseudo-procedure is polynomial in time and respects the behavior

We have shown that all the ingredients bounded recursion schema can be encoded in $\mathsf{SIFP_{RA}}$ by means of poly-time programs. We can now address the claim of Lemma 15.

Proof of Lemma 15. For each function $f \in \mathcal{POR}$ we define a program $\mathfrak{L}_f$ such that: $[\![\mathfrak{L}_f]\!](x_1, \dots, x_n) = f(x_1, \dots, x_n)$ by induction on the structure of f. The correctness of $\mathfrak{L}_f$ is given by the following invariant properties:

- The result of the computation is stored in R.
- The inputs are stored in the registers of the group X.
- The function $\mathfrak{L}.$ does not change the values it accesses as input.

We define the function $\mathfrak{L}_f$ as follows.

- $\mathfrak{L}_E := R \leftarrow \epsilon$.
- $\mathfrak{L}_{S_0} := R \leftarrow X_0.\mathsf{0}$.
- $\mathfrak{L}_{S_1} := R \leftarrow X_0.\mathsf{1}$.
- $\mathfrak{L}_{P^n_i} := R \leftarrow X_i$.
- $\mathfrak{L}_C := R \leftarrow X_1 \sqsubseteq X_2$.
- $\mathfrak{L}_Q := \mathsf{Flip}(X_1)$.

The basic functions' correctness is trivial. Moreover, it is simple to see that the only translation containing $\mathsf{Flip}(e)$ for some $e \in \mathcal{L}(\mathsf{Exp})$ is the translation of Q.

The encoding of the composition and of the bounded recursion are defined as follows:

$$
\begin{aligned}
\mathfrak{L}_{g(h_1(x_1,\dots,x_n,\omega),\dots h_k(x_1,\dots,x_n,\omega),\omega)} := {} & \mathfrak{L}_{h_1}(X_1, \dots, X_n)\\
& S_1 \leftarrow R;\\
& \dots\\
& \mathfrak{L}_{h_k}(X_1, \dots, X_n)\\
& S_k \leftarrow R;\\
& Y_1 \leftarrow X_1;\\
& \dots\\
& Y_{\max(n,k)} \leftarrow X_{\max(n,k)};\\
& X_1 \leftarrow S_1;\\
& \dots\\
& X_k \leftarrow S_k;\\
& \mathfrak{L}_g(X_1, \dots, X_k)\\
& X_1 \leftarrow Y_1\\
& \dots\\
& X_{\max(n+1,k+1)} \leftarrow Y_{\max(n+1,k+1)};
\end{aligned}
$$

described above. Precisely, the *trunc* pseudo-procedure is introduced in Definition 143, its polynomial complexity is shown in Lemma 38 and its correctness with respect to the behavior described above is shown in Lemma 39.

The correctness of this encoding with respect to the invariants is a consequence of the IHs. Furthermore, suppose that f does not contain Q in its definition, then none of g and h_i for $1 \leq i \leq k$ does. So, by IH and for the construction we did, we know that $\mathfrak{L}_f$ does not contain any $\mathsf{Flip}(e)$ statement. Supposing that g takes n parameters, bounded recursion is encoded as follows:

```
$\mathfrak{L}_{ite(g,h_1,h_2,t)}$ :=$Y_0 \leftarrow X_{n+1}$;
        $X_{n+1} \leftarrow \epsilon$;
        $\mathfrak{L}_g(X_1, \dots, X_n)$
        $X_{n+2} \leftarrow R$;
        **while**$(X_{n+1} \sqsubset Y_0)\{$
          **if**$(X_{n+1}.\mathtt{0} \sqsubseteq Y_0)\{$
            $\mathfrak{M}_t(X_1, \dots, X_n, X_{n+1})$
            $T \leftarrow R$;
            $\mathfrak{L}_{h_0}(X_1, \dots, X_n, X_{n+1}, X_{n+2})$
            $trunc(T, R)$;
            $X_{n+2} \leftarrow R$;
            $X_{n+1} \leftarrow X_{n+1}.\mathtt{0}$;
       $\}$  **if**$(X_{n+1}.\mathtt{1} \sqsubseteq Y_0)\{$
            $\mathfrak{M}_t(X_1, \dots, X_n, X_{n+1})$
            $T \leftarrow R$;
            $\mathfrak{L}_{h_1}(X_1, \dots, X_n, X_{n+1}, X_{n+2})$
            $trunc(T, R)$;
            $X_{n+2} \leftarrow R$;
            $X_{n+1} \leftarrow X_{n+1}.\mathtt{1}$;
       $\}\}$
        $R \leftarrow X_{n+2}$;
```

This encoding is quite cumbersome: we suppose that the bounded recursive function takes $n+1$ parameters as input and that one of those, the $n+1$-th is the recursion bound. The correctness of this piece of code with respect to the invariant properties mentioned above can be shown by induction on the value of the induction parameter σ together with another invariant property: at the end of the evaluation X_{n+1} contains σ.

$\boldsymbol{\epsilon}$ In this case, the evaluation skips the outermost while, so the correctness of the overall code is a consequence of the induction hypothesis on g. At the end of the evaluation, $\Sigma(X_{n+1}) = \boldsymbol{\epsilon}$.

$\tau\mathtt{b}$ In this case we know that the code behaves correctly for input τ. Suppose that the input is now $\tau\mathtt{b}$; according to the semantics of the **while**$()\{\}$ statement, the execution of the code on $\tau\mathtt{b}$ is identical to the execution of the code on τ (which respects the invariant properties by induction hypothesis), but with another cycle at the end. This last cycle matches the value of $\mathtt{b}$. Suppose it to be $\mathtt{0}$. The code computes the size bound $\mathfrak{M}_t$, without changing the values in the other registers, then it computes h_0 and truncates it, then prepares the inputs for the next cycle (simulated call). The correctness of the overall procedure is a consequence of the IHs and the correctness of the *trunc* pseudo-procedure (Lemma 39).

The proof that f does not contain $\mathsf{Flip}(e)$ statements for any e is identical to the case of composition. $\square$

Even intuitively, it is easy to see that the schema $\mathfrak{L}_P$ we have introduced in Lemma 15 is poly-time with respect to the cost model described in Definition 84.[15] This result is a consequence of all the other complexity results mentioned in this section. On top of it, we will show that the implementation of a $\mathcal{POR}$ function on an STM is poly-time, i.e. that it is equivalent to a function in **SFP**.

Thus, we can state the first step of the second part of Lemma 9.

Corollary 6. *For each $f \in \mathcal{POR}$ there is a poly-time program $P \in \mathsf{SIFP}_{\mathsf{RA}}$ such that for each $x, y \in \mathbb{S}$:*

$$\mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | f(x,\omega) = y\}\right) = \mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right).$$

Proof. Consequence of Lemmas 15 and 40 $\square$

As a consequence of Corollary 6, we know that for showing

$$\forall f \in \mathcal{POR}.\exists g \in \mathbf{SFP}.\mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | f(x,\omega) = y\}\right) = \mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | g(x,\omega) = y\}\right),$$

it suffices to show that:

$$\forall P \in \mathsf{SIFP}_{\mathsf{RA}}.\exists g \in \mathbf{SFP}.\mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right) = \mu\left(\{\omega \in \{\mathtt{0},\mathtt{1}\}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right).$$

To prove this second claim, we define a k-taped *on-demand* STM, where k is linear in the amount of registers used by P. Thus, it is important to show that, fixed $f \in \mathcal{POR}$, the corresponding program $\mathfrak{L}_f$ uses a constant amount of registers, which is uniquely determined by f.

Remark 5. *The number of registers used by $\mathfrak{L}_f$ is finite.*

Definition 91. Programs are finite production, for this reason we can define a function $\#_r^{\mathsf{Stm}} : \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}}) \longrightarrow \mathcal{P}(Id)$ which records the identificators of the registers used by a $\mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ program. Since $\forall f \in \mathcal{POR}.\mathfrak{L}_f \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$, the claim holds. Such function can defined as follows:

$$\begin{aligned}
\#_r^{\mathsf{Stm}}(\mathsf{Flip}(e)) &:= \#_r^{\mathsf{Exp}}(e)\\
\#_r^{\mathsf{Stm}}(Id \leftarrow e) &:= \{Id\} \cup \#_r^{\mathsf{Exp}}(e)\\
\#_r^{\mathsf{Stm}}(\mathbf{while}(e)\{s\} &:= \#_r^{\mathsf{Exp}}(e) \cup \#_r^{\mathsf{Stm}}(s)\\
\#_r^{\mathsf{Stm}}(p; s) &:= \#_r^{\mathsf{Stm}}(p) \cup \#_r^{\mathsf{Stm}}(s)
\end{aligned}$$

$$\begin{aligned}
\#_r^{\mathsf{Exp}}(\epsilon) &:= \emptyset\\
\#_r^{\mathsf{Exp}}(e.\mathtt{0}) &:= \emptyset\\
\#_r^{\mathsf{Exp}}(e.\mathtt{1}) &:= \emptyset\\
\#_r^{\mathsf{Exp}}(Id) &:= \{Id\}\\
\#_r^{\mathsf{Exp}}(e \sqsubseteq Id) &:= \{Id\} \cup \#_r^{\mathsf{Exp}}(e)\\
\#_r^{\mathsf{Exp}}(e \wedge Id) &:= \{Id\} \cup \#_r^{\mathsf{Exp}}(e)\\
\#_r^{\mathsf{Exp}}(\neg e) &:= \#_r^{\mathsf{Exp}}(e).
\end{aligned}$$

[15] A formal proof of this result is given in Chapter 6, Lemma 40.

Proof. We can show by induction that:

1. Each register which appears in a program $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ is in $\#_r\mathsf{Stm}(P)$.
2. $\forall P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}}).\#_r^{\mathsf{Stm}}(P)$ is finite.

As a consequence $|\#_r^{\mathsf{Stm}}(P)|$ is exactly an upper bound to number of registers used by a $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ □

3.3.3 From $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$

In the previous section, we have shown that all the $\mathcal{POR}$ functions can be represented by means of poly-time $\mathsf{SIFP}_{\mathsf{RA}}$ programs. Unfortunately, this is not sufficient to prove the result we are aiming to, namely that each $\mathcal{POR}$ function can be represented by a **SFP** function (Lemma 9). However, the proof of Lemma 9 can be built starting from the correspondence between $\mathcal{POR}$ and $\mathsf{SIFP}_{\mathsf{RA}}$. However, as we explained above, $\mathsf{SIFP}_{\mathsf{RA}}$ is not directly suitable for a direct encoding on Stream machines for many reasons. One of those is the fact that it adopts a *random access to the oracle*, while **SFP** does not. To bridge this gap, we defined the $\mathsf{SIFP}_{\mathsf{LA}}$ formalism: the dialect of SIFP which does not use the $\mathsf{Flip}(e)$ primitive, but a simpler modality of access to the oracle: $\mathsf{RandBit}()$. This primitive does not allow to specify the coordinate to be queried to the oracle. Instead, it accesses the bits of a stream $\eta : \mathbb{N} \longrightarrow \mathbb{B}$ in a sequential way. This is why we say that $\mathsf{SIFP}_{\mathsf{LA}}$ adopts a form of *linear access to the oracle.* In this section we show that each $\mathsf{SIFP}_{\mathsf{RA}}$ program can be represented by means of a $\mathsf{SIFP}_{\mathsf{LA}}$ program equivalent to the first with respect to the measure of the oracles mapping each input to the output. Namely:

Lemma 16. *For each total program $P \in \mathsf{SIFP}_{\mathsf{RA}}$ there is a $Q \in \mathsf{SIFP}_{\mathsf{LA}}$ such that:*

$$\forall x, y.\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | [\![P]\!](x, \omega) = y\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | [\![Q]\!](x, \eta) = y\}\right).$$

Moreover, if P is poly-time Q is poly-time, too.

This result is analogous to what we proved in Corollary 6. In particular, we show that each program in $\mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$ can be simulated by means of a program in $\mathcal{L}(\mathsf{Stm}_{\mathsf{LA}})$ coherently with Lemma 16. We derive Lemma 16 as a corollary of the proof that $\mathsf{SIFP}_{\mathsf{RA}}$ can be simulated in $\mathsf{SIFP}_{\mathsf{LA}}$ with respect the *small-step* semantic relations defined in Section 3.3.1.2, Definitions 81 and 82. Indeed, the idea behind those semantics is to enrich the *big-step* operational semantic with some pieces of information necessary to build an induction proof of the reduction from $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$, in particular:

- We add a value Ψ in order to keep track of the constraints associated to each sequence of transitions. In this way, we can directly prove Lemma 16 as a simulation result. In other words, the transitive closures of $\rightsquigarrow_{\mathsf{LA}}$ and $\rightsquigarrow_{\mathsf{RA}}$ shape this parameter in order to build, together with the reduction, a complete representation of the measurable set of functions which bring the reduction to a certain configuration.
- The book-keeping information which is contained in Ψ can be used to enrich each configuration of the evaluation of P with the associative table which is built by the corresponding program P' during its evaluation.

The main issue we encountered during our reduction was proving that the random access can be simulated building, in a specific register, an associative table, which records all the queries which have been previously simulated. This approach requires also that:

- At each simulated query, the destination program looks up this table;
- If it finds the queried coordinate, then it returns the result stored in the table otherwise:
  - It reduces $\mathsf{Flip}(e)$ to a call of $\mathsf{RandBit}()$ which outputs either $\mathtt{b} = \mathsf{0}$ or $\mathtt{b} = \mathsf{1}$.
  - It records the couple $\langle e, \mathtt{b}\rangle$ in the associative table and returns the $\mathtt{b}$.

We believe that it is not too much of a problem to see, at least intuitively, that this kind of simulation preserves the probability measure associated to any possible input-output pair during the translation of a $\mathsf{SIFP_{RA}}$ program into an equvalent $\mathsf{SIFP_{LA}}$. However, the formal proof of Lemma 16 requires some effort to be given in its entirety. It is structured as follows:

- We show that the two operational semantics given for $\mathsf{SIFP_{RA}}$ and $\mathsf{SIFP_{LA}}$ are equally expressive, in Section 3.3.3.1.
- We define a relation Θ between configurations of the *small-step* semantics of $\mathsf{SIFP_{RA}}$ and $\mathsf{SIFP_{LA}}$. This is done in Section 3.3.3.2.
- We prove a simulation result between the two semantics with respect to the Θ relation.

3.3.3.1 Relating Small Step Semantics and Big Step Semantics

In this section, our goal is to define a characterization of the function evaluated by a $\mathcal{L}(\mathsf{Stm_{LA}})$ program basing on the *small-step* semantics rather than on the *big-step* operational semantics we used before. From a non-quantitative point of view, the equivalence of those two semantics is almost trivial: indeed we can characterize the notion of *value* computed by a program of $\mathsf{SIFP_{LA}}$ and $\mathsf{SIFP_{RA}}$ using this the *small-step* semantics, simply decomposing the *big-step* in many *small-steps* of computation. Intuitively, if it is possible to reach a configuration with a single step, it is also possible to decompose it by single operations and to represent the whole computation placing a *final configuration marker* — the **halt**; instruction — at the end of the program and using the transitive closure of the transition relation. This is done in Characterizations 1 and 2, respectively for $\mathsf{SIFP_{LA}}$'s and $\mathsf{SIFP_{RA}}$'s semantics. However, we also need to relate the *big-step* and the *small steps* semantics under a quantitative point of view: to do so, we rely on the additional book-keeping value Ψ — which can either be in $\mathbb{S}$ or in $adt(M)$ — introduced specifically for this goal in the small step semantics of $\mathsf{SIFP_{LA}}$ and $\mathsf{SIFP_{RA}}$. This is done in Lemmas 17 and 18.

Finally all the four main results of this section rely on the relations $\succ$, which were introduced in Definitions 87 and 86 to the aim of relating oracles and their Ψ-representation in the setting of the *small-step* semantics. Indeed, in Characterizations 1 and 2, those two homonym relations are employed to show that the Ψ value collected by the *small-step* semantics is coherent with the possible values of the oracle function leading the program to the same final store by using the *big-step* semantic instead of the *small-step* one. On the other hand, in Lemmas 17 and 18, it is used implicitly to show that, fixed a program and an input, all the possible transitions are captured by the *small-step* semantics.

Characterization 1 (Characterization of the Notion of Function evaluated bu a $\mathsf{SIFP_{LA}}$ program)**.** *Each program $P \in \mathsf{SIFP_{LA}}$ is such that for each $\sigma, \tau \in \mathbb{S}$ and for each $\eta \in \mathbb{B}^{\mathbb{N}}$ it holds that:*

$$[\![P]\!](\sigma, \eta) = \tau \leftrightarrow \langle P; \mathbf{halt};, [][X_1 \leftarrow \sigma], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Sigma, \Psi\rangle.$$

For some $\Psi \in \{\mathsf{0}, \mathsf{1}\}^*$ such that $\eta \succ \Psi$, and a store Σ such that $\Sigma(R) = \tau$.

Characterization 2 (Characterization of the Notion of Function evaluated bu a $\mathsf{SIFP}_{\mathsf{RA}}$ program)**.** *Each program $P \in \mathsf{SIFP}_{\mathsf{RA}}$ is such that for each $\sigma, \tau \in \mathbb{S}$ and for each $\omega \in \mathbb{B}^{\mathbb{S}}$ it holds that:*

$$[\![P]\!](\sigma, \omega) = \tau \leftrightarrow \langle P; \mathbf{halt};, [][X_1 \leftarrow \sigma], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \mathbf{halt};, \Sigma, \Psi\rangle.$$

For some $\Psi \in adt(M)$ such that $\omega \succ \Psi$, and a store Σ such that $\Sigma(R) = \tau$.

The proofs of these results are given on the syntax of the program P. [16]

Lemma 17 (Quantitative Correspondence for $\mathsf{SIFP}_{\mathsf{LA}}$)**.** *For every* total $\mathsf{SIFP}_{\mathsf{LA}}$ *program P, and for each store Σ there are a finite sequence of strings $\Psi_1 \dots \Psi_k$ and a finite sequence of stores $\Sigma_1, \dots, \Sigma_k$ such that:*

$$(\forall 1 \le i \le k. \langle P; \mathbf{halt};, \Sigma, \boldsymbol{\epsilon}\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Sigma_i, \Psi_i\rangle) \wedge \sum_{1 \le i \le k} \mu(\Psi_i) = 1$$

and

$$\forall i, j \in \mathbb{N}. i \neq j \rightarrow \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi_i\} \cap \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi_j\} = \emptyset.$$

Proof. The first part is a consequence of the totality of P, while the second is a consequence of Lemma 41 (Chapter 6): it shows that for each $P \in \mathsf{SIFP}_{\mathsf{LA}}$, $\Psi \in \mathbb{S}$ and for each store $\Sigma \in \{\mathtt{0}, \mathtt{1}\}^{Id}$, said $W := \{\Psi' \in \mathbb{S} | \langle P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle P, \Sigma, \Psi'\rangle\}$, then $\sum_{\Phi \in W} \mu(\Phi) = \mu(\Psi)$, which entails the claim we are aiming to. Finally, the disjointedness of the sets of oracles can be seen by induction on the length of the derivation of the relation $\rightsquigarrow^*_{\mathsf{LA}}$:

0 Trivial because the $\{\Psi_i\}_{i \in \mathbb{N}}$ sequence is composed by one only element.

$n+1$ The claim is a consequence of the IH and of the fact that the rules defining $\rightsquigarrow_{\mathsf{LA}}$ have only one possible target configuration for the majority of the operators, while the only rule which generates two different configurations is the one for the $\mathsf{RandBit}()$ statement. However, these configurations are tagged with $\Psi\mathtt{0}$ and $\Psi\mathtt{1}$ respectively, so the sets of the claim are respectively:

$$\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\} \cap \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\mathtt{0}\}$$

and

$$\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\} \cap \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\mathtt{1}\}$$

which are mutually disjoint, because $\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\mathtt{1}\}$ and $\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi\mathtt{0}\}$ are themselves disjoint.

□

A similar result for the $\mathsf{SIFP}_{\mathsf{RA}}$ language follows, as well.

Lemma 18 (Quantitative Correspondence for $\mathsf{SIFP}_{\mathsf{RA}}$)**.** *For every* total $\mathsf{SIFP}_{\mathsf{RA}}$ *program P, and for each store Σ there are a finite sequence of $\mathbb{SB}$-maps $\Psi_1 \dots \Psi_k$ and a finite sequence of stores $\Sigma_1, \dots, \Sigma_k$ such that:*

$$(\forall 1 \le i \le k. \langle P; \mathbf{halt};, \Sigma, \boldsymbol{\epsilon}\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \mathbf{halt};, \Sigma_i, \Psi_i\rangle) \wedge \sum_{1 \le i \le k} \mu(\Psi_i) = 1$$

and

$$\forall i, j \in \mathbb{N}. i \neq j \rightarrow \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi_i\} \cap \{\eta \in \mathbb{B}^{\mathbb{N}} | \eta \succ \Psi_j\} = \emptyset.$$

Proof. Analogous to the proof of Lemma 17. □

[16]For sake of readability, we do not show them in this section: they are given in Chapter 6, Section 6.2.7.

$$\begin{tikzcd}
\langle P_{\mathsf{RA}}; Q_{\mathsf{RA}}, \Sigma, \Psi\rangle \arrow[r, rightsquigarrow] \arrow[d, "\Theta"] & \langle Q_{\mathsf{RA}}, \Sigma', \Psi'\rangle \arrow[d, "\Theta"] \\
\langle P_{\mathsf{LA}}; Q_{\mathsf{LA}}, \Gamma, \Phi\rangle \arrow[r, rightsquigarrow, "*"] & \langle Q_{\mathsf{LA}}, \Gamma', \Phi'\rangle
\end{tikzcd}$$

Figure 3.2: Commutation schema between $\mathsf{SIFP}_{\mathsf{RA}}$ and $\mathsf{SIFP}_{\mathsf{LA}}$

3.3.3.2 The Θ Relation

We define the encoding from $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$ — actually, we only need the encoding of $\mathsf{Flip}()$ in $\mathsf{SIFP}_{\mathsf{RA}}$ — and prove that an appropriate simulation result holds with respect to the programs' step semantics. In other words, we want to show that there is a relation:

$$\Theta \subseteq \big(\mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \times (\mathsf{Id} \to \mathbb{S}) \times \mathcal{P}(\mathbb{B}^{\mathbb{S}})\big) \times \big(\mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}}) \times (\mathsf{Id} \to \mathbb{S}) \times \mathcal{P}(\mathbb{B}^{\mathbb{N}})\big)$$

associating to each triple $\langle P_{\mathsf{RA}}, \Sigma, \Psi\rangle$ other triples $\langle P_{\mathsf{LA}}, \Gamma, \Phi\rangle$ which weakly simulate the relation $\rightsquigarrow_{\mathsf{RA}}$ with respect to $\rightsquigarrow_{\mathsf{LA}}$. This is depicted by Figure 3.2. In particular, Θ is defined upon three co-operating functions or relations:

- A function
$$\alpha : \mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \longrightarrow \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}})$$
which maps the program $P_{\mathsf{RA}} \in \mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}})$ into its corresponding $P_{\mathsf{LA}} \in \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}})$ with respect to the simulation relation.
- A relation
$$\beta \subseteq ((\mathsf{Id} \to \mathbb{S}^*) \times \mathsf{SIFP}_{\mathsf{RA}} \times adt(M)) \times (\mathsf{Id} \to \mathbb{S}^*)$$
which is intended to capture the store-to-store relations between the two configurations. This is done by two conditions:
  - The two stores are identical if restricted to the domain of the left-hand-side store. The intuition between the relation is that the α function causes the employment of some additional registers by the $\mathsf{SIFP}_{\mathsf{LA}}$ program which should not be taken into account by the simulation relation.
  - The store on the right hand side must record the associative table in a specific register, which is determined by the structural syntax of the program in $\mathsf{SIFP}_{\mathsf{RA}}$.
- A function
$$\gamma : adt(M) \longrightarrow \mathbb{S}$$

which transforms the constraints on the oracle gathered by the relation $\rightsquigarrow_{\mathsf{RA}}$ to the information collected by the relation $\rightsquigarrow_{\mathsf{RA}}$.

This section will be organized as follows:

- First, we define the translation of the $\mathsf{Flip}(e)$ statement: the fl_k pseudo-procedures for $k \in \mathbb{N}$. These are the $\mathsf{SIFP}_{\mathsf{LA}}$ programs implementing the associative table and its access policies. In this way, the translation fo a $\mathsf{SIFP}_{\mathsf{RA}}$ program into $\mathsf{SIFP}_{\mathsf{LA}}$ i capable to simulate *random access to the oracle* within a formalism supporting a weaker form of access to randomness.
- Then, we define the functions α, β and γ employing fl_k for the definition of α;
- We combine these definitions in order to obtain Θ;
- We prove that Θ is a (weak) simulation relation with respect to $\rightsquigarrow_{\mathsf{RA}}$ and $\rightsquigarrow_{\mathsf{LA}}$.

The fl_k pseudo-procedures In order to define the pseudo-procedure which simulates the $\mathsf{Flip}(e)$ primitive, we will reuse some of the functions and the data-structures we defined for the reduction from $\mathcal{POR}$ to $\mathsf{SIFP}_{\mathsf{RA}}$. In particular, all the functions within the class $\mathcal{POR}^-$ can be defined without recurring to Q, so they are in $\mathsf{SIFP}_{\mathsf{LA}}$ as well. This is a consequence of Lemma 15 and means that for every $f \in \mathcal{POR}^-$, we can employ $\mathfrak{L}_f$ to define the translation of $\mathsf{Flip}(e)$. This is done in Corollary 7 below. After doing that, we show that it is always possible to avoid name clashes when defining *syntactical* compositions of programs, as done in Remark 6. Thus we introduce the translation of $\mathsf{Flip}(e)$ in Definition 92.

Corollary 7. *All the functions $f \in \mathcal{POR}$ defined in Section 3.2 apart from χ and apply are such that $\mathfrak{L}_f \in \mathsf{SIFP}_{\mathsf{LA}} \cap \mathsf{SIFP}_{\mathsf{RA}}$.*

Proof. Is is a consequence of the definitions of those functions and of Lemma 15: there, together with the correctness of $\mathfrak{L}_f$, we showed that if a function f is in $\mathcal{POR}^-$, then $\mathfrak{L}_f$ does not contain any $\mathsf{Flip}(e)$ expression. For this reason it is in $\mathsf{SIFP}_{\mathsf{LA}}$, too. □

To avoid name clashes, we define a procedure which renames the registers of a program whose name changes do not affect the program's semantics, i.e. all the registers apart from $\{X_i\}_{i\in\mathbb{N}}$ and the register R. This function has not been introduced before because the definition of the pseudo-procedure notation prevents itself name cashes requiring the α-conversions of the program's register names. Moreover, all the functions defined the throughout $\mathfrak{L}.$ operator were intended to respect the same invariants while, within the definition of the α function, we are plugging them in contexts where these properties are not necessarily respected. To sum up, in the definition of $\mathfrak{L}_f$, name captures were prevented by invariant properties which $\alpha(P)$ does not guarantee. So, we need to define a way to prevent that phenomenon. Finally, to define effectively the β relation, we need a way to distinguish the names of the registers which are used by both $P \in \mathsf{SIFP}_{\mathsf{RA}}$ and $\alpha(P) \in \mathsf{SIFP}_{\mathsf{LA}}$ from the registers used only for simulation purposes by the $\alpha(P)$ only; in particular, we need a way to identify the register storing the associative table. This is done introducing a sort of boundary between the indexes of the registers used by the two programs, which can be easily determined, as stated by Lemma 19. The register storing the associative table will be easily determined consequently.

Lemma 19. *For each program $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$, $\exists t \in \mathbb{N}.\forall k \geq t.Y_k, X_k, S_k \notin \#_r^{\mathsf{Stm}}(P)$.*

Proof. Let h be the maximum index (subscript) used by a register in $\#_r^{\mathsf{Stm}}(P)$. Let $t := h + 1$. It holds that $\forall k \geq t.Y_k, X_k, S_k \notin \#_r^{\mathsf{Stm}}(P)$. □

Thus, we introduce the program α-conversion notation

Notation 14 (Program α-conversion)**.** We denote P_n^α the program $P \in \mathbf{SIMP}$ in which all the name of the registers apart from the family X_i and R have been reassigned to an appropriate register S_j for $j \geq n$.

As expected, this conversion preserves the semantics of the program, as showed by the following Remark.

Remark 6.

$$\forall P \in \mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \cup \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}}).\forall k \in \mathbb{N}.k \geq \#_r^{\mathsf{Stm}}(P) + 1 \rightarrow [\![P_k^\alpha]\!] = [\![P]\!]$$

where $[\![\cdot]\!]$ *is the function in Definitions 79.*

Proof. By induction on the syntax of P and showing an analogous result for expressions, namely that expressions evaluate in the same way in the original store and in the α-converted store. $\square$

Broadly speaking, the simulation of the $\mathsf{Flip}(e)$ statement is defined by means of a sequence of pseudo-procedures. We cannot employ a single pseudo-procedure because the address of the register storing the value of the associative table cannot be fixed constantly for every $P \in \mathsf{SIFP}_{\mathsf{RA}}$, but it depends on the code of the starting program. This family of programs employs the encoding for finite functions defined in Section 6.1 to represent the associative table. Furthermore, this allows us to employ the $\mathsf{SIFP}_{\mathsf{LA}}$ translations of the data manipulators we defined in Section 6.1 for reduction from **SFP** to $\mathcal{POR}$. In particular, we employ the *sim* schema, which performs a lookup into the associative table in order to find a pair whose first element matches with the queried coordinate, and thus it returns the second element of that pair. Together with *sim*, we also reuse the *ral* primitive, which allows us to construct pairs and to edit the associative table adding, as last element a new associated pair.

The overall behavior of this function has already been described at the beginning of Section 3.3.3, but can be summarized as follows:

1. It checks whether the register containing the associative table is empty or not. If it is the case, it initializes it with an empty associative table.
2. It checks whether the associative table contains the queried coordinate or not. If it is not the case, it simulates the query extracting a new random bit by means of the $\mathsf{RandBit}()$ primitive which outputs either $\mathtt{b} = \mathtt{0}$ or $\mathtt{b} = \mathtt{1}$ and records it in the associative table.
3. It queries the associative table, returning the queried result.

Definition 92. The family of programs $\mathit{fl}_k()$ is defined as follows:

```
$\mathit{fl}_k$ :=while($Y_k \sqsubseteq \epsilon$){
        $Y_k \leftarrow \langle\rangle_{\mathbb{L}}^{0}$;
    }
    $Y_{k+1} \leftarrow X_1$;
    $Y_{k+2} \leftarrow X_2$;
    $X_1 \leftarrow Y_{k+3}$;
    $X_2 \leftarrow Y_k$;
    $\mathfrak{L}_{sim}{}^{\alpha}_{\max(|\#_r^{\mathsf{Stm}}(\mathfrak{L}_{sim})|+1,k)}$
```

```
$\mathbf{while}(R \sqsubseteq \epsilon)\{$
    $X_1 \leftarrow \langle\rangle^0_{\mathbb{L}};$
    $X_2 \leftarrow Y_{k+3};$
    $\mathfrak{L}_{ral}{}^{\alpha}_{\max(|\#^{\mathsf{Stm}}_r(\mathfrak{L}_{ral})|+1,k)}$
    $X_1 \leftarrow R;$
    $\mathsf{RandBit}();$
    $X_2 \leftarrow R;$
    $\mathfrak{L}_{ral}{}^{\alpha}_{\max(|\#^{\mathsf{Stm}}_r(\mathfrak{L}_{ral})|+1,k)}$
    $X_2 \leftarrow R;$
    $X_1 \leftarrow Y_k;$
    $\mathfrak{L}_{ral}{}^{\alpha}_{\max(|\#^{\mathsf{Stm}}_r(\mathfrak{L}_{ral})|+1,k)}$
    $Y_k \leftarrow R;$
$\}$
$X_1 \leftarrow Y_{k+3};$
$X_2 \leftarrow Y_k;$
$\mathfrak{L}_{sim}{}^{\alpha}_{\max(|\#^{\mathsf{Stm}}_r(\mathfrak{L}_{sim})|+1,k)}$
$X_1 \leftarrow Y_{k+1};$
$X_2 \leftarrow Y_{k+1};$
```

Basically, this piece of code implements an associative function, which is stored in the register Y_k. It uses exactly the same encoding we gave in Definition 127. This allows us to reuse the function *sim* we defined in $\mathcal{POR}$ to simulate finite functions — i.e. to search the associative table. If a certain coordinate is present in the function's domain, we are certain that the second **while**(){} will not be executed by the correctness of *sim* (Lemma 29) and by Lemma 15, so it ends returning the value which is associated to that coordinate in the table (the simulated value of $\omega(\Sigma(Y_{k+3}))$). Otherwise, the body of the cycle is executed once only and ends up adding an entry for the coordinate $\omega(\Sigma(Y_{k+3}))$ to the associative table.

The α function. The fl_k pseudo-procedures family covers the definition of the function α for the $\mathsf{Flip}(e)$ statements, all the other cases, the translation is identical to the source program.

Definition 93 (α Function)**.** The function $\alpha : \mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \longrightarrow \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}})$ is defined as an instance of α' which itself is a function $\mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}}) \times \mathbb{N} \longrightarrow \mathcal{L}(\mathsf{Stm}'_{\mathsf{LA}})$ defined by induction on the syntax of $\mathcal{L}(\mathsf{Stm}'_{\mathsf{RA}})$.

$$\alpha(P) := \alpha'(P, |\#^{\mathsf{Stm}}_r(P)| + 1).$$

$$\begin{aligned}
\alpha'(Id \leftarrow e, n) &:= Id \leftarrow e\\
\alpha'(\mathbf{halt};, n) &:= \mathbf{halt};\\
\alpha'(s;t) &:= \alpha(s,n); \alpha'(t,n)\\
\alpha'(\mathbf{while}(e)\{s\}, n) &:= \mathbf{while}(e)\{\alpha'(s,n)\}\\
\alpha'(\mathsf{Flip}(e), n) &:= Y_{n+3} \leftarrow e; \mathit{fl}_n.
\end{aligned}$$

Where $|\#^{\mathsf{Stm}}_r(P)| + 1$ is such that $Y_{|\#^{\mathsf{Stm}}_r(P)|+1} \notin \#^{\mathsf{Stm}}_r(P)$ is a consequence of Corollary 19. The family of programs[17] fl_n are defined in Definition 92.

As we discussed above, to define β, we pose two conditions on the store of $\alpha(P)$ for $P \in \mathsf{SIFP}_{\mathsf{RA}}$:

- It contains an associative table representing the simulated oracle.
- It is identical to the store of P with respect to its domain.

Definition 94 (β-Relation)**.** A triple $\langle \Sigma_1, P, (k_1, \mathtt{b}_1) :: \ldots :: (k_h, \mathtt{b}_h) :: \varepsilon\rangle$ is in β relation with a store Σ_2 if and only if the two conditions below hold:

- Σ_2 contains an associative table encoded as described in Section 6.1 in the register $Y_{|\#_r^{\mathsf{Stm}}(P)|+1}$, namely:
$$\Sigma_2(Y_{|\#_r^{\mathsf{Stm}}(P)|+1})) = \langle\langle k_h, \mathtt{b}_h\rangle^2_{\mathbb{L}}, \ldots, \langle k_1, \mathtt{b}_1\rangle^2_{\mathbb{L}}\rangle^k_{\mathbb{L}}.$$
- Σ_2 is identical to Σ_1 with respect to the registers which are used by the source program, namely:
$$\Sigma_2|_{dom(\Sigma_1)} = \Sigma_1.$$

If so, we write:
$$\beta(\langle \Sigma_1, P, (k_1, \mathtt{b}_1) :: \ldots :: (k_h, \mathtt{b}_h) :: \varepsilon\rangle, \Sigma_2).$$

Finally, the $\gamma : adt(M) \longrightarrow \mathbb{S}$ function relates the $\mathbb{SB}$-map to the corresponding string $\sigma \in \mathbb{S}$. This function is basically a translation between two different representations of the set of oracles which are coherent with a sequence of transitions.

Definition 95 (Function γ)**.** The function $\gamma : adt(M) \longrightarrow \mathbb{S}$ is defined as follows:

$$\gamma(\varepsilon) := \boldsymbol{\epsilon}$$
$$\gamma((k, \mathtt{b}) :: M) := \gamma(M)\mathtt{b}.$$

Notice that we are basically reversing the order of the elements within the list of pairs $(k, \mathtt{b})$ and the output string. This is not a mistake, because the $\rightsquigarrow_{\mathsf{RA}}$ semantics uses right-associative lists[18], so that new pairs are inserted on the left of the sequence. On the other hand, $\rightsquigarrow_{\mathsf{LA}}$ appends the bits on the right of the string thus, in this case, the new bit is on the right of the string.

The Θ Relation Combining α, β and γ we can impose enough constraints to define a weak simulation relation between two pairs of configurations for the $\rightsquigarrow_{\mathsf{LA}}$ and $\rightsquigarrow_{\mathsf{RA}}$ relations. As an intuition, if two configurations c, d are in Θ, then they are such that if c' reduces in one step to a configuration c', then also d reduces to another configuration d', possibly employing more transitions than c, which is the definition of *weak simulation*, [18].

Definition 96 (Θ Relation)**.** A triple $\langle P, \Sigma_1, \Psi\rangle$ is in $\langle Q, \Sigma_2, \Phi\rangle$ if and only if the following conditions hold:

- $\alpha'(P, k) = Q$ for some $k \geq |\#_r^{\mathsf{Stm}}(P)| + 1$ — for sake of readability, we will expressi this condition simply as $\alpha(P) = Q$;
- $\beta(\langle \Sigma_1, P, \Psi\rangle, \Sigma_2)$;
- $\gamma(\Psi) = \Phi$;
- $\mu(\Psi) = \mu(\Phi)$.

[17]Saying that fl_n are programs instead of pseudo-procedures, we mean that the names of the registers they use must remain the same after their inlining.

[18]Indeed, $adt(M)$ is a right associative list of $\mathbb{S} \times \mathbb{B}$ pairs.

3.3.3.3 Simulation result

It is now possible to establish that Θ is indeed a weak simulation relation.

Lemma 20 (Θ Weak Simulation)**.** *Θ is a* weak simulation relation *with respect to the small step operational semantics $\rightsquigarrow_{\mathsf{RA}}$ and $\rightsquigarrow_{\mathsf{LA}}$. Formally, if $\langle P; P', \Sigma_1, \Psi\rangle$ is Θ-related with $\langle Q; Q', \Sigma_2, \Phi\rangle$, then:*

$$\langle P; P', \Sigma_1, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle P', \Sigma_1', \Psi'\rangle \to \langle Q; Q', \Sigma_2, \Phi\rangle \rightsquigarrow_{\mathsf{LA}}{}^{*} \langle Q', \Sigma_2', \Phi'\rangle.$$

and $\langle P', \Sigma_1', \Psi'\rangle$ is Θ-related with $\langle Q', \Sigma_2', \Phi'\rangle$.

Proof. We proceed by cases on P.

$Id \leftarrow e$ By the construction of β (Definition 94) we know that the two triplets $\langle Id \leftarrow e; Q, \Sigma, \Psi\rangle$ and $\Theta(Id \leftarrow e; Q, \Sigma, \Psi)$ have stores which are identical on all the registers used by P (namely: $\in \#_r^{\mathsf{Stm}}(P)$). This entails that $Id \leftarrow e$ and $\alpha(Id \leftarrow e)$ both evaluate to the same assignment $[Id \leftarrow \sigma]$ because both the expressions evaluate to the same value (as a consequence of Lemma 42, Chapter 6). So, the ending stores are identical on $\#_r^{\mathsf{Stm}}(P)$. Moreover the value contained in $Y_{|\#_r^{\mathsf{Stm}}(P)|+1}$ is unchanged by the $\mathsf{SIFP_{LA}}$ program, so the condition on its value imposed by β still holds, to this end, notice that the source program cannot assign values to the register storing the associative table. Even Ψ and Φ do not change during these reduction steps, so the condition concerning their measure is verified in the target configurations, too.

while$(e)\{s\}$ By construction of β we know that any triple which is in Θ-relation with $\langle \mathbf{while}(e)\{s\}; P', \Sigma_1, \Psi\rangle$ works with a store Σ_2, which is identical to Σ with respect to all the registers used by P (namely: $\#_r^{\mathsf{Stm}}(P)$). This entails that e evaluates to the same expression σ_e in both P and $\alpha(P)$. This implies that if the guard of the cycle is `1` in P, it is `1` in $\alpha(P)$, too. We proceed by cases on this proposition.

– If it is true, then:

$$\langle \mathbf{while}(e)\{s\}; Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle s; \mathbf{while}(e)\{s\}; Q, \Sigma, \Psi\rangle$$

and

$$\begin{aligned}&\langle \alpha(\mathbf{while}(e)\{s\}; Q), \beta(\Sigma, \Psi), \gamma(\Psi)\rangle \rightsquigarrow_{\mathsf{LA}}\\ &\quad\langle \alpha(s); \mathbf{while}(e)\{\alpha(s)\}; \alpha(Q), \beta(\Sigma, \Psi), \gamma(\Psi)\rangle =\\ &\qquad\langle \alpha(s; \mathbf{while}(e)\{s\}; Q), \beta(\Sigma, \Psi), \gamma(\Psi)\rangle,\end{aligned}$$

which proves the claim.

– Conversely, if the proposition does not hold, then:

$$\langle \mathbf{while}(e)\{s\}; Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma, \Psi\rangle$$

and

$$\langle \alpha(\mathbf{while}(e)\{s\}; Q), \beta(\Sigma, \Psi), \gamma(\Psi)\rangle \rightsquigarrow_{\mathsf{LA}} \langle \alpha(Q), \beta(\Sigma, \Psi), \gamma(\Psi)\rangle,$$

which proves the claim. The condition on the identity of the measures comes form the fact that Ψ is not changed by the semantics of the **while**$(e)\{s\}$ construct.

$\cdot;\cdot$ The result comes from vacuity of the premise.

$\mathsf{Flip}(e)$ Suppose $\langle \mathsf{Flip}(e); P, \Sigma_1, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle P, \Sigma_1[R \leftarrow b], \Psi'\rangle$ and that such triplet is in Θ relation with

$$\langle Y_{|\#_r^{\mathsf{Stm}}(P)|+3} \leftarrow e; \mathit{fl}_{|\#_r^{\mathsf{Stm}}(P)|+1}; Q, \Sigma_2, \Phi\rangle.$$

Let $\langle e, \Sigma\rangle \rightharpoonup \sigma$. Moreover, suppose that $(\sigma, b) \in \Psi$ for $b \in \{0, 1\}$. It holds that $\langle e, \Sigma_2\rangle \rightharpoonup \sigma$ for definition of β (Definition 94) and Lemma 42. We can rewrite the premise applying the reduction rule described for the positive cylinder, obtaining:

$$\langle \mathsf{Flip}(e); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma[R \leftarrow b], \Psi\rangle.$$

It also holds that:

$$\langle Y_{|\#_r^{\mathsf{Stm}}(P)|+3} \leftarrow e; \mathit{fl}_{|\#_r^{\mathsf{Stm}}(P)|+1}; Q, \Sigma_2, \Phi\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle Q, \Gamma, \Phi\rangle.$$

By definition of β, we know that $\Sigma_2(Y_{|\#_r^{\mathsf{Stm}}(P)|+1})$ contains the encoding of an associative table T such that $\langle \sigma, b\rangle \in T$. In this case, $\alpha(\mathsf{Flip}(e))$ performs a lookup in T and returns the value it founds in the table, i.e. b. Doing so, $\alpha(\mathsf{Flip}(e))$ is completely transparent on the registers used by the program P, because it backups the values in X_1 and X_2 and because of the α-conversions of the pieces of code embedded in the definition: according to the definition of the names' conversion (Notation 14), all the names are reassigned apart from the X_is and R. Anyway, Lemma 15 ensures that X_is are accessed in read-only mode and that R contains the output of the function, which is b by correctness of *sim* (Lemma 29). Moreover all the other registers used by fl_k are above the range of registers used by P. For this reason we can say that Γ is equal to $\Sigma_2[R \leftarrow 1]$ in R, X_1, X_2 and in all the registers used by P and that $\Sigma_2(Y_{|\#_r^{\mathsf{Stm}}(P)|+1}) = \Gamma(Y_{|\#_r^{\mathsf{Stm}}(P)|+1})$. Even in this case the measure of Ψ and Φ are preserved by the transitions. Conversely, suppose that $(\sigma, 0) \notin \Psi \wedge (\sigma, 1) \notin \Psi$. It holds that:

$$\langle \mathsf{Flip}(e); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma[R \leftarrow 1], \Psi\rangle \wedge$$
$$\langle \mathsf{Flip}(e); Q, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{RA}} \langle Q, \Sigma[R \leftarrow 0], \Psi\rangle.$$

We will take in exam only the first case, the second is analogous. Suppose that Ψ is a generic $\mathbb{SB}$-map composed by k elements, then:

$$\langle \alpha(\mathsf{Flip}(e); Q), \Sigma_2, \gamma(\Psi)\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \alpha(Q), \Gamma, \gamma(\Psi)1\rangle.$$

It also holds that:

$$\langle \alpha(Q), \Gamma, \gamma(\Psi)1\rangle = \langle \alpha(Q), \Gamma, \gamma((\sigma, 1) :: \Psi))\rangle$$

For the definition of $\rightsquigarrow_{\mathsf{LA}}$ in the case of $\mathsf{RandBit}()$ and the definition of fl_k, the program puts 1 in R and adds a new entry in the associative table, as its rightmost element, so the relation β between the two stores still hold. We know that $\mu(\Psi) = \mu(\Phi)$. This, but we also know that $\mu((\sigma, b) :: \Psi) = \frac{1}{2}\mu(\Psi)$, because σ does not appear in Φ's keys. Similarly $\mu(\Phi b) = \frac{1}{2}\mu(\Phi)$. □

Given that Θ is a weak simulation with respect to the $\rightsquigarrow_{\mathsf{LA}}$ and $\rightsquigarrow_{\mathsf{RA}}$ relations, we can finally prove Lemma 16. Namely that:

$$\forall P \in \mathsf{SIFP}_{\mathsf{RA}} \exists Q \in \mathsf{SIFP}_{\mathsf{LA}}. \forall x, y \in \mathbb{S}. \mu(\{\omega \in \mathbb{O} | \llbracket P\rrbracket(x, \omega) = y\}) = \mu(\{\eta \in \{0, 1\}^{\mathbb{N}} | \llbracket Q\rrbracket(x, \eta) = y\}).$$

Proof of Lemma 16. We show only one of the two directions, the second is analogous to the first. It holds that the set

$$\{\omega \in \mathbb{O} | \llbracket P\rrbracket(x, \omega) = y\}$$

is equal to:

$$\bigcup\{\omega \in \mathbb{O} | \exists \Psi, \Gamma. \langle P; \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \mathbf{halt};, \Gamma, \Psi\rangle \wedge \omega \succ \Psi \wedge \Gamma(R) = y\}.$$

This is due to Characterization 2. Moreover, according to Lemma 18, we can suppose that all the instances of this union are pairwise disjoint. So, we can call this sequence of $\mathbb{SB}$-maps $\Psi_1, \dots, \Psi_k$ and observe that:

$$\forall 1 \leq i \leq k. \mu(\Psi_i) = \mu\left(\{\omega \in \mathbb{O} | \langle P; \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \mathbf{halt};, \Gamma, \Psi_i\rangle \wedge \omega \succ \Psi_i \wedge \Gamma(R) = y\}\right).$$

For this reason, the left hand side of the claim can be rewritten as follows:

$$\mu\left(\{\omega \in \mathbb{O} | [\![P]\!](x, \omega) = y\}\right) = \sum_{1 \leq i \leq k} \mu(\Psi_i).$$

Thanks to Lemma 16, we know that for every $\mathbb{SB}$-map Ψ and $x, y \in \mathbb{S}$,

$$\{\omega \in \mathbb{O} | \langle P; \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \mathbf{halt};, \Gamma, \Psi\rangle \wedge \omega \succ \Psi \wedge \Gamma(R) = y\}$$
$$\Downarrow$$
$$\{\omega \in \mathbb{O} | \langle \alpha(P); \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma, \gamma(\Psi)\rangle \wedge \omega \succ \gamma(\Psi) \wedge \Gamma(R) = y\}.$$

This means that there is a sequence of sets $\Phi_1, \dots, \Phi_k$ such that:

$$\forall 1 \leq i \leq k. \mu(\Psi_i) = \mu(\Phi_i).$$

The claim is a consequence of the following observations:

$$\begin{aligned}
\sum_{1 \leq i \leq k} \mu(\Phi_i) &= \sum_{1 \leq i \leq k} \mu(\{\eta \in \{0,1\}^{\mathbb{N}} | \exists \Phi_i. \langle \alpha(P); \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma, \gamma(\Phi_i)\rangle \wedge \eta \succ \gamma(\Phi_i) \wedge \Gamma(R) = y\}) \\
&= \mu(\bigcup_{1 \leq i \leq k} \{\eta \in \{0,1\}^{\mathbb{N}} | \exists \Phi_i. \langle \alpha(P); \mathbf{halt};, [][X_1 \leftarrow x], \varepsilon\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma, \gamma(\Phi_i)\rangle \wedge \eta \succ \gamma(\Phi_i) \wedge \Gamma(R) = y\}) \\
&= \mu(\{\eta \in \{0,1\}^{\mathbb{N}} | [\![\alpha(P)]\!](x, \eta) = y\}).
\end{aligned}$$

Again, this is a consequence of Characterization 1 and Lemma 17.

Finally, suppose that a P is poly-time, then $\alpha(P)$ is poly-time as well. Indeed all the statements different from $\mathsf{Flip}(e)$ can be simulated in a linear number of steps, while in order to establish the complexity of $\alpha(\mathsf{Flip}(e))$, that function has certainly a poly-time complexity in $|x|$: its complexity depends on the complexity of the family of programs fl_k, so it requires a constant number of steps plus the cost of the simulation of $\mathfrak{L}_{sim}$ and $\mathfrak{L}_{ral}$. Since these functions are in $\mathcal{POR}$, their time complexity is polynomial in the size of their inputs (consequence of Lemma 9), but this cost is itself polynomial because the associative table grows at most of a constant size each step. This concludes the proof. □

3.3.4 From $\mathsf{SIFP}_{\mathsf{LA}}$ to $\mathbf{SFP}_{\mathbf{OD}}$

We have shown that $\mathcal{POR}$ can be reduced to a formalism which does not support random access to the tape, namely $\mathsf{SIFP}_{\mathsf{LA}}$. In order to complete the proof of Lemma 9, we want to show that $\mathsf{SIFP}_{\mathsf{LA}}$ can be reduced to $\mathbf{SFP}_{\mathbf{OD}}$: the variant of **SFP** defined on a variation on Stream Machines which are capable to read characters from the oracle tape *on-demand*. Later, we show that $\mathbf{SFP}_{\mathbf{OD}}$ can be reduced to **SFP**, concluding the proof of the second part of Theorem 9. This section is structured as follows:

- First, we define the formalism of the on-demand stream machines, so to define $\mathbf{SFP}_{\mathbf{OD}}$;

- We prove that $\mathsf{SIFP}_{\mathsf{LA}}$ can be reduced to $\mathbf{SFP_{OD}}$.

As for ordinary Stream Machines (Definition 45), we limit our discussion to single-tape machines: which naturally scale to multi-tape ones.

Definition 97 (On-Demand Stream Machine)**.** An *On-Demand Stream Machine* is a quadruple $M := \langle \mathcal{Q}, \Sigma, \delta, q\rangle$, where:

- $\mathcal{Q}$ is a finite set of states ranged over by q_i and similar meta-variables.
- Σ is a finite set of characters ranged over by q_i and similar meta-variables.
- $\delta : \mathcal{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}, \natural\} \longrightarrow \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$.
- $q \in \mathcal{Q}$ is an initial state.

Moreover, we want δ such that $\forall c \in \hat{\Sigma}, q \in \mathcal{Q}$:

- If $\delta(q, c, \natural)$ is defined, then $\delta(q, c, \mathtt{0})$ and $\delta(q, c, \mathtt{1})$ are not.
- $\delta(q, c, \mathtt{0}) \neq \delta(q, c, \mathtt{1})$.

Notice that the only difference between Definition 97 and Definition 45 concerns the δ function: indeed, we introduced a new kind of transitions, labelled with $\natural$, which do not cause the oracle-tapeto advance. The configuration of *on-demand stream machines* can be represented in the same way of the configurations of ordinary stream machines, i.e. by means of a tuple which contains the current state and the configuration of the tapes. The main difference between these machines' transition function and ordinary STMs' one lies in the necessity for configurations which cause no advancements on the oracle-tape.

Definition 98 (On-Demand Stream Machine Transition Function)**.** Given an *on-demand* stream machine $M = \langle \mathcal{Q}, \Sigma, \delta, q\rangle$, we define the *partial transition function* $\vdash_\delta \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \{\mathtt{0}, \mathtt{1}\}^{\mathbb{N}} \longrightarrow \hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \{\mathtt{0}, \mathtt{1}\}^{\mathbb{N}}$ between two configurations of M_S as:

$$\begin{aligned}
\langle \sigma, q, c\tau, \eta\rangle &\vdash_\delta \langle \sigma c', q', \tau, \eta\rangle && \text{if } \delta(q, c, \natural) = \langle q', c', R\rangle\\
\langle \sigma c_0, q, c_1\tau, \eta\rangle &\vdash_\delta \langle \sigma, q', c_0 c_1'\tau, \eta\rangle && \text{if } \delta(q, c_1, \natural) = \langle q', c_1', L\rangle\\
\langle \sigma, q, c\tau, \mathtt{b}\eta\rangle &\vdash_\delta \langle \sigma c', q', \tau, \eta\rangle && \text{if } \delta(q, c, \mathtt{b}) = \langle q', c', R\rangle\\
\langle \sigma c_0, q, c_1\tau, \mathtt{b}\eta\rangle &\vdash_\delta \langle \sigma, q', c_0 c_1'\tau, \eta\rangle && \text{if } \delta(q, c_1, \mathtt{b}) = \langle q', c_1', L\rangle.
\end{aligned}$$

The family of *on-demand stream machine reachability functions* are defined exactly as in Definition 50, even the notion sof final configuration (Notation 8), function evaluated by a Stream Machine (Definition 51) and of polynomial Stream Machine (Definition 52) scale naturally to the corresponding ones for on-demand stream machines. Finally, we can define the class $\mathbf{SFP_{OD}}$.

Definition 99 ($\mathbf{SFP_{OD}}$)**.**

$$\mathbf{SFP} := \{f \in \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \mid \text{There is a poly-time canonical OD STM } M \text{ such that } f = f_M\}.$$

We will not present the reduction from $\mathsf{SIFP}_{\mathsf{LA}}$ to $\mathbf{SFP_{OD}}$ extensively because this kind of reductions are administrative and many similar reductions are already presented in the literature. For these reason we will only describe the *on-demand* stream machine which corresponds to the encoded $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{LA}})$.

Proposition 6. *For every $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{LA}})$ there is a $M_P \in \mathbf{SFP}$ such that for every $x \in \mathbb{S}$ and $\eta \in \mathbb{B}^{\mathbb{S}}$, $P(x, \eta) = P(x, \eta)$. Moreover, if P is poly-time, then M_P is poly-time.*

Proof. The construction relies on the fact that we can implement a $\mathsf{SIFP}_{\mathsf{LA}}$ program by means of a multi-tape *on-demand stream machine* machine which uses a tape to store the values of each register, plus an additional tape containing the partial results obtained during the evaluation of the expressions and another tape containing η. We will denote with e the tape used for storing the result coming from the evaluation of the expressions.

The machine works thanks to some invariant properties:

- On each tape the values are stored to the immediate right of the head.
- The result of the last expression evaluated is stored on the e tape to the immediate right of the head.

The value of a SIFP expression can be easily computed using the e tape. We show it by induction on the syntax of the expression:

- Each access to the value stored in a register basically consist in a copy of the content of the corresponding tape to the e tape, which is a simple operation, due to the invariants properties mentioned above.
- Concatenations ($f.\mathtt{0}$ and $f.\mathtt{1}$) are easily implemented by the addition of a character at the end of the e tape which contains the value of f, as stated by the induction hypothesis on the invariant properties.
- The binary expression are non-trivial, but since one of the two operands is a register identifier, the machine can directly compare e with the tape which corresponding to the identifier, and to replace the content of e with the result of the comparison, which in all cases $\mathtt{0}$ or $\mathtt{1}$.

All these operations can be implemented without consuming any character on the oracle tape and with linear time with respect to the size of the expression's value. To each statement s_i, we assign a sequence of machine states, $q^I_{s_i}, q^1_{s_i}, q^2_{s_i}, \dots, q^F_{s_i}$.

- Assignments consist in a copy of the value in e to the tape corresponding to the destination register and a deletion of the value on e by replacing its symbols with $\circledast$ characters. This can be implemented without consuming any character on the oracle tape.
- The sequencing operation $s; t$ can be implemented inserting in δ a composed transition from q^F_s to q^I_t, which does not consume the oracle tape.
- A **while**(){s}tatement $s := \mathbf{while}(f)\{t\}$ requires the evaluation of f and then passing to the evaluation of t, if $f \rightharpoonup \mathtt{1}$, or stepping to the next transition if it exists and $f \not\rightharpoonup \mathtt{1}$. After the evaluation of the body, the machine returns to the initial state of this statement, namely: q^I_s.
- A $\mathsf{RandBit}()$ statement is implemented consuming a character on the tape and copying its value on the tape which corresponds to the register R.

The following invariants hold at the beginning of the execution and are kept true throughout M_P's execution. In particular, if we assume P to be poly-time, after the simulation of each statement it holds that:

- The length of the non blank portion of the first tapes corresponding to the register is polynomially bounded because their contents are precisely the contents of P's registers, which are polynomially bounded as a consequence of the hypotheses on their polynomial time complexity.

- The head of all the tapes corresponding to the registers point to the leftmost symbol of the string thereby contained.

It is well-known that the reduction of the number of tapes on a poly-time Turing Machine comes with a polynomial overhead in time; for this reason, we can conclude that the poly-time *multi-tape* on-demand stream machine we introduced above can be *shrinked* to a poly-time *canonical* on-demand stream machine. This concludes the proof. □

As a trivial consequence of Proposition 6, we establish that each poly-time $\mathsf{SIFP}_{\mathsf{LA}}$ program P can be reduced to a poly-time *on demand stream machine.*

Corollary 8. *For each poly-time $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{LA}})$ there is a poly-time* on demand stream machine *machine M_P such that:*

$$\mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | [\![P]\!](x,\eta) = \tau\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | M_P(x,\eta) = \tau\}\right).$$

Proof. It is a consequence of Proposition 6, indeed:

$$\{\eta \in \mathbb{B}^{\mathbb{N}} | [\![P]\!](x,\eta) = \tau\} = \{\eta \in \mathbb{B}^{\mathbb{N}} | M_P(x,\eta) = \tau\}.$$

□

The main results of Sections 3.3.2 and 3.3.3 can be combined in the following statement:

Corollary 9. $\forall f \in \mathcal{POR}.\exists P \in$ *poly-time* $\mathsf{SIFP}_{\mathsf{RA}}$*:*

$$\forall x, y.\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | f(x,\omega) = y\}\right) = \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right).$$

Proof. The proof is a conjunction of Lemmas 15 and 40. □

As a consequence of Corollary 9 and Proposition 6, we prove that $\mathcal{POR}$ is directly related with $\mathbf{SFP_{OD}}$.

Corollary 10. *For each $f \in \mathcal{POR}$ there exists an $\mathbf{SFP_{OD}}$ machine M_f such that:*

$$\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | f(x,\eta) = \tau\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | M_P(x,\eta) = \tau\}\right).$$

Proof. From Corollary 9 we get that:

$$\forall x, y.\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | f(x,\omega) = y\}\right) = \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right)$$

for some poly-time $P \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$. Lemma 16 shows that:

$$\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | [\![P]\!](x,\omega) = y\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | [\![Q]\!](x,\eta) = y\}\right)$$

for some poly-time $Q \in \mathcal{L}(\mathsf{Stm}_{\mathsf{RA}})$. Finally, Corollary 8 states that:

$$\mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | [\![Q]\!](x,\eta) = \tau\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | M_P(x,\eta) = \tau\}\right).$$

Putting together all these equivalences, we get:

$$\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | f(x,\omega) = y\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | M_P(x,\eta) = \tau\}\right).$$

The introduction of the existential on M_P concludes the proof.

□

3.3.5 From $\mathbf{SFP_{OD}}$ to SFP

The aim of this section is to show that each on-demand stream machine can be reduced to an equivalent STM. This would allow us to conclude that the $\mathcal{POR}$ class of function is equivalent to the **SFP** formalism, as stated by Theorem 9. In particular, the result we are aiming to is formally stated by Lemma 22, completing the chain of reductions which connect the $\mathcal{POR}$ class of function to **SFP**.

Lemma 21. *For every $M = \langle \mathcal{Q}, \Sigma, \delta, q_0\rangle \in \mathbf{SFP_{OD}}$, the machine $N = \langle \mathcal{Q}, \Sigma, H(\delta), q_0\rangle \in \mathbf{SFP}$ is such that for every $n \in \mathbb{N}$, for every configuration of M $\langle \sigma, q, \tau, \eta\rangle$ and for every $\sigma', \tau' \in \mathbb{S}, q \in \mathcal{Q}$:*

$$\mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | \exists \eta'.\langle \sigma, q, \tau, \eta\rangle \triangleright^n_\delta \langle \sigma', q', \tau', \eta'\rangle\}\right) = \mu\left(\{\chi \in \mathbb{B}^{\mathbb{N}} | \exists \chi'.\langle \sigma, q, \tau, \xi\rangle \triangleright^n_{H(\delta)} \langle \sigma', q', \tau', \chi'\rangle\}\right).$$

Even in this case, the proof relies on a reduction. In particular, we show that given an on-demand stream machine M is is possible to build stream machine N which is equivalent to M. This is done by removing the $\natural$ transitions form the transition function δ and replacing them with ordinary transitions. The proof of the correctness of the reduction will follow the schema of the first part of Theorem 9, when we gave the encoding of STMs in $\mathcal{POR}$, but with some simplifications:

- We do not need to extend transition functions to total ones (in the sense of Lemma 4) because our encoding will preserve the number of steps used by the reduction and so it will not execute the machine for an over-approximated number of steps.
- The configuration of an on-demand stream machine and the configuration of an ordinary stream machine, are the same type of object.

This section is structured as follows:

- First, we introduce the encoding which turns a transition function for the on demand stream machine machine M into a transition function for an equivalent machine in **SFP**.
- Subsequently, we directly prove Lemma 22.

Intuitively, the encoding from an *on-demand* stream machine M to an ordinary stream machine takes the transition function δ of M and substitutes each transition not causing the oracle tape to shift — i.e. tagged with $\natural$ — with two distinct transitions, with respectively `0` and `1` instead of the symbol $\natural$. This causes the resulting machine to produce an identical transition but shifting the head on the oracle tape on the right.

Definition 100 (Encoding from On-Demand to Canonical Stream Machines)**.** We define the encoding from an On-Demand Stream Machine to a Canonical Stream Machine as below:

$$H := \langle \mathbb{Q}, \Sigma, \delta, q_0\rangle \mapsto \left\langle \mathbb{Q}, \Sigma, \bigcup \Delta_H(\delta), q_0\right\rangle.$$

where Δ_H is defined as follows:

$$\begin{aligned}
\Delta_H(\langle p, c_r, \mathtt{0}, q, c_w, d\rangle) &:= \{\langle p, c_r, \mathtt{0}, q, c_w, d\rangle\}\\
\Delta_H(\langle p, c_r, \mathtt{1}, q, c_w, d\rangle) &:= \{\langle p, c_r, \mathtt{1}, q, c_w, d\rangle\}\\
\Delta_H(\langle p, c_r, \natural, q, c_w, d\rangle) &:= \{\langle p, c_r, \mathtt{0}, q, c_w, d\rangle, \langle p, c_r, \mathtt{1}, q, c_w, d\rangle\}.
\end{aligned}$$

Lemma 22. *For every on-demant stream machine $M = \langle \mathcal{Q}, \Sigma, \delta, q_0\rangle$, the stram machine $N = \langle \mathcal{Q}, \Sigma, H(\delta), q_0\rangle$ is such that for every $n \in \mathbb{N}$, for every configuration of M $\langle \sigma, q, \tau, \eta\rangle$ and for every $\sigma', \tau' \in \mathbb{S}, q \in \mathcal{Q}$:*

$$\mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | \exists \eta'.\langle \sigma, q, \tau, \eta\rangle \triangleright_\delta^n \langle \sigma', q', \tau', \eta'\rangle\}\right) = \mu\Big(\{\chi \in \mathbb{B}^{\mathbb{N}} | \exists \chi'.\langle \sigma, q, \tau, \xi\rangle \triangleright_{H(\delta)}^n \langle \sigma', q', \tau', \chi'\rangle\}\Big).$$

Proof. The definition of $\triangleright_\delta^n$ (Definition 50) allows us to rewrite the statement:

$$\exists \eta'.\langle \sigma, q, \tau, \eta\rangle \triangleright_\delta^n \langle \sigma', q', \tau', \eta'\rangle$$

as

$$\begin{aligned}\exists \eta', \eta'' \in \mathbb{B}^{\mathbb{N}}.\exists c_1, \ldots, c_k.\langle \sigma, q, \tau, c_1c_2\ldots c_k\eta'\rangle \triangleright_\delta^{n_1} \langle \sigma_1, q_{i_1}, \tau_1, c_1c_2\ldots c_k\eta'\rangle \triangleright_\delta^1 \langle \sigma_1', q_{i_1}', \tau_1, c_2\ldots c_k\eta'\rangle \wedge\\ \langle \sigma_1', q_{i_1}', \tau_1, c_2\ldots c_k\eta'\rangle \triangleright_\delta^{n_2} \langle \sigma_2, q_{i_2}, \tau_2, c_2\ldots c_k\eta'\rangle \triangleright_\delta^1 \langle \sigma_2', q_{i_2}', \tau_2', c_3\ldots c_k\eta'\rangle \wedge\\ \langle \sigma_2', q_{i_2}', \tau_2', c_3\ldots c_k\eta'\rangle \triangleright_\delta^{n_3} \ldots \triangleright_\delta^{n_{k+1}} \langle \sigma', q', \tau', \eta'\rangle.\end{aligned}$$

According to Lemma 43 (Section 6.2.8) and Definitions 99 and 100, we can write:

$$\begin{aligned}&\exists \eta'', c_1, \ldots, c_k \in \mathbb{B}.\exists n_1, \ldots, n_{k+1} \in \mathbb{N}.\forall \xi_1, \ldots, \xi_{k+1} \in \mathbb{S}.|\xi_1| = n_1 \wedge \ldots |\xi_{k+1}| = n_{k+1} \wedge\\ &\langle \sigma, q, \tau, c_1c_2\ldots c_k\eta'\rangle \triangleright_\delta^{n_1} \langle \sigma_1, q_{i_1}, \tau_1, c_1c_2\ldots c_k\eta'\rangle \triangleright_\delta^1 \langle \sigma_1', q_{i_1}', \tau_1, c_2\ldots c_k\eta'\rangle \wedge\\ &\langle \sigma_1', q_{i_1}', \tau_1, c_2\ldots c_k\eta'\rangle \triangleright_\delta^{n_2} \langle \sigma_2, q_{i_2}, \tau_2, c_2\ldots c_k\eta'\rangle \triangleright_\delta^1 \langle \sigma_2', q_{i_2}', \tau_2', c_3\ldots c_k\eta'\rangle \wedge\\ &\langle \sigma_2', q_{i_2}', \tau_2', c_3\ldots c_k\eta'\rangle \triangleright_\delta^{n_3} \ldots \triangleright_\delta^{n_{k+1}} \langle \sigma', q', \tau', \eta'\rangle\\ &\iff\\ &\langle \sigma, q, \tau, \xi_1c_1\xi_2c_2\ldots c_k\xi_{k+1}\eta''\rangle \triangleright_{H(\delta)}^{n_1} \langle \sigma_1, q_{i_1}, \tau_1, c_1\xi_2c_2\ldots c_k\xi_{k+1}\eta''\rangle \triangleright_{H(\delta)}^1 \langle \sigma_1', q_{i_1}', \tau_1, \xi_2c_2\ldots c_k\xi_{k+1}\eta''\rangle \wedge\\ &\langle \sigma_1', q_{i_1}', \tau_1, \xi_2c_2\ldots c_k\xi_{k+1}\eta''\rangle \triangleright_{H(\delta)}^{n_2} \langle \sigma_2, q_{i_2}, \tau_2, c_2\ldots c_k\xi_{k+1}\eta''\rangle \triangleright_{H(\delta)}^1 \langle \sigma_2', q_{i_2}', \tau_2', \xi_3c_3\ldots c_k\xi_{k+1}\eta''\rangle \wedge\\ &\langle \sigma_2', q_{i_2}', \tau_2', \xi_3c_3\ldots c_k\eta''\rangle \triangleright_{H(\delta)}^{n_3} \ldots \triangleright_{H(\delta)}^{n_{k+1}} \langle \sigma', q', \tau', \eta''\rangle.\end{aligned}$$

Intuitively, this holds because it suffices to take the n_is as the length of longest sequence of non-shifting transitions of the on-demand sream machine. With this assumption, the conclusion follows as a consequence. Thus, we can express the sets of the claim as follows:

$$\begin{aligned}\{\eta \in \mathbb{B}^{\mathbb{N}} | \exists \eta'.\langle \sigma, q, \tau, \eta\rangle \triangleright_\delta^n \langle \sigma', q', \tau', \eta'\rangle\} &= \{\eta \in \mathbb{B}^{\mathbb{N}} | \forall 0 \le i \le k.\eta(i) = c_i\rangle\}\\ \{\chi \in \mathbb{B}^{\mathbb{N}} | \exists \chi'.\langle \sigma, q, \tau, \xi\rangle \triangleright_{H(\delta)}^n \langle \sigma', q', \tau', \chi'\rangle\} &= \{\chi \in \mathbb{B}^{\mathbb{N}} | \forall 1 \le i \le k.\chi(n_i + i) = c_i \wedge \chi(0) = c_1\rangle\}.\end{aligned}$$

The conclusion is trivial because both the set can be expressed as cylinders with the same measure. □

As a corollary, we get the result we are aiming to:

Corollary 11. *For every* $\mathbf{SFP_{OD}}$ *machine $M := \langle \mathcal{Q}, \Sigma, \delta, q\rangle$, $N := \langle \mathcal{Q}, \Sigma, H(\delta), q\rangle$ is such that:*

$$\forall x, y.\mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | M(x, \eta) = y\}\right) = \mu\left(\{\eta \in \mathbb{B}^{\mathbb{N}} | N(x, \eta) = y\}\right).$$

Proof. The result comes from the expansion of the definition of function computed by a Stream Machine 51. Then, from Definitions 99 and 100, we observe that a final configuration on M is a final configuration on N, too. Finally, applying Lemma 22 on the initial configurations of M and N (which are identical by definition), we get the result we are aiming to. □

3.3.6 Finalizing the proof

In this section we prove the second claim of Theorem 9, namely that for each $g \in \mathcal{POR}$, there is an $f \in \mathbf{SFP}$ such that for every $x, y \in \mathbb{S}$:

$$\mu\big(\{\omega \in \mathbb{O} \mid g(x,\omega) = y\}\big) = \mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} \mid f(x,\eta) = y\}\big).$$

The proof is a straightforward conjunction of the main results of the previous sections.

Proof of Theorem 9, Second Part. The claim is a consequence of Corollary 9, Lemma 16, Corollary 8, Corollary 11. ☐

3.4 From SFP to PPT

In this last section we show that the set of all the **SFP** functions and that of **PPT** function are tightly linked. In particular the measure of the set of oracles which link an input σ and an output τ throughout a computation of a **SFP** function M is equal to the probability that a certain **PPT** function computes τ on the same input σ, and vice-versa. To do so, we give a formal definition of the **PPT** functions leveraging the formalism of the Probabilistic Turing Machines (already described in Definiton 15). On top of it, we define a semantic association a function to each machine. Finally, **PPT** is basically defined as the set of functions computed by a Probabilistic Turing Machine in a polynomial number of steps. After that, we show that there is a bijection between the class of Stream machines and PTM with the property we are aiming to.

Definition 101 (Probabilistic Turing Machine)**.** A Probabilistic Turing Machine (PTM) $M := \langle \mathcal{Q}, \Sigma, \delta_0, \delta_1, q_0 \rangle$ is a quintuple where:

- $\mathcal{Q}$ is a finite set of states ranged over by q_i and similar meta-variables.
- q_0 is the initial state.
- Σ is a finite set of characters ranged over by c_i and similar meta-variables.
- $\delta_0 : \mathcal{Q} \times \hat{\Sigma} \longrightarrow \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ is a transition function which describes the new configuration reached by a Stream Machine. L, R are two fixed and distinct symbols, and $\hat{\Sigma} = \Sigma \cup \{\circledast\} \wedge \circledast \neq \mathtt{0} \wedge \circledast \neq \mathtt{1}$.
- $\delta_1 : \mathcal{Q} \times \hat{\Sigma} \times \longrightarrow \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ is another transition function.

As for **SFP**, we discuss only *canonical* PTMs, winch use $\{\mathtt{0}, \mathtt{1}\}$ as alphabet and are single-taped.

Definition 102 (Canonical PTM)**.** A Canonical PTM is a single-tape PTM in which:

- $\Sigma = \{\mathtt{0}, \mathtt{1}\}$;
- $L = \mathtt{0}, R = \mathtt{1}$.

Differently from STMs, the function computed by a *canonical* PTM does not map an input in $\mathbb{S}$, and an oracle in $\mathbb{B}^{\mathbb{N}}$ to another input, but maps a string in $\mathbb{S}$ to a probability distribution on the set $\mathbb{S}$. This is why we cannot prove that $\mathbf{PPT} = \mathbf{SFP}$, but we will prove a different form of equivalence between the two classes. The definition of function computed by a PTM passes through the notion of configuration and a random variables associated to the machine. Configurations are defined as follows:

Definition 103 (Configuration of a Probabilistic Turing Machine)**.** The *configuration of a PTM* M is a triple $\{\sigma, q, \tau\}$ where:

- $\sigma \in \hat{\Sigma}^*$ is the portion of the first tape on the left of the head.
- $q \in \mathcal{Q}$ is the current state of M.
- $\tau \in \hat{\Sigma}^*$ is the portion of the first tape on the right of the head.

As we established above, a PTM computes *random variables*, instead of simple values. In order to give a formal definition of the random variable computed by a PTM, we must take in account a probability space of all the possible sequences of choices made by a PTM. This is a triple $(\Omega, \mathcal{F}, P)$. A suitable field was introduced in Definition 35 for every countable set A, in order to define a measure for the σ-algebra $\mathbb{O}$. Together with this field, it was even introduced a class of measure functions (Definition 36). Thus, we can define the PTM's Probability Space as follows:

Definition 104 (PTM Probably Space)**.** We call $\mathscr{C}_{\mathbb{S}}$ the probability space defined as follows:

$$\mathscr{C}_{\mathbb{S}} := \langle \mathbb{B}^{\mathbb{S}}, \mathcal{F}_{\mathbb{S}}, \mu_{\mathbb{S}} \rangle$$

where $\mathcal{F}_{\mathbb{S}}$ is the field of cylinders obtained instantiating Definition 35 with $A = \mathbb{S}$ and $\mu_{\mathbb{S}}$ is the measure function induced on this set by Definition 36.

Definition 105 (Sequence of Random Variables associated to a Probabilistic Turing Machine)**.** Given a PTM M, a configuration $\{\sigma, q, \tau\}$ and the probability space $\mathscr{C}_{\mathbb{S}}$ for $\mathbb{B}^{\mathbb{N}}$ (Definition 35), we define the following *sequence of random variables*:

$$\forall \eta \in \mathbb{B}^{\mathbb{N}}.X^{\{\sigma,q,\tau\}}_{M,0} := \eta \mapsto \{\sigma, q, \tau\}$$

$$\forall \eta \in \mathbb{B}^{\mathbb{N}}.X^{\{\sigma,q,\tau\}}_{M,n+1} := \eta \mapsto \begin{cases} \delta_0(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) & \text{if } \eta(n) = 0 \land \exists\langle \sigma', q'\tau'\rangle.\delta_0(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) = \langle \sigma', q', \tau'\rangle \\ X^{\{\sigma,q,\tau\}}_{M,n}(\eta) & \text{if } \eta(n) = 0 \land \neg\exists\langle \sigma', q'\tau'\rangle.\delta_0(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) = \langle \sigma', q', \tau'\rangle \\ \delta_1(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) & \text{if } \eta(n) = 1 \land \exists\langle \sigma', q'\tau'\rangle.\delta_1(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) = \langle \sigma', q', \tau'\rangle \\ X^{\{\sigma,q,\tau\}}_{M,n}(\eta) & \text{if } \eta(n) = 1 \land \neg\exists\langle \sigma', q'\tau'\rangle.\delta_1(X^{\{\sigma,q,\tau\}}_{M,n}(\eta)) = \langle \sigma', q', \tau'\rangle. \end{cases}$$

Intuitively the variable $X^{\{\sigma,q,\tau\}}_{M,n}$ describes the configuration reached by the machine after exactly n transitions. As we did for the Stream Machines, we need do give a formal definition of random variable computed by a PTM. To do so, we define some sort of *final* random variable.

Definition 106 (Final configuration of a PTM)**.** We say that a PTM M has final configuration X_t if and only if $\forall t' > t.X^{\{\sigma,q,\tau\}}_{M,t'} = X^{\{\sigma,q,\tau\}}_{M,t}$.

We are interested in finding a characterization the **PPT** class of functions. For this reason, we define the notion of *poly-time* PTM.

Definition 107 (Poly-time Probabilistic Turing Machine)**.** We say that a PTM M is polynomial in time if and only if

$$\exists p \in POLY.\forall \sigma.X^{\langle \epsilon, q_0, \sigma\rangle}_{M,p(\sigma)} \text{ is final.}$$

Definition 108 (Random Variable computed by a Probabilistic Turing Machine)**.** We say that a PTM M computes $Y_{M,\sigma}$ if and only if $\exists t \in \mathbb{N}.\forall \sigma.X^{\langle \sigma, q_0, \tau\rangle}_{M,t}$ is final. In such case $Y_{M,\sigma}$ is the longest suffix of $\pi_1(X^{\langle \sigma, q_0, \epsilon\rangle}_{M,t})$, which does not contain $\circledast$.

On top of the definition above, we can build up a formal definition of **PPT**:

Definition 109 (**PPT** class)**.** **PPT** is the class of distributions which is computable by a poly-time PTM.

It is easy to see that the class of the Polynomial Probabilistic Turing Machine has a strong correspondence with the class of **SFP**: the former hard-codes the randomness on a specific tape, while the latter encodes its as a behavior. It is proved formally by the following proposition:

Proposition 7 (Polynomial PTM and **SFP** equivalence)**.** *For each poly-time STM N, there is a poly-time PTM M, machine N such that:*

$$\forall \sigma, \tau. \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | N(\sigma, \eta) = \tau\}) = Pr[Y_{M,\sigma} = \tau]$$

and vice-versa.

Proof. We observe that, as stated in Definition 108, the random variable $Y_{\cdot,\cdot}$ is equal to the random variable $X^{\langle\sigma,q_0,\epsilon\rangle}_{M,t}$ for some $t \in \mathbb{N}$. According to Definition 105, the random variable $X^{\langle\sigma,q_0,\epsilon\rangle}_{M,t}$ is defined on the σ-algebra $\mathscr{C}_{\mathbb{S}}$, using exactly $\mu_{\mathbb{S}}$ as probability measure. For these reasons, the claim can be restated as follows:

$$\forall \sigma, \tau. \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | N(\sigma, \eta) = \tau\}) = \mu(Y^{-1}_{M,\sigma}(\tau))$$
$$\forall \sigma, \tau. \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | N(\sigma, \eta) = \tau\}) = \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | Y_{M,\sigma}(\eta) = \tau\}).$$

In order to prove what we stated above, we will show a stronger result: namely that there is bijection I : STMs $\longrightarrow$ PTM such that:

$$\forall n \in \mathbb{N}. \{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright^n_\delta \langle \tau, q, \psi, n\rangle\} = \{\eta \in \mathbb{B}^{\mathbb{N}} | X^{\langle\epsilon,q_0,\sigma\rangle}_{I(N),n}(\eta) = \langle \tau, q, \psi\rangle\} \tag{3.1}$$

which entails:

$$\{\eta \in \mathbb{B}^{\mathbb{N}} | N(\sigma, \eta) = \tau\} = \{\eta \in \mathbb{B}^{\mathbb{N}} | Y_{I(N),\sigma}(\eta) = \tau\}. \tag{3.2}$$

For this reason, it suffices to show I and prove that 3.1 holds. I is defined splitting the transition function of N in two transition functions. A transition from δ is assigned to δ_0 if it matches the `0` character on the oracle-tape, otherwise it is assigned to δ_1. I can be defined as follows:

$$I \coloneqq \langle \mathcal{Q}, \Sigma, \delta, q_0\rangle \mapsto \langle \mathcal{Q}, \Sigma, \Delta_0(\delta), \Delta_1(\delta), q_0\rangle$$
$$\Delta_i(\delta) \coloneqq \{\langle p, c_r, i, q, c_w, d\rangle \in \delta\}.$$

I is bijective. Indeed, its inverse is a function, too. I^{-1} takes two transition functions (ideally the two transition functions of M) and joins them in a single transition function which behaves as δ_0 if the tape has `0` under the head, and as δ_1 otherwise. Now we prove the 3.1 by induction on the number of steps required by N to compute its output value.

0 In this case we know that:

$$\{\eta \langle \sigma, q_0, \tau, \eta\rangle \triangleright^0_\delta \langle \epsilon, q_0, \sigma, \eta\rangle\} = \mathbb{B}^{\mathbb{N}} = \{\eta \in \mathbb{B}^{\mathbb{N}} | X^{\langle\epsilon,q_0,\sigma\rangle}_{I(N),0}(\eta) = \langle \epsilon, q, \sigma\rangle\},$$

which proves the thesis.

$n+1$ In this case we must show that:

$$\{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright_{\delta}^{n+1} \langle \tau, q, \psi, \eta'\rangle\} = \{\eta \in \mathbb{B}^{\mathbb{N}} | X_{I(N),n+1}^{\langle \epsilon, q_0, \sigma\rangle}(\eta) = \langle \tau, q, \psi\rangle\},$$

which proves the claim. We also know that:

$$\forall m \leq n.\{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright_{\delta}^{m} \langle \tau, q, \psi, \eta''\rangle\} = \{\eta \in \mathbb{B}^{\mathbb{N}} | X_{I(N),m}^{\langle \epsilon, q_0, \sigma\rangle}(\eta) = \langle \tau, q, \psi\rangle\},$$

but it is easy to show that $\{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright_{\delta}^{n+1} \langle \tau, q, \psi, \eta'\rangle\} =$

$$\begin{gathered}\{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright_{\delta}^{n} \langle \tau', q', \psi', \mathtt{0}\eta'\rangle \vdash_{\delta} \langle \tau, q, \psi, \eta'\rangle\} \\ \cup \\ \{\eta \in \mathbb{B}^{\mathbb{N}} | \langle \sigma, q_0, \tau, \eta\rangle \triangleright_{\delta}^{n} \langle \tau', q', \psi', \mathtt{1}\eta'\rangle \vdash_{\delta} \langle \tau, q, \psi, \eta'\rangle\}.\end{gathered} \tag{3.3}$$

Concerning $\{\eta \in \mathbb{B}^{\mathbb{N}} | X_{I(N),n+1}^{\langle \epsilon, q_0, \sigma\rangle}(\eta) = \langle \tau, q, \psi\rangle\}$, it is equal to

$$\begin{gathered}\{\eta \in \mathbb{B}^{\mathbb{N}} | X_{I(N),n}^{\langle \epsilon, q_0, \sigma\rangle}(\eta) = \langle \tau', q', \psi'\rangle \wedge \eta(n) = 0 \wedge \Delta_0(\delta)(\langle \tau', q', \psi'\rangle) = \langle \tau, q, \psi\rangle\} \\ \cup \\ \{\eta \in \mathbb{B}^{\mathbb{N}} | X_{I(N),n}^{\langle \epsilon, q_0, \sigma\rangle}(\eta) = \langle \tau', q', \psi'\rangle \wedge \eta(n) = 1 \wedge \Delta_1(\delta)(\langle \tau', q', \psi'\rangle) = \langle \tau, q, \psi\rangle\}.\end{gathered} \tag{3.4}$$

It is easy to see that the sets in (3.3) and (3.4) are pairwise equal thanks to the IH and the definition of I. Claim (3.2) is a consequence of the definition if Y. Both the machines require the same number of steps. For this reason, if the first is poly-time also the second one is so. The opposite direction comes from the fact that I is a bijection.

□

This result completes the chain of reductions from $\mathcal{POR}$ to **PPT** and vice-versa.

3.4.1 Main Result

In the previous sections, we have described all the reductions encoding **SFP** in $\mathcal{POR}$ and vice-versa, so we can finally state and prove the main result of this chapter. As we established before, the equivalence relation we are proving, is not a strict identity between two classes of functions for many reasons: some of the formalisms are intrinsically dis-homogeneous, for example $\mathcal{POR}$ functions map strings $\sigma \in \mathbb{S}$ and an oracle function in $\mathbb{O}$, while, for instance **PPT** maps strings in probability distributions, which means that these sets are disjoint. However, the results we showed in the previous sections can be used to prove the main result of this chapter: the statement of Conjecture 1.[19]

Theorem 10. *It holds that:*

- $\forall f \in \mathbf{PPT}.\exists g_f \in \mathcal{POR}.Pr[f(x) = y] = \mu(\{x \in \{\mathtt{0}, \mathtt{1}\}^{\mathbb{S}} | g_f(x, \omega) = y\})$;
- $\forall g \in \mathcal{POR}.\exists f_g \in \mathbf{PPT}.\mu(\{x \in \{\mathtt{0}, \mathtt{1}\}^{\mathbb{S}} | g(x, \omega) = y\}) = Pr[f_g(x) = y]$.

Proof. Consequence of Theorem 9 and Proposition 7. □

Furthermore, joining this result with Theorem 3, we get the **PPT** representability result we were aiming to:

[19]Since we showed that Conjecture 1 actually holds, we restate it as a theorem, namely Theorem 10.

Theorem 4 (Σ^b_1-Representability of **PPT** Functions)**.** *It holds that:*

$$\forall G \in \Sigma^b_1.RS^1_2 \vdash \forall x \exists! y.G(x,y) \to \exists f_G \in \mathbf{PPT}.\forall x, y \in \mathbb{S}.\mu\left(\llbracket G(x,y) \rrbracket\right) = Pr[f_G(x) = y]$$

and that:

$$\forall f \in \mathbf{PPT}.\exists G_f \in \Sigma^b_1..RS^1_2 \vdash \forall x \exists! y.G(x,y) \wedge \forall x, y \in \mathbb{S}.\mu\left(\llbracket G(x,y) \rrbracket\right) = Pr[f_G(x) = y].$$

Proof. Consequence of Theorems 3 and 10. □

Chapter 4

On the equivalence between FP and Cobham's Algebra

In the literature, it is widely agreed that Cobham's function algebra *et simila* — e.g. Ferreira's PTCF — are sound and complete with respect to the polynomial time computable functions **FP**. Unfortunately, as far as we know, there is no proof of such result which is clear, detailed, self-contained and easily available. For this reason, we decided to give, as a corollary of our results, a proof of that fact. Even in this case, we show the two inclusions separately. We will first address the inclusion of **FP** in a Cobham style function algebra in Section 4.2, then, in Section 4.3, we show that even the opposite direction holds. To do so, we employ the definitions of Turing Machines given in Chapter 1 to show that they can be reduced FSTMs, then we give a definition of a Cobham-Style function algebra $\mathcal{F}_{\mathsf{Cob}}$ and prove it is equivalent to $\mathcal{POR}^{-}$, then leveraging some results we gave in Chapter 3, we conclude that the the class of **FP** functions is equivalent to $\mathcal{F}_{\mathsf{Cob}}$.

4.1 The class $\mathcal{F}_{\mathsf{Cob}}$

Following [14] and [9], we introduce the class $\mathcal{F}_{\mathsf{Cob}}$ which will be proved to precisely characterize string-functions computable in polynomial time.

Definition 110 (The Class $\mathcal{F}_{\mathsf{Cob}}$)**.** The class $\mathcal{F}_{\mathsf{Cob}} : \mathbb{S}^n \longrightarrow \mathbb{S}$ is the smallest class containing initial functions:

- $E_{\mathcal{D}}(x) = \emptyset$;
- $P^i_{\mathcal{D}}(x_1, \dots, x_n) = x_i$, for $1 \leq i \leq n$;
- $C^{\mathsf{0}}_{\mathcal{D}}(x) = x\mathsf{0}$ and $C^{\mathsf{1}}_{\mathcal{D}}(x) = x\mathsf{1}$;
- $Q_{\mathcal{D}}(x, y) = \mathsf{1}$ if and only if $x \subseteq y$, $Q_{\mathcal{D}}(x, y) = \mathsf{0}$ otherwise;

and closed under composition and bounded recursion, where a function f is defined by bounded recursion from g, h_0, h_1 as:

$$
\begin{aligned}
f(x_1, \dots, x_n, \boldsymbol{\epsilon}) &= g(x_1, \dots, x_n);\\
f(x_1, \dots, x_n, y\mathsf{0}) &= h_0(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))|_{t(x_1, \dots, x_n, y)};\\
f(x_1, \dots, x_n, y\mathsf{1}) &= h_1(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))|_{t(x_1, \dots, x_n, y)};
\end{aligned}
$$

with t term of $\mathcal{L}$.

The class $\mathcal{F}_{\mathsf{Cob}}$ is equi-expressive with respect to the class $\mathcal{POR}^-$: indeed, even if the former class takes in input an additional functional argument, its functions cannot access it by means of the $\mathcal{POR}$'s querying function Q.

Indeed, the $\mathcal{POR}^-$ has been defined only to decouple the combinatorial concerns of the reduction from **SFP** to $\mathcal{POR}$ from the measure-theoretic ones. The $\mathcal{POR}^-$ class by itself, has not got any concrete reason to be defined in that peculiar way. Thus, the definition of the $\mathcal{F}_{\mathsf{Cob}}$ class is meant to remove from the definition of $\mathcal{POR}^-$ the superfluous right-most argument which does not affect at all the result of the computation. As a consequence, it is easy to show equivalence of $\mathcal{POR}^-$ with the class $\mathcal{F}_{\mathsf{Cob}}$ we have defined above.

First, we show that each function in $\mathcal{F}_{\mathsf{Cob}}$ can be represented by a function in $\mathcal{POR}^-$ which behaves exactly like it, independently form the value of its oracle ω.

Remark 7 ($\mathcal{F}_{\mathsf{Cob}}$ Soundness)**.** $\forall f \in \mathcal{F}_{\mathsf{Cob}}.\exists g_f \in \mathcal{POR}^-.\forall \omega, \vec{x}.f(\vec{x}) = g_f(\vec{x}, \omega)$.

Proof. By induction on the proof that a function is in Ferreira's $\mathcal{F}_{\mathsf{Cob}}$.

- If the function is $E_{\mathcal{D}}$, E respects the property.
- If the function is $P^n_j{}_{\mathcal{D}}$, the correspondent $\mathcal{POR}$ projector respects the claim.
- If the function is $C^{\mathsf{b}}{}_{\mathcal{D}}$, S_{b} verifies the claim.
- $Q_{\mathcal{D}}$ can be encoded by means of bounded recursion on notation, as :

$$\begin{aligned} \mathit{pref}(x, \epsilon, \omega) &:= \mathit{eq}(x, \mathtt{0}, \omega); \\ \mathit{pref}(x, y\mathtt{b}, \omega) &:= \mathtt{if}(\mathtt{1}, \mathit{pref}(x, y, \omega), \mathit{eq}(x, y\mathtt{b}, \omega))|_{\mathtt{1}} \end{aligned}$$

  where the functions $\mathtt{if}, \mathit{eq}$ are in $\mathcal{POR}^-$. The definitions can found in Section 6.1.2. The proof of the correctness can be done by induction of y.

  - ϵ In this case *pref* returns $\mathtt{1}$ if and only if $x = y = \epsilon$.
  - $\tau\mathtt{b}$ Suppose that x is a prefix of $\tau\mathtt{b}$, then they are equal or x is a strong prefix of $\tau\mathtt{b}$. In the first case, the functions returns $\mathtt{1}$ as a consequence of the correctness of $\mathtt{if}$ and *eq*, proven in Section 6.1.2. Otherwise x is a prefix of y, thus the result is a consequence oh the IH. Suppose that x is not a prefix of y, they cannot be equal, so the result comes from the IH.

- The composition and recursion on notation cases follow by IH.

□

Finally we state that each function $f \in \mathcal{POR}^-$ can be represented by means of a function $\mathcal{F}_{\mathsf{Cob}}$ which, without taking any functional argument, behaves exactly like f does.

Remark 8 ($\mathcal{F}_{\mathsf{Cob}}$ Completeness)**.** $\forall g \in \mathcal{POR}^-.\exists f_g \in \mathcal{F}_{\mathsf{Cob}}.\forall \omega, \vec{x}.g(\vec{x}, \omega) = f_g(\vec{x})$.

Proof. By induction on the proof that $g \in \mathcal{POR}$

- If the function is E, $E_{\mathcal{D}}$ is a witness for the existential.
- If the function is P^n_j , the correspondent $\mathcal{F}_{\mathsf{Cob}}$ projector has the claimed property.
- If the function is S_{b}, $C^{\mathsf{b}}{}_{\mathcal{D}}$ verifies the claim.

- $\mathcal{POR}^-$'s C can be encoded by means of bounded recursion on notation, taking:

$$C(\boldsymbol{\epsilon}, z_\epsilon, z_0, z_1, \omega) := z_\epsilon;$$
$$C(x\mathsf{b}, z_\epsilon, z_0, z_1, \omega) := z_b|_{z_0 z_1}.$$

- The composition and recursion on notation cases follow by IH.

□

4.2 From FP to a Cobham style Function Algebra

In this section, we show that each $f \in \mathbf{FP}$ function is in $\mathcal{F}_{\mathsf{Cob}}$ too. To do so, we start by reducing TMs to FSTM. The reduction is purely combinatorial: intuitively, we extend the transition function of the starting TM to match all the possible characters of the secondary tape. Thus we get a FSTM which behaves exactly like the starting TM. Then Lemma 3 shows that there is a function g in $\mathcal{POR}^-$ such that $\forall x, y \in \mathbb{S}.\forall\omega \in \mathbb{O}.f(x, y) = g(x, y, \omega)$. Finally by Remark 8, we have that there is a function g' in $\mathcal{F}_{\mathsf{Cob}}$ such that $\forall\omega.g(x, y, \omega) = g(x, y)$. This entails that $f = g$ and, thus, that $\mathbf{FP} \subseteq \mathcal{F}_{\mathsf{Cob}}$.

Lemma 23. *Each ordinary poly-time computable function f there is a function in $\mathcal{PTF}$ g such that:*

$$\forall x_1, x_2.f(x_1) = g(x_1, x_2).$$

Proof. Given the (canonical) Turing Machines which computes f, $M_f = \langle \mathcal{Q}, q_0, \Sigma, \delta\rangle$, we can define the Finite Stream Turing Machine which computes g as $N = \langle \mathcal{Q}, q_0, \Sigma, \Delta(\delta)\rangle$ where the functional Δ extends all the transitions of the original machine in order to match each of the characters on the secondary tape. Formally:

$$\Delta(\delta) := \bigcup\{\{\langle p, c_r, \mathsf{0}, q, c_w, d\rangle, \langle p, c_r, \mathsf{1}, q, c_w, d\rangle, \langle p, c_r, \circledast, q, c_w, d\rangle | \langle p, c_r, q, c_w, d\rangle \in \delta\}.$$

Proceeding by induction on the steps of the ordinary Turing Machine that the Finite Stream Turing Machine N (restricted to its work tape), behaves exactly as the Turing Machine M, it si possible to show that N and M produce the same output configuration with respect to their work tapes. □

Corollary 12. $\mathbf{FP} \subseteq \mathcal{F}_{\mathsf{Cob}}$.

Proof. Suppose that $f \in \mathbf{FP}$. We must show that $f \in \mathcal{F}_{\mathsf{Cob}}$, too. Lemma 23 shows that there is a $f' \in \mathcal{PTF}$ such that

$$\forall x_1, x_2 \in \mathbb{S}.f(x_1) = f'(x_1, x_2).$$

From Lemma 3, we know that there is a function $g \in \mathcal{POR}^-$ such that:

$$\forall x_1, x_2 \in \mathbb{S}.\forall\omega \in \mathbb{O}.f'(x_1, x_2) = g(x_1, x_2, \omega).$$

Consequently, from Remark 8, we observe that $\exists g' \in \mathcal{F}_{\mathsf{Cob}}$ such that:

$$\forall x_1, x_2 \in \mathbb{S}\forall\omega \in \mathbb{O}.g(x_1, x_2, \omega) = g'(x_1, x_2).$$

Joining together these results, and choosing some values for ω, e.g. $\omega = \lambda x.\mathsf{0}$, we obtain that:

$$\forall x_1, x_2 \in \mathbb{S}.f(x) = g'(x_1, x_2).$$

Take in exam the following function:

$$g''(x) := g'(x, \boldsymbol{\epsilon}),$$

it is trivial that $g'' \in \mathcal{F}_{\mathsf{Cob}}$ because such class is closed under composition. This concludes the proof, indeed we have that $f = g''$ and $g \in \mathcal{F}_{\mathsf{Cob}}$. □

4.3 From a Cobham style Function Algebra to FP

In this section, we prove that the Cobham style function algebra $\mathcal{F}_{\mathsf{Cob}}$ is sound with respect to ordinary Turing Machines. The is a mere consequence of some results we already established: Remark 7 proofs the soundness of this class with respect to $\mathcal{POR}^-$, then Theorem 9 proves a quantitative soundness of $\mathcal{POR}^-$ with respect to **SFP**, thus the main concern of the proof is showing that in the functions in **SFP** we obtain there in this specific case are in **FP**, too. This is done showing a reduction from STMs to TMs. Before diving the claim and the detailed proof, we would like to point out that we are not meaning to show that, *in general*, a STM can be reduced to an equivalent TM. But that this can be done under the strong assumption that the value on the oracle tape does not affect the result of the computation. This particular result is part of the proof of Lemma 24, in which we show that $\mathcal{PTF}$ is sound with respect to **FP**.

Lemma 24. *Each function $f \in \mathcal{F}_{\mathsf{Cob}}$ is in* **FP***, too.*

Proof. Suppose that $f \in \mathcal{F}_{\mathsf{Cob}}$. According to Remark 7, there is a function $g \in \mathcal{POR}^-$ such that:

$$\forall x \in \mathbb{S}.\forall \omega \in \mathbb{O}.f(x) = g(x, \omega)$$

so, said $f(x) = y$, we have that:

$$\mu(\{\omega \in \mathbb{O} | g(x, \omega) = y\}) = 1.$$

Furthermore, from Theorem 9, we know that there is a $h \in$ **SFP** such that:

$$\mu(\{\eta \in \{\mathsf{0}, \mathsf{1}\}^{\mathbb{N}} | h(x, \eta) = y\}) = 1,$$

which entails that:

$$\forall \eta \in \{\mathsf{0}, \mathsf{1}\}^{\mathbb{N}}.h(x, \eta) = y$$

due to the polynomial time bound of **SFP**. Moreover, as a consequence of Lemmas 15, 16, Proposition 6 and Lemma 22, we can suppose without lack of generality, that said $M_h = \langle \mathcal{Q}, \Sigma, \delta, q_0 \rangle$ the STM computing h, δ is such that:

$$\forall c \in \Sigma.\forall q \in \mathcal{Q}.\delta(q, c, \mathsf{0}) = \delta(c, q, \mathsf{1}). \qquad (*)$$

This because the program $P_g \in \mathsf{SIFP}_{\mathsf{RA}}$ whose semantics is equal to g does not employ $\mathsf{Flip}(e)$ expressions. Moreover, even the $\mathsf{SIFP}_{\mathsf{LA}}$ implementation of P_g does not use to $\mathsf{RandBit}()$ expression, thus it is identical to P_g. Consequently, its $\mathbf{SFP_{OD}}$ correspondent function can be defined upon a machine which does not consume the oracle tape, thus, M_h is a machine whose transition function δ has the property described above. Finally, we can define a standard TM which behaves *exactly* as M_h does: $M := \langle \mathcal{Q}, \Sigma, \Delta(\delta), q_0 \rangle$, with

$$\Delta(\delta) := \{\langle p, c_r, q, c_w, d \rangle | \exists \mathsf{b} \in \mathbb{B}.\langle p, c_r, \mathsf{b}, q, c_w, d \rangle \in \delta\}.$$

Since δ satisfies Equation $(*)$, $\Delta(\delta)$ is a function, so M is a TM. We show that for every $k \in \mathbb{N}$, said $\overline{\eta} = \lambda x.\mathsf{0}$, it holds that:

$$\langle \sigma, q, \tau, \overline{\eta}\rangle |\!\triangleright^k_\delta \langle \sigma', q', \tau', \overline{\eta}\rangle \to \langle \sigma, q, \tau\rangle \triangleright^k_{\Delta(\delta)} \langle \sigma', q', \tau'\rangle.$$

The proof is by induction on k:

0 The claim is trivial.

$n+1$ Suppose that:

$$\langle \sigma, q, \tau, \overline{\eta}\rangle |\!\triangleright^n_\delta \langle \sigma'', q'', \tau'', \overline{\eta}\rangle \Vdash_\delta \langle \sigma', q, \tau', \overline{\eta}\rangle.$$

The claim comes from the IH observing that $\delta(q, c, \mathsf{0}) = \Delta(\delta)(q, c)$ and thus that:

$$\langle \sigma'', q'', \tau''\rangle \vdash_{\Delta(\delta)} \langle \sigma', q, \tau'\rangle$$

as a consequence of the definition of $\Vdash_\delta$ and $\vdash_\delta$ (Definitions 49 and 3).

As a consequence of this result, we know that for every $x \in \mathbb{S}$ the function computed by M and called h' is such that:

$$\forall x \in \mathbb{S}.h'(x) = h(x, \overline{\eta}).$$

Moreover, if the first machine is poly-time, then even the second is so. To summarize, supposing $f \in \mathcal{F}_{\mathsf{Cob}}$, we know that there is a function $g \in \mathcal{POR}^-$ such that:

$$\forall x \in \mathbb{S}.\forall \omega \in \mathbb{O}.f(x) = g(x, \omega).$$

then there is also a function h in **SFP** such that:

$$\forall x \in \mathbb{S}.\forall \eta \in \mathbb{B}^{\mathbb{N}}.f(x) = h(x, \eta),$$

which entails that:

$$\forall x \in \mathbb{S}.f(x) = h(x, \overline{\eta}).$$

So, since $\forall x \in \mathbb{S}.h'(x) = h(x, \overline{\eta})$, we conclude that:

$$\forall x \in \mathbb{S}.f(x) = h'(x).$$

Finally, since $h' \in \mathbf{FP}$, we obtain $f \in \mathbf{FP}$.

□

4.4 Concluding the proof

In Section 4.2, we have shown that the $\mathcal{F}_{\mathsf{Cob}}$ function algebra is complete with respect to the class **FP**. Subsequently, in Section 4.3, we have shown that $\mathcal{F}_{\mathsf{Cob}}$ is even sound with respect to **FP**. These two results, if taken together, show that the class of functions computed by the Cobham style function algebra $\mathcal{F}_{\mathsf{Cob}}$ coincides with the class of functions computed by polynomial Turing machines, namely **FP**. This result have been for long part of the folklore surrounding Cobham's function algebra, but as far as we know, among the literature there was not any detailed, self-contained and exhaustive proof of that fact.

Theorem 11. **FP** $= \mathcal{PTF}$.

Proof. Consequence of Lemmas 23 and 24.

□

Chapter 5

Characterizing complexity classes

In Chapter 3, we proved that every **PPT** function is Σ^b_1 representable in RS^1_2 a result which grounds the characterizations of some well-known probabilistic complexity classes. In this chapter we show how it is possible, for instance, to characterize the classes **BPP**, **RP**, co-**RP** and **ZPP**. To do so we introduce an extension of $\mathcal{L}$ strongly inspired by the arithmetic MQPA introduced in [1]. We call this novel language $\mathcal{L}^{\mathsf{MQ}}$. The language $\mathcal{L}^{\mathsf{MQ}}$ extends $\mathcal{L}$ expressiveness by means of a *measure quantifier* $\mathbf{C}^{s/t}$ which is intended to model measure-related assertions such as:

$$\mu\left(\llbracket F \rrbracket\right) \geq \frac{|s|}{|t|}.$$

Leveraging the quantitative semantics for $\mathcal{L}$ formulæ, this quantifier allows for a characterization of probabilistic complexity classes such as the aforementioned ones.

This chapter is structured as follows:

- In Section 5.1, we define the language $\mathcal{L}^{\mathsf{MQ}}$ and its semantics.
- In Section 5.2, we show the characterizations of the aforementioned probabilistic complexity classes.

5.1 The $\mathcal{L}^{\mathsf{MQ}}$ Language

The language $\mathcal{L}^{\mathsf{MQ}}$ is the first-order language with equality obtained extending $\mathcal{L}$ with the measure quantifier $\mathbf{C}^{s/t}$ and his De Morgan dual $\mathbf{D}^{s/t}$. This novel language uses the same term grammar of $\mathcal{L}$ and, apart the measure quantifiers, the well-formed fromulaæ of $\mathcal{L}^{\mathsf{MQ}}$ are exactly the ones of $\mathcal{L}$. In what follows we suppose that the notational conventions adopted for $\mathcal{L}$ are adopted for $\mathcal{L}^{\mathsf{MQ}}$, as well.

Definition 111 (Terms of $\mathcal{L}^{\mathsf{MQ}}$)**.** Let $x, y, \dots$ denote variables. Terms are defined by the following grammar:

$$t, s ::= x \;\Big|\; \epsilon \;\Big|\; \mathtt{0} \;\Big|\; \mathtt{1} \;\Big|\; t \frown s \;\Big|\; t \times s.$$

Definition 112 (Formulæ)**.** Let $x, y, \dots$ denote variables and $t, s, \dots$ terms. Formulæ are defined by the following grammar:

$$F, G ::= \mathtt{Flip}(t) \;\Big|\; t = s \;\Big|\; t \subseteq s \;\Big|\; \neg F \;\Big|\; F \wedge G \;\Big|\; F \vee G \;\Big|\; F \to G \;\Big|\; \exists x.F \;\Big|\; \forall x.F \;\Big|\; \mathbf{C}^{s/t}F \;\Big|\; \mathbf{D}^{s/t}F.$$

Unlikely $\mathcal{L}$, due to the presence of measure quantifiers, it makes no sense to define a qualitative semantics for $\mathcal{L}^{\mathsf{MQ}}$, so we will only define a *quantitative semantics* for this language. Intuitively, the *quantitative* semantics of $\mathcal{L}^{\mathsf{MQ}}$ is an extension of the semantics of $\mathcal{L}$ form Definition 37 in which measure quantified formulæ of the form $\mathbf{C}^{s/t}.F$ are quantitatively true, meaning that their semantics id $\mathbb{O}$, if and only if the $[\![F]\!]$ is grater or equal to $\frac{|s|}{|t|}$. Finally, in Remark 9, we show that $\mathcal{L}^{\mathsf{MQ}}$ is a conservative extension of $\mathcal{L}$.

Definition 113 (Interpretation for Terms)**.** Given an environment $\xi : \mathcal{G} \mapsto \mathbb{S}$, a term t in $\mathcal{L}^{\mathsf{MQ}}$, the *interpretation of* t *in* ξ is the string $[\![t]\!]_\xi \in \mathbb{S}$ defined as follows:

$$
\begin{aligned}
[\![\epsilon]\!]_\xi &:= \boldsymbol{\epsilon} & [\![x]\!]_\xi &:= \xi(x) \in \mathbb{S} \\
[\![\mathtt{0}]\!]_\xi &:= \mathtt{0} & [\![t \frown s]\!]_\xi &:= [\![t]\!]_\xi[\![s]\!]_\xi \\
[\![\mathtt{1}]\!]_\xi &:= \mathtt{1} & [\![t \times s]\!]_\xi &:= [\![t]\!]_\xi \times [\![s]\!]_\xi.
\end{aligned}
$$

Definition 114 (Quantitative Semantics)**.** Given a formula F and an environment $\xi : \mathcal{G} \to \mathbb{S}$, where $\mathcal{G}$ is the set of term variables, the *interpretation of* F *in* ξ, $[\![F]\!]_\xi$, is the (measurable) set of functions inductively defined as follows:

$$
\begin{aligned}
[\![\mathtt{Flip}(t)]\!]_\xi &:= \{\omega \mid \omega([\![t]\!]_\xi) = \mathtt{1}\} & [\![\neg G]\!]_\xi &:= \mathbb{O} \setminus [\![G]\!]_\xi \\
[\![t = s]\!]_\xi &:= \begin{cases} \mathbb{O} & \text{if } [\![t]\!]_\xi = [\![s]\!]_\xi \\ \emptyset & \text{otherwise} \end{cases} & [\![G \vee H]\!]_\xi &:= [\![G]\!]_\xi \cup [\![H]\!]_\xi \\
& & [\![G \wedge H]\!]_\xi &:= [\![G]\!]_\xi \cap [\![H]\!]_\xi \\
[\![t \subseteq s]\!]_\xi &:= \begin{cases} \mathbb{O} & \text{if } [\![t]\!]_\xi \subseteq [\![s]\!]_\xi \\ \emptyset & \text{otherwise} \end{cases} & [\![G \to H]\!]_\xi &:= (\mathbb{O} \setminus [\![G]\!]_\xi) \cup [\![H]\!]_\xi \\
& & [\![\exists x.G]\!]_\xi &:= \bigcup_{i \in \mathbb{S}} [\![G]\!]_{\xi\{x \leftarrow i\}} \\
[\![\mathbf{C}^{s/t}F]\!]_\xi &:= \begin{cases} \mathbb{O} & \text{if } |[\![t]\!]_\xi| > 0 \text{ and } \mu([\![F]\!]) \geq \frac{|[\![s]\!]_\xi|}{|[\![t]\!]_\xi|} \\ \emptyset & \text{otherwise} \end{cases} & [\![\forall x.G]\!]_\xi &:= \bigcap_{i \in \mathbb{S}} [\![G]\!]_{\xi\{x \leftarrow i\}}. \\
[\![\mathbf{D}^{s/t}F]\!]_\xi &:= \begin{cases} \mathbb{O} & \text{if } |[\![t]\!]_\xi| > 0 \text{ and } \mu([\![F]\!]) < \frac{|[\![s]\!]_\xi|}{|[\![t]\!]_\xi|} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

where $|[\![t]\!]_\xi|$ denotes the length of $[\![t]\!]_\xi \in \mathbb{S}$ and μ is the measure function of Definition 36.

Intuitively, the semantics of the measure quantifiers evaluates the terms s and t within the environment ξ, then it computes the ratio between the length of s and the length of t and compares the values thus obtained with the measure of the semantics of the quantified formula. An example of the application of these quantifiers is the following:

Example 3. Take in exam the formula F defined below:

$$F := \forall x.\mathbf{C}^{1/1}\mathbf{D}^{11/111}\mathtt{Flip}(x).$$

The semantics of F is $\mathbb{O}$, indeed, be x a general term within $\mathbb{S}$ then it is true that $\mu\left([\![\mathtt{Flip}(x)]\!]\right) = \frac{1}{2}$ regardless from the value of x. Thus, $[\![\mathbf{D}^{11/111}\mathtt{Flip}(x)]\!] = \mathbb{O}$ and $[\![F]\!] = \mathbb{O}$ because $\mu(\mathbb{O}) = 1 = \frac{|\mathtt{1}|}{|\mathtt{1}|}$.

The previous example also emphasizes that the terms within s, t within $\mathbf{C}^{s/t}$ quantified formulæ can cause cumbersome notations. For this reason, we introduce a simplified notation which allows us to replace these terms with natural numbers.

Notation 15. For every $n, m \in \mathbb{N}$, we represent with $\mathbf{C}^{n/m}F$ (respectively $\mathbf{D}^{n/m}$) the formula $\mathbf{C}^{\mathtt{1}^n/\mathtt{1}^m}F$ (respectively $\mathbf{D}^{\mathtt{1}^n/\mathtt{1}^m}F$).

Remark 9. *$\mathcal{L}^{\mathsf{MQ}}$ is conservative extension of $\mathcal{L}$ with respect to their quantitative semantics. Formally, said $\overline{[\![\cdot]\!]}$ $\mathcal{L}$'s quantitative semantics, it holds that:*

$$\forall F \in \mathcal{L}.\overline{[\![F]\!]} = [\![F]\!].$$

Proof. Immediate by induction on F. □

5.2 Characterizations

In this section, we examine how it is possible to define *semantical characterizations* of well-known probabilistic complexity classes within $\mathcal{L}^{\mathsf{MQ}}$. With *semantical characterizations*, we mean that these characterizations are done leveraging the notion of semantic consequence ($\models$) rather than the syntactical notion of provability ($\vdash$). This is a consequence of the Σ^b_1-representability property (Theorem 4) which captures **PPT** functions by means of the following statement:

$$\forall f \in \mathbf{PPT}.\exists G_f \in \Sigma^b_1.RS^1_2 \vdash \forall x\exists! y.G(x,y) \wedge \mu\left([\![G(x,y)]\!]\right) = Pr[f_G(x) = y].$$

Due to the explicit presence of G's semantics in correspondence with $Pr[f_G(x) = y]$, Theorem 4 describes a *semantical notion* of representability instead of a syntactical one.

5.2.1 BPP

The probabilistic complexity class **BPP** has been introduced in Definition 17. It is the class of languages having a poly-time PTM M deciding L with probability of error smaller than $\frac{1}{3}$. This class is intended to capture both time feasibility, and arbitrarily low probability of error. Indeed, due to the polynomial time bound required to the machine, **BPP** languages can be decided with low time consuming algorithms; moreover, running the same machine a polynomial number of times, it is possible to reduce the probability of error to an arbitrarily small value, still requiring polynomial time complexity. Using our framework, we can define **BPP** as follows:

Definition 115 (**BPP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **BPP** if and only if, said f_L the characteristic function of L, there is a Polynomial Probabilistic Turing Machine M such that $\forall \sigma \in L.Pr[Y_{M,\sigma} = f_L(\sigma)] \geq \frac{2}{3}$.

Exploiting Theorem 4, we can give a characterization of **BPP** replacing the PTM M of Definition 115 with a Σ^b_1 formula of $\mathcal{L}$ which is provable under RS^1_2. Doing so, we end up with Characterization 3.

Characterization 3. *We say that a language L is in* **BPP** *if (and only if)*

$$\exists G \in \Sigma^b_1.RS^1_2 \vdash \forall x\exists! y.G(x,y) \wedge \forall \sigma \in L. \models \mathbf{C}^{2/3}G(\sigma, \mathtt{1}) \wedge \forall \sigma \in \overline{L}. \models \mathbf{C}^{2/3}\exists y.G(\sigma, y) \wedge y \neq \mathtt{1}.$$

Proof. The result above is a consequence of the existence of a function $f_L : \{\mathtt{0},\mathtt{1}\}^* \longrightarrow \mathbb{B}$ which decides L: according to Theorem 4 (second claim), there is a RS^1_2-provable formula $G \in \Sigma^b_1$ asserting the totality of f_L. Moreover, it holds that:

$$\forall x, y \in \mathbb{S}.\mu([\![G(x,y)]\!]) = Pr[F_l(x) = y].$$

Thus, if $x \in L$, then $Pr[F_l(x) = \mathtt{1}] \geq \frac{2}{3}$, so it even holds that $\mathbf{C}^{2/3}G(x, \mathtt{1})$. Similarly, if $x \in \overline{L}$, $Pr[F_l(x) \neq \mathtt{1}] \geq \frac{2}{3}$, so $\mathbf{C}^{2/3}G(x, \mathtt{0})$. For the opposite direction, we can take in exam the **PPT** function F_G whose existence is guaranteed by Theorem 4 (first claim). The opposite direction can thus be shown moving an argument identical to the one above. □

5.2.2 RP and co-RP

The **RP** complexity class (Definition 18) is very similar to **BPP**, the only difference is that **RP** contains all the languages which are decidable with *one-sided* probabilistic error, instead of *two-sided* probabilistic error, as for **BPP**. Precisely, a language L is in **RP** if and only if there is a Probabilistic Turing Machine M always refusing strings which are not in L and accepting L's elements with probability at least $\frac{2}{3}$. This can be formally stated by this Definition:

Definition 116 (**RP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **RP** if and only if there is a Polynomial Probabilistic Turing Machine M such that:

$$\forall \sigma \in L. Pr[Y_{M,\sigma} = \mathtt{1}] \geq \frac{2}{3};$$
$$\forall \sigma \notin L. Pr[Y_{M,\sigma} \neq \mathtt{1}] = 1.$$

As for **BPP**, we can characterize the **RP** complexity class in $\mathcal{L}^{\mathsf{MQ}}$ with a slight modification on the characterization of **BPP**, i.e. requiring that:

$$\forall \sigma \in \overline{L}. \models \mathbf{C}^{1/1} G(\sigma, \mathtt{0}),$$

instead of requiring that

$$\forall \sigma \in \overline{L}. \models \mathbf{C}^{2/3} G(\sigma, \mathtt{0}).$$

i.e. requiring no probabilistic error for all the strings which are not in the language. Formally:

Characterization 4. *We say that a language L is in* **RP** *if (and only if)*

$$\exists G \in \Sigma^b_1 . RS^1_2 \vdash \forall x \exists ! y. G(x, y) \land \forall \sigma \in L. \models \mathbf{C}^{2/3} G(\sigma, \mathtt{1}) \land \forall \sigma \in \overline{L}. \models \mathbf{C}^{1/1} \exists y. G(\sigma, y) \land y \neq \mathtt{1}.$$

Proof. Analogous to the proof of Characterization 3. □

The complementary of **RP** is co-**RP** (Definition 19. It is defined as the class of languages L such that there is a PTM M accepting all the strings of L with no probabilistic error and refusing the strings which not belonging to L with probability of error smaller than $\frac{1}{3}$. This can be formally stated as follows:

Definition 117 (co-**RP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in co-**RP** if and only if there is a Polynomial Probabilistic Turing Machine M such that:

$$\forall \sigma \in L. Pr[Y_{M,\sigma} = \mathtt{1}] = 1;$$
$$\forall \sigma \notin L. Pr[Y_{M,\sigma} \neq \mathtt{1}] \geq \frac{2}{3}.$$

The characterization comes naturally, applying the same approach of those for **BPP** and **RP**.

Characterization 5. *We say that a language L is in co-***RP** *if (and only if)*

$$\exists G \in \Sigma^b_1 . RS^1_2 \vdash \forall x \exists ! y. G(x, y) \land \forall \sigma \in L. \models \mathbf{C}^{1/1} G(\sigma, \mathtt{1}) \land \forall \sigma \in \overline{L}. \models \mathbf{C}^{2/3} \exists y. G(\sigma, y) \land y \neq \mathtt{1}.$$

Proof. Analogous to the proof of Characterization 3. □

5.2.3 ZPP

In this section, we study a characterization for the **ZPP** complexity class (Definition 20), containing all the *Zero Probabilistic-error Poly-time* languages, namely all the languages which can be decided by a PTM with zero probabilistic error.

Definition 118 (**ZPP**)**.** We say that a language $L \subseteq \mathbb{S}$ is in **ZPP** if and only if there is a Polynomial Probabilistic Turing Machine M such that:

$$\forall \sigma \in \mathbb{S}.Pr[Y_{M,\sigma} = \mathtt{1} \wedge \sigma \notin \mathbb{S}] = 0;$$
$$\forall \sigma \in \mathbb{S}.Pr[Y_{M,\sigma} = \mathtt{0} \wedge \sigma \in \mathbb{S}] = 0;$$
$$Pr[Y_{M,\sigma} \neq \mathtt{1} \wedge Y_{M,\sigma} \neq \mathtt{0}] < \frac{1}{3}.$$

We would like to point out that Definition 20 and 118 are not the standard one, which is based on Las Vegas algorithms, i.e. algorithms with *expected* polynomial complexity. However, within this work, we employed this one because **PPT** functions are Monte Carlo algorithm, i.e. probabilistic algorithms with *worst case* polynomial complexity. For this reason, if is a more convenient starting point for the development of an $\mathcal{L}^{\mathsf{MQ}}$ characterization of that class.
Theorem 1 states that **ZPP** = **RP** ∩ co-**RP**, for this reason, it allows us to define a characterization of **ZPP** leveraging the characterizations of **RP** and co-**RP**.

Characterization 6. *We say that a language L is in* **ZPP** *if (and only if)*

$$\exists F \in \Sigma^b_1.RS^1_2 \vdash \forall x \exists! y.F(x,y) \wedge \forall \sigma \in L. \models \mathbf{C}^{2/3} F(\sigma, \mathtt{1}) \wedge \forall \sigma \in \overline{L}. \models \mathbf{C}^{1/1} \exists y.F(\sigma, y) \wedge y \neq \mathtt{1}$$

and

$$\exists G \in \Sigma^b_1.RS^1_2 \vdash \forall x \exists! y.G(x,y) \wedge \forall \sigma \in L. \models \mathbf{C}^{1/1} G(\sigma, \mathtt{1}) \wedge \forall \sigma \in \overline{L}. \models \mathbf{C}^{2/3} \exists y.G(\sigma, y) \wedge y \neq \mathtt{1}.$$

Proof. Consequence of Theorem 1 and of Characterizations 4, 5. □

Chapter 6

Omitted Concepts and Results

In this chapter we collect some of the technical results which are employed in other parts of this work, hoping to foster the understanding of the main thread of the proof, without lacking of completeness. In the first section (Section 6.1), we show how it is possible to define a co-articulated collection of encodings of data structures on the set of binary strings $\mathbb{S}$. Then we define $\mathcal{POR}$ functions — Actually $\mathcal{POR}^-$ functions — which are the embedding of canonical operations on these encodings. The existence of these functions entails that, within $\mathcal{POR}$, it is possible to represent and manipulate natural numbers, lists, sets and other more complex data structures. All these results, taken together be the basis of the proof that **SFP** can be reduced to $\mathcal{POR}$.

In the following section (Section 6.2), we collect the auxiliary Lemmas employed in the previous chapters, divided per section. This is aimed to separate low-level results from the main structure of the proof.

6.1 Encodings

In the proof sketch of Lemma 3, we discussed that we can show the inclusion of $\mathcal{PTF}$ in $\mathcal{POR}$ building a function which manipulates the encoding of a machine's configuration followings behavior described by δ. This work requires the definition of appropriate encodings of these structures by means of strings, together with the definition of manipulating functions which can be used to emulate upon these encodings all the operations of a FSTM. Among the literature, there are many construction of similar encodings, so that they are rarely taken re-defined extensively. While it is usually more common to refer to the possibility to encode certain data structures within a set (typically $\mathbb{N}$) in an non-constructive fashion. The reasons why we are re-defining a complete encoding for numbers, lists and other data structures on the set $\mathbb{S}$ despite the complicatedness of this process is that we want our proof to be as self-contained and detailed as possible, in order to establish in the most convincing way our results. For this reason, this section is aimed to show that $\mathcal{POR}^-$ is expressive enough to represent all the data structures which are needed to support the simulation of FSTM in $\mathcal{POR}$. In particular, in the first section we will show how we will represent the data, then in the following section, we will show how we can use $\mathcal{POR}^-$ to define functions which can be used to manipulate data encodings in order to implement complex behaviors such as the simulation of a FSTM.

6.1.1 Statical encodings

Before getting into more complex encodings, we start with a toy problem: the encoding of the constant strings. We did not mention such values before, but they are needed, for instance, as a notational support, because they allow us to write string values directly into $\mathcal{POR}^-$ function rather than writing explicitly the functions computing them. It may seem superfluous to show that there is a $\mathcal{POR}^-$ function for all the constant strings, but it will be useful to introduce the same pattern which we will follow later for the definition of more complex encodings.

Definition 119 (Encoding of Constants)**.** We define the injection $\underline{\cdot}_{\mathbb{S}} : \mathbb{S} \longrightarrow \mathbb{S}$ as follows:

$$\underline{\sigma}_{\mathbb{S}} := \sigma.$$

For sake of readability, from now on, the notation $\underline{\cdot}_{\mathbb{S}}$ will be omitted. This definition, by itself does not carry any information concerning the possibility to adopt it within $\mathcal{POR}$. To this aim, we show that this encoding is suitable for $\mathcal{POR}^-$, which means that for every $s \in \mathbb{S}$ there is $\mathcal{POR}^-$ function computing exactly $\underline{\sigma}_{\mathbb{S}}$. If we show that a similar result holds, then defining a $\mathcal{POR}^-$ function we are allowed to use string expressions as parameters, because strings can be computed by $\mathcal{POR}^-$ functions and because $\mathcal{POR}^-$ is closed under composition.

Remark 10 (Representability of $\mathbb{S}$ in $\mathcal{POR}^-$)**.** $\forall \sigma \in \mathbb{S}.\exists f_\sigma \in \mathcal{POR}.\forall x \in \mathbb{S}.\forall \omega \in \mathbb{O}.f_\sigma(x, \omega) = \underline{\sigma}_{\mathbb{S}}$

Proof. Take as f_σ:

$$\begin{aligned} f_{\boldsymbol{\epsilon}}(x, \omega) &:= E(x, \omega) : \\ f_{\tau\mathtt{b}}(x, \omega) &:= S_{\mathtt{b}}(f_\tau(x, \omega), \omega). \end{aligned}$$

The claim can be obtained by induction on σ. □

We choose to represent the natural number n as a string composed by $n+1$ 1s [1], in this way, definitions of arithmetical functions will be simpler.

Definition 120 (Encoding of Natural numbers)**.** We define the injection $\underline{\cdot}_{\mathbb{N}} : \mathbb{N} \longrightarrow \mathbb{S}$ as follows:

$$\underline{n}_{\mathbb{N}} := \mathtt{1}^{n+1}.$$

This encoding is suitable for $\mathcal{POR}^-$, because we can show that each string which is the encoding of a natural number can be represented by a term in $\mathcal{POR}^-$, formally:

Remark 11 (Representability of $\mathbb{N}$ in $\mathcal{POR}^-$)**.** $\forall n \in \mathbb{N}.\exists f_n \in \mathcal{POR}.\forall x, \omega.f_n(x, \omega) = \underline{n}_{\mathbb{N}}$

Proof. Take as f_n:

$$\begin{aligned} f_0(x, \omega) &:= S_{\mathtt{1}}(E(x, \omega), \omega); \\ f_{n+1}(x, \omega) &:= S_{\mathtt{1}}(f_n(x, \omega), \omega). \end{aligned}$$

The claim can be obtained by induction on n. □

In Computer Science, conventionally, the true Boolean value is usually represented with `1` and the false value is represented with `0`. We will follow this convention.

[1] We took this decision because we do not want $\boldsymbol{\epsilon}$ to be neither the encoding of a natural number nor of any other value (indexes, Booleans, tuples, ...). This will allow us to use it as return value in case of errors, if necessary.

Definition 121 (Encoding of Boolean values)**.** We define the injection $\underline{\cdot}_{\mathbb{B}} : \mathbb{B} \longrightarrow \mathbb{S}$ as follows:

$$\underline{0}_{\mathbb{B}} := \mathtt{0}$$
$$\underline{1}_{\mathbb{B}} := \mathtt{1}.$$

Remark 12 (Representability of $\mathbb{B}$ in $\mathcal{POR}^-$)**.** $\forall \mathsf{b} \in \mathbb{B}.\exists f_{\mathsf{b}} \in \mathcal{POR}.\forall x, \omega.f_{\mathsf{b}}(x, \omega) = \underline{n}_{\mathbb{B}}$

Proof. Take as f_n:

$$f_{\mathtt{0}}(x, \omega) := S_{\mathtt{0}}(E(x, \omega), \omega)$$
$$f_{\mathtt{1}}(x, \omega) := S_{\mathtt{1}}(E(x, \omega), \omega).$$

The claim is correct by definition of the functions $f_{\mathtt{0}}$ and $f_{\mathtt{1}}$. □

In order to encode in $\mathcal{POR}^-$ any polynomial FSTM, we need to represent some complex data structures: we argued that for each $f \in \mathcal{PTF}$, there is a function g implementing it. This can be done by defining a general schema for g which depends on a representation of the machine transition function δ. In this way, changing only the function which encodes δ, we are be able to decouple the encoding of the machine's mechanics, by the machine's transition function.

To encode the information contained in the finite function δ (which can be considered a big but finite table), we need to define some compositional data structures. From a set theoretic point of view, functions are sets of tuples so, for sake of simplicity, we chose to represent these different data structures — namely sets, lists, tuples and functions — uniformly as lists, which generalize all of them.

6.1.1.1 Lists

In order to represent lists, we will adapt an encoding from Odifreddi's *Classical Recursion Theory* [21, p. 183] which was introduced as a representation of tuples. Before doing so, we would like to formally define the set of lists, in order to clarify which kind of data we will be working on.

Definition 122 (Lists of strings)**.** We say that l is a list of strings if and only if $l \in \mathbb{L}$, where $\mathbb{L} := \bigcup_{i \in \mathbb{N}} \mathbb{S}^i$.

Before presenting the encoding, we need to introduce two auxiliary functions, the *doubling* function $\mathcal{D}$ and the *halving* function $\mathcal{H}$, such epithets are due to the fact that they respectively double and halve the length of their input. The function $\mathcal{D}$ interleaves the bits in its input with $\mathtt{1}$s and $\mathcal{H}$ removes those characters.

The $\mathcal{D}$ function can be seen as the mapping of a string σ through an encoding $\mathbb{B} \longrightarrow \mathbb{B}^2$, and $\mathcal{H}$ is the left-inverse of $\mathcal{D}$. In this way we can represent lists as sequences of characters $c \in \mathbb{B}^2$. Two of those four characters are used to encode the values in $\mathbb{B}$ which compose members of list, while one of the two remaining characters (we chose $\mathtt{00}$) is used as separator. This way we will be able to embed multiple strings in a single sequence of bits encoding a list and to extract the values contained in it.

Definition 123 (List Element Encoding and Decoding Functions)**.** Encoding and decoding functions are defined as follows:

$$\mathcal{D}(\sigma\mathtt{0}) := \mathcal{D}(\sigma)\mathtt{10}$$
$$\mathcal{D}(\sigma\mathtt{1}) := \mathcal{D}(\sigma)\mathtt{11}.$$

$$\mathcal{H}(\sigma\mathtt{10}) := \mathcal{H}(\sigma)\mathtt{0}$$
$$\mathcal{H}(\sigma\mathtt{11}) := \mathcal{H}(\sigma)\mathtt{1}.$$

We can define the encoding of lists as follows:

Definition 124 (List Encoding)**.** For every $n \in \mathbb{N}$, the family of list encoding injections $\langle\cdot,\dots,\cdot\rangle^n_{\mathbb{L}} : \mathbb{S}^n \longrightarrow \mathbb{S}$ is defined as described below:

$$\langle x_1,\dots,x_n\rangle^n_{\mathbb{L}} := \mathtt{00}\mathcal{D}(x_n)\mathtt{00}\dots\mathtt{00}\mathcal{D}(x_1)\mathtt{00}\mathcal{D}(x_1)\mathtt{00}\mathcal{D}(\underline{n}_{\mathbb{N}})\mathtt{00}.$$

Following the pattern which we used for the previous encodings we should show that all the values which encode tuples can be represented by functions in $\mathcal{POR}^-$. To do so, we need to show that natural correspondent of $\mathcal{D}$ and $\mathcal{H}$ are in $\mathcal{POR}^-$, together with the fact that the function which computes the concatenation is in $\mathcal{POR}^-$, too.

Definition 125 (String Concatenation function)**.**

$$\begin{aligned} concat(x,\boldsymbol{\epsilon},\omega) &:= x; \\ concat(x,y\mathtt{b},\omega) &:= S_{\mathtt{b}}(concat(x,y,\omega),\omega)|_{xy\mathtt{b}}. \end{aligned}$$

Lemma 25 (String concatenation is in $\mathcal{POR}^-$)**.** *Formally: $concat \in \mathcal{POR} \land \forall x_1,x_2 \in \mathbb{S}.\forall\omega \in \mathbb{O}.concat(x_1,x_2,\omega) = x_1x_2$.*

Proof. Such function is defined by bounded recursion, so the result can be proven by induction on x_2. □

Lemma 26 ($\mathcal{D}$ is in $\mathcal{POR}^-$)**.** *There is a function doub in $\mathcal{POR}^-$ such that $\forall x \in \mathbb{S},\forall\omega \in \mathbb{O}.doub(x,\omega) = \mathcal{D}(x)$.*

Proof. It can be shown that the doubling function, *doub*, can be defined by bounded recursion:

$$\begin{aligned} doub(\boldsymbol{\epsilon},\omega) &:= \boldsymbol{\epsilon}; \\ doub(y\mathtt{1},\omega) &:= S_{\mathtt{1}}(S_{\mathtt{1}}(doub(y,\omega),\omega))|_{(\mathtt{11})\times(y\mathtt{1})}; \\ doub(y\mathtt{0},\omega) &:= S_{\mathtt{0}}(S_{\mathtt{1}}(doub(y,\omega),\omega))|_{(\mathtt{11})\times(y\mathtt{1})}. \end{aligned}$$

□

Remark 13. *All the finite lists can be represented by functions in $\mathcal{POR}^-$. Namely: $\forall l \in \mathbb{L}.\exists f_l \in \mathcal{POR}.\forall x \in \mathbb{S},\omega \in \mathbb{O}.f_l(x,\omega) = \langle l\rangle^{|l|}_{\mathbb{L}}$.*

Proof. This comes form the fact that $\langle l\rangle^n_{\mathbb{L}}$ can be defined by composition of functions which are in $\mathcal{POR}^-$, namely constant functions (e.g. $f_{\mathtt{00}}$), *concat*, f_n, D_f. □

We can also state representability results dealing with functions and sets, instead of lists. We will not state a similar result for tuples directly because, according to Definition 122, tuples are *exactly* specific cases of lists. Concerning sets and functions, we can represents sets as specific lists with at most one copy per element, and finite functions from $\mathbb{S}$ to $\mathbb{S}$ (but not only) with their graph using lists as generalized couples and sets as well.

Definition 126 (Finite Set Encoding)**.** The family of set encoding injections $\underline{\cdot}_{\mathcal{P}_{fin}(\mathbb{S})} : \mathcal{P}_{fin}(\mathbb{S}) \longrightarrow \mathbb{S}$ is defined as the family of function described below:

$$\underline{\{x_1,\dots,x_n\}}_{\mathcal{P}_{fin}(\mathbb{S})} := \langle x_1,\dots,x_n\rangle^n_{\mathbb{L}}.$$

where $x_1,\dots,x_n$ are taken in ascending order according to the value of the natural number encoded by $\mathtt{1}x_i$. This definition il well-founded because sets do not contain repetitions.

Definition 127 (Function Encoding)**.** The family of Finite Function Encoding injections $\underline{\cdot}_{\mathbb{S}^{\mathbb{S}}}$: $\mathbb{S}^{\mathbb{S}} \longrightarrow \mathbb{S}$ is defined as the family of functions described below:

$$\underline{\{\langle x_1, y_1\rangle, \dots, \langle x_n, y_n\rangle\}}_{\mathbb{S}^{\mathbb{S}}} := \langle\langle x_1, y_1\rangle^2_{\mathbb{L}}, \dots, \langle x_n, y_n\rangle^2_{\mathbb{L}}\rangle^n_{\mathbb{L}}.$$

Corollary 13. *Each finite function from $\mathbb{S}$ to $\mathbb{S}$ and each set included in $\mathbb{S}$ can be represented by functions in $\mathcal{POR}^-$. Namely:*

$$\forall g \in \mathcal{P}_{fin}(\mathbb{S}^{\mathbb{S}}).\exists f_g \in \mathcal{POR}.\forall x \in \mathbb{S}, \omega \in \mathbb{O}.f_g(x, \omega) = \underline{g}_{\mathbb{S}^{\mathbb{S}}}$$

$$\forall s \in \mathcal{P}_{fin}(\mathbb{S}).\exists f_s \in \mathcal{POR}.\forall x \in \mathbb{S}, \omega \in \mathbb{O}.f_s(x, \omega) = \underline{s}_{\mathcal{P}_{fin}(\mathbb{S})}.$$

Proof. This claim is a direct consequence of Definitions 127 and 126 and of Remark 13. □

Another result which we need to state is that the graph of any function $\delta : \mathcal{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}\} \longrightarrow \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}$ can be represented by a term in $\mathbb{S}$. Formally:

Corollary 14 (Representability of δ)**.** *There is an injection $f : (\mathcal{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}\} \longrightarrow \mathcal{Q} \times \hat{\Sigma} \times \{L, R\}) \longrightarrow \mathbb{S}$.*

Proof. We have suggested that the constants L and R can be represented by means of $\mathtt{0}$ and $\mathtt{1}$ respectively, this describes an injection $i_1 : \{L, R\} \longrightarrow \mathbb{S}$. Similarly, we have suggested that we can be represented through their indexes. This identifies an injection $i_2 : \mathcal{Q} \longrightarrow \mathbb{N}$, we have also shown that there exists a natural injection $\underline{\cdot}_{\mathbb{N}} : \mathbb{N} \longrightarrow \mathbb{S}$; so, the composition of i_2 and $\underline{\cdot}_{\mathbb{N}}$ is an injection $\mathcal{Q} \longrightarrow \mathbb{S}$. Since $\hat{\Sigma}$ is a finite set, it is countable too, so there is a bijection $i_3 : \hat{\Sigma} \longrightarrow \mathbb{N}$, which composed with $\underline{\cdot}_{\mathbb{N}}$ yields another injection $\hat{\Sigma} \longrightarrow \mathbb{S}$. These observations, together with Remark 13 prove that there is an injection $i_4 : \left(\mathcal{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}\}\right) \longrightarrow \mathbb{S}$ and another injection $i_5 : \left(\mathcal{Q} \times \hat{\Sigma} \times \{L, R\}\right) \longrightarrow \mathbb{S}$. Finally, the composition of these injection with the one described by Corollary 13 yields the desired injection. □

With this result we have shown that we can encode all the information contained in a FSTM transition function in a string. Together with the previous result, we need to state that even all the possible machine configuration can be encoded by means of strings. Namely, that there exists an injection from an exhaustive representation of a FSTM's configuration and $\mathbb{S}$ because we want to simulate FSTMs by manipulating the encoding of their configurations.

Corollary 15 ((Partial) Representability of Machine's Configurations)**.** *There is an injection $f : (\hat{\Sigma}^* \times \mathcal{Q} \times \hat{\Sigma}^* \times \hat{\Sigma}^*) \longrightarrow \mathbb{S}$.*

Proof. As for the proof of Corollary 14, we observe that there is an injection $i_2 : \mathcal{Q} \longrightarrow \mathbb{N}$. Since $\hat{\Sigma}$ is a finite set, it is countable too, so there is a bijection $i_3 : \hat{\Sigma} \longrightarrow \mathbb{N}$, which composed with $\underline{\cdot}_{\mathbb{N}}$ yields another injection $\hat{\Sigma} \longrightarrow \mathbb{S}$. There is another injection $\underline{\cdot}_{\mathbb{T}} : \hat{\Sigma}^* \longrightarrow \mathbb{L}$ defined as follows:

$$\underline{\cdot}_{\mathbb{T}}(\boldsymbol{\epsilon}) := \langle \underline{i_3(\circledast)}_{\mathbb{N}} \rangle^1_{\mathbb{L}}$$
$$\underline{\cdot}_{\mathbb{T}}(c_0 c_1 \dots c_n) := \langle \underline{i_3(c_n)}_{\mathbb{N}}, \dots, \underline{i_3(c_1)}_{\mathbb{N}}, \underline{i_3(c_0)}_{\mathbb{N}} \rangle^{n+1}_{\mathbb{L}}.$$

In other words, for each $\sigma \in \hat{\Sigma}^*$, it suffices to take the list which contains in position k the mapping of the k-th element of σ through i_3, and as encoding of the empty tape a list with a single instance of the blank character. We took this decision for a technical reason: the rightmost and the leftmost characters in an empty tape exist and are $\circledast$, this will turn out to be useful in the definition of the machine's simulation in $\mathcal{POR}^-$. We also know that for every $\sigma \in \mathbb{S}$ there is an injection $\langle\cdot\rangle^{|\sigma|}_{\mathbb{L}} : \mathbb{S}^{|\sigma|} \longrightarrow \mathbb{S}$. The composition of the aforementioned injections with $\langle\cdot\rangle^4_{\mathbb{L}}$ yields the claimed one. □

We are interested in capturing $\mathcal{PTF}$ so, before going further, we want to instantiate encodings described in Corollaries 15 and 14 for the class of the Finite Stream Stream Machines.

Definition 128 (Encodings for Canonical Stream Machines)**.** The Encodings for Canonical Stream Machines are defined from the one described in Corollaries 15 and 14, choosing in particular:

- $q_i \mapsto i$ for i_2.
- $L \mapsto \mathtt{0}, R \mapsto \mathtt{1}$.
- For i_3: $\mathtt{0} \mapsto 0, \mathtt{1} \mapsto 1, \circledast \mapsto 2$
- the encoding for tapes $\underline{\cdot}_{\mathbb{T}}$, defined as follows:

$$\begin{aligned}\underline{\cdot}_{\mathbb{T}}(\epsilon) &:= \langle \underline{i_3(\circledast)}_{\mathbb{N}} \rangle^{1}_{\mathbb{L}}\\ \underline{\cdot}_{\mathbb{T}}(c_0c_1 \dots c_n) &:= \langle \underline{i_3(c_n)}_{\mathbb{N}}, \dots, \underline{i_3(c_1)}_{\mathbb{N}}, \underline{i_3(c_0)}_{\mathbb{N}} \rangle^{n+1}_{\mathbb{L}}.\end{aligned}$$

6.1.2 Dynamic Encoding

In this section we will introduce some manipulating functions in $\mathcal{POR}^-$ for the data encodings defined above. Throughout these functions, we can easily define complex program behaviors, so that composing them we are able to construct increasingly complex behaviors such as the implementation in $\mathcal{POR}$ of any FSTM.

6.1.2.1 A Simple Boolean Algebra.

Since we choose to use `0` and `1` to represent boolean values, it is natural to represent predicates with functions $\mathbb{S}^n \times \mathbb{O} \longrightarrow \mathbb{B}$. Given two values, x_1 and x_2, we can define a function that returns x_1 if a predicate outputs `1`, x_2 if that predicate outputs `0` and ϵ otherwise. Such function behaves as an `if`-expression.

Definition 129 (`if`)**.** The `if` expression is defined as follows:

$$\mathtt{if}(x_2, x_1, y, \omega) := C(y, \epsilon, x_1, x_2, \omega).$$

Notice that according to this syntax, the first argument of the `if` function is the expression resulting from the true branch, the second the one corresponding to the false branch, and the third is the so-called *guard* expression.

Remark 14. `if` $\in \mathcal{POR}^-$

Proof. Trivial: the definition of the function is given recurring to C. □

From now on, we will omit similar proofs. Thanks to the `if` function we can now also easily define some basic connectives of classical Propositional Logic.

Definition 130 (Logical Connectives)**.** Conjunction, disjunction and negation are defined as follows:

$$\begin{aligned}(P_1 \wedge P_2)(\vec{x}, \omega) &:= \mathtt{if}(P_2(\vec{x}, \omega), \mathtt{0}, P_1(\vec{x}, \omega), \omega)\\ (P_1 \vee P_2)(\vec{x}, \omega) &:= \mathtt{if}(\mathtt{1}, P_2(\vec{x}, \omega), P_1(\vec{x}, \omega), \omega)\\ (\neg P)(\vec{x}, \omega) &:= \mathtt{if}(\mathtt{0}, \mathtt{1}, P(\vec{x}, \omega), \omega).\end{aligned}$$

Now we define some predicates which are helpful in the development of the list manipulators. In particular, in order to show that $halv \in \mathcal{POR}^-$, it is useful to show that there is a predicate $odd \in \mathcal{POR}^-$ which is $\mathtt{1}$ if and only if the length of the remaining part of the string is even. Depending on the value of this predicate, proceeding by iteration it will be possible to deciding whether to keep or remove a certain bit while decoding a value stored within a list.

Definition 131. Basic logical predicates in $\mathcal{POR}^-$ are defined by the functions below:

$$\begin{aligned} odd(\boldsymbol{\epsilon}, \omega) &:= \mathtt{0}; \\ odd(y\mathtt{b}, \omega) &:= \neg\big(even(y)\big)|_{\mathtt{0}}. \\[1em] even(x, \omega) &:= \neg\big(odd(x, \omega)\big); \\ eq(x, y, \omega) &:= S(x, y, \omega) \wedge S(y, x, \omega). \end{aligned}$$

Remark 15. *Observe that the odd and even predicates work as their opposites for the encoding of natural numbers:*

$$\begin{aligned} \forall \sigma \in \mathbb{S}. \forall \mathtt{b} \in \mathbb{B}. odd(\sigma) &\leftrightarrow even(\sigma b). \\ \forall n \in \mathbb{N}. odd(\underline{n}_{\mathbb{N}}) &\leftrightarrow n \textit{ is even.} \end{aligned}$$

Finally, encoding more advanced data manipulators, such as list manipulators, it will often be the case to test the rightmost or leftmost bit of a string. In order to do so, we need to define two other predicates:

Definition 132. Given a string, predicates

- *res*, i.e. Right Most Element String
- *les*, i.e. Left Most Element String

are defined as follows:

$$\begin{aligned} res(\boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon}; \\ res(y\mathtt{0}, \omega) &:= \mathtt{0}|_{\mathtt{1}}; \\ res(y\mathtt{1}, \omega) &:= \mathtt{1}|_{\mathtt{1}}. \\[1em] les(\boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon} \\ les(y\mathtt{0}, \omega) &:= \mathtt{if}\big(\mathtt{0}, les(y, \omega), C(y, \boldsymbol{\epsilon}, \omega), \omega\big)|_{\mathtt{1}} \\ les(y\mathtt{1}, \omega) &:= \mathtt{if}\big(\mathtt{1}, les(y, \omega), S(y, \boldsymbol{\epsilon}, \omega), \omega\big)|_{\mathtt{1}}. \end{aligned}$$

Remark 16.

$$\begin{aligned} \forall \sigma \in \mathbb{S}. res(\sigma\mathtt{1}) = \mathtt{1} \wedge res(\sigma\mathtt{0}) = \mathtt{0}. \\ \forall \sigma \in \mathbb{S}. les(\mathtt{1}\sigma) = \mathtt{1} \wedge les(\mathtt{0}\sigma) = \mathtt{0}. \end{aligned}$$

Proof. Direct consequence of the definitions of *res* and *les*. □

6.1.2.2 Strings

The FSTMs' reachability function is defined on top of configurations, in particular, the strings which represent the tape are obtained from the strings in the initial configuration, applying some of the following transformations:

- Appending on the left or on the right.
- Deleting a character on the left or on the right.

Definition 133 (String Appenders and Removers)**.** We define string appenders

- *ras*, i.e. *Right Appender for Strings*
- *las*, i.e. *Left Appender for Strings*
- *rrs*, i.e. *Right Remover for Strings*
- *lrs*, i.e. *Left Remover for Strings*

as follows:

$$\begin{aligned}
reverse(\boldsymbol{\epsilon}, \omega) &:= E(x, \omega)\\
reverse(x\mathtt{b}, \omega) &:= concat(\mathtt{b}, reverse(x, \omega), \omega)\\
ras(x, y, \omega) &:= concat(x, y, \omega)\\
las(x, y, \omega) &:= reverse(ras(reverse(x, \omega), \omega), \omega)\\
rrs(\boldsymbol{\epsilon}, \omega) &:= E(x, \omega)\\
rrs(x\mathtt{b}, \omega) &:= x|_x\\
lrs(\boldsymbol{\epsilon}, \omega) &:= reverse(rrs(reverse(x, \omega), \omega), \omega).
\end{aligned}$$

The correctness of these functions can be seen by induction on the length of their arguments.

6.1.2.3 Natural Numbers

The execution of a FSTM in $\mathcal{POR}^-$ can modeled with the self-application of the machine's transition function to its output a polynomial number of times. However, the only iteration construct of $\mathcal{POR}^-$ is bounded recursion, so, to iterate a function a polynomial number of step, we need to show that the schema computing the k-th application of a function for arbitrary k is in $\mathcal{POR}$. Moreover, due to the polynomial time bound of $\mathcal{PTF}$ functions we need to compute a string whose size is polynomial in the size of the input. This would provide us a sufficiently long string to feed to the self-application schema in order to simulate any FSTM for a sufficient number of steps. For this reason, we defined the encodings of numbers in a way that their size is linear in the size of the encoded number. This will allow us to use them directly as iteration parameters for the bounded recursion schema. Finally, we said that the time bound of a FSTM is polynomial, so to iterate a function a polynomial number of steps, we could simply compute reduce the task to the computation of polynomial values in our encoding of naturals.

Definition 134 (Successor)**.** We define a successor function $S : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{S}_{\mathbb{N}}$ that computes the successor of a number as follows:

$$\begin{aligned}
S(\boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon};\\
S(y\mathtt{b}, \omega) &:= S_1(y, \omega)|_{y1}.
\end{aligned}$$

Definition 135 (Predecessor)**.** If $y \in \mathbb{S}_{\mathbb{N}}$ is the encoding of a number, the *pd* function calculates its predecessor y simply removing its last digit (if present):

$$\begin{aligned} pd(\boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon}; \\ pd(y\mathtt{b}, \omega) &:= y|_y. \end{aligned}$$

Remark 17. $\forall \sigma, c_1, c_2.pd(pd(\sigma c_1 c_2)) = \sigma$.

Proof. The claim is a trivial consequence of Definition 135. □

Definition 136 (Sum)**.** The sum of two numbers *sum* is implemented using the operation of concatenation *concat* introduced in Definition 125 as:

$$sum(x, y, \omega) := pd(concat(x, y, \omega), \omega).$$

The encoding of the difference between two numbers within $\mathcal{POR}^-$ is cumbersome because it recurs on the definition of the self application schema we introduced above, namely: the $sa_{\cdot,\cdot}$ functor. Intuitively, it applies *rrs* n times to x, where $\underline{n}_{\mathbb{N}}$ is y.

Definition 137 (Difference)**.** The function *diff*, which computes the difference between two natural numbers, is defined as follows:

$$diff(x, y, \omega) := sa_{rrs,x}(x, y, \omega).$$

Remark 18. *The function diff is in $\mathcal{POR}^-$ and follows its intended semantics.*

Proof. Lemma 5 states that the function $sa_{rrs,x}$ is in $\mathcal{POR}^-$ and that $sa_{rrs,x}(x, \underline{n}_{\mathbb{N}}, \omega) = rrs^n(x, \omega)$ this is sufficient to prove the claim, in accordance with the definition of *rrs* (Definition 133). □

In order to multiply two values x, y, we can remove their last digit — so that their size is equal to the number that they encode — concatenate x to itself y-times and return the successor of the number we get.

Definition 138 (Multiplication)**.** Multiplication between two natural numbers, *mult*, is defined as follows:

$$\begin{aligned} mult^*(x, \boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon}; \\ mult^*(x, y\mathtt{b}, \omega) &:= \big(sum(mult^*(x, y, \omega), x, \omega)\big)|_{x \times y\mathtt{1}}; \\ mult(x, y, \omega) &:= S\big(mult^*(pd(x, \omega), pd(y, \omega), \omega), \omega\big). \end{aligned}$$

With this encoding the computation of an exponential function would require an exponential size for the representation of the output, but our iteration is bounded by a term $\mathcal{L}$ and the size of the number in our encoding is linear in its value[2]. Nevertheless, we can show how to compute an exponentiation, and so how to represent monomials and polynomials.

Definition 139 (Exponentiation)**.** Given a $k \in \mathbb{N}$, the function computing the k-th exponentiation of such value is defined as follows:

$$\begin{aligned} @0(x, \omega) &:= \underline{1}_{\mathbb{N}}; \\ @(n+1)(x, \omega) &:= mult(@n(x), x, \omega). \end{aligned}$$

[2]We prove such result in Lemma 10

Notice that all these functions, which are clearly in $\mathcal{POR}$, behave as desired. Indeed the following claims are easily verifiable:

- For any $n, m \in \mathbb{N}$ and $\omega \in \mathbb{O}$, $succ(\underline{n}_{\mathbb{N}}, \omega) = \underline{n+1}_{\mathbb{N}}$;
- For any $n \in \mathbb{N}^+$ and $\omega \in \mathbb{O}$, $pd(\underline{n}_{\mathbb{N}}, \omega) = \underline{n-1}_{\mathbb{N}}$;
- For any $n, m \in \mathbb{N}$, and $\omega \in \mathbb{O}$, $sum(\underline{n}_{\mathbb{N}}, \underline{m}_{\mathbb{N}}, \omega) = \underline{n+m}_{\mathbb{N}}$;
- For any $n, m \in \mathbb{N}$ and $\omega \in \mathbb{O}$, such that $n \geq m$, $diff(\underline{n}_{\mathbb{N}}, \underline{m}_{\mathbb{N}}, \omega) = \underline{n-m}_{\mathbb{N}}$;
- For any $n, m \in \mathbb{N}$ and $\omega \in \mathbb{O}$, $mult(\underline{n}_{\mathbb{N}}, \underline{m}_{\mathbb{N}}, \omega) = \underline{n \cdot m}_{\mathbb{N}}$;
- For any $n, m \in \mathbb{N}$ and $\omega \in \mathbb{O}$, $@m(\underline{n}_{\mathbb{N}}, \omega) = \underline{n^m}_{\mathbb{N}}$.

Corollary 16 (Polynomials are in $\mathcal{POR}^-$)**.** *For each polynomial $p : \mathbb{N}^k \longrightarrow \mathbb{N}$ there is a function $\underline{p} : \mathbb{S}^k \times \mathbb{O} \longrightarrow \mathbb{S}$ such that for each $x_1, \dots, x_k \in \mathbb{N}$, and for each $\omega \in \mathbb{O}$, $\underline{p}(\underline{x_1}_{\mathbb{N}}, \dots, \underline{x_k}_{\mathbb{N}}, \mathbb{O}) = \underline{p(x_1, \dots, x_k)}_{\mathbb{N}}$.*

Proof. $\mathcal{POR}^-$ is closed under composition ad it contains multiplications and sums. □

6.1.2.4 Lists

In the previous Section, we reduced all the composed data structures to the case of lists. For this reason, it comes natural to develop a tool-set of functions which can be used to manipulate lists, which will be used as building-blocks to define the function simulator. We start showing that the list member representation can be decoded in $\mathcal{POR}^-$, namely that $halv \in \mathcal{POR}^-$.

Lemma 27. *There is a function halv in $\mathcal{POR}^-$ such that $\forall x \in \mathbb{S}.\forall \omega \in \mathbb{O}.halv(x, \omega) = \mathcal{H}(x)$*

Proof. It can be shown giving the following definition:

$$\begin{aligned} halv(\boldsymbol{\epsilon}, \omega) &:= \boldsymbol{\epsilon}; \\ halv(y\mathtt{0}, \omega) &:= concat\big(halv(y, \omega), \mathtt{if}(\mathtt{0}, \boldsymbol{\epsilon}, odd(y, \omega), \omega)\big)|_{y\mathtt{0}}; \\ halv(y\mathtt{1}, \omega) &:= concat\big(halv(y, \omega), \mathtt{if}(\mathtt{1}, \boldsymbol{\epsilon}, odd(y, \omega), \omega)\big)|_{y\mathtt{0}}. \end{aligned}$$

□

Moreover, we ought show that it's the left-inverse of *doub* (Lemma 26), as stated by the following lemma.

Lemma 28 (Left-Inverse of *doub*)**.** $\forall \sigma \in \mathbb{S}, \omega \in \mathbb{O}.halv\big(doub(\sigma, \omega), \omega\big) = \sigma.$

Proof. The proof is by (right) induction on σ.

$\boldsymbol{\epsilon}$. The thesis comes from a trivial rewriting of the two functions' bodies.

τc. The thesis is:

$$\begin{aligned} halv\big(doub(\tau c, \omega), \omega\big) &= \tau c; \\ halv\big(doub(\tau, \omega)\mathtt{1}c, \omega\big) &= \tau c. \end{aligned}$$

By induction on σ we can also prove that $\forall \sigma.odd\big(doub(\sigma)\big) = \mathtt{0}$. So we can simplify our claim as follows:

$$halv\big(doub(\tau, \omega)\mathtt{1}c, \omega\big) = \tau c;$$

$$\begin{aligned} halv\big(doub(\tau,\omega)\mathtt{1},\omega\big)c &= \tau c; \\ halv\big(doub(\tau,\omega)\mathtt{1},\omega\big) &= \tau. \end{aligned}$$

We argued that $\forall\sigma.odd\big(doub(\sigma)\big) = \mathtt{0}$. So, we can state the claim as:

$$\begin{aligned} halv\big(doub(\tau,\omega)\mathtt{1},\omega\big) &= \tau; \\ halv\big(doub(\tau,\omega),\omega\big) &= \tau. \end{aligned}$$

which is the IH.

□

Thanks to the previous result, we can leverage *halv* to define list appenders, extractors and removers.

Definition 140 (List Extractors, Appenders and Removers)**.** We define lists' manipulator

- *rel*, i.e. *Right Extractor for Lists*;
- *lel*, i.e. *Left Extractor for Lists*;
- *ral*, i.e. *Right Appender for Lists*;
- *lal*, i.e. *Left Appender for Lists*;
- *rrl*, i.e. *Right Remover for Lists*;
- *lrl*, i.e. *Left Remover for Lists.*

and the auxiliary functions

- *rmsep*, i.e. the functions which removes an encoded character.
- *rel′*, i.e. the functions which returns the rightmost element of a list (size included).
- *srrl*, i.e. the right semi-remover, which given a string shaped as $\sigma\mathtt{00}\tau\mathtt{00}$ returns $\sigma\mathtt{00}$ where σ is as long as possible. This function is dual with respect to *rel*.

Starting from the auxiliary functions, we define them as:

$$\begin{aligned} rmsep(x,\omega) &:= rrs(rrs(x,\omega),\omega) \\ rel'(\boldsymbol{\epsilon},\omega) &:= E(x,\omega) \\ rel'(y\mathtt{0},\omega) &:= \mathtt{if}(\boldsymbol{\epsilon}, concat(rel'(y,\omega),\mathtt{0}), (\neg res(y,\omega)) \wedge odd(y,\omega),\omega)|_{y\mathtt{0}} \\ rel'(y\mathtt{1},\omega) &:= concat(rel'(y,\omega),\mathtt{1},\omega))|_{y_0} \\ srrl'(\boldsymbol{\epsilon},\omega) &:= E(x,\omega) \\ srrl'(y\mathtt{0},\omega) &:= \mathtt{if}(y\mathtt{0}, srrl'(y,\omega), \neg(res(y,\omega)) \wedge odd(y,\omega),\omega)|_{y\mathtt{0}} \\ srrl'(y\mathtt{1},\omega) &:= srrl'(y,\omega)|_{y_0} \\ srrl(x,\omega) &:= srrl'(rmsep(x,\omega),\omega). \end{aligned}$$

Thus, leveraging the auxiliary functions, we can define thour function as follows:

$$\begin{aligned} rel(t,\omega) &:= halv(rel'(rmsep(t),\omega),\omega) \\ lel(x,\omega) &:= reverse(rel(reverse(x,\omega),\omega),\omega) \end{aligned}$$

$$\begin{aligned}
\mathit{ral}(x,y,\omega) &:= \mathit{srrl}(x,\omega)\mathtt{00}\mathit{doub}(y,\omega)\mathtt{00}\mathit{doub}(\mathit{rel}(x,\omega)\mathtt{1},\omega)\mathtt{00}\\
\mathit{lal}(x,y,\omega) &:= \mathtt{00}\mathit{doub}(y,\omega)\mathit{srrl}(y,\omega)\mathit{doub}(\mathit{rel}(x,\omega)\mathtt{1},\omega)\mathtt{00}\\
\mathit{rrl}(x,\omega) &:= \mathit{srrl}(\mathit{srrl}(x,\omega),\omega)\mathit{doub}(\mathit{rrs}(\mathit{rel}(x,\omega),\omega)\omega)\mathtt{00}\\
\mathit{lrl}(x,\omega) &:= \mathit{srrl}(\mathit{reverse}(\mathit{srrl}(\mathit{reverse}(x,\omega),\omega),\omega)\mathit{doub}(\mathit{rrs}(\mathit{rel}(x,\omega),\omega)\omega)\mathtt{00}.
\end{aligned}$$

Remark 19 (Correctness of lists operators)**.** *Lists operators of Definition 140 follow their intended specification, or the described ones for the auxiliary functions.*

Proof. Respectively:

rmsep Trivial, comes from the definition od *rrs*.

rel' Suppose that y encodes a list, i.e. $y = \{\mathtt{10},\mathtt{11}\}^*\mathtt{00}\{\mathtt{10},\mathtt{11}\}^*$. If the input is $y\mathtt{0}$, we continue by cases on the guard of the `if` function: $\neg(\mathit{res}(y,\omega))$ entails that $y = y'\mathtt{0}$, so the input of the function is $y'\mathtt{00}$ finding two consecutive `0`s means that we reached the separator, indeed the output is the same of rel' is its input. If the guard is not true, the `0` at the end of the input is the second character of an encoded `0` in the rightmost value; even if the input is $y\mathtt{1}$, the current character is the encoding of a value, so we get the claim by induction.

srrl' We have already argued that this function is the dual of the rel' function. The proof is a slight variation of the previous.

srrl Trivial by the proof of the correctness of srrl'

rel The claim is a consequence of the correctness of rel' and of Lemma 28.

lel The claim comes from the fact that the definition of *rel* checks that a sequence of two characters is found and that the current character is in even position (the remaining part of the encoding has odd length) and that

- By induction on σ we can show that $\forall\sigma.\mathit{doub}(\sigma)$ has odd length.
- If σ is the encoding of a list, the separators `00` are always followed by an even number of characters. This can be show by induction on the number of elements of the list.

ral, *lal* The correctness comes from the definition of lists' encoding and from the correctness of all the functions which appear in the definitions of *ral* and *lal*.

rrl The correctness comes from the definition of lists' encoding and from the correctness of all the functions composed in the definition of this function.

lrl The correctness comes from the definition of lists' encoding and from the fact that *srrl*, as *rel* does, is safe with respect to the reversing of a string.

□

Thanks to these basic manipulators, we have a sufficiently expressive tool-set of functions, which enable us to show that it is possible to define list parametric and constant list projectors, namely functions $\pi_k : \mathbb{S} \times \mathbb{O} \longrightarrow \mathbb{S}$ for $k \in \mathbb{N}$, which given an encoding of a list and any oracle, they return the k-th element of that list if it exists, ϵ otherwise.

Together with these constant projectors, we can also define a *parametric* projector, in which the index of the projection is given as an input.

Definition 141 (List-Projectors)**.** A family of *constant projectors*, π_n, fo $n \in \mathbb{N}$ is defined as follows:

$$\pi_n(x, \omega) := rel(sa_{rrll,x}(x, \underline{n}_{\mathbb{N}}, \omega), \omega).$$

A family of *parametric projector*, π, for $n \in \mathbb{N}$ is defined as follows:

$$\pi(x, n, \omega) := rel(sa_{rrl,x}(x, n, \omega), \omega).$$

Remark 20. *Constant and parametric projectors with index $i > 0$ are correct with respect to their intended specification. Moreover, π_0 returns the number of elements in the list and for indexes of projections j greater than the number of elements in the list, π_j, returns $\boldsymbol{\epsilon}$.*

Proof. Correctness is a consequence of the correctness of sa, rrl and rel functions which has already been proven. □

6.2 Auxiliary Results

6.2.1 The Theory RS^1_2

Remark 21. *For each countable set A, the cylinder measure μ is a function.*

Proof. Assume $X \subseteq \{\mathtt{0}, \mathtt{1}\}^A$, and that:

$$X = \{\omega \in \{\mathtt{0}, \mathtt{1}\}^A \mid \omega|_{K_1} \in H_1\}$$
$$X = \{\omega \in \{\mathtt{0}, \mathtt{1}\}^A \mid \omega|_{K_2} \in H_2\},$$

we proceed by cases on the identity of the K_i and H_i for $i \in \{1, 2\}$.

- If the K_is and the H_is are identical, the conclusion is trivial.
- Suppose $K_1 = K_2$, while $H_1 \neq H_2$. This wound entail $X \neq X$, which is absurd.
- Suppose $K_1 \neq K_2$, while $H_1 = H_2$. This is absurd because $H_i \subseteq \{\mathtt{0}, \mathtt{1}\}^{K_i}$.
- Finally, we proceed by double induction on the elements of $K_1 \setminus K_2$ and $K_2 \setminus K_1$. If such value is 0, we get the claim, otherwise, suppose that there are $n + 1$ one such elements. Without loss of generality, call that element k and assume it in K_1. Let $K_2' = K_2 \cup \{k\}$ and
  $$H_2' = \{w \in \{\mathtt{0}, \mathtt{1}\}^{K_2'} \mid w|_{K_2} \in H\}.$$
  It holds that:
  $$\frac{|H_2'|}{2^{|K_2|}} = \frac{2 \cdot |H_2|}{2^{|K_2+1|}} = \frac{2 \cdot |H_2|}{2 \cdot 2^{|K_2|}}$$
  which proves the claim applying the IH.

□

Remark 22 (Cylinder measure)**.** *For each countable set A, μ is a measure function.*

Proof. First, we show that μ is defined on all the elements of the field on $\{\mathtt{0}, \mathtt{1}\}^A$.

- If X is a cylinder, then the conclusion is trivial.

- If such element is the union of two elements of the field X_1 and X_2, then μ is defined on them, so they can be expressed as cylinders. So they are two elements defined as
$$X_1 = \{\omega \in \{\texttt{0}, \texttt{1}\}^A \mid \omega|_K \in H\}$$
$$X_2 = \{\omega \in \{\texttt{0}, \texttt{1}\}^A \mid \omega|_{K'} \in H'\}.$$
Thus we get that:
$$X = X_1 \cup X_2 = \{\omega \in \{\texttt{0}, \texttt{1}\}^A \mid \omega|_{K\cup K'} \in H''\},$$
with
$$H'' := \{w \in \{\texttt{0}, \texttt{1}\}^{K\cup K'} \mid w|_K \in H \vee w|_{K'} \in H'\}.$$
So, μ is defined on X.
- Finally, if X is obtained by complementation, we get that:
$$X = \mathbb{B}^A \setminus X' = \{\omega \in \{\texttt{0}, \texttt{1}\}^A \mid \omega|_K \notin H'\} = \{\omega \in \{\texttt{0}, \texttt{1}\}^A \mid \omega|_K \in \mathbb{B}^K \setminus H'\},$$
which is itself a cylinder, so μ is defined on it.

Now we show that μ is a probability measure over the field generated by $\{\texttt{0}, \texttt{1}\}^A$.

- It is trivial to see that $0 \leq \mu(X) \leq 1$ for every cylinder X.
- Similarly, it is easily possible to verify that $\mu(\emptyset) = 0$ and that $\mu(\{\texttt{0}, \texttt{1}\}^A) = 1$.
- Finally, suppose that $A_1, A_2, \dots$ is a sequence of pairwise disjoint sets such that $A = \bigcup_{i=1}^{\infty} A_i$ is a cylinder, then A can be expressed as follows:
$$A = \{\omega \in \{\texttt{0}, \texttt{1}\}^A | \omega|_K \in \bigcup_{i=1}^{\infty} H_i\},$$
with $H = \bigcup_{i=1}^{\infty} H_i$ finite. Moreover, all the H_is are finite and pairwise disjoint. So
$$\mu(A) = \frac{|\bigcup_{i=1}^{\infty} H_i|}{2^{|K|}} = \frac{\sum_{i=1}^{\infty} |H_i|}{2^{|K|}} = \sum_{i=1}^{\infty} \frac{|H_i|}{2^{|K|}} = \sum_{i=1}^{\infty} \mu(H_i).$$

Finally, applying [5, Theorem 3.1], we get the claim. □

6.2.2 The Class SFP

Proof of Proposition 4. The proof is by induction on n.

$n = 0$. Then, $\triangleright_M^n$ is the identity function.

$n + 1$. By IH $\triangleright_M^n$ is a function. Since $\vdash_\delta$ is a function, $\triangleright_M^{n+1} = \triangleright_M^n \circ \vdash_\delta$ is a function too. □

6.2.3 From $\mathcal{PTF}$ to $\mathcal{POR}^-$

Lemma 29. *If γ is a function $\mathbb{S} \longrightarrow \mathbb{S}$ such that $|\gamma| = n$, define $x_\gamma = \langle\langle x_1, y_1\rangle_{\mathbb{L}}^2, \dots, \langle x_n, y_n\rangle_{\mathbb{L}}^2\rangle_{\mathbb{L}}^n$, then $sim(x, x_\gamma, \omega) = \gamma(x)$, otherwise $sim(x, x_\gamma, \omega) = \boldsymbol{\epsilon}$.*

Proof. Let $\omega \in \mathbb{O}$ any oracle. Suppose x' is in γ's domain. It means that there is a tuple $t = \langle x', \overline{y}\rangle^2_{\mathbb{L}}$ among the elements of x_γ, so there is $k \in \mathbb{N}$ such that $t = \pi_k(x_\gamma, \omega)$. Thus, we can prove that if the third argument of *sim*$'$ is the number of elements in t, then $\forall k \leq m < \pi_0(t, \omega).\mathit{sim}'(x_\gamma, x, \underline{m}_{\mathbb{N}}, \omega) = \overline{y}$; this result can be established by induction on m. Leveraging the fact that γ is a function, so in its graph there cannot be two pairs with the same first projection. Then, the correctness of *sim* comes as consequence because it is an instance of *sim*$'$ with a value which is greater or equal to any possible value of k, namely the number of pairs in x_γ. Now suppose that x' is not in γ's domain, then there is not a k such that $\pi_k(x_\gamma) = \langle x', \overline{y}\rangle^2_{\mathbb{L}}$, so *sim*$'$ returns $\boldsymbol{\epsilon}$. □

Definition 142 (Apply)**.** The function $\mathit{apply}(\cdot, \cdot, \omega) \in \mathcal{POR}^-$ is defined as follows:

$$\mathit{apply}(x_\delta, x_c, \omega) := \mathtt{if}\big(x_c, g(x_c, x_\delta, \omega), \neg(\mathit{eq}(\xi(x_c, x_\delta, \omega), \boldsymbol{\epsilon}, \omega), \omega), \omega\big),$$

where:

$$g(x, y, \omega) = \mathtt{if}\big(f_1(x, y, \omega), f_2(x, y, \omega), \mathit{eq}(\xi(x, y, \omega), \mathtt{0}, \omega), \omega\big)$$

$$\begin{aligned} f_1(x, y, \omega) = \langle &\chi(\mathit{rrl}(\pi_1(x, \omega), \omega), \omega), \pi_3(\xi(x, y, \omega), \omega), \\ &\mathit{lal}(\mathit{lal}(\mathit{lrl}(\pi_3(x, \omega), \omega), \pi_1(\xi(x, y, \omega)), \omega), \mathit{rel}(\pi_1(x, \omega), \omega), \omega) \\ &\mathit{lrl}(\pi_4(x, \omega), \omega)\rangle_{\mathbb{L}} \\ f_2(x, y, \omega) = \langle &\mathit{ral}(\mathit{rrl}(\pi_1(x, \omega), \omega), \pi_1(\xi(x, y, \omega), \omega), \omega), \pi_3(\xi(x, y, \omega), \omega), \\ &\chi(\mathit{lrl}(\pi_3(x, \omega), \omega), \omega), \mathit{lrl}(\pi_4(x, \omega), \omega)\rangle_{\mathbb{L}} \end{aligned}$$

$$\begin{aligned} \chi(x, \omega) &= \mathtt{if}\big(\underline{\circledast}_{\mathbb{T}}, x, \mathit{eq}(\pi_0(x, \omega), \underline{0}_{\mathbb{N}}, \omega), \omega\big) \\ \xi(x, y, \omega) &= \mathit{sim}\big(y, \langle \pi_2(x, \omega), \mathit{lel}(\chi(\pi_3(x, \omega), \omega), \omega), \mathit{lel}(\chi(\pi_4(x, \omega), \omega), \omega)\big). \end{aligned}$$

Lemma 4 (Correctness of *apply*)**.** *The function apply is such that for any FSTM M with transition function δ, said x_δ and $h(c)$ respectively, the given encodings of δ and the encoding of a configuration c,*[3]

$$\forall \omega \in \mathbb{O}. \vdash_\delta (c) = d \to \mathit{apply}(x_\delta, h(c), \omega) = h(d);$$
$$\forall \omega \in \mathbb{O}. (c \not\vdash_\delta) \to \mathit{apply}(x_\delta, h(c), \omega) = h(c).$$

Proof. It should not be too much of a problem to see that the function χ return a tape which contains an instance of the blank character if and only if the tape passed as argument is empty, otherwise it returns its argument. Thanks to this observation, we prove that ξ is defined as the invoking of *sim* on:

1. The machine's transition function δ;
2. The tuple containing:
   (a) The current state $\pi_2(x_c, \omega)$;
   (b) The current character $\mathit{lel}(\chi(\pi_3(x_c, \omega), \omega), \omega)$;
   (c) The first character on the oracle tape, i.e. the *random* bit $\mathit{lel}(\chi(\pi_3(x_c, \omega), \omega), \omega)$.

[3]These encodings exist as a consequence of Corollaries 14 and 15. In what follows, the functions thereby described will be considered the canonical encoding of FSTM transition functions and configurations within $\mathcal{POR}$.

This call to the *sim* function returns either the encoding of the image of this inputs through δ (if defined), or $\boldsymbol{\epsilon}$ otherwise, as a consequence of Lemma 29. Suppose that the result of ξ is $\boldsymbol{\epsilon}$, then the claim is trivially true, since the function returns exactly its input, as required by the claim. Otherwise, suppose that ξ returns a value different from $\boldsymbol{\epsilon}$. It means that δ is defined on its input, so, the guard of the inner `if` expression checks whether the result describes a right or left movement of the head and, depending on the response, behaves differently. Observe that the constant L is represented by `0` as stated in Definition 128. We will show only the case in which the head moves right, for the left movement the proof is analogous. The configuration is made up as follows:

1. The left portion of the tape tape $ral(rrl(\pi_1(x_c,\omega),\omega),\pi_1(\chi(x_c,x_\delta,\omega),\omega))$ is obtained removing the current character by means of $rrl(\pi_1(x_c,\omega),\omega)$, appending the character which is written by the head $\pi_1(\chi(x_c,x_\delta,\omega),\omega)$ by means of *ral*.
2. The current state is obtained projecting the third element of χ.
3. The right portion of the work tape tape loses its leftmost element, becoming $\underline{\circledast}_{\mathbb{T}}$ if empty, as described by $\chi(lrl(\pi_3(x_c,\omega),\omega),\omega)$.
4. The second tape loses its leftmost character.

The function χ prevents to obtain empty list as encodings of a tape, as required by Definition 128. □

Lemma 5. *For each $f : \mathbb{S}^{k+1} \times \mathbb{O} \longrightarrow \mathbb{S} \in \mathcal{POR}$, if there is a term $t \in \mathcal{L}$ such that $\forall x, \vec{z}, \omega. f(x,\vec{z},\omega)|_t = f(x,\vec{z},\omega)$ then there is also a function $sa_{f,t} : \mathbb{S}^{k+2} \times \mathbb{O} \longrightarrow \mathbb{S}$ such that:*

$$\forall n \in \mathbb{N}.\forall x \in \mathbb{S}, \omega \in \mathbb{O}.sa_{f,t}(x,\underline{n}_{\mathbb{N}},\vec{z},\omega) = \underbrace{f(f(f(x,\vec{z},\omega),\vec{z},\omega),\ldots)}_{n\ times}.$$

Proof. Given $f \in \mathcal{POR}^-$ and $t \in \mathcal{L}_{\mathbb{PW}}$, let $sa_{f,t}$ be defined as follows:

$$sa'_{f,t}(x,\boldsymbol{\epsilon},\vec{z},\omega) := x$$
$$sa'_{f,t}(x,y\mathtt{b},\vec{z},\omega) := f\big(sa'_{f,t}(x,y,\omega),\vec{z},\omega\big)|_t$$

$$sa_{f,t}(x,y,\vec{z},\omega) := sa'_{f,t}(x,rrs(y,\omega),\vec{z},\omega).$$

The function sa is correct as $\underline{n}_{\mathbb{N}}$ has size $n+1$. We prove that if $|y| = n$, then $sa'_{f,t}(x,y,\vec{z},\omega) = f(f(f(\vec{x},\vec{z},\omega),\vec{z},\omega)\ldots)$, nested n times, by induction on n.

- If $n = 0$, $sa_{f,t}$ reduces to $sa'_{f,t}$ with argument ϵ, so the result is x.
- $n+1$. By applying IH and the definition of $sa'_{f,t}$.

The correctness of sa comes as a consequence of the correctness of the function rrs (which removes the rightmost digit of a string) and of the definition of sa.

□

Lemma 6 (*apply* Size Growth)**.** *For all $x_c \in \mathbb{S}$ being the encoding of an FSTM configuration and for each encoding of a transition function δ in a string $x_\delta \in \mathbb{S}$ and for each $\omega \in \mathbb{O}$, there is a $k \in \mathbb{N}$ such that $apply(x_c,x_\delta,\omega)|_{x_c\mathtt{1}^k}$.*

Proof. At each step, the function *apply* manipulates three values:

- The portion of the tape on the left of the head;
- The portion of the tape on the right of the head;
- The current state.

The manipulation of the tapes is done through shifting of characters and rewriting. An ordinary shifting operation does not change the overall size of the encoded configuration, while the rewriting can cause such phenomenon, due to the different sizes of the encoded characters, but fixed a machine, the maximum size of its characters is fixed. Say that value $k_1 \in \mathbb{N}$. Replacing a character on the encoded tape with it takes less than $2 \cdot k_1$ bits. Even if a tape contains an infinite sequence of $\circledast$ and the machine moves in that direction, the overall result is the appending of a character on the other portion of the tape this takes exactly $2 \cdot k_1$ bits, plus 2 additional bits to store the new size of the portion of the tape which grows by one. Then, we must take in account the difference in size due to the new state, but since the numbers of state in a machine is fixed, there is a constant k_2 which bounds the size of any state, so rewriting a state takes less than $2 \cdot k_2$ additional bits. Finally, we must take in account the representation of the second tape: at each step, we strip off a cell from its representation since the head moves always on the right, so it causes no term growth. The value of k we are looking for is $2 \cdot (1 + k_1 + k_2)$.

□

Lemma 30. *The $\mathcal{POR}$ function dectape* : $\mathbb{S} \longrightarrow \mathbb{S}$ *is such that if* $\underline{\cdot}_{\mathbb{T}}$ *is the encoding for tapes proposed in Definition 128, then it holds that* $\forall \sigma \in \{\mathtt{0}, \mathtt{1}, \circledast\}^*, \omega. f(\underline{\sigma}_{\mathbb{T}}, \omega) = \tau$ *and* τ *is the longest suffix of* σ *without* $\circledast$.

Proof of Lemma 30. Let us consider the function *dectape*(x, ω), described in Definition 64. If $x = \underline{\sigma}_{\mathbb{T}}$ is the encoding of a tape $\sigma \in \hat{\Sigma}^*$, $y = \underline{n}_{\mathbb{N}}$ and $n \leq |\sigma|$, so $\rho(x, y, \omega)$ returns the n-th projection of σ from the right. This is a consequence of the correctness of the function used in the definition of ρ, which we have already discussed. In order to show the correctness of *dectape*, we prove that *dectape*$'$ is a generalization of *dectape*, for which if $y = \underline{n}_{\mathbb{N}}$ with $n > 0$, and $\underline{\sigma}_{\mathbb{T}} = x$, then *dectape*$'(x, y, \omega)$ is the longest subfix without $\circledast$, of σ's n-th prefix. The proof is by induction on n.

- Case 0. In this claim we have with the condition $n > 0$.
- Case $n + 1$. According to the IH, the recursive call returns the longest suffix of the n-th prefix of the string. Form the consideration above, thus In this case, intuitively, ρ checks the $n + 1$-th character from the left of the encoding for $\sigma = \sigma' c$. If the character c is $\circledast$, its encoding will be `111`, so the condition in the guard of `if` is verified and the whole expression reduces to $\boldsymbol{\epsilon}$, which is indeed the longest suffix of $\sigma'\circledast$ without $\circledast$. Otherwise, the function returns the decoding of the *dectape*$'(x, y, \omega) = \tau$, which is the longest suffix of σ' without $\circledast$, thus the function returns τc. This string is free from $\circledast$ as a consequence of the IH and is also the longest suffix of σ with this property: otherwise, τ would not be the longest suffix of σ' free from $\circledast$, contradicting the IH.

We conclude the proof by summing the consideration above and the fact that the definition of *dectape* is defined as an instance of *dectape*$'$ with x equal to the size of σ. So, the substring returned by *dectape* is actually a suffix. □

6.2.4 Extractor Function in $\mathcal{POR}$

Lemma 31. *The function $dy(\underline{n}_{\mathbb{N}}, \omega)$ is bijective with respect to its first argument.*

Proof. By the definition of the function, we know that it is in $\mathcal{POR}^-$, so the function is constant with respect to its second parameter. Moreover, it is clearly an injection because different numbers have different binary encodings and $\forall n > 0 \in \mathbb{N}.\forall\omega \in \mathbb{O}.bin(n, \omega)$ has $\mathtt{1}$ as leftmost bit (it can be shown by induction on n, leveraging the definition of bin).

So if we take two distinct binary encodings of natural numbers n, m and call them $\mathtt{1}\sigma$ and $\mathtt{1}\tau$. It must hold that $\sigma \neq \tau$, otherwise we would have $n = m$.

So we just need to show that it is surjective. We know that dy is computed removing a bit which is always $\mathtt{1}$. This entails that taken a string $\sigma \in \mathbb{S}$, it is the image of the natural number n such that the binary encoding of $n + 1$ is $\mathtt{1}\sigma$. This number always exist. □

Lemma 32 (Correctness of e)**.** $\forall i \in \mathbb{N}.\forall j \leq i.\forall\omega \in \mathbb{B}^{\mathbb{S}}.e(\underline{i}_{\mathbb{N}}, \omega)(j) = \omega(dy(\underline{j}_{\mathbb{N}}, \omega))$ *and the length of* $e(\underline{i}_{\mathbb{N}}, \omega)$ *is exactly* $i + 1$.

Proof. By induction on i:

0 For the first claim we have that:

$$e(\mathtt{1}, \omega)(0) = \boldsymbol{\epsilon}Q\big(dy(\mathtt{1}, \omega), \omega\big) = \omega(dy(\mathtt{1}, \omega)).$$

While, the second claim is trivial.

i+1 By the IH on the second claim, we get that for $j = i + 1$, the j-th element of $e(\underline{i+1}_{\mathbb{N}}, \omega)$ is exactly $Q\big(dy(\underline{i+1}_{\mathbb{N}}, \omega), \omega\big)$, which is equal to $\omega(dy(\underline{i+1}_{\mathbb{N}}, \omega))$. For smaller values of j, the first claim is a consequence of the definition of e and the IH. The second claim is trivial.

□

Lemma 33.

$$\eta \sim_{dy} \omega \to \forall n \in \mathbb{N}.\eta_n = e(\underline{n}_{\mathbb{N}}, \omega).$$

Proof. By contraposition: suppose that the consequence does not hold:

$$\eta_n \neq e(\underline{n}_{\mathbb{N}}, \omega).$$

As a consequence of the correctness of e (Lemma 32), this means that there is an $i \in \mathbb{N}$ such that $\eta(i) \neq \omega(dy(\underline{i}_{\mathbb{N}}, \omega))$, which is a contradiction. □

6.2.5 The SIFP language

Lemma 34. *The relation $\rightharpoonup$ of Definition 76 is a function* $\rightharpoonup: \mathcal{L}(\mathsf{Exp}) \times (\mathsf{Id} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^*) \times \mathbb{O} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^*$.

Proof. By induction on the syntax of the expression, we have the following cases:

- If the expression is $\boldsymbol{\epsilon}$, a bit concatenation or an identifier, we can apply only a single rule.
- If the expression is a predicate, two rules can be applied to the expression, but in all the cases, these rules are mutually exclusive.

□

Lemma 35. *The relation* $\triangleright$ *of Definition 77 is a function* $\mathcal{L}(\mathsf{Stm}_{\mathsf{RA}}) \times (\mathsf{Id} \longrightarrow \{0,1\}^*) \times \mathbb{O} \longrightarrow (\mathsf{Id} \longrightarrow \{0,1\}^*)$.

Proof. By induction on the syntax of the program P.

- If the statement is $\mathsf{Flip}(e)$, the consequence comes from the fact that ω is a function.
- If the statement is an assignment, the result is a consequence of Lemma 34.
- If the statement is $\cdot;\cdot$, the result comes from the IH.
- If the statement a **while**(){}, the result comes from the observation that the two rules are mutually exclusive, due to Lemma 34, and to the IH if the condition is true.

□

Lemma 36. *The relation* $\triangleright$ *of Definition 78 is a function* $\mathcal{L}(\mathsf{Stm}_{\mathsf{LA}}) \times (\mathsf{Id} \longrightarrow \{0,1\}^*) \times \mathbb{O} \longrightarrow (\mathsf{Id} \longrightarrow \{0,1\}^*)$.

Proof. By induction on the syntax of the program P.

- If the statement is $\mathsf{RandBit}()$, the consequence comes from the fact that b is an unique value.
- If the statement is an assignment, the result is a consequence of Lemma 34.
- If the statement is $\cdot;\cdot$, the result comes from the IH.
- If the statement a **while**(){}, the result comes from the observation that the two rules are mutually exclusive, due to Lemma 34, and to the IH if the condition is true.

□

6.2.6 From $\mathcal{POR}$ to the $\mathsf{SIFP}_{\mathsf{RA}}$ Language

Lemma 10. *The size of a term in* $\mathcal{L}$ *is poynomial in the size of its variables.*

Proof of Lemma 10. The proof is by induction on the production of the term.

- If the term is a digit or the empty string, it has no variables in it and its size is 1, which is a constant. All constants are polynomials with no variables, so the claim is proved.
- If the term is a variable, its size consists in the size of its variable, which is a polynomial in the size of the variables of the term.
- If the term is the concatenation of two terms $t \frown s$, its size is the sum of the sizes of its subterms, which are polynomials for the IH. Since the sum of polynomials is still polynomial, the union of the variables of t and s is polynomial in its turn.
- If the term is the product of two terms $t \times s$, the size of s is a polynomial p_s in the size of the variables in s, and the size of t is still polynomial p_t in the size of the variables in t. Hence, the size of the term $t \times s$ is given by $p_t p_s$, which is polynomial in the size of the variables in $t \times s$.

□

Lemma 11. *For each $f \in \mathcal{POR}$, the following holds:*

$$\forall x_1, \dots, x_n . \forall \omega . \exists p \in \mathsf{POLY} . |f(x_1, \dots, x_n, \omega)| \leq p(|x_1|, \dots, |x_n|).$$

Proof of Lemma 11. The proof is by induction on the structure of $f \in \mathcal{POR}$. Base cases:

- if f is E, we introduce the the constant 1.
- If f is P_i we introduce the polynomial $\sum_{i=0}^{n} |x_i| + 1$. It is easy to see that such value is greater or equal than the size of each input.[4]
- If f is C, we introduce the constant 1.
- If f is S_0 or S_1, we introduce the polynomial $|x| + 1$.
- If f is Q, we introduce the constant 1.

Inductive cases:

- Composition. Consequence of the IH

  $$\forall 1 \leq i \leq k . \exists p_i \in \mathsf{POLY}\big(h_i(x_1, \dots, x_n, \omega)| \leq p_i(|x_1|, \dots, |x_n|)\big).$$

  So, a similar bound q exists for the external function f. The composition of q with the sequence of polynomials p_i is still a polynomial and bounds the size of the composition of the function by IH.
- Bounded Recursion. By IH, we know that the size of g is bounded by a polynomial p_g in its inputs. Moreover, the size of the value computed by f in the inductive cases is polynomial in its input, as it is truncated to the size of a term in $\mathcal{L}$, whose size is polynomial in its variables (that are the inputs of f), by Lemma 10. Call this polynomial p_t. We introduce the polynomial $p_g + p_t$ Then, we proceed on induction on the length of the string τ that is passed as recursion bound. If such string has length 0, it is $\boldsymbol{\epsilon}$. So, the function f coincides with g, and the size of its output is smaller than p_g for induction hypothesis. Otherwise, the size of the value computed by f is polynomial in its input, as it is truncated to the size of a $t \in \mathcal{L}$, whose size is smaller than p_t by induction hypothesis

$\square$

Lemma 37 (Complexity of $\mathfrak{M}$)**.** *$\forall t \in \mathcal{L}. \mathfrak{M}_t$ can be computed in number of steps which is polynomial in the size of the variables in t with respect to Definition 84.*

Proof. We proceed by induction on the syntax of t.

ϵ If the term is ϵ, $\mathfrak{M}_t$ consists in two steps.

$\mathtt{0}, \mathtt{1}$ If the term is a digit, $\mathfrak{M}_t$ consists in two steps.

x If the term is a variable, $\mathfrak{M}_t$ consists in two steps.

$t \frown s$ From the Lemmas 13 and 12 we know that *copyb* requires a constant number of steps, and that each time *copyb* is executed, Z grows by one. Moreover, we know that the pseudo-procedure respects the invariant that Z is a prefix of S. For this reason, the complexity of the **while**( ){ } statement is linear in the size of Z. Finally, the complexity takes in account two polynomial due the recursive hypothesis on $\mathfrak{M}$ and two steps for the two assignments before the **while**( ){ }. The overall sum of these complexities is still a polynomial.

[4] Actually, we are over-killing the bound: introducing the polynomial $|x_i|$ for each P_i^n is sufficient.

$t \times s$ We will distinguish the three levels of **while**( ){ }s by calling them *outer*, *middle* and *inner*. For Lemmas 12 and 13, the inner **while**( ){ }s take at most $|T|$ steps, such value is a polynomial over the free variables of t according to Lemma 11 and the induction hypothesis. The value of T remains constant after its first assignment, for this reason all the inner cycles require a polynomial number of steps. The middle cycles implementing the `if` construct are executed only once for each outer cycle as stated in the definition of the `if` syntax in Notation 12. Moreover, they add a constant number of steps to the complexity of the inner cycles. This means that, modulo an outer cycle, the complexity is still polynomial. For the same argument of Lemma 12, the outer cycle takes at most $|S|$ steps which is a polynomial according to Lemma 11 and the induction hypothesis.

□

Definition 143 (Truncating pseudo-procedure)**.** The *trunc*(T, R) pseudo-procedure is a SIFP program with free names T and R, defined as follows:

```
$trunc(T, R) :=Q \leftarrow R;$
    $R \leftarrow \epsilon;$
    $Z \leftarrow \epsilon;$
    $Y_0 \leftarrow \epsilon;$
    while($Z \sqsubset T$){
      if($Z.0 \sqsubseteq T$){
        $copyb(R, Q, Y_0)$
        $Z \leftarrow Z.0;$
        }
      if($Z.1 \sqsubseteq T$){
        $copyb(R, Q, Y_0)$
        $Z \leftarrow Z.1;$
        }
    }
```

Lemma 38 (Complexity of truncation)**.** *The pseudo-procedure trunc requires a number of steps which is at most polynomial in the sizes of its free names with respect to Definition 84.*

Proof. By Lemma 12 we know that the pseudo-procedure *copyb* requires a constant number of steps. Furthermore, the cycles implementing the `if` are executed only once for each outer cycle as a consequence of to what stated ater the introduction of the `if` syntax in Notation 12. Finally the number of outer cycles is bounded by the $|T|$, so the whole complexity of the pseudo-procedure is polynomial (linear) in $|T|$. □

Lemma 39 (Correctness of truncation)**.** *The pseudo-procedure trunc truncates the register R to its $|T|$-th prefix.*

Proof. We proceed by induction on T.

ϵ Trivially we have $R = \epsilon$ since the cycle is not executed.

σb In this case, only one of the sub-cycles implementing the `if` is executed (they are mutually-exclusive), another bit of Q is stored in R according to Lemma 13, and Q is unchanged after the execution of *copyb*. Theses arguments prove the claim. The register Y has no practical implications since it's only used in order to leverage the lemmas on *copyb*.

$\square$

Lemma 40 (Complexity of $\mathsf{SIFP}_{\mathsf{RA}}$)**.** *$\forall f \in \mathcal{POR}.\mathfrak{L}_f$ takes a number of steps which is polynomial in the size of the arguments of f with respect to Definition 84.*

Proof. We proceed by induction on the proof of the fact that f is indeed in $\mathcal{POR}$. All the base cases (E, S_0,S_1, P^n_i, C,Q) are trivial. The inductive steps follow easily:

- In the case of composition, we know that the claim holds for all the pseudo-procedures $\mathfrak{L}_\cdot$. The program requires a finite number of assignments more, so its complexity is still polynomial.
- In the case of iteration we can move the same argument of Lemma 12 to prove that the outer cycle is executed only $|Z|$ times, which is a polynomial in an argument of the encoded function. Moreover, the implementation of the `if` construct relies on cycles, but as a consequence of the definition of the `if` syntax in Notation 12, they are executed only once for each outer cycle. So the claim comes from Lemmas 38, 37, from the fact that the composition of polynomials is still polynomial and from the IH.

$\square$

6.2.7 From $\mathsf{SIFP}_{\mathsf{RA}}$ to $\mathsf{SIFP}_{\mathsf{LA}}$

Proof of Characterization 1. In order to strengthen the IH, we will prove a stronger result:

$$\langle P, \Gamma, \eta\rangle \triangleright \langle \Sigma, \eta'\rangle \leftrightarrow \exists \Psi' \in adt(M).\langle P; \mathbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^*_{\mathsf{LA}} \langle \mathbf{halt};, \Sigma, \Psi'\rangle \wedge \Psi\eta = \Psi'\eta'.$$

This can be done by induction on the syntax of P.

$Id \leftarrow e$ In this case, since $\rightharpoonup$ is a function (Lemma 34), in both cases e reduces to the same string σ, thus the *big-step* semantics returns $\Gamma[Id \leftarrow \sigma]$, and

$$\langle Id \leftarrow e; \mathbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma[Id \leftarrow \sigma], \Psi\rangle.$$

The second part of the claim holds because Ψ and η do not change during the transition.

$\mathsf{RandBit}()$ In this case, the *big-step* semantics acts as follows:

$$\langle \mathsf{RandBit}(), \Gamma, \mathtt{b}\eta\rangle \triangleright \langle \Gamma[R \leftarrow \mathtt{b}], \eta\rangle$$

On the other hand:

$$\langle \mathsf{RandBit}(); \mathbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma[R \leftarrow \mathtt{1}], \Psi\mathtt{1}\rangle$$

and

$$\langle \mathsf{RandBit}(); \mathbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{LA}} \langle \mathbf{halt};, \Gamma[R \leftarrow \mathtt{0}], \Psi\mathtt{0}\rangle.$$

Moreover, it holds that $\Psi\mathtt{b}\eta = \Psi\mathtt{0}\eta \vee \Psi\mathtt{b}\eta = \Psi\mathtt{1}\eta$ proving the claim

$s;t$ Suppose that:
$$\langle s;t,\Gamma,\eta\rangle \rhd \langle \Sigma,\eta''\rangle.$$
According to the definition of $\rhd$, this means that:
$$\langle s,\Gamma,\eta\rangle \rhd \langle \Gamma',\eta'\rangle$$
and that:
$$\langle t,\Gamma',\eta'\rangle \rhd \langle \Sigma,\eta''\rangle.$$
For the IH, this is equivalent to:
$$\langle s;\mathbf{halt};,\Gamma,\Psi\rangle \rightsquigarrow^{*}_{\mathsf{LA}} \langle \mathbf{halt};,\Gamma',\Psi'\rangle.$$
and
$$\langle t;\mathbf{halt};,\Gamma',\Psi'\rangle \rightsquigarrow^{*}_{\mathsf{LA}} \langle \mathbf{halt};,\Sigma,\Psi''\rangle.$$
Thus, we proved that:
$$\langle s;t;\mathbf{halt};,\Gamma,\Psi\rangle \rightsquigarrow^{*}_{\mathsf{LA}} \langle \mathbf{halt};,\Sigma,\Psi''\rangle.$$
But also we know that:
$$\Psi\eta = \Psi'\eta' = \Psi''\eta''.$$

$\mathbf{while}(e)\{s\}$ We go by cases on the value of e in Γ. If it does not reduce to `1`, we have that:
$$\langle \mathbf{while}(e)\{s\},\Gamma,\eta\rangle \rhd \langle \Gamma,\eta\rangle \wedge \langle \mathbf{while}(e)\{s\};\mathbf{halt};,\Gamma,\Psi\rangle \rightsquigarrow^{1}_{\mathsf{LA}} \langle \mathbf{halt};,\Gamma,\Psi\rangle.$$
So, the conclusion is trivial. Otherwise, if e reduces to `1`, we have that:
$$\langle s,\Gamma,\eta\rangle \rhd \langle \Gamma',\eta'\rangle \wedge \langle \mathbf{while}(e)\{s\},\Gamma',\eta'\rangle \rhd \langle \Sigma,\eta''\rangle.$$
By induction hypothesis, we know that:
$$\langle s;\mathbf{halt};,\Gamma,\Psi\rangle \rightsquigarrow^{*}_{\mathsf{LA}} \langle \mathbf{halt};,\Gamma',\Psi'\rangle$$
and that:
$$\langle \mathbf{while}(e)\{s\};\mathbf{halt};,\Gamma',\Psi'\rangle \rightsquigarrow^{*}_{\mathsf{LA}} \langle \mathbf{halt};,\Sigma,\Psi''\rangle.$$
Moreover, we also know that:
$$\Psi\eta = \Psi'\eta' \wedge \Psi'\eta' = \Psi''\eta''$$
But, since e reduces to `1`, we know that:
$$\langle \mathbf{while}(e)\{s\};\mathbf{halt};,\Gamma,\Psi\rangle \rightsquigarrow^{1}_{\mathsf{LA}} \langle s;\mathbf{while}(e)\{s\};\mathbf{halt};,\Gamma,\Psi'\rangle.$$
Summing together these three last conclusions, we get the claim we where aiming to.

We will not show the other direction, which is analogous, apart from the fact that it requires to observe that for every $h, h \in \mathbb{N}$:
$$\langle P,\Sigma,\Psi\rangle \rightsquigarrow^{h}_{\mathsf{LA}} \langle P',\Sigma',\Psi'\rangle \wedge \langle P,\Sigma,\Psi\rangle \rightsquigarrow^{h+k}_{\mathsf{LA}} \langle P'',\Sigma'',\Psi''\rangle$$
entails that $\Psi' \subseteq \Psi''$. The final result comes instantiating the universal quantifiers with the initial values as for Definition 79. In particular, we get $\eta \succ \Psi$, because $\boldsymbol{\epsilon}\eta = \Psi\eta'$ for some η', thus Ψ is a prefix of η. □

Proof of characterization 2. In order to strengthen the IH, we will prove a stronger result: if $\omega \succ \Psi$, then

$$\langle P, \Gamma, \omega\rangle \rhd \Sigma \leftrightarrow \exists \Psi' \in adt(M).\langle P; \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \textbf{halt};, \Sigma, \Psi'\rangle \wedge \omega \succ \Psi'.$$

This can be done by induction on the syntax of P.

$Id \leftarrow e$ In this case, since $\rightharpoonup$ is a function (Lemma 34), in both cases e reduces to the same string σ, thus the *big-step* semantics returns $\Gamma[Id \leftarrow \sigma]$, and

$$\langle Id \leftarrow e; \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{RA}} \langle \textbf{halt};, \Gamma[Id \leftarrow \sigma], \Psi\rangle.$$

The second part of the claim holds because Ψ is unchanged by the transition.

$\mathsf{Flip}(e)$ Even in this case, since $\rightharpoonup$ is a function, the *big-step* semantics acts as the *small-step* semantics acts, in particular, suppose that $\langle e, \Gamma\rangle \rightharpoonup \sigma$:

$$\langle \mathsf{Flip}(e), \Gamma, \omega\rangle \rhd \Gamma[R \leftarrow \omega(\sigma)].$$

On the other hand, supposing $\forall b \in \{\mathtt{0}, \mathtt{1}\}, (\sigma, b) \notin \Psi$, we have:

$$\langle \mathsf{Flip}(e); \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{RA}} \langle \textbf{halt};, \Gamma[R \leftarrow \omega](\sigma), (\sigma, b) :: \Psi\rangle$$

for $b \in \{\mathtt{0}, \mathtt{1}\}$. Since we know that $\omega \succ \Psi$, then also $\omega \succ (\sigma, b) :: \Psi$ for *some* $b \in \{\mathtt{0}, \mathtt{1}\}$. Conversely, if we suppose that the pair $(\sigma, \mathtt{b})$ is in Ψ, we have:

$$\langle \mathsf{Flip}(e); \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{RA}} \langle \textbf{halt};, \Gamma[R \leftarrow \omega](\sigma), \Psi\rangle,$$

for the definition fo $\rightsquigarrow_{\mathsf{LA}}$. The coherency claim, in this case, is trivial.

$s; t$ Suppose that

$$\langle s; t, \Gamma, \omega\rangle \rhd \Sigma.$$

According to the definition of $\rhd$, this means that:

$$\langle s, \Gamma, \omega\rangle \rhd \Gamma'$$

and that:

$$\langle t, \Gamma', \omega\rangle \rhd \Sigma.$$

For the IH, this is equivalent to:

$$\langle s; \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \textbf{halt};, \Gamma', \Psi'\rangle \wedge \omega \succ \Psi'$$

and

$$\langle t; \textbf{halt};, \Gamma', \Psi'\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \textbf{halt};, \Sigma, \Psi''\rangle \wedge \omega \succ \Psi''.$$

Thus, we proved that:

$$\langle s; t; \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^*_{\mathsf{RA}} \langle \textbf{halt};, \Sigma, \Psi''\rangle \wedge \omega \succ \Psi''.$$

$\textbf{while}(e)\{s\}$ We go by cases on the value of e in Γ. If it does not reduce to $\mathtt{1}$, we have that:

$$\langle \textbf{while}(e)\{s\}, \Gamma, \omega\rangle \rhd \langle \Gamma, \omega\rangle \wedge \langle \textbf{while}(e)\{s\}; \textbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^1_{\mathsf{RA}} \langle \textbf{halt};, \Gamma, \Psi\rangle,$$

so the conclusion is trivial. Otherwise, if e reduces to $\mathtt{1}$, we have that:

$$\langle s, \Gamma, \omega\rangle \triangleright \langle \Gamma', \omega\rangle \wedge \langle \mathbf{while}(e)\{s\}, \Gamma', \omega\rangle \triangleright \langle \Sigma, \omega\rangle.$$

By induction hypothesis, we know that:

$$\langle s; \mathbf{halt};, \Gamma, \Psi\rangle \rightsquigarrow^{*}_{\mathsf{RA}} \langle \mathbf{halt};, \Gamma', \Psi'\rangle \wedge \omega \succ \Psi',$$

and that:

$$\langle \mathbf{while}(e)\{s\}; \mathbf{halt};, \Gamma', \Psi'\rangle \rightsquigarrow^{*}_{\mathsf{RA}} \langle \mathbf{halt};, \Sigma, \Psi''\rangle \wedge \omega \succ \Psi''.$$

But since e reduces to $\mathtt{1}$, we know that:

$$\langle \mathbf{while}(e)\{s\}; \mathbf{halt};, \Gamma, \Psi'\rangle \rightsquigarrow^{1}_{\mathsf{RA}} \langle s; \mathbf{while}(e)\{s\}; \mathbf{halt};, \Gamma, \Psi\rangle \wedge \omega \succ \Psi.$$

Summing together these three last conclusions, we get the claim we where aiming to.

The final result comes instantiating the universal quantifiers with the values described in Definition 79. The opposite direction can be obtained by a proof very similar to this one, reversing the order of the observations. Even in this case observing that for every $h, h \in \mathbb{N}$:

$$\langle P, \Sigma, \Psi\rangle \rightsquigarrow^{h}_{\mathsf{RA}} \langle P', \Sigma', \Psi'\rangle \wedge \langle P, \Sigma, \Psi\rangle \rightsquigarrow^{h+k}_{\mathsf{RA}} \langle P'', \Sigma'', \Psi''\rangle$$

entails that all the pairs $\langle k, \mathtt{b}\rangle$ of Ψ' belongs to Ψ'', too. □

Lemma 41. *For each* $\mathsf{SIFP}_{\mathsf{LA}}$ *program P, for each $\Psi \in \mathbb{S}$ and for each store $\Sigma \in \{\mathtt{0}, \mathtt{1}\}^{Id}$, said*

$$W := \{\Psi' \in \mathbb{S} | \langle P, \Sigma, \Psi\rangle \rightsquigarrow_{\mathsf{LA}} \langle P, \Sigma, \Psi'\rangle\},$$

then $\sum_{\Phi \in W} \mu(\Phi) = \mu(\Psi)$.

Proof. By cases on the definition of the program:

- The rules for assignments and while loops do not change the value of Ψ, so the result is trivial.
- $\mathsf{RandBit}()$ associates to each configuration two different images, one which adds $\mathtt{0}$ at the end of Ψ and one which adds $\mathtt{1}$ at the end of Ψ. For this reason, the measure is given by

$$\begin{aligned}\mu(\Psi\mathtt{0}) + \mu(\Psi\mathtt{1}) &= \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta_{|\Psi\mathtt{0}|} = \Psi\mathtt{0}\}) + \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta_{|\Psi\mathtt{1}|} = \Psi\mathtt{1}\}) \\ &= \frac{1}{2}\mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta_{|\Psi|} = \Psi\}) + \frac{1}{2}\mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta_{|\Psi|} = \Psi\}) \\ &= \mu(\{\eta \in \mathbb{B}^{\mathbb{N}} | \eta_{|\Psi|} = \Psi\}).\end{aligned}$$

- $\mathsf{Flip}(e)$, supposing that $\langle e, \Sigma\rangle \rightharpoonup \sigma$ and that $(\sigma, b) \notin \Psi$ for each $b \in \mathbb{B}$, we observe that $\rightsquigarrow_{\mathsf{RA}}$ associates to each configuration a pair of configurations in which Ψ has been replaced by $(\sigma, b) :: \Psi$ for $b \in \{\mathtt{0}, \mathtt{1}\}$. We show that the claim holds by cases. Suppose that $\sigma \notin \Psi$. this case is very similar to the previous one, indeed:

$$\begin{aligned}\mu\left((\sigma, \mathtt{0}) :: \Psi\right) + \mu\left((\sigma, \mathtt{1}) :: \Psi\right) &= \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in (\sigma, \mathtt{0}) :: \Psi.\omega(\sigma) = \mathtt{b}\}\right) \\ &\quad + \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in (\sigma, \mathtt{1}) :: \Psi.\omega(\sigma) = \mathtt{b}\}\right) \\ &= \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = \mathtt{0}\}\right) \\ &\quad + \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = \mathtt{1}\}\right)\end{aligned}$$

$$= 2 \cdot \left(\frac{1}{2}\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\}\right)\right)$$
$$= \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\}\right).$$

Indeed we can rewrite

$$\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = b\}$$

as

$$\frac{1}{2}\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\}\right),$$

because $\sigma \notin \Psi$. Otherwise, if $\sigma \in \Psi$

$$\begin{aligned}\mu((\sigma, \mathtt{0}) :: \Psi) + \mu((\sigma, \mathtt{1}) :: \Psi) &= \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in (\sigma, \mathtt{0}) :: \Psi.\omega(\sigma) = \mathtt{b}\}\right) \\ &\quad + \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in (\sigma, \mathtt{1}) :: \Psi.\omega(\sigma) = \mathtt{b}\}\right) \\ &= \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = \mathtt{0}\}\right) \\ &\quad + \mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = \mathtt{1}\}\right),\end{aligned}$$

which is equal to $\mu\left(\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\}\right)$ because one there is $b \in \{\mathtt{0}, \mathtt{1}\}$ such that:

$$\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \subseteq \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = b\}$$

and thus, said $\overline{b}$ its complementary element, it holds that:

$$\{\omega \in \mathbb{B}^{\mathbb{S}} | \forall(\sigma, \mathtt{b}) \in \Psi.\omega(\sigma) = \mathtt{b}\} \cap \{\omega \in \mathbb{B}^{\mathbb{S}} | \omega(\sigma) = b\} = \emptyset.$$

Conversely, if we suppose that $(\sigma, b) \in \Psi$ for some $b \in \mathbb{B}$ we the semantics $\rightsquigarrow_{\mathsf{RA}}$ admits only one transition towards a configuration in which $\Psi' = \Psi$, so the claim in this case is trivial.

□

Lemma 42 (Reduction of expressions)**.** $\forall e \in \mathcal{L}(\mathsf{Exp}).\forall\Sigma, \Gamma.(\forall G \in \#_r^{\mathsf{Exp}}(e))\Sigma(e) = \Gamma(e) \to \langle e, \Sigma\rangle \rightharpoonup \sigma \to \langle e, \Gamma\rangle \rightharpoonup \sigma$.

Proof. By induction on the syntax of e.

- ϵ is the only base case, and the conclusion is trivial.
- For all the other cases, the conclusion is a consequence of the IH(es).

□

6.2.8 From $\mathbf{SFP_{OD}}$ to SFP

Lemma 43. *For every $M = \langle \mathcal{Q}, \Sigma, \delta, q_0\rangle \in \mathbf{SFP_{OD}}$, the there is a $N = \langle \mathcal{Q}, \Sigma, H(\delta), q_0\rangle \in \mathbf{SFP}$ such that for every $\langle \sigma, q, \tau, \eta\rangle$ configuration of M, if $\xi = c_1c_2 \dots c_k$ with $c_i \in \mathbb{B}$ for $1 \le i \le k$, then*

1. *if $\langle \sigma, q, \tau, \xi\eta\rangle \triangleright_\delta^k \langle \sigma', q', \tau', \xi\eta\rangle$, then $\langle \sigma, q, \tau, \xi\eta\rangle \triangleright_{H(\delta)}^k \langle \sigma', q', \tau', \eta\rangle$.*
2. *if $\langle \sigma, q, \tau, \mathtt{b}\eta\rangle \triangleright_\delta^1 \langle \sigma', q', \tau', \eta\rangle$, then $\langle \sigma, q, \tau, \mathtt{b}\eta'\rangle \triangleright_{H(\delta)}^1 \langle \sigma', q', \tau', \eta'\rangle$.*

Proof. We start with the first claim, the proof is by induction on k.

- If $k = 0$, the thesis is trivial.
- If $k = j + 1$, the hypothesis induction states that:

  $$\forall \chi = c_1 c_2 \dots c_j. \left(\langle \sigma, q, \tau, \chi\eta \rangle \rhd_\delta^j \langle \sigma'', q'', \tau'', \chi\eta \rangle \right) \to \left(\langle \sigma, q, \tau, \chi\eta \rangle \rhd_{H(\delta)}^j \langle \sigma'', q'', \tau'', \eta \rangle \right),$$

  we know that:

  $$\left(\langle \sigma, q, \tau, c_1 c_2 \dots c_{j+1}\eta \rangle \rhd_\delta^{j+1} \langle \sigma', q', \tau', c_1 c_2 \dots c_{j+1}\eta \rangle \right),$$

  so we derive that:

  $$\left(\langle \sigma, q, \tau, c_1 c_2 \dots c_j (c_{j+1}\eta) \rangle \rhd_\delta^j \langle \sigma'', q'', \tau'', c_1 c_2 \dots c_j (c_{j+1}\eta) \rangle \right) \rhd_\delta^1 \langle \sigma', q', \tau', c_1 c_2 \dots c_{j+1}\eta \rangle.$$

  We can feed this sentence to the induction hypothesis obtaining:

  $$\left(\langle \sigma, q, \tau, c_1 c_2 \dots (c_{j+1}\eta) \rangle \rhd_{H(\delta)}^j \langle \sigma', q', \tau', c_{j+1}\eta \rangle \right).$$

  Finally, we observe that since

  $$\langle \sigma'', q'', \tau'', c_1 c_2 \dots c_j (c_{j+1}\eta) \rangle \rhd_\delta^1 \langle \sigma', q', \tau', c_1 c_2 \dots c_{j+1}\eta \rangle,$$

  then, according to Definition 100, the function δ contains a transition labeled with $\natural$ which matches exactly the current configuration, moreover, according to Definition 100, $H(\delta)$ contains a pair of transitions which match both the possible characters on the tape, namely `0` or `1`. so:

  $$\langle \sigma'', q'', \tau'', c_{j+1}\eta \rangle \rhd_{H(\delta)}^1 \langle \sigma', q', \tau', \eta \rangle.$$

  This concludes the derivation.

The second claim comes from the definition fo $H(\delta)$: the oracle consuming transition which are in δ are in $H(\delta)$, too. □

Chapter 7

Conclusions

This research was aimed to address the study of probabilistic complexity classes by means of a bounded arithmetic. To this end, we defined the function algebra $\mathcal{POR}$ (Section 2.1.2) and we introduced the $\mathcal{L}$ first-order language and the RS_2^1 bounded arithmetic (Section 2.1.3), together with a notion of Σ_1^b-representability within RS_2^1, establishing that the $\mathcal{POR}$ functions are exactly the Σ_1^b-representable functions of RS_2^1, Theorem 3. In particular, this result relies on the extension of the standard *qualitative* semantics of first-order formulæ with the introduction of a non-standard *quantitative* semantics (Definition 37) associating to each formula a measurable set. This novel semantics is employed in the notion of Σ_1^b-representability to describe the probability distributions computed by the represented functions.

Probabilistic complexity classes are usually defined basing on the class of Probabilistic Turing Machines. For this reason, we showed that the $\mathcal{POR}$ functions — and consequently the class of Σ_1^b-representable functions — are exactly the PTM poly-time computable functions (**PPT**).

The proof of this result required some effort. First, to bridge the gap between the definitions of the classes $\mathcal{POR}$ and **PPT**, we defined an intermediate class of functions **SFP**, which is grounded on the TM-like paradigm of the Stream Turing Machines and is meant to be equivalent to both $\mathcal{POR}$ and **PPT**. Then, we proposed a series of reductions connecting $\mathcal{POR}$ and **SFP**.

The reduction from **SFP** to $\mathcal{POR}$ (Section 3.2) is quite straightforward: it relies on the fact that in $\mathcal{POR}$ it is possible to define a function which, taking in input the initial configuration of a polynomial Stream Turing Machine, outputs the corresponding final configuration.

On the other hand, we encountered some obstacles attempting a direct reduction from $\mathcal{POR}$ to **SFP** (Section 3.3). The biggest has to do with the different ways adopted by $\mathcal{POR}$ and **SFP** to access randomness. In particular, the reduction of each $\mathcal{POR}$ function relies to at most to exponentially many random bits in the size of their input, while **SFP** functions can access only polynomially many random bits during their reduction. Our proof relies on three intermediate formalisms which were introduce to bridge the separate the paradigm related concerns of the reduction — indeed we were reducing a functional paradigms towards a TM-like one — from the probabilistic aspects of the reduction process.

At the end of Chapter 3, we managed to show that a form of equivalence holds between $\mathcal{POR}$ and **SFP**. Finally, we also managed to show that the class of functions **SFP** is equivalent to the **PPT** class (Proposition 7). Thanks to this last result, we manage to extend the Σ_1^b representability result (Theorem 3) from the class of $\mathcal{POR}$ function to the class of **PPT** functions (Theorem 4).

After Chapter 3, we employed the large amount of technical results given in those pages to prove, as a corollary, the equivalence of the Cobham style function algebra $\mathcal{F}_{\mathsf{Cob}}$ and the class **FP**.

This was done because similar results are widely agreed among the literature, although being almost completely folklore: as far as we know, up to now, there were not exhaustive and self-contained proofs of this result. In this case, the proof was quite simple: the first inclusion relies on the fact that $\mathcal{POR}$ is a generalization of $\mathcal{F}_{\mathsf{Cob}}$, but that all the non-random $\mathcal{POR}$ functions have an equivalent **FP** function. Similarly, the second inclusion relies on the observation that **SFP** is a generalization of **FP** and that under the hypothesis that an **SFP** function does not depend on random choices, it is in $\mathcal{F}_{\mathsf{Cob}}$ as well.

After we have found a characterization of all the **PPT** functions within the language $\mathcal{L}$, in Chapter 5, we addressed the problem of defining characterization of thinner probability complexity classes, such as **BPP**, **RP**, co-**RP** and **ZPP**. To this aim, we extended the $\mathcal{L}$'s language and semantics with a non-standard measure quantifiers $\mathbf{C}^{s/t}$, whose quantitative semantics has measure equal to 1 if and only if the semantics of the formula they are quantifying has measure greater or equal to $\frac{|s|}{|t|}$. The language thus obtained has been named $\mathcal{L}^{\mathsf{MQ}}$ and was proved being a conservative extension of $\mathcal{L}$, Remark 9.

Due to this extension, we were capable to reformulate the definitions of **BPP**, **RP**, co-**RP** within $\mathcal{L}^{\mathsf{MQ}}$. Then, these characterizations have been employed together with a standard result of probabilistic complexity theory — **ZPP** = **RP**$\cap$ co-**RP** (Theorem 1) — to develop a characterization of **ZPP**.

7.1 Future work

In the next months, we plan to investigate the characterizations of probabilistic complexity classes given in Chapter 5 in order to determine whether it is possible to develop equivalent characterizations within a standard first-order arithmetical language. This would pass through reduction of the measure quantifier $\mathbf{C}^{s/t}$ to a standard existential quantifier and the reduction of the predicate `Flip` to a first-order formula.

A similar result would pave the way to the identification of a *recursively enumerable* subset of those non-recursively enumerable complexity classes. Indeed, none of **BPP**, **RP**, co-**RP** and **ZPP** are known to be such and thus are considered *semantical* complexity classes.

To do so, we plan to reduce our non-standard logic $\mathcal{L}^{\mathsf{MQ}}$ to a standard first-order logic. This would also cause the reduction of the *quantitative* semantics to a *qualitative* semantics.

Concerning the reduction of the non-standard quantifiers, we conjecture that, due to the syntactical constraints on the formulæ within our characterizations, they can be reduced to a threshold-existential quantifier, [19]. Intuitively, this quantifier, denoted by $\exists^{\geq}kx.F$ holds if and only if there are at least k interpretations of the variable x which satisfy the formula F. We believe that this reduction is possible: intuitively, stating an assertion similar to $\mu(\llbracket F \rrbracket) \geq \frac{s}{t}$ is equivalent to stating that "over N classes of `Flip` possible interpretations, at least $\frac{s}{t}$ of them satisfy the quantified formula". Thus, the quantitative semantics of the $\mathcal{L}^{\mathsf{MQ}}$ would be encoded inside the language of the logic enriched with threshold quantifiers. This would come with the substitution of the $\mathtt{Flip}(t)$ predicate with a new formula $\phi(t, s)$ which is equivalent to $\mathtt{Flip}(t)$ under the hypothesis that s encodes an interpretation of such predicate. Intuitively, we want this formula ϕ to mean "if the interpretation of flip is s, then $\mathtt{Flip}(t)$ would hold". Precisely, we plan to define ϕ leveraging the encoding of lists and functions of Section 6.1.1, thus we could define the predicate as a $\mathcal{POR}^-$ function and leverage Theorem 5 to conclude that the ϕ formula we are aiming to exists.

Subsequently, we plan to adapt some works for counting quantifiers elimination — such as for example [25, 8]. Those works describes procedures for eliminating counting quantifiers from Presburger's Arithmetic. Even if Presburger's Arithmetic is a small and even decidable fragment of Peano Arithmetic, we think that it is possible to develop similar techniques aimed to *reduce* counting quantifiers to standard existentials instead of eliminating them. Moreover, we are confident about the possible outcomes of this process because we are facilitated by working in the peculiar context of our characterizations rather than in a general one.

Once the characterizations proposed in Chapter 5 will be expressed in the language of first-order logic, it will be possible to identify recursively enumerable subsets of probabilistic semantic classes by means of the provability relation $\vdash$. For instance, assume that **BPP** can be described by means of a class of first-order formulæ $\{\psi_i\}_{i\in\mathbb{N}}$, in the sense that each of these formulæ corresponds to a problem in **BPP**, then it would be possible to choose a proof system — for example PA's axioms — and to recursively enumerate all the derivations proving one of those formulæ, namely all the ψ_i such that $\mathsf{PA} \vdash \psi_i$. All the formulæ enumerated this way would correspond to a language in **BPP**. Let **PBPP** be this class of languages: it would certainly hold that $\mathbf{PBPP} \subseteq \mathbf{BPP}$. However, we believe that $\mathbf{BPP} \not\subseteq \mathbf{PBPP}$: contrarily, we would be surprised discovering a similar result, due to the incompleteness of PA and to the widely agreed conjecture that **BPP** is indeed a non recursively enumerable complexity class [4].

Finally, it would be worth to investigate the set **PBPP** under an extensional perspective: which **BPP** problems are in **PBPP**? Which problems are in $\mathbf{PBPP} \setminus \mathbf{BPP}$?

Bibliography

[1] M. Antonelli, U. Dal Lago, and P. Pistone. “On Counting Propositional Logic and Wagner’s Hierarchy”. In: *Proc. ICTCS ’21.* Ed. by CEUR Workshop Proceedings. Vol. 3072. 2021, pp. 107–121.

[2] M. Antonelli, U. Del Lago, and P. Pistone. “On Measure Quantifiers in First-Order Arithmetic”. In: *Connecting with Computabiliy.* Ed. by L. De Mol et al. Springer, 2021, pp. 12–24.

[3] M. Antonelli et al. *Randomized Bounded Arithmetic — Technical Report.* `https://github.com/davidedavoli/RBA`. 2022.

[4] S. Arora and B. Barak. *Computational complexity: A modern approach.* Cambridge University Press, 2009.

[5] P. Billingsley. *Probability and Measure.* Wiley, 1995.

[6] S.R. Buss. “Bounded Arithmetic”. PhD thesis. Princeton University, 1986.

[7] S.R. Buss. “First-Order Proof Theory of Arithmetic”. In: *Handbook of Proof Theory.* Ed. by S.R: Buss. Elsavier, 1998.

[8] Dmitry Chistikov, Christoph Haase, and Alessio Mansutti. “Presburger arithmetic with threshold counting quantifiers is easy”. In: *ArXiv* abs/2103.05087 (2021).

[9] Alan Cobham. “The Intrinsic Computational Difficulty of Functions”. In: *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics).* Ed. by Yehoshua Bar-Hillel. North-Holland Publishing, 1965, pp. 24–30.

[10] Stephen Cook and Alasdair Urquhart. “Functional Interpretations of Feasibly Constructive Arithmetic”. In: *Annals of Pure and Applied Logic* 63.2 (1993), pp. 103–200. DOI: `10.1016/0168-0072(93)90044-e`.

[11] H. B. Curry. “Functionality in Combinatory Logic*”. In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. DOI: `10.1073/pnas.20.11.584`. eprint: `https://www.pnas.org/doi/pdf/10.1073/pnas.20.11.584`. URL: `https://www.pnas.org/doi/abs/10.1073/pnas.20.11.584`.

[12] William Feller. “An Introduction to Probability Theory and its Applications, Volume 1”. In: (1968).

[13] F. Ferreira. “Polynomial-Time Computable Arithmetic”. In: *Logic and Computation.* Ed. by W. Sieg. Vol. 106. Contemporary Mathematics. AMS, 1990.

[14] F. Ferreira. “Polyonmial Time Computable Arithmetic and Conservative Extesions”. Ph.D. Dissertation. Dec. 1988.

[15] G. Ferreira and I. Oitavem. “An Interpretation of S_2^1 in Σ_1^b-NIA”. In: *Portugaliae Mathematica* 63 (2006), pp. 427–450.

[16] J. Gill. “Computational Complexity of Probabilistic Turing machines”. In: *J. Comput.* 6(4) (1977), pp. 675–695.

[17] Oded Goldreich and David Zuckerman. “Another proof that bpp?ph (and more)”. In: *Electronic Colloquium on Computational Complexity - ECCC* (Jan. 1997). DOI: `10.1007/978-3-642-22670-0_6`.

[18] Roberto Gorrieri and Cristian Versari. *Introduction to concurrency theory: transition systems and CCS*. Springer, 2015.

[19] E. Gradel, M. Otto, and E. Rosen. “Two-variable logic with counting is decidable”. In: *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. 1997, pp. 306–317. DOI: `10.1109/LICS.1997.614957`.

[20] H. A. Howard. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[21] Piergiorgio Odifreddi. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier, 1992.

[22] R. Parikh. “Existence and feasibility in arithmetic”. In: *Journal of Symbolic Logic* 36 (1971), pp. 494–508.

[23] Hartley Rogers Jr. *Theory of recursive functions and effective computability.* MIT press, 1987.

[24] E.S. Santos. “Probabilistic Turing Machines and computability”. In: *AMS* 22.3 (1969), pp. 704–710.

[25] Nicole Schweikardt. “Arithmetic, First-Order Logic, and Counting Quantifiers”. In: *ACM Transactions on Computational Logic* 6 (Dec. 2002). DOI: `10.1145/1071596.1071602`.

[26] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

[27] Larry J. Stockmeyer. “The polynomial-time hierarchy”. In: *Theoretical Computer Science* 3.1 (1976), pp. 1–22. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(76)90061-X`. URL: `https://www.sciencedirect.com/science/article/pii/030439757690061X`.

[28] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT press, 1993.