TASK C

Contents

Introduction. In the previous sections we have proved that there is a strong correspondance between the set of functions $\mathcal{POR}$ and the Σ_1^b formulas of $\mathcal{L}_{\mathbb{PW}}$, precisely that:

$$\forall G \in \Sigma_1^b . \exists f_G \in \mathcal{POR} . \big(S_3^1, \omega \vdash \forall x . \exists! y . G(x,y) \wedge \omega \in [\![G(x,y)]\!]\big) \leftrightarrow f_G(x,\omega) = y$$

and that

$$\forall f \in \mathcal{POR} . \exists G_f \in \Sigma_1^b . \big(S_3^1, \omega \vdash \forall x . \exists! y . G(x,y) \wedge \omega \in [\![G(x,y)]\!]\big) \leftrightarrow f_G(x,\omega) = y$$

this means that each Σ_1^b formula of the arithmetic $\mathcal{L}_{\mathbb{PW}}$ can be mapped to a $\mathcal{POR}$ function and viceversa. For this reason, if we prove that the $\mathcal{POR}$ functions are exactly the **PPT** functions, we could be able to arithmetize probabilistic complexity classes such as **BPP**.

More precisely, the goal of this part is to show that the following claim holds

Conjecture 1. *The* **PPT** *functions are exactly the functions in* $\mathcal{POR}$

Outline. In order to prove Conjecture 1, we need to:

(1) Define a formalism which can be proved equivalent to the **PPT** functions with few effort. Such formalism is **SFP**. Although it will be introduced in Section 0.1.1, we will prove the correspondance between **PPT** and **SFP** only later, while studying an arithmetical characterization of **BPP**, in Section **??**. This delay should not be too much of a problem because the definition of **SFP** almost the same of **PPT** so, in our opinion, their equivalence is almost trivial.
(2) On top of the definition of **SFP**, define its encoding in $\mathcal{POR}$ and prove it correct. This is carried out in Section 0.2.
(3) Finally, proce that each function in $\mathcal{POR}$ is in **SFP**, too. This reduction is presented in Section 0.2.

We will prove that the functions that can be computed in the **SFP** formalism are precisely the ones in $\mathcal{POR}$. The present Section is tripartite. First, in Section 0.1.1, we will define **SFP**, then in Section 0.2, we will establish that all the functions that are calculated by an **SFP**-machine are in $\mathcal{POR}$. Finally, in Section 0.3, we will prove viceversa that each $\mathcal{POR}$-function can be computed by an **SFP**-machine.

0.1. **SFP and other tools.**

0.1.1. *On* **SFP** *functions.* As we mentioned above, the definition of **SFP** is intended to capture the concept of Probabilistic Polynomial Time function. For this reason, when defining **SFP**, we would like to ground it on top of some kind of probabilistic Turing Machines.

Unfortunately, Probabilistic Turing Machines as proposed in Definition 1 are not suitable for the reduction we are aiming to. The randomness of these machines is implicit, since its represented by a distribution of probability, while $\mathcal{POR}$ encodes it by means of an oracle function $\omega : \mathbb{S} \longrightarrow \mathbb{B}$.

Definition 1 (Probabilistic Turing Machine, [**barak**])**.** A probabilistic Turing machine (**PTM**) is a Turing machine with two transition functions δ_0, δ_1. To execute a **PTM** M on an input x, we choose in each step with probability $\frac{1}{2}$ to apply the transition function δ_0 and with probability $\frac{1}{2}$ to apply δ_1. This choice is made independently of all previous choices.

For these reason we decided to use a slight variation of such model, which uses a explicit read-only random tape instead of a probability function as source od randomness for determining the transition. This is what we call a Stream Machine.

Definition 2 (Stream Machines, informally)**.** Stream machines are ordinary $k+1$-taped Turing Machines which use one of their tape for storing an infinite stream of characters. This tape, which we will also call *oracle*, is read from left to right, one character for transition.

The formalism of the Stream Machine is an useful intermediate between $\mathcal{POR}$ and **PPT**. Indeed:

- It uses a tape as explicit source of randomness, which can be easily represented by means of a function $\eta : \mathbb{N} \longrightarrow \mathbb{B}$. This behaviour is consistent with $\mathcal{POR}$, which uses functions $\omega : \mathbb{S} \longrightarrow \mathbb{B}$.
- It is a particular instance of a multi-tape Turing Machine, so many well known results about this formalism can be reused for the reduction from $\mathcal{POR}$ to **SFP**.
- It is almost intuitively quivalent to the **PTM** formalism, so it will be easy to prove that some interesting classe of function can be characterized as subset of the funcions computed by stream machines.

The first class of functions which we will characterize as a subset of Stream Machine is exactly **SFP**.

Definition 3 (**SFP**, informally)**. SFP** is the class of functions $f : \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \longrightarrow \mathbb{S}$ which can be computed by a Stream Machine in polynomial time.

In order to formalize the definition above, we need to formalize Stream Machines, too:

Definition 4 (Stream Machine)**.** A *stream machine* is a quadruple $M := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q \rangle$, where:

- $\mathbb{Q}$ is a finite set of states ranged over by $q_1, \ldots, q_n$
- Σ is a finite set of characters ranged over by $c_1, \ldots, c_n$
- $\delta_{\mathbf{SFP}} : \mathbb{Q} \times \hat{\Sigma} \times \{\mathbf{0}, \mathbf{1}\} \longrightarrow \mathbb{Q} \times \hat{\Sigma} \times \{L, R\}$ is a transition function that describes the new configuration reached by an **SFP**-machine. L, R are two fixed constants and $\hat{\Sigma} = \Sigma \cup \{\circledast\}$; $\circledast$ represents the *blank character*, so $\circledast \notin \Sigma$
- $q \in \mathbb{Q}$ is an initial state,

Note that the assumption $\Sigma = \{\mathbf{1}, \mathbf{0}\}$ would not be reductive.

Notation 1. From now on we will denote the blank character $\circledast$ as $c_{|\Sigma|+1}$.

Usually, the configuration of an ordinary Turing Machine is a tuple which keeps records the current state and some strings which represent the state of the machine tape(s). Using strings for representing the configuration of the tapes is possible only tape contains a finite sequence of

characters, but a Stream Machine uses a tape with an infinite sequence of **0** and **1** as source of randomness. For this reason, while defining the configuration of this class of Turing Machines, we decided to represent the value of the oracle tape by means of a function $\eta \in \mathbb{B}^{\mathbb{N}}$.

Definition 5 (Configuration of a Stream Machine)**.** The *configuration of a stream machine* M_S is a quadruple $\langle \sigma, q, \tau, \eta \rangle$, where;

- $\sigma \in \hat{\Sigma}^*$ is the portion of the first tape to the left of the head
- $q \in \mathbb{Q}$ is the current state of M_S
- $\tau \in \hat{\Sigma}$ is the portion of the first tape to the right of the head.
- $\eta \in \{\mathbf{0}, \mathbf{1}\}^{\mathbb{N}}$ is the portion of the second tape that has not been read yet.

Now we would like to formalize the fact that at each step, the machine queries a new value of its oracle tape. While the shifting of the work tape can be naturally defined by prefixing and postfixing of characters to strings, the same operation is not that natural for the oracle tape because of its infinite size. For this reason we need to define some shifting operation between a function $\mathbb{N} \longrightarrow \mathbb{B}$ and a string $\sigma \in \mathbb{S}$.

Definition 6 (Shifting operation)**.** Given a string $\sigma \in \mathbb{S}$ and a function $\eta : \mathbb{N} \longrightarrow \mathbb{B}$, we define the shifting of η by the prefix σ, denoted $\sigma\eta$ by induction on σ as follows:

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

This allows us to give a formal definition of the machine's transitions.

Definition 7 (Stream Machine Transition Function)**.** Given a stream machine $M = \langle \mathbb{Q}, \Sigma, \delta, q \rangle$, we define the partial transition function $\vdash_\delta \hat{\Sigma}^* \times \mathbb{Q} \times \hat{\Sigma}^* \times \{\mathbf{0}, \mathbf{1}\}^{\mathbb{N}} \longrightarrow \hat{\Sigma}^* \times \mathbb{Q} \times \hat{\Sigma}^* \times \{\mathbf{0}, \mathbf{1}\}^{\mathbb{N}}$ between two configurations of M_S as:

$$\begin{aligned}
\langle \sigma, q, c\tau, \mathbf{0}\eta \rangle &\vdash_\delta \langle \sigma c', q', \tau, \eta \rangle && \text{if } \delta(q, c, \mathbf{0}) = \langle q', c', R \rangle \\
\langle \sigma c_0, q, c_1\tau, \mathbf{0}\eta \rangle &\vdash_\delta \langle \sigma, q', c_0 c_1' \tau, \eta \rangle && \text{if } \delta(q, c_1, \mathbf{0}) = \langle q', c_1', L \rangle \\
\langle \sigma, q, c\tau, \mathbf{1}\eta \rangle &\vdash_\delta \langle \sigma c', q', \tau, \eta \rangle && \text{if } \delta(q, c, \mathbf{1}) = \langle q', c', R \rangle \\
\langle \sigma c_0, q, c_1\tau, \mathbf{1}\eta \rangle &\vdash_\delta \langle \sigma, q', c_0 c_1' \tau, \eta \rangle && \text{if } \delta(q, c_1, \mathbf{1}) = \langle q', c_1', L \rangle.
\end{aligned}$$

The function obtained composing n times the reachability function describes the configuration reached by the machine after n steps of computation. This is stated as follows:

Definition 8 (Stream Machine Reachability Functions)**.** Given a stream machine $M := \langle \mathbb{Q}, \Sigma, \delta, q_0 \rangle$, we denote with $\{\triangleright_M^n\}_n$ the smallest family of relations for which:

$$\langle \sigma, q, \tau, \eta \rangle \triangleright_M^0 \langle \sigma, q, \tau, \eta \rangle$$
$$\langle \sigma, q, \tau, \eta \rangle \triangleright_M^n \langle \sigma', q', \tau', \eta' \rangle \wedge \langle \sigma', q', \tau', \eta' \rangle \vdash_\delta \langle \sigma'', q', \tau'', \eta'' \rangle \rightarrow \langle \sigma, q, \tau, \eta \rangle \triangleright_M^{n+1} \langle \sigma'', q'\tau'', \eta'' \rangle.$$

We would also like to point out that our machine doesn't use final states: the computation is considered finished whenever the transition function isn't defined on the current configuration. This choice isn't reductive, because we can imagine to add a final state q_F and a transition from all the configurations on which the transition function is undefined to q_F.

For seek of completeness, we want to show that each $\triangleright_M^n$ relation is a function.

Proposition 1. *If $M := \langle \mathbb{Q}, \Sigma, \delta, q_0 \rangle$ is a Stream Machine, it holds that $\forall n. \triangleright_M^n$ is a function.*

Proof. The proof is by induction on n.

- Let $n = 0$. In this case $\triangleright_M^n$ is the identity function.
- We assume the claim to hold for n, and prove it for $n+1$. For IH, $\triangleright_M^n$ is a function. Then, since $\vdash_\delta$ is a function, $\triangleright_M^{n+1} = \triangleright_M^n \circ \vdash_\delta$ is a function, too.

□

Now we would like to give a formal definition of the function computed by a Stream Machine. This will allow us to define **SFP** as the set of functions which can be computed by polynomial Stream Machines.

First, we build a notion of final configuration.

Notation 2 (Final configuration)**.** Given a stream machine $M := \langle \mathbb{Q}, \Sigma, \delta, q_0 \rangle$ and a configuration $\langle \sigma, q, \tau, \eta \rangle$, we denote the condition $\neg \exists \sigma', q', \tau', \eta' . \langle \sigma, q, \tau\eta \rangle \vdash_\delta \langle \sigma', q', \tau', \eta' \rangle$ as $\langle \sigma, q, \tau, \eta \rangle \nvdash_\delta$.

This allows us to define the function computed by a Stream Machine.

Definition 9 (Value Computed by a Stream Machine)**.** Given a machine $M := \langle \mathbb{Q}, \Sigma, \delta, q_0 \rangle$, M computes γ on input σ and oracle η if and only if

$$\exists n . \langle \boldsymbol{\epsilon}, q_0, \sigma, \eta \rangle \triangleright_M^n \langle \gamma, q', \tau, \psi \rangle \nvdash_\delta$$

for some τ, q', ψ. In that case, we write $M(\sigma, \eta) = \gamma$.

This outlines a function $f_M : \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \longrightarrow \mathbb{S}$ for each Stream Machine M.

Definition 10 (Polynomial Stream Machine)**.** An *Polynomial Stream Machine* is a Stream Machine $M_{\mathbf{SFP}} := \langle \mathbb{Q}, \Sigma, \delta, q_0 \rangle$ such that,

$$\exists p \in \mathsf{POLY} . \forall \sigma, \eta, n . n > p(|\sigma|) \wedge \langle \boldsymbol{\epsilon}, q_0, \sigma, \eta \rangle \triangleright_{M_{\mathbf{SFP}}}^n \langle \gamma, q', \tau, \psi \rangle \rightarrow \bot .$$

Finally, we can define the set **SFP**.

Definition 11 (**SFP**)**.**

$$\mathbf{SFP} := \{ f \in \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \mid \text{There exists an Polynomial Stream Machine } M \text{ such that } f = f_M \}$$

Because of the last definitions, later, we will abbrevite *Polynomial Stream Machines* with "**SFP** machines". This because we have defined **SFP** as the set of functions computable by *Polynomial Stream Machines*.

We would like to stress out the fact that the definitions given so far deal with single-input single-tape machines only. This isn't reductive: we could naturally extend our definitions to multi-input and multi-tape machines, in a similar way of how thee extensions are defined for canonical Turing Machines. Moreover, in the following parts, we will use multi-tape Stream Machines for simplifying some proofs.

Now, it is quite simple to outine a road-map which will allow us to prove Conjecture 1.

(1) Prove that each function in **SFP** can be computed by a function in $\mathcal{POR}$.
(2) Prove that each function in $\mathcal{POR}$ can be computed by a function in **SFP**.
(3) Prove that $\mathbf{SFP} = \mathbf{PPT}$.

Before proceding with the proofs of the results stated above, we'd like to point out some observations:

(1) $\mathcal{POR}$ and **SFP** are two inherently different sets:

$$\mathcal{POR} \subseteq \bigcup_{i \in \mathbb{N}} \mathbb{S}^i \times \mathbb{O} \rightarrow \mathbb{S}$$

$$\mathbf{SFP} \subseteq \bigcup_{i \in \mathbb{N}} \mathbb{S}^i \times \{\mathbf{0}, \mathbf{1}\}^{\mathbb{N}} \rightarrow \mathbb{S}$$

(2) Functions in **SFP** are determined by a polynomial prefix of their oracle, while a similar result seems (and actually is) not true for $\mathcal{POR}$.

These facts will require us some effort in the follwing encodings and will drive us to state results which are weaker than the ones we proved for $\mathcal{L}_{\mathbb{PW}}$ and $\mathcal{POR}$.

For instance, take in exam an hypothetic reduction from **SFP** to $\mathcal{POR}$: it should not be difficult to imagine that a machine $M(x,\eta) \in \mathbf{SFP}$ can be simulated by a function $f(x,\omega) \in \mathcal{POR}$, which at every simulated step queries a different coordinate of its oracle ω.

But the correspondance between η and ω, even if intuitive, isn't trivial under a technical perspective because of the necessity to link functions in $\mathbb{O}$ to functions in $\mathbb{B}^{\mathbb{N}}$; this fact drove us to define some mathematical structures, called *Reduction Trees* which will help us to prove the correctness of our encoding. *Reduction Trees* keep track of all the possible sequences of coordinates used by a probabilistic function for querying the oracle and producing the result.

Moreover, when dealing with Σ_1^b formulas in $\mathcal{L}_{\mathbb{PW}}$ and $\mathcal{POR}$ functions, we could state lemmas without any measure theoretic claim inside. This was due to the identity between the oracles passed as parameters to $\mathcal{POR}$ functions and the oracles in the semantics of a formula. For instance this allowed us to prove that:

$$\forall G \in \Sigma_1^b.\exists f_G \in \mathcal{POR}.\big(S_3^1, \omega \vdash \forall x.\exists! y.G(x,y) \wedge \omega \in [\![G(x,y)]\!]\big) \leftrightarrow f_G(x,\omega) = y$$

which itself entails that

$$\forall G \in \Sigma_1^b.\exists f_G \in \mathcal{POR}.\forall x,y.\{\omega \in \mathbb{O} | f_G(x,\omega) = y\} = [\![G(x,y)]\!]$$

Obviously, a similar identity can't be proved for $\mathcal{POR}$ and **SFP** because we defined them on top of different sets of oracle-functions. In order to fill this gap, we decided to show a weaker (but strong enough) result: the encodings between $\mathcal{POR}$ and **SFP** preserve the measure of the sets of oracles driving an input x to an output y. Formally:

$$\forall f \in \mathcal{POR}.\exists M_f \in \mathbf{SFP}.\forall x,y.\mu\big(\{\omega \in \mathbb{O} | f(x,\omega) = y\}\big) = \mu\big(\{\eta \in \mathbb{B}^{\mathbb{N}} | M_f(x,\eta) = y\}\big)$$

This oultines the necessity to define cylinders of oracle functions on top of which we will be able to define the notion of measure.

0.1.2. *Cylindes and Reduction Trees.* By definition, the result of a random function isn not only determined by its input, but it depends on the source of randomness adopted, too. Depending on the value of the oracle function on some specific coordinates, the probabilistic function will drive to different outputs.

For this reason we need to introduce two important structures which capture this phenomenon:

(1) Cylinder sets, for describing the oracles which drive a probabilistic function to compute the same output y on input x.
(2) Reduction Trees, which describe the coordinates of the oracle which are useful for producing an input and how the value of the oracle on these coordinates affects the final result.

Definition 12 (Cylinder set)**.** Given a set $\mathbb{Q} = \mathbb{B}^{A}$[1], we define the positive cylinder associated to $\mathbb{Q}$ on coordinate x as:

$$P(x) = \{\psi \in \mathbb{Q} | \psi(x) = \mathtt{1}\}$$

[1]In the following part of our discussion we will be mostly intrested in cylinders of $\mathbb{O}$ and $\mathbb{B}^{\mathbb{N}}$. For this reason we will not specify whether we are considering cylinders of a set or the others, because it can be inferred by the argument of the function

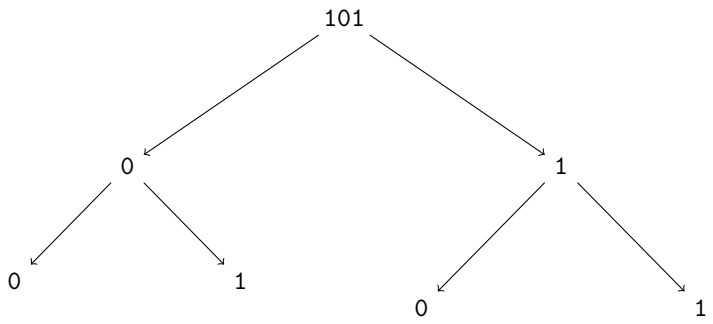

FIGURE 1. $\mathbb{S}$-Reduction-Tree associated to $Q(Q(\mathtt{101},\cdot),\cdot)$

similarly, we define the negative cylinders as:

$$N(x) = \{\psi \in \mathbb{Q} | \psi(x) = \mathtt{0}\}$$

Definition 13 (Reduction Tree)**.** An A-Reduction-Trees a production of the following grammar:

$$\mathit{Tree} ::= T\ a\ (\mathit{Tree})\ (\mathit{Tree}) \mid \mathit{nil}$$

with $a \in A$.

For deterministic paradigms, reduction trees are degenerate, because they consist in lists, but when dealing with probabilistic paradigms, they acquire significance because it's possible to associate one of these structures to any fuctions ad any input.

Example 1. *The $\mathbb{S}$-Reduction-Treessociated to the $\mathcal{POR}$ function $Q(Q(x,\cdot),\cdot)$ and input* $\mathtt{101}$ *is represented in Figure 1*

We can associate a sequence of cylinder sets to each path from the root of a reduction tree to one of its leafs. The intersection of such set is exactly the set of oracles which drive the function to produce the value on the leaf. We can state define these sequence of cylinders by induction:

Definition 14 (Cylinder Paths of a Reduction Tree)**.** Given a A-Reduction-Tree, we define the set of cylinder paths as follows:

$$\mathit{cylp}(L\ b) := \mathbb{B}^A$$
$$\mathit{cylp}(T\ a\ t_0\ t_2) := \{N(a), t \mid t \in \mathit{cylp}(t_0)\} \cup \{P(a), t \mid t \in \mathit{cylp}(t_1)\}$$

Definition 15 (Reduction Tree associated to a $\mathcal{POR}$ function)**.** We say that a $\mathbb{S}$-Reduction-Tree t is associated to a function $f \in \mathcal{POR}$ and input x if and only if $\mathit{cylp}(t) = C(s_0^x), \ldots, C(s_n^x)$ and $f(x, \bigcap_{i=0}^n C(s_i^x))$ is a singleton. If it is the case, we write $t \in RT_{\mathcal{POR}}(f, x)$.

Similarly, we can define the notion of reduction Tree associated to a $\mathcal{POR}$ function:

Definition 16 (Reduction Tree associated to a $\mathcal{POR}$ function)**.** We say that a $\mathbb{N}$-Reduction-Tree t is associated to a **SFP** function M and input x if and only if $\mathit{cylp}(t) = C(s_0^x), \ldots, C(s_n^x)$ and $M(x, \bigcap_{i=0}^n C(s_i^x)) =$ is a singleton. In this case we write $t \in RT_{\mathbf{SFP}}(M, x)$.

Finally, these definitions allow us to state the claims which we will prove them in the following sections:

Proposition 2. *For every $f : \mathbb{S} \times \mathbb{O} \in \mathcal{POR}$ there exists a $M_f \in$ **SFP** such that for every $x \in \mathbb{S}$ and for every $t \in RT_{\mathcal{POR}}(f, x)$ there exists a Reduction Tree $\bar{t} \in RT_{\mathbf{SFP}}(M_f, x)$ such that $\bar{t}$ is structurally identical to t and for every path C in $\mathit{cylp}(t)$, the corresponding path D in $\mathit{cylp}(\bar{t})$ is such that $f(x, C) = M_f(x, D)$.*

Proposition 3. *For every $M : \mathbb{S} \times \mathbb{B}^{\mathbb{N}} \in$ **SFP** there exists a $f_M \in \mathcal{POR}$ such that for every $x \in \mathbb{S}$ and for every $t \in RT_{\mathbf{SFP}}(M, x)$ there exists a Reduction Tree $\bar{t} \in RT_{\mathcal{POR}}(f_M, x)$ such that $\bar{t}$ is structurally identical to t and for every path C in $cylp(t)$, the corresponding path D in $cylp(\bar{t})$ is such that $M(x, C) = f_M(x, D)$.*

0.2. **From SFP to $\mathcal{POR}$ functions.**
Basic data structures and functions. Now, let us suppose that we want to implement **SFP** on a formalism B which manipulates sequences of character in a set $\mathbb{D}$. To do so we need some primitive data structures which allow us to represent a generical machine $M \in$ **SFP** and which support some basic operation on top of which we can easily implement the behaviour of the machine M.

This choice isn't univocal, but chosing subsets of $\mathbb{D}$ on which it is simple to build a *natural* omomorphism with the orignal sructres will simplify our job.

Specifically, we decided to implement the following data structures:

- *Natural numbers*
- *Boolean values*
- *Strings* of a binary alphabet,
- *Tuples*.

The reason which drove us to take these specific sets was the fact that natural numbers will allow us to encode enumerable sets, which are necessary, for example, for describing the machine's state and characters. Boolean values are necessary for implementing control structures, which will simplify the implementation of the machine. Moreover, Tuples will be useful for the implementation of collections, such as sets and lists. For instance, binary strings will be implemented as tuples of binary values and will be used for representing the machine's tapes.

The structures mentioned above, if taken all together, form an algebraic structure characterized by a finite collection of data sets and a finite set of functions closed on those sets. We want the encoding from **SFP** to the destination formalism B to preserve such algebraic structure, namely: to be an isomorphism.

For instance, suppose that we want to represent the function $odd : \mathbb{N} \longrightarrow \mathbb{B}$ in our destination formalism. It is natural that we also want the encoded function odd' to be a function from the *encoding of* $\mathbb{N}$ to the *encoding of* $\mathbb{B}$. And that, be our encoding $\bar{\cdot}$ $\forall n \in \mathbb{N}. \forall b \in \mathbb{B}. odd(n) = b \rightarrow odd''(\overline{n}) = \overline{b}$.

We can state this fact formally as follows:

Definition 17. We say that a morphism $\Phi(\langle\langle X_1, \ldots, X_n\rangle, O\rangle) =: \langle\langle \Phi_{\mathbb{D}_{X_1}}, \ldots, \Phi_{\mathbb{D}_{X_n}}\rangle, \Phi_Q(O)\rangle$ is a correct encoding of the structure $\langle\langle X_1, \ldots, X_n\rangle, O\rangle$, with $O \subseteq X_i^{X_j}$ to the structure: $\langle\langle \mathbb{D}_{X_1}, \ldots, \mathbb{D}_{X_n}\rangle, Q\rangle$ if and only if:

- All the $\Phi_{X_i} : X_i \longrightarrow D_{X_i}$ are injective functions.
- The functor Φ_Q is such that $\forall X_i. \forall X_j. \forall o \in O. \Phi_Q(o(x_j, x_i)) = \Phi_Q(o)(\Phi_{X_j}(x_j), \Phi_{X_i}(x_i))$.

Suppose that we have a correct encoding from

Notation 3. Let us use $\mathbb{D}_S$ to denote the image of the set S, model its encoding over $\mathbb{D}$, $\widetilde{n} \in \mathbb{D}_{\mathbb{N}}$ to denote the encoding of the number $n \in \mathbb{N}$ in the domain $\mathbb{D}$, $\mathbb{1} \in \mathbb{D}_{\{\mathbf{0},\mathbf{1}\}}$ to denote the encoding of the true Boolean value and $\in \mathbb{D}_{\{\mathbf{0},\mathbf{1}\}}$ to denote the false Boolean value as represented in $\mathbb{D}$, $\sigma, \tau \in \mathbb{D}$ to indicate the representation of a string over the alphabet $\{\mathbf{0}, \mathbf{1}\}$ (a string in $\mathbb{S}$) in the domain $\mathbb{D}$, and range the tuples on the following meta-variables: $t_0, t_1, \ldots, t_n$.

On top of the data structures that we have introduced, we need some basic functions. In particular, in order to perform computations on natural numbers, we need at least the following

functions: *addition*, denoted by $+$, *subtraction*, denoted by $-$, *multiplication*, denoted by $\cdot$, *exponentiation*, represented with the common power notation. The latter function will be useful to express the complexity bound of an **SFP**-machine that is a polynomial. All these functions have signature: $\mathbb{D}_{\mathbb{N}} \times \mathbb{D}_{\mathbb{N}} \longrightarrow \mathbb{D}_{\mathbb{N}}$. For each of these functions, call it $*$, it must hold that: $(\forall n, m, o \in \mathbb{N})(n * m = o \rightarrow \widetilde{n} * \widetilde{m} = \widetilde{o})$.

Boolean values can be defines as a subset of natural numbers. So, if we are able to define $\mathbb{D}_{\mathbb{N}}$, we are also able to define $\mathbb{D}_{\{\mathbf{0},\mathbf{1}\}}$ as its subset $\{\widetilde{\mathbf{0}}, \widetilde{\mathbf{1}}\}$ and to implement Boolean values and functions as subsets of the natural numbers and functions defined on such values. For example, we can assign $\widetilde{\mathbf{0}}$ to the false Boolean value and $\widetilde{\mathbf{1}}$ to the true one. Due to Booleans we are able to define a *conditional function* over a generical set S, i.e. the $\texttt{if} : \mathbb{D}_{\{\mathbf{0},\mathbf{1}\}} \times \mathbb{D}_S \times \mathbb{D}_S \longrightarrow \mathbb{D}_S$, which respects its commonly intended specification, and logical connectives. In particular, the `if`-function is an important control function to determine the configuration that follows the current one. We need also the bounded iteration structure, i.e. a `for` expression, which allows us to simulate the execution of an **SFP**-machine up to its polynomial bound. For all our data structures we need a binary function *eq* that returns $\widetilde{\mathbf{1}}$ if its parameters are equal with respect to the identity and $\widetilde{\mathbf{0}}$ otherwise.

Binary strings are needed to access the ω tape. In particular, we will use a simple function of random access with the following signature: $\cdot[\cdot] : \mathbb{D}_{\{\mathbf{0},\mathbf{1}\}^{\mathbb{N}}} \times \mathbb{D}_{\mathbb{N}} \longrightarrow \{\widetilde{\mathbf{0}}, \widetilde{\mathbf{1}}\}$, such that:

$$\forall \sigma \in \mathbb{D}_{\{\mathbf{0},\mathbf{1}\}^{\mathbb{N}}}. \forall n \in \mathbb{D}_{\mathbb{N}}. \sigma[\widetilde{n}] = \widetilde{\mathbf{1}} \leftrightarrow \text{the } n\text{-th bit of } \sigma \text{ is } \widetilde{\mathbf{1}}.$$

Let $\mathbb{T}_S^n$ denotes the set of homogeneous tuples of elements in S with cardinality n.

Finally, we need the following functions for handling tuples:

- A family of constructors, which are used to build tuples of finite dimension. These functions are denoted by angular brackets and have the following signature:

  $$\langle \cdot, \cdots, \cdot \rangle : \mathbb{D}_S^n \longrightarrow \mathbb{D}_{\mathbb{T}_S^n}.$$

  For example, $\langle \widetilde{\mathbf{0}}, \widetilde{\mathbf{1}} \rangle$ is the instantiation of the tuple's constructor on the encoding of $\mathbf{0}$ and $\mathbf{1}$ as first and second arguments.
- A function which computes the size of a tuple, denoted as $|\cdot| : \mathbb{D}_{\mathbb{T}_s^n} \longrightarrow \mathbb{D}_{\mathbb{N}}$.
- A family of projectors, which are used to extract values from a tuple, and denoted as $\pi_i : \mathbb{D}_{\mathbb{N}} \times \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_S$, where i is the position of the element returned by the projector. If the index of the element is greater than the tuple's size, we assume that the projection function returns a default value.
- Four manipulators:
  - A unary function, called *rmr* : $\mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n-1}}$, which deletes the rightmost element of a tuple.
  - A unary function, called *rml* : $\mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n-1}}$, which deletes the leftmost element of a tuple.
  - A unary function, called *addr* : $\mathbb{D}_S \times \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n+1}}$, which inserts an element in the rightmost position of a tuple.
  - A unary function, called *addl* : $\mathbb{D}_S \times \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n+1}}$, which inserts an element in the leftmost position of a tuple.

Specifically, the last two modifiers will be useful when handling tapes and transition, since they allow us to simulate the movement of the machine's head.

Definition 18 (Correctness of Encoding for Tuples)**.** An implementation of tuples is *correct* if and only if:

$$\forall n.\forall 1 \leq i \leq n.\pi_i\big(\langle x_1,\ldots,x_n\rangle\big) = x_i$$
$$\exists x.\forall i \geq n.\pi_i\big(\langle x_1,\ldots,x_n\rangle\big) = x.$$

Complex Data Structures and Functions. On top of the data structures we have just defined, we can introduce the encoding of an **SFP**-machine and an emulating interpreter as described below.

The encoding of the transition function, $\delta_{\mathbf{SFP}}$, is finite as the domain of $\delta_{\mathbf{SFP}}$ is. Let $\{t_0,\ldots,t_N\}$ be the *finite* subset of the set $(\mathbb{Q},\hat{\Sigma},\{\mathbf{0},\mathbf{1}\})\times(\mathbb{Q},\hat{\Sigma}\times\{L,R\})$, describing the function.

Definition 19 (Encoding: Transition Function)**.** The encoding of a machine's transition function is defined as:

$$enct\big(\{t_0,\ldots,t_N\}\big) = \langle g(t_0,\ldots,g(t_{|\delta_{\mathbf{SFP}}|})\rangle$$

and g as:

$$g\big(\langle\langle q_i,c_j,b\rangle,\langle q_k,c_l,D\rangle\rangle\big) := \begin{cases}\langle\widetilde{i},\widetilde{j},\widetilde{k},\widetilde{l},\widetilde{\mathbf{0}},b\rangle & \text{if } D=L\\ \langle\widetilde{i},\widetilde{j},\widetilde{k},\widetilde{l},\widetilde{\mathbf{1}},b\rangle & \text{otherwise.}\end{cases}$$

Observe that the definition above only uses the data structures introduced in the previous paragraph, namely natural values and tuples. Furthermore, the transition function is represented extensively, by enumerating all its members.

The representations of tuples and configurations are defined in a the same way.

Definition 20 (Encoding: Tape)**.** We encode all the finite portion of a tape $\sigma := c_i,\ldots,c_k$ as:

$$tenc(c_i,\ldots,c_k) := \langle\widetilde{i},\ldots,\widetilde{k}\rangle.$$

Definition 21 (Encoding: Configuration)**.** The representation of the configuration of a stream machine is defined as the 4-tuple $\langle\sigma,\widetilde{i},\tau,\widetilde{k}\rangle$, such that:

- $\sigma = tenc(\sigma')$, with σ' is the shortest portion of the tape that starts from the cell on the immediate left of the head and is followed (on its left) by an infinite sequence of blank characters ⊛.
- $\widetilde{i}$ is the encoding of the index of the current state q_i.
- $\tau = tenc(\tau')$, where τ' is the shortest portion of the tape that starts from the cell under the head, continues on its right and is followed (on its right) by an infinite sequence of blank characters ⊛.
- $\widetilde{k}$ is the encoding of the length of the prefix of the oracle tape that has already been consumed.

Remark 1. *The encoding of the initial state of a stream machine,* $\langle\epsilon,q_0,\sigma,\omega\rangle$ *is* $\big\langle\langle\rangle,\widetilde{\mathbf{0}},tenc(\sigma),\widetilde{\mathbf{0}}\big\rangle$.

Now that we have described all the static aspects of the stream machine, we can define some functions that allow us to emulate the dynamic behavior of the machine.

Lemma 1 (**SFP** Implementation)**.** *Each formalism which works on a domain* $\mathbb{D}$ *in which it is possible to express the data structures, functions and control primitives described by our encoding and that is closed under composition, is at least as expressive as the stream machines are.*

Proof Sketch. Here we will only show a sketch of the proof. For a more comprehensive one, the reader is invited to consult the Section **??** of the appendix.
Let $M_S := \langle\mathbb{Q},\Sigma,\delta_{\mathbf{SFP}},q_0\rangle$ be a stream machine and $t := enct(\mathtt{T})$. We can encode the transition function of a machine by means of a tuple which has an entry for each different configuration. The configuration entry is describex by means of a tuple which contains the integers corresponding to the values on the tapes and the current state, together with the information about the first

head movement and the new character. We can argue that this piece of information is enough, because the second tape is read-only and left to right. Given an encoding t, and a current configuration c, we can define the function *matcht* which computes the matching transition, by cases, matching the representation of the current configuration with the values stored in t. After that, with the control structures that we described above, we can define a function *apply* which applies a transition to a state, resulting with a new configuration. Finally we need a function that emulates the execution of the machine for a fixed number of steps, passed as a parameter. A step consists in a lookup of the configuration that matches c in t and in the application of such transition to t, ending up with a new configuration c'. We call such function *step* and define it by induction on the number of steps. Finally we define $eval_M(\sigma, \mathbf{1})$ fixing a polynomial number of steps to the *step* function.

□

Given this result, we will only need to represent in the $\mathcal{POR}$ formalism all the functions that we have described above to conclude the proof.
Expressivity of the $\mathcal{POR}$ Formalism. It can be also shown that all the encodings and functions over the set $\mathbb{D}$ can be expressed in $\mathcal{POR}$. We will skip here the meticulous details of this implementation, just considering a few examples. Let us start with data structures. The set $\mathbb{D}$ corresponds to the set $\mathbb{S}$ of binary strings. Strings over $\mathbb{B}$ are native in $\mathcal{POR}$ so they need not be implemented ($\mathbb{D}_{\mathbb{S}}$ is $\mathbb{S}$ itself). Moreover, the only random access to such strings will be the reading of some bits of the oracle ω, for which $\mathcal{POR}$ has the primitive function, Q. Numbers will be represented in unary notation, starting from the $\mathbf{1}$ string, i.e. $\mathbb{D}_{\mathbb{N}} = \mathbf{1}^+$.

As seen in Section **??**, binary strings are the only datatype in the $\mathcal{POR}$ formalism. In order to prove the expressibility in $\mathcal{POR}$ of our encoding of **SFP**-machines, some auxiliary functions are need. Just to take an example into account, the constant string is expressed in $\mathcal{POR}$ as follows.

Definition 22 (Constant String)**.** A *constant string* $c_0c_1 \dots c_n$ is expressed in $\mathcal{POR}$ as:

$$c_0c_1 \dots c_n := S_{c_n}(\cdots(S_{c_1}(S_{c_0}(E(\mathbf{1}, \omega), \omega), \omega)\cdots), \omega).$$

It is also required to show how to express natural numbers and functions. Also in this case, let us just consider one example of encoding.

Definition 23. The encoding of $\mathbb{N}$ over $\mathbb{S}$ is defined as follows:

$$\overline{\overline{0}} := S_{\mathbf{1}}(\boldsymbol{\epsilon}, \omega)$$
$$\overline{\overline{n+1}} := S_{\mathbf{1}}(\overline{\overline{n}}, \omega).$$

Arithmetic functions can easily be on natural numbers by induction. Then, complex cases, such as predicates and tuples are considered.

As anticipated when discussing the encoding of Booleans in Section **??**, we can represent them as $\overline{\overline{0}}$ and $\overline{\overline{1}}$. We can also leverage the case-by-case definition of bounded iteration for giving the definition of the `if` statement as above.

$$\mathtt{if}'(x_1, x_2, \epsilon, \mathbf{1}) := \epsilon$$
$$\mathtt{if}'(x_1, x_2, y\mathtt{0}, \mathbf{1}) := x_2|_{x_1x_2}$$
$$\mathtt{if}'(x_1, x_2, y\mathtt{1}, \mathbf{1}) := x_1|_{x_1x_2}$$
$$\mathtt{if}(t, f, c, \mathbf{1}) := \mathtt{if}'(t, f, c, \mathbf{1})$$

In order to represent the tuples, we use Odifreddi's notation as described in [**Odifreddi**]. The encoding uses of a couple of functions $\mathcal{D}$ and its left inverse $\mathcal{H}$.

Definition 24 (Encoding and Decoding Functions)**.**

$$\mathcal{D}(\sigma 0) := \mathcal{D}(\sigma)10$$
$$\mathcal{D}(\sigma 1) := \mathcal{D}(\sigma)11$$

$$\mathcal{H}(\sigma 10) = \mathcal{H}(\sigma)0$$
$$\mathcal{H}(\sigma 11) = \mathcal{H}(\sigma)1$$

Definition 25 (Tuple Constructors)**.** We define the family of tuple constructors as the family of function defined as below and indexed by n:

$$\langle x_0, x_1, \ldots, x_n\rangle_\omega := 00\mathcal{D}(x_n)00\ldots 00\mathcal{D}(x_1)00\mathcal{D}(x_n)00\mathcal{D}(\overline{\overline{n}})00$$

Lemma 2. *Our encoding of stream machines can be expressed by functions of $\mathcal{POR}$.*

Proof. We can show that with the functions described above, it is possible to implement the functions *matcht*, *apply*, *step* and *eval* that we described in Lemma 1. For a more comprehensive construction proof of this result, the reader can consult Section **??** of the Appendix. □

Concluding the Proof. Now, by Lemma 1, we have an encoding-formalism which is provably (at least) as expressive as stream machines and, by Lemma 2, such an encoding formalism is definable by $\mathcal{POR}$-functions. So, we can conclude that every function that can be expressed by an **SFP**-machine, can be expressed in $\mathcal{POR}$ as well.

Proposition 4. *For each* **SFP***-machine $M_{\mathbf{SFP}} := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$, there is a $\mathcal{POR}$-function f, such that:*

$$f(\sigma, \omega) = M_{\mathbf{SFP}}(\sigma, \omega).$$

Furthermore,

Lemma 3. *The encoding of the initial state of an* **SFP***-machine $M_{\mathbf{SFP}}(\sigma, \omega)$ inn $\mathcal{POR}$ is polynomial in the size of σ.*

Lemma 4. *The representation in $\mathcal{POR}$ of the complexity bound of an* **SFP***-machine $M_{\mathbf{SFP}}(\sigma, \omega)$ in $\mathcal{POR}$ is polynomial in the size of σ*

0.3. **From $\mathcal{POR}$-functions to SFP-machines.** The first encoding was direct, but maybe too technical. On the contrary, the proof of its converse, i.e. that each function in $\mathcal{POR}$ can be computed by an **SFP**-machine, is indirect and relies on three intermediate formalisms, namely the $\mathcal{POR}^-$-formalism, the **SIMP**-formalism, and the multi-tape Turing machine's one. The proof proceeds as follows:

(1) We first introduce the $\mathcal{POR}^-$-formalism, and encode each $\mathcal{POR}$-function in a $\mathcal{POR}^-$-function.
(2) Then, we define the **SIMP**-formalism with its syntax and operational semantics.
(3) We encode each $\mathcal{POR}^-$-function in a correct **SIMP**-program.
(4) We implement an interpretation for **SIMP**-language on a multi-tape Turing machine, and show that the number of steps required by the interpreter is polynomially bounded by the size of the input.
(5) Finally, we show that each multi-tape Turing machines can be encoded in an **SFP**-machine with a polynomial overhead.

0.3.1. *The $\mathcal{POR}^-$-formalism.* The class of $\mathcal{POR}^-$ functions is defined by removing from the $\mathcal{POR}$-formalism the query function, Q. We basically introduce this class to prove that each oracle, ω, can be queried only on its initial prefix, whose size is polynomial in the size of the input. For this reason, the necessity of an infinite long sequence of random bits vanishes. Furthermore, proceeding in this way, we deal with the oracle as with a standard argument of a $\mathcal{POR}^-$-function. So, we do not need to copy such tape in the corresponding **SFP**: differently, we would have to start copying a polynomially big prefix the oracle at the very beginning of the execution of the machine without even knowing the size of the input.

Definition 26 (The Class $\mathcal{POR}^-$)**.** The *class* $\mathcal{POR}^-$ is the smallest class of functions in the form $\mathbb{S}^{n+1}$ to $\mathbb{S}$ containing:

- The function E^-, such that $E^-(x,\nu) = \boldsymbol{\epsilon}$
- The function $P^-n_i(x_1,\ldots,x_n,\nu) = x_i$, for $n, \in \mathbb{N}$ and $1 \leq i \leq n$
- The function $H^-_n(x_1,\ldots,x_n,\nu) = \nu$ for $n \in \mathbb{N}$
- The function $C^-_b(x,\nu) = x \frown b$, for $\mathbf{b} \in \{\mathbf{0},\mathbf{1}\}$
- The function
$$Q^-(x,y,\nu) = \begin{cases} \mathbf{1} & \text{if } x \subseteq y \\ \mathbf{0} & \text{otherwise} \end{cases}$$

and closed under:

- Composition: f^- is defined from $g, h_1, h_k \in \mathcal{POR}^-$ as:
$$f(x_1,\ldots,x_n,\nu) = g\big(h_1(x_1,\ldots,x_n,\nu),\ldots,h_k(x_1,\ldots,x_n,\nu),\nu\big).$$
- Bounded iteration with bound t: f^- is defined from $g, h_0, h_1 \in \mathcal{POR}^-$
$$\begin{aligned} f(x_1,\ldots,x_n,\boldsymbol{\epsilon},\nu) &= g(x_1,\ldots,x_n,\nu) \\ f(x_1,\ldots,x_n,y\mathbf{0},\nu) &= h_0\big(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y,\nu)\big)|_{t(x_1,\ldots,x_n,y,\nu)} \\ f(x_1,\ldots,x_n,y\mathbf{1},\nu) &= h_1\big(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y,\nu)\big)|_{t(x_1,\ldots,x_n,y,\nu)}. \end{aligned}$$

For simplicity's sake, we will represent with $ite(g,h_1,h_2,t)$ the function obtained by applying the bounded iteration rule to $g, h_0, h_1 \in \mathcal{POR}^-$ and with bound t.

In order to show that all the $\mathcal{POR}$ functions are also in $\mathcal{POR}^-$, we will prove that all the functions in $\mathcal{POR}$ use only a polynomial prefix of the oracle ω.

Lemma 5. *For each $f \in \mathcal{POR}$, the following holds:*
$$\forall x_1,\ldots,x_n.\forall\omega.\exists p \in \mathsf{POLY}\big(f(x_1,\ldots,x_n,\omega)| \leq p(|x_1|,\ldots,|x_n|\big).$$

Proof. The result comes by infuciton on the syntax of f, an thanks to an auxilairy lemma which bounds the size of the terms of $\mathcal{L}_{\mathbb{PW}}$. For a comprehensive proof of the result, the reader is invited to consult Section **??** of the appendix. □

A similar result can be stated for the functions $g \in \mathcal{POR}^-$, namely:

Lemma 6. *For any $f \in \mathcal{POR}^-$,*
$$\forall x_1,\ldots,x_n.\forall\nu.\exists p \in \mathsf{POLY}\big(|f(x_1,\ldots,x_n,\nu)| \leq p(|x_1|,\ldots,|x_n|,|\nu|)\big).$$

Since all the strings obtained by a $\mathcal{POR}$ function are polynomially bounded, we can conclude that every $\mathcal{POR}$-function queries its oracle function only with strings of polynomial length.

For this reason, the $\mathcal{POR}^-$ functions are expressive enough to enocde all the $\mathcal{POR}$ functions: an additional and polynomially long sequence of **0** and **1** bits can substitute the oracle function. Let us start the formalization due to the definition above.

Lemma 7 (Prefix)**.** *Fore any $f \in \mathcal{POR}$,*

$$\exists p \in \mathsf{POLY}.\forall x_1,\ldots,x_k.\forall \omega,\omega'\big(\omega_{p(|x_1|,\ldots,|x_k|)} = \omega'_{p(|x_1|,\ldots,|x_k|)} \to f(x_1,\ldots,x_k,\omega) = f(x_1,\ldots,x_k,\omega')\big).$$

Proof. See Theorem **??** of the Appendix. □

In order to fully substitute the oracle, we need to perform some further encodings. In particular, we need a $\mathcal{POR}^-$-function that simulates the access to the oracle:

$$\begin{aligned} lft(\boldsymbol{\epsilon},\nu) &:= \boldsymbol{\epsilon} \\ lft(y\mathbf{b},\nu) &:= y|_y \end{aligned}$$

$$\begin{aligned} nlft(x,\boldsymbol{\epsilon},\nu) &:= x \\ nlft(x,y\mathbf{b},\nu) &:= lft(nlft(x,y,\nu),\nu)|_y \end{aligned}$$

$$access(x,y\mathbf{b},\nu) := lst(nlft(rv(\nu,\nu),x,\nu),\nu).$$

With a little abuse of notation, we reused the function lft and rv that we showed being in $\mathcal{POR}$. Although we will not define it explicitly, the same construction can be used in order to prove that such functions are in $\mathcal{POR}^-$ too.

Remark 2. *Notice that the following holds:*

$$\begin{aligned} \forall\sigma,\nu.lft(\sigma,\nu) &\subseteq \sigma \\ \forall\sigma,\tau,\nu.nlft(\sigma,\tau,\nu) &\subseteq \sigma \\ \forall\sigma,\tau,\nu.nlft(\sigma,\tau\mathbf{b},\nu) &\subseteq nlft(\sigma,\tau,\nu). \end{aligned}$$

Lemma 8 (Implementation of $\mathcal{POR}$ in $\mathcal{POR}^-$)**.** *For any $f \in \mathcal{POR}$,*

$$\exists p \in \mathsf{POLY}.\exists g \in \mathcal{POR}^-.\forall q \in \mathsf{POLY} p \lesssim q.\forall x_1,\ldots,x_n,\omega\big(f(x_1,\ldots,x_n,\omega) = g(x_1,\ldots,x_n,\omega_{q(x_1,\ldots,x_n)})\big).$$

Proof. Lemma **??** shows that there is a bound on the size of the portion of the oracle that is actually read by f. Wr use such bound for p. We proceed by induction on the definition of f.

- If f is E, Lemma **??** provides bound 0. Indeed, $E(x,\omega) = E^-(\omega,\nu)$, where ν is any polynomially prefix of ω.
- If f is S_0, S_1, C, or P^i the proof is similar to the previous one.
- If f is Q, its implementation in $\mathcal{POR}^-$ is $access(x,\nu)$. So, $Q(\sigma,\omega) = access(\sigma,\nu)$, where ν is any prefix of ω longer than $|\sigma|$.
- In the case of composition, by IH we know that for any i, such that $1 \leq i \leq k$, $\exists p_i \in \mathsf{POLY}.\forall q_i \in \mathsf{POLY}\big(p_i \lesssim q_i \to h_i(x_1,\ldots,x_n,\omega) = h'_i(x_1,\ldots,x_n,\omega_{q_i(|x_1|,\ldots,|x_n|)})\big)$, with $h'_i \in \mathcal{POR}^-$ for each i. Similarly, we know that $\exists p_f \in \mathsf{POLY}.\forall q_f \in \mathsf{POLY}.p \lesssim q.f(x_1,\ldots,x_k,\omega) = f'(x_1,\ldots,x_k,\omega_{q_f(|x_1|,\ldots,|x_k|)})$, with $f' \in \mathcal{POR}^-$. We can start by introducing the polynomials p_i and p_f by IH. Then, we compose p_f with the polynomials that express the size of the arguments of such functions, which exist for Lemma **??**, obtaining p'_f that expresses the actual size of the prefix for ω, read by f. Now, we can apply Lemma **??**, in order to obtain a polynomial which is universally greater than the starting ones, call it p. Finally, since any polynomial which is universally greater than p is also universally greater than p_is and p_f, we can conclude by applying IH.
- In the case of bounded recursion, by IH $\exists p_g \in \mathsf{POLY}.\forall_{q_g} \in \mathsf{POLY}\big(p_q \lesssim q_g \to g(x_1,\ldots,x_k,\omega) = g'(x_1,\ldots,x_k,\omega_{q_g(|x_1|,\ldots,|x_k|)})\big)$, with $g' \in \mathcal{POR}^-$, is bounded by a polynomial p_g in its input. Similarly, we know that $\exists p_0,p_1 \in \mathsf{POLY}.\forall q_0,q_1 \in \mathsf{POLY}\big(p_0 \lesssim q_0 \wedge p_1 \lesssim q_1 \to h_0(x_1,\ldots,x_k,x_{k+1},\omega) = h'_0(x_1,\ldots,x_k,x_{k+1},\omega_{q_0(|x_1|,\ldots,|x_k|,|x_{k+1}|)}) \wedge h_1(x_1,\ldots,x_k,x_{k+1},\omega) =$

$h'_1(x_1,\dots,x_k,x_{k+1},\omega_{p_1(|x_1|,\dots,|x_k|,|x_{k+1}|)})\big)$. As for the previous case, we use Lemma **??**, to obtain an upper bound to the size of the recursive call that allows us to express p_0 and p_1 in function of $x_1,\dots,x_k$, call them p'_0 and p'_1. Now, we can introduce a polynomial universally greater than the one obtained by applying Lemma **??** to get p. This allows us to use all the IHs and to conclude the sub-derivation.

□

So, we can conclude that the $\mathcal{POR}^-$ formalism coincides with Ferreira's **BRS** [**Ferreira90**]. **BRS** is proved to be polynomially interpretable by a single tape Turing machine, which is a particular class the **SFP** one.[2] For the sake of self-containment, we explicitly prove that the $\mathcal{POR}^-$-formalism can be interpreted by a single tape Turing machine with polynomial complexity.

0.3.2. *The* **SIMP** *formalism.* The **SFP** paradigm, which is a subclass of the Turing machine's model, is far from being functional, while $\mathcal{POR}$ is such. As these two formalisms are radically different, a direct encoding of the $\mathcal{POR}$ – or even of the $\mathcal{POR}^-$ – formalism in **SFP** would be complicated. In order to simplify the whole encoding, we will pass through an intermediate imperative paradigm, the *String's Imperative and Minimal Paradigm*, **SIMP**. As we disucssed above, similar results have already been stated; for this reason we will only show some definitions, the claim and a scketch of the proof. We invite the curious readers to consult Section **??** of the Appendix for a complete and a rigorous proof.

The **SIMP** paradigm is defined by an enumerable set of correct paradigms and an operational semantics.[3]

Definition 27 (SIMP-Language)**.** The *language of* **SIMP** *programs* is $L(\mathsf{Stm})$, i.e. the set of strings produced by the non-terminal symbol Stm defined by:

$$\mathsf{Id} ::= X_i \mid Y_i \mid S_i \mid R \mid Q \mid Z \mid T$$
$$\mathsf{Exp} ::= \epsilon \mid \mathsf{Exp}.\mathbf{0} \mid \mathsf{Exp}.\mathbf{1} \mid \mathsf{Id} \mid \mathsf{Exp} \sqsubseteq \mathsf{Exp} \mid \mathsf{Exp} \wedge \mathsf{Exp} \mid \neg\mathsf{Exp}$$
$$\mathsf{Stm} ::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm};\mathsf{Stm} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\} \mid \mathbf{skip};.$$

with $i \in \mathbb{N}$.

Definition 28 (Semantics of **SIMP**)**.** On the **SIMP** language we can define two relations:

- The relation $\rightharpoonup$ which describes the semantics of the expressions.
- The relation $\triangleright$ which describes the semantics of the statements.

Combining the two relations above, we can describe the structural operational semantics of the **SIMP** language.

Definition 29. The value described by a correct **SIMP** program, P, is $\mathcal{F}(P) : L(\mathsf{Stm}) \longrightarrow (\mathbb{S}^n \longrightarrow \mathbb{S})$, where F is defined as above:[4]

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

Now that we have define the notion of value described by a program, we can state the reasult which we are aiming to, namely that each $\mathcal{POR}^-$ function can be represented by a **SIMP** program.

---

[2] Unfortunately, as far as the authors know, the proof of this result is not available

[3] Notice that, by a slight abuse of notation which is quite common in the literature, e.g. [**Winskel**], in what follows, we will use $\boldsymbol{\epsilon}, \mathbf{0}, \mathbf{1}$ in the language of **SIMP**.

[4] We use the prefixed notation for $\triangleright$, instead of the infixed one, in order to express the store associated to the program and the starting store that are between the curly brackets.

Lemma 9 (Implementation of $\mathcal{POR}^-$ in **SIMP**). *For each $f \in \mathcal{POR}^-$, there is a $P \in L(\mathsf{Stm})$:*

$$\forall x_1 \dots x_n . F(P)(x_1, \dots, x_n) = f(x_1, \dots, x_n).$$

The proof of this result uses a large number of thecnical and shows by induction the full encoding any synctactic set of $\mathcal{POR}^-$ functions. For these reasons, we decided to show the proof of this result in the appendix only; for more details, the reader can consult Section **??** of the Appendix.

Another important claim is that our constructions are polynomial, i.e. each programs requires at most polynomial number of $\triangleright$ transitions to terminate.

Proposition 5 (Complexity of **SIMP**). *For each $f \in \mathcal{POR}^-$, $\mathbb{L}(f)$ takes a number of steps which is polynomial in the size of the arguments of f.*

The last non-trivial reduction which we need to show is the one from **SIMP** towards a multi-taped Turing Machine.

Proposition 6. *Each program $p \in$ **SIMP**, which is polynomial and uses k registers can be simulated with polynomial complexity on a $k+2$-tape Turing machine which uses a $\Sigma = \{\mathbf{0}, \mathbf{1}\}$ and $*$ as blank character.*

We won't even give a fromal proof of such result in the Appendix because showing the machine itself would reuire a too large and over-detailed construction. We will only give a rigorous description of the machine's behaviour, without showing it explicitly. The description can be found at the end of Section **??** of the Appendix.

Finally, Proposition 6 entails the results which we were addressing:

Corollary 1. *Each polynomial* **SIMP***-function can be executed with a polynomial complexity on a single-tape Turing machine, which uses a $\Sigma = \{\mathbf{0}, \mathbf{1}\}$ and $*$ as a blank character.*

Corollary 2. *Each polynomial* **SIMP***-function can be computed by an* **SFP***-machine.*

Proof. The result comes from the fact that **SFP**-machines are extensions of single-tape Turing machines, and that a **SIMP**-program can be executed on a single-tape Turing machine with a polynomial complexity. □

Proposition 7. *Each $f \in \mathcal{POR}$ can be computed by an* **SFP***-machine.*

Proof. By Lemmas 8, every function which is in $\mathcal{POR}$ is in $\mathcal{POR}^-$ too. By Lemma **??** too. Finally, we conclude by Corollary 2. □

Theorem 1. $\mathcal{POR} = \mathbf{SFP}$.

Proof. By considering Proposition **??** and Proposition 7. □