1. Task C

In the current section we will prove that every function that can be computed in the **SFP** formalism can be expressed in the $\mathcal{POR}$ formalism, in the following section we will show that the converse holds too. In order to prove the former so we will proceed as follows:

- We will first give a formal definition of the **SFP** machines.
- Then we will define a set of the data structures and functions that can be used in order to encode a **SFP** and to simulate its execution.
- After that, we will prove that all the data structures and functions that we described can be defined in the $\mathcal{POR}$ formalism.
- Finally we will prove our result, namely that all the functions that are calculated by a **SFP** machine are $\mathcal{POR}$ functions, too.

1.1. Definition of the SFP formalism.

Definition 1 (Stream machine)**.** *A Stream Machine is a quadruple* $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$ *where:*

- $\mathbb{Q}$ *is a finite set of states ranged over by* $q_0, q_1, \dots, q_n$.
- Σ *is a finite set of characters ranged over by* $c_0, c_1, \dots, c_n$.
- $\delta_{\mathbf{SFP}} : \mathbb{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}\} \longrightarrow \mathbb{Q} \times \hat{\Sigma} \times \{L, R\}$ *is a transition function that describes the new configuration reached by a* **SFP** *machine.* L, R *are two fixed constants, and* $\hat{\Sigma} = \Sigma \cup \{*\} \wedge * \neq \mathtt{0} \wedge * \neq \mathtt{1}$.
- $q \in \mathbb{Q}$ *is an initial state.*

In the former definition, $*$ denotes a generic blank character that is not part of Σ; mind that assuming $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ would not be reductive. From now on we will denote the blank character $*$ as $c_{|\Sigma|+1}$.

Definition 2 (Configuration of a Stream Machine)**.** *The configuration of a stream machine* M *is a quadruple* $\langle \sigma, q, \tau, \omega\rangle$ *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.
- $\tau \in \hat{\Sigma}^*$ *is the portion of the first tape to the right of the head.*
- $\omega \in \{0,1\}^{\mathbb{N}}$ *is the portion of the second tape that hasn't been read yet.*

Definition 3 (Stream Machine Reachability function)**.** *Given a stream machine* $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$, *we define the partial transition function* $\vdash_{\delta_{\mathbf{SFP}}} : \hat{\Sigma}^* \times \mathbb{Q} \times \hat{\Sigma}^* \times \{0,1\}^{\mathbb{N}} \longrightarrow \hat{\Sigma}^* \times \mathbb{Q} \times \hat{\Sigma}^* \times \{0,1\}^{\mathbb{N}}$ *between two configuration of* M_S *as:*

$$
\begin{aligned}
\langle \sigma, q, c\tau, \mathtt{0}\omega\rangle &\vdash_{\delta_{\mathbf{SFP}}} \langle \sigma c', q', \tau, \omega\rangle && \textit{if } \delta_{\mathbf{SFP}}(q, c, \mathtt{0}) = \langle q', c', R\rangle\\
\langle \sigma c_0, q, c_1\tau, \mathtt{0}\omega\rangle &\vdash_{\delta_{\mathbf{SFP}}} \langle \sigma, q', c_0c_1'\tau, \omega\rangle && \textit{if } \delta_{\mathbf{SFP}}(q, c_1, \mathtt{0}) = \langle q', c_1', L\rangle\\
\langle \sigma, q, c\tau, \mathtt{1}\omega\rangle &\vdash_{\delta_{\mathbf{SFP}}} \langle \sigma c', q', \tau, \omega\rangle && \textit{if } \delta_{\mathbf{SFP}}(q, c, \mathtt{1}) = \langle q', c', R\rangle\\
\langle \sigma c_0, q, c_1\tau, \mathtt{1}\omega\rangle &\vdash_{\delta_{\mathbf{SFP}}} \langle \sigma, q', c_0c_1'\tau, \omega\rangle && \textit{if } \delta_{\mathbf{SFP}}(q, c_1, \mathtt{1}) = \langle q', c_1', L\rangle
\end{aligned}
$$

Definition 4. *Given a stream machine* $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$, *we denote with* $\{\vdash^n_{\delta_{\mathbf{SFP}}}\}_n$ *the smallest family of relations relation for which:*

$$\langle \sigma, q, \tau, \omega\rangle \vdash^0_{\delta_{\mathbf{SFP}}} \langle \sigma, q, \tau, \omega\rangle$$

$$\langle \sigma, q, \tau, \omega\rangle \vdash^n_{\delta_{\mathbf{SFP}}} \langle \sigma', q', \tau', \omega'\rangle \wedge \langle \sigma', q', \tau', \omega'\rangle \vdash_{\delta_{\mathbf{SFP}}} \langle \sigma'', q', \tau'', \omega''\rangle \to \langle \sigma, q, \tau, \omega\rangle \vdash^{n+1}_{\delta_{\mathbf{SFP}}} \langle \sigma'', q', \tau'', \omega''\rangle$$

Lemma 1. $\forall n. \vdash^n_{\delta_{\mathbf{SFP}}}$ *is a function.*

Proof. By induction on n

0 In this case the $\vdash^n_{\delta_{\mathbf{SFP}}}$ is the identity function.

$n+1$ As induction hypothesis we have that $\vdash^n_{\delta_{\mathbf{SFP}}}$ is a function, then, since $\vdash_{\delta_{\mathbf{SFP}}}$ is a function, and $\vdash^{n+1}_{\delta_{\mathbf{SFP}}} = \vdash^n_{\delta_{\mathbf{SFP}}} \circ \vdash_{\delta_{\mathbf{SFP}}}$, we have the thesis.

$\square$

Notation 1. *Given a stream machine $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0 \rangle$ and a configuration $\langle \sigma, q, \tau, \omega \rangle$ we denote with $\langle \sigma, q, \tau, \omega \rangle \not\vdash_{\delta_{\mathbf{SFP}}}$ the following condition:*

$$\neg \exists \sigma', q', \tau', \omega'. \langle \sigma, q, \tau, \omega \rangle \vdash_{\delta_{\mathbf{SFP}}} \langle \sigma', q', \tau', \omega' \rangle$$

Definition 5 (Value computed by a Stream Machine)**.** *Given a machine $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0 \rangle$, we say that M_S computes γ on input σ and oracle ω if and only if:*

$$\exists n. \langle \epsilon, q_0, \sigma, \omega \rangle \vdash^n_{\delta_{\mathbf{SFP}}} \langle \gamma, q', \tau, \psi \rangle \not\vdash_{\delta_{\mathbf{SFP}}}$$

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

Definition 6 (**SFP** Stream Machine)**.** *We say that a stream machine $M_{\mathbf{SFP}} := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0 \rangle$ is a* **SFP** *Stream Machine if and only if:*

$$\exists p \in \mathsf{POLY}. \forall \sigma, \omega, n. \langle \epsilon, q_0, \sigma, \omega \rangle \vdash^n_{\delta_{\mathbf{SFP}}} \langle \gamma, q', \tau, \psi \rangle \not\vdash_{\delta_{\mathbf{SFP}}} \rightarrow n \leq p(|\sigma|)$$

The result that we are addressing in the current section can be now restated as:

Lemma 2 (Representation of Stream Machines in $\mathcal{POR}$)**.** *For every deterministic* **SFP** *machine $M_{\mathbf{SFP}} := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0 \rangle$, there exists a $\mathcal{POR}$ function f such that $f(\sigma, \omega) = M_{\mathbf{SFP}}(\sigma, \omega)$.*

1.2. **Encoding of a SFP Machine, Bottom-up.**

1.2.1. *Basic data structures and functions.* The previous subsection of this work outlines some data structures and functions that we respectively need to represent and compute in order to simulate the execution of a **SFP** machines; for example, since we represented **SFP** machines as tuples of elements, we will need to represent such structures. Furthermore, since we defined the transition function by cases, we need some control structures that will allow us to implement that specific behaviour.

If we want to implement the **SFP** machine in a formalism that works on a domain $\mathbb{D}$, we need to implement in $\mathbb{D}$ the following data structures:

- *Natural* numbers, that will allow us to define an encoding of characters, and states.
- *Boolean* values.
- *Strings* on a binary alphabet.
- *tuples*, by means of which we will encode tapes and configurations.

Notation 2. *We will represent the data structures described above by means of the following notation:*

- *We will use the $\mathbb{D}_S$ notation in order to denote the image of the set S modulo its encoding over $\mathbb{D}$.*
- *We will use the $\overline{n} \in \mathbb{D}_{\mathbb{N}}$ symbol in order to denote the encoding of the number $n \in \mathbb{N}$ in the domain $\mathbb{D}$.*
- *We will use the $\mathtt{1} \in \mathbb{D}_{\{\mathtt{0},\mathtt{1}\}}$ symbol in order to denote the encoding of the true boolean value and the $\mathtt{0}$ symbol for denoting the false boolean value as represented in $\mathbb{D}$.*
- *We will use the symbols $\sigma, \tau \in \mathbb{D}_{\mathbb{S}}$ in order to denote the representation of a string over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ (a string in $\mathbb{S}$) in the domain $\mathbb{D}$.*
- *We will 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. To perform computations on natural numbers, we need at least the following functions:

- Addition, denoted with the $+$ symbol.
- Subtraction, denoted with the $-$ symbol.
- Multiplication, denoted with the $\cdot$ symbol.
- Exponentiation, that we will represent with the common power notation.

The latter function will be useful for expressing the complexity bound of a **SFP** machine, that is a polynomial. All the functions above mentioned need to have the follwing signature:

$$\mathbb{D}_{\mathbb{N}} \times \mathbb{D}_{\mathbb{N}} \longrightarrow \mathbb{D}_{\mathbb{N}}$$

And, if $*$ is a function of the above mentioned, it must hold that:

$$\forall n, m, o \in \mathbb{N}. n * m = o \rightarrow \overline{n} * \overline{m} = \overline{o}$$

Boolean values can be defined as a subset of natural numbers. It means that if we are able to deifne the set $\mathbb{D}_{\mathbb{N}}$, we will be able to define the set $\mathbb{D}_{\{\mathtt{0},\mathtt{1}\}}$ as a subset of the above mentioned set and to implement boolean values and boolean functions as a subset of the natural numbers and of the function defined on such values. For example assigning `0` to the false boolean value and `1` to the true boolean value will do the job. By means of booleans we will be able to define:

- A conditional function over a generical set S, i.e. the $\mathtt{if} : \mathbb{D}_{\{\mathtt{0},\mathtt{1}\}} \times \mathbb{D}_S \times \mathbb{D}_S \longrightarrow \mathbb{D}_S$ function, which respects its commonly intended specification.
- Logical connectives.

The `if` function is an important control function that we will employ in order to determine the configuration that follows the current one. Together with the `if` structure, we will need another simple control structure: the bounded iteration, i.e. a `for` expression. It will allow us to simulate the execution of a **SFP** machine up to its polynomial bound. For all our data structures we need a binary function *eq* that returns `1` if its parameters are equal with respect to the identity and `0` otherwise.

We need binary strings because we will need to access handle the ω tape, in particular we will use a simple function of random access, with the following signature:

$$\cdot[\cdot] : \mathbb{D}_{\{\mathtt{0},\mathtt{1}\}^{\mathbb{N}}} \times \mathbb{D}_{\mathbb{N}} \longrightarrow \{\mathtt{0}, \mathtt{1}\}$$

such that:

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

Definition 7. *Let $\mathbb{T}^n_S$ be 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 we will use in order to build tuples of finite dimension. We will represent this function putting its elements between angular brackets. These functions will have the following sigature: $\langle \cdot, \dots, \cdot \rangle : \mathbb{D}^n_S \longrightarrow \mathbb{D}_{\mathbb{T}^n_S}$. For example $\langle \overline{\mathtt{0}}, \overline{\mathtt{1}} \rangle$, will be the instantiation of the tuple's constructor on the encoding of `1` and `0` as first and second argument.
- A function which computes the size of a tuple, that we denote with $| \cdot | : \mathbb{D}_{\mathbb{T}^n_S} \longrightarrow \mathbb{D}_{\mathbb{N}}$.
- A family of projectors, which we will use in order to extract values from a tuple. We will denote these unary functions with $\pi_i : \mathbb{D}_{\mathbb{N}} \times \mathbb{D}_{\mathbb{T}^n_S} \longrightarrow DD_S$ 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 will return a default value.
- Four manipulators:
  - An unary function which deletes the rightmost element of a tuple, which we call $rmr : \mathbb{D}_{\mathbb{T}^n_S} \longrightarrow \mathbb{D}_{\mathbb{T}^{n-1}_S}$.

– An unary function which deletes the leftmost element of a tuple, which we call $rml : \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n-1}}$.
– A binary function which inserts an element in the rightmost position of a tuple, which we call $addr : \mathbb{D}_S \times \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n+1}}$.
– A function that inserts an element in the leftmost position of a tuple, which we call $addl : \mathbb{D}_S \times \mathbb{D}_{\mathbb{T}_S^n} \longrightarrow \mathbb{D}_{\mathbb{T}_S^{n+1}}$.

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

Definition 8 (Correctness of an encoding of tuples)**.** *We say that an implementation of tuples is correct if and only if:*

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

1.2.2. *Complex data structures and functions.* On top of the data structures that we have recently defined, we can define the encoding of a **SFP** machine and an emulating interpreter as described below.

Definition 9 (Encoding of the transition function)**.** *The encoding of a machine's transition function is defined as above:*

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

And g as:

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

The encoding of the transition function $\delta_{\mathbf{SFP}}$ is finite, since the domain of $\delta_{\mathbf{SFP}}$ is so. Let $\{t_0, \dots, t_N\}$ be the *finite* subset of the set $(\mathbb{Q} \times \hat{\Sigma} \times \{\mathtt{0}, \mathtt{1}\}) \times (\mathbb{Q} \times \hat{\Sigma} \times \{L, R\})$ that describes the function. It is possible to observe that the definition above is only given by means of data structures that we have defined in the previous section, which are natural values and tuples. Furthermore, we represent the transition functions extensively, enumerating all its members.

In the same way we define the representation of tapes ad configurations:

Definition 10 (Encoding of a portion of a tape)**.** *We encode all the finite portions of a tape $\sigma := c_i, \dots c_k$ as:*

$$tenc(c_i, \dots c_k) := \langle \bar{i}, \dots, \bar{k} \rangle$$

Definition 11 (Representation of the configuration of a stream machine)**.** *The representation of the configuration of a stream machine is defined as the 4-uple $\langle \sigma, \bar{i}, \tau, \bar{k} \rangle$ where:*

- *$\sigma = tenc(\sigma')$, where σ' is the shortest portion of the tape that starts from the cell on the immediate* left *of the head is followed (on its* left*) by an infinite sequence of blank characters $*$.*
- *$\bar{i}$ 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 (again on its* right*) by an infinite sequence of blank characters $*$.*
- *$\bar{k}$ 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* $\langle \langle\rangle, \overline{0}, tenc(\sigma), \overline{0}\rangle$.

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

1.2.3. *A first result.* We are going to prove a lemma that states that the previously defined functions can be used to emulate the execution of a Stream Machine. Once this result will be proved, in order to demonstrate the lemma that we are addressing, i.e. 2, we will only need to represent in the $\mathcal{POR}$ formalism all the functions that we have described before.

Lemma 3 (Implementation of **SFP**). *Each formalism which works on a domain* $\mathbb{D}$ *in which it's possible to express the data structures, functions and control primitives described in the subsections 1.2.1 and 1.2.2 and that is closed under composition is at least as expressive as the Stream Machines are.*

Proof. Let $M_S := \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$ be a stream machine and $t := enct(\delta_{\mathbf{SFP}})$ the encoding of its transitions as defined in 9, let $\langle \sigma, \overline{i}', \tau, \overline{p}\rangle$ be the current configuration encoded as defined in 11. We can define on t the function which computes the matching transition as follows:

$$matcht(\langle\rangle, \langle \sigma, \overline{i}', \tau, \overline{p}\rangle, \omega) := \mathsf{0}$$

$$matcht(\langle\langle \overline{i}, \overline{j}, \overline{k}, \overline{l}, d, b\rangle, t_0, \dots, t_m\rangle, \langle \sigma, \overline{i}', \tau, \overline{p}\rangle, \omega) :=$$
$$:= \begin{cases} \langle \overline{i}, \overline{j}, \overline{k}, \overline{l}, d, b\rangle & \text{if } \overline{i} = \overline{i}' \wedge \overline{j} = \pi_1(\tau) \wedge \omega(\overline{p}) = b \\ matcht(\langle t_0, \dots, t_m\rangle, \langle \sigma, \overline{i}', \tau, \overline{p}\rangle, \omega) & \text{otherwise} \end{cases}$$

Now we need to define a function that applies a transition to a state:

$$apply(0, \langle \sigma, \overline{i}', \tau, \overline{p}\rangle) := \langle \sigma, \overline{i}', \tau, \overline{p}\rangle$$
$$apply(\langle \overline{i}, \overline{j}, \overline{k}, \overline{l}, d, b\rangle, \langle \sigma c_1, \overline{i}', c_2\tau, \overline{p}\rangle) := \begin{cases} \langle addr(\sigma c_1, \overline{l}), \overline{k}, \tau, \overline{p}+1\rangle & \text{if } d = R \\ \langle \sigma, \overline{k}, addl(c_1, addl(\overline{l}, \tau)), \overline{p}+1\rangle & \text{otherwise} \end{cases}$$

Finally we need a function that emulates the execution of the machine for a fixed number of steps, passed as a parameter. We define it by induction on the number of steps.

$$step(t, s, 0, \omega) := s$$
$$step(t, s, n+1, \omega) := apply(matcht(t, step(t, s, n, \omega), \omega), step(t, s, n, \omega), n)$$

Finally we define $eval_M(\sigma, \omega)$ as follows

$$eval_{\langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle, n}(\sigma, \omega) := step(enct(\delta_{\mathbf{SFP}}), \langle\langle\rangle, \overline{0}_\omega, tenc(\sigma), \overline{0}\rangle, \overline{n}, \omega)$$

If n is sufficiently big, $eval_{\langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle, n}(\sigma, \omega) = M_S(\sigma, \omega)$

□

1.3. **Expressivity of the $\mathcal{POR}$ formalism.** The aim of this section is to show that all the encodings and functions over the set $\mathbb{D}$ that we introduced in Section 1.2 can be expressed in the $\mathcal{POR}$ formalism. To do so, we need to define some even simpler functions that we will use to build the others.

Before doing so, we will give some intuitions about the actual implementation of the structures:

- The set $\mathbb{D}$ consists in the set $\mathbb{S}$, which is the set of the binary strings.
- The strings over the set $\{\mathtt{0}, \mathtt{1}\}$ are native in $\mathcal{POR}$, so they won't need to be implemented, i.e. $\mathbb{D}_{\mathbb{S}}$ is $\mathbb{S}$ itself; furthermore, the only random access to such strings will be the reading of some bits of the oracle ω for which $\mathcal{POR}$ has a primitive function f_q.
- Numbers will be represented in unary notation, starting from the $\mathtt{1}$ string. This means that $\mathbb{D}_{\mathbb{N}} = \mathtt{1}^+$.

1.3.1. *Preliminaries on Strings.* Since the binary strings are the only datatype that is present in the $\mathcal{POR}$ formalism, we would like to express some basic operations on such data that will turn out to be useful in the following parts.

Given a variable name x, and an oracle ω, every constant can be represented in $\mathcal{POR}$ as follows:

Definition 12 (Constant Strings)**.** *A constant string $c_0c_1 \dots c_n$ can be computed in $\mathcal{POR}$ by means of the following function:*

$$c_0c_1 \dots c_n \coloneqq C_{c_n}(\dots(C_{c_1}(C_{c_0}(E(x, \omega), \omega), \omega), \dots), \omega)$$

Similarly, the concatenation between two generic strings can be defined as follows:

Definition 13 (Concatenation of Strings)**.**

$$\begin{aligned}
concat(x, \epsilon, \omega) &\coloneqq x\\
concat(x, y\mathtt{0}, \omega) &\coloneqq C_0(concat(x, y, \omega), \omega)|_{xy}\\
concat(x, y\mathtt{1}, \omega) &\coloneqq C_1(concat(x, y, \omega), \omega)|_{xy}
\end{aligned}$$

We also define the meta-language notation $|\cdot| : \mathbb{S} \longrightarrow \mathbb{N}$ that represents the size of a string, further we will introduce a $\mathcal{POR}$ function that computes the size of a tuple $|\cdot|_\omega : SS_{\mathbb{T}^n_{\mathbb{S}}} \longrightarrow \mathbb{S}_{\mathbb{N}}$. Those functions are not the same object.

Since constants and concatenations of strings are representable in $\mathcal{POR}$, we will use respectively their explicit representation and juxtaposition for representing those operations.

Notation 3 (Constant Strings)**.** *When introducing constant string we will use their explicit notation instead of writing their definition in $\mathcal{POR}$, for example:*

$$concat(C_0(C_1(E(x, \omega))), C_1(C_0(E(x, \omega)), \omega)$$

Will be written as $\mathtt{1001}$ *and $E(x, \omega)$ will be written as ϵ.*

Before going further with the encoding of numerals and arithmetical operations, we would like to point out the fact that all the $\mathcal{POR}$ functions need to have an oracle as parameter: for example, we passed ω to the *concat* function. When dealing with data structures, and manipulating functions, it will often be useless but, at least inside the definitions, we will use it.

The implementation of some functions will be easier if we can use a function that reverses the strings. Such a function can be defined in the $\mathcal{POR}$ formalism as follows:

Definition 14 (Reversing Function)**.**

$$\begin{aligned}
rv(\epsilon, \omega) &\coloneqq \epsilon\\
rv(y\mathtt{0}, \omega) &\coloneqq \mathtt{0}rv(y, \omega)|_{y\mathtt{0}}\\
rv(y\mathtt{1}, \omega) &\coloneqq \mathtt{1}rv(y, \omega)|_{y\mathtt{0}}
\end{aligned}$$

1.3.2. *Natural Numbers.* All the finite natural numbers can be represented as follows:

Definition 15 (Encoding of $\mathbb{N}$ over $\mathbb{S}$).

$$\begin{aligned} \overline{0}_\omega &:= C_1(\epsilon, \omega) \\ \overline{n+1}_\omega &:= C_1(\overline{n}_\omega, \omega) \end{aligned}$$

Remark 2 (Set $\mathbb{S}_\mathbb{N}$). *As we pointed out before, there's a bijection between* $\mathtt{1}^+ \subset \mathbb{S}$ *and* $\mathbb{N}$*, so* $\mathbb{S}_\mathbb{N} = \mathtt{1}^+$.

Remark 3 (Size of numbers).

$$\forall n \in \mathbb{N}. |\overline{n}_\omega| = n + 1$$

Remark 4 (Appropriateness of $\mathbb{S}_\mathbb{N}$). *We say that* $\mathbb{S}_\mathbb{N}$ *is an appropriate representation of* $\mathbb{N}$*, meaning that:*

- $\forall e \in \mathbb{S}_\mathbb{N}. \exists! n \in \mathbb{N}. \overline{n}_\omega = e$
- $\forall n \in \mathbb{N}. \exists! e \in \mathbb{S}_\mathbb{N}. \overline{n}_\omega = e$

Proof. The results can be respectively obtained by induction on the size of the element and on the value of n. □

The successor of a number n, passed to the formal parameter y, can be calculated simply adding an `1` at the end of n, i.e. as follows:

Definition 16 (Successor function). *We define a function* $S : \mathbb{S}_\mathbb{N} \longrightarrow \mathbb{S}_N N$ *that computes the successor of a number:*

$$\begin{aligned} S(\epsilon, \omega) &:= \epsilon \\ S(y\mathtt{0}, \omega) &:= C_\mathtt{0}(\epsilon, \omega)|_{y\mathtt{11}} \quad (*) \\ S(y\mathtt{1}, \omega) &:= C_\mathtt{1}(C_\mathtt{1}(y), \omega)|_{y\mathtt{11}} \end{aligned}$$

Since the representation of a number is only composed by `1`s, the row marked with $(*)$ is useless when dealing with natural numbers. All the other functions that we will implement for the manipulation of numbers will present a similar issue, but sometimes, those definition will turn out to be useful.

Definition 17 (Predecessor of a Natural Number). *If* $y \in \mathbb{S}_\mathbb{N}$ *is the encoding of a number, the pd function calculates its predecessor simply by removing its last digit.*

$$\begin{aligned} pd(\epsilon, \omega) &:= \epsilon \\ pd(y\mathtt{0}, \omega) &:= y|_y \\ pd(y\mathtt{1}, \omega) &:= y|_y \end{aligned}$$

Remark 5.

$$\forall \sigma, c_1, c_2. pd(pd(\sigma c_1 c_2)) = \sigma$$

Proof. The claim is a trivial consequence of the definition of *pd*. □

Notation 4 (Oracle Included Function). *From now on, we will use the notation* op_ω *to denote an operator op that uses* $\omega \in \mathbb{B}$ *as oracle, i.e. the expression* $x op_\omega y$ *will be a shorthand for* $op(x, y, \omega)$.

Definition 18 (Sum of two Natural Numbers)**.** *The sum of two numbers $+_\omega$ can be easy implemented by using the operation of concatenation that we introduced in 1.3.1.*

$$\overline{n} +_\omega \overline{m} = pd(concat(\overline{n}, \overline{m}, \omega))$$

The $\mathcal{POR}$ encoding of the difference between two numbers is quite cumbersome, because the only form of recursion that is allowed by such formalism is on the longest non trivial prefix of a single argument. Intuitively we can decrease the measure of both the numbers since the second is ϵ; at that time, we have decreased the first argument one too many, so we need to return its successor.

Definition 19 (Difference of two Natural Numbers)**.** *We define the function $-_\omega$ which encodes the difference between two natural numbers as follows*

$$\begin{aligned} x -_\omega \epsilon &:= S(x, \omega) \\ x -_\omega y\mathtt{0} &:= pd(x, \omega) -_\omega y|_x \\ x -_\omega y\mathtt{1} &:= pd(x, \omega) -_\omega y|_x \end{aligned}$$

In order to multiply two values x and y, we can remove their last digit, so that their size is equal to the number that they encode, then concatenate x to itself y time and return the successor of the number that we get; formally:

Definition 20 (Multiplication of two Natural Numbers)**.** *We define the multiplication between two natural numbers $\cdot_\omega$ as follows:*

$$\begin{aligned} x \cdot^*_\omega \epsilon &:= \epsilon \\ x \cdot^*_\omega y\mathtt{0} &:= ((x \cdot^*_\omega y) +_\omega x)|_{x \times y\mathtt{1}} \\ x \cdot^*_\omega y\mathtt{1} &:= ((x \cdot^*_\omega y) +_\omega x)|_{x \times y\mathtt{1}} \end{aligned}$$

$$x \cdot_\omega y := S(pd(x, \omega) \cdot^*_\omega pd(y, \omega))$$

With this encoding we cannot go much further since the computation of an exponential would require an exponential size for the representation of the output, but our iteration is bounded by a term in $L_{\mathbb{W}}$, but the size of a number in our encoding is linear in its value. However, we can still represent monomials, and so polynomials, as follows:

Definition 21 (Monomials)**.** *Given a $k \in \mathbb{N}$, we define the function that computes the value $\overline{n^k}_\omega$ as follows:*

$$\forall k \in \mathbb{N}. \overline{n}^k_\omega := (\prod_{i=0}^{k})_\omega \overline{n}$$

where:

$$(\prod_{i=0}^{k})_\omega e_i := 1 \cdot_\omega e_0 \cdot_\omega e_1 \cdot_\omega \dots \cdot_\omega e_k$$

1.3.3. *Boolean algebra.* Before going further, it's time to define some predicates. As it happens in the definition of the $\mathcal{POR}$ function Q, we say that a predicate $P(\vec{x})$ is true if (and only if) it returns $\mathtt{1}$, it's false if it returns $\mathtt{0}$, otherwise it's undefined. Let's start with some zero-order logic and predicates.

Given that x_1 and x_2 are values, we can define a function that returns x_1 if a condition y is met, x_2 if such condition is false, and ϵ otherwise. Such function behaves as an `if` expression:

Definition 22 (`if` expression)**.**

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

Definition 23 (Logical connectives)**.** *Thanks to the maththttif function, we can define some basic (and complete) connectives for the propositional logic:*

$$\begin{aligned}(P_1 \wedge P_2)(\vec{x}, \omega) &:= \mathtt{if}(P_2(\vec{x}, \omega), \mathtt{0}, P_1(\vec{x}, \omega), \omega)\\ (P_1 \vee P_2)(\vec{x}, \omega) &:= \mathtt{if}(\mathtt{1}, P_2(\vec{x}, \omega), P_1(\vec{x}, \omega), \omega)\\ (\neg P)(\vec{x}, \omega) &:= \mathtt{if}(\mathtt{0}, \mathtt{1}, P(\vec{x}, \omega), \omega)\end{aligned}$$

Now we define some predicates that will help us to develop the tuple's encoding. In particular, we will represent such structures, representing their values with an encoding that prefixes a `1` to each bit of their binary representation. For this reason, when we will decode a tuple's value it will be useful to know whether the length of the remaining part of such value is even or odd in order to decide whether it's the case to keep or remove a certain bit.

Definition 24 (Basic Logical predicates)**.** *The basic logical predicates in $\mathcal{POR}$ arer the function defined below.*

$$\begin{aligned}odd(\epsilon, \omega) &:= \mathtt{0}\\ odd(y\mathtt{0}, \omega) &:= \neg(odd(y))|_{\mathtt{0}}\\ odd(y\mathtt{1}, \omega) &:= \neg(odd(y))|_{\mathtt{0}}\\ even(x, \omega) &:= \neg odd(x, \omega)\\ eq(x, y, \omega) &:= Q(x, y, \omega) \wedge Q(y, x, \omega)\end{aligned}$$

It's important to observe that the *odd* and *even* predicates work as their opposites for the encoding of natural numbers, i.e. the following remark holds:

Remark 6 (Even and odd's idiosincrasy)**.**

$$\begin{gathered}\forall \sigma \in \{\mathtt{0}, \mathtt{1}\}^*.\forall b \in \{\mathtt{0}, \mathtt{1}\}.odd(\sigma) \leftrightarrow even(\sigma b)\\ \forall n \in \mathbb{N}.odd(\overline{n}_\omega) \leftrightarrow n \textit{ is even.}\end{gathered}$$

Before defining tuples, let's proceed with the definition of two string-specific predicates:

Definition 25 (String specific Predicates)**.** *For working with strings we define the followinf predicates which respectively extract the rightmost and the leftmost bit of the string.*

$$\begin{aligned} lst(\epsilon,\omega) &:= \epsilon\\ lst(y\mathtt{0},\omega) &:= \mathtt{0}|_{\mathtt{1}}\\ lst(y\mathtt{1},\omega) &:= \mathtt{1}|_{\mathtt{1}}\\ \\ fst(\epsilon,\omega) &:= \epsilon\\ fst(y\mathtt{0},\omega) &:= If(\mathtt{0}, fst(y,\omega), Q(y,\epsilon,\omega),\omega)|_{\mathtt{1}}\\ fst(y\mathtt{1},\omega) &:= If(\mathtt{1}, fst(y,\omega), Q(y,\epsilon,\omega),\omega)|_{\mathtt{1}} \end{aligned}$$

Remark 7.

$$\forall\sigma\in\{\mathtt{0},\mathtt{1}\}^*.lst(\sigma\mathtt{1})=\mathtt{1}\wedge lst(\sigma\mathtt{0})=\mathtt{0}$$
$$\forall\sigma\in\{\mathtt{0},\mathtt{1}\}^*.fst(\mathtt{1}\sigma)=\mathtt{1}\wedge lst(\mathtt{0}\sigma)=\mathtt{0}$$

1.3.4. *tuples.* In order to represent the tuples, we use Odifreddi's notation as described in (vol 2 p.183). The encoding, that we have briefly introduced in 1.3.3 makes use of a couple of functions $\mathcal{D}$ and its left inverse $\mathcal{H}$.

Definition 26 (Encoding and Decoding Functions)**.**

$$\begin{aligned} \mathcal{D}(\sigma\mathtt{0}) &:= \mathcal{D}(\sigma)\mathtt{10}\\ \mathcal{D}(\sigma\mathtt{1}) &:= \mathcal{D}(\sigma)\mathtt{11}\\ \\ \mathcal{H}(\sigma\mathtt{10}) &= \mathcal{H}(\sigma)\mathtt{0}\\ \mathcal{H}(\sigma\mathtt{11}) &= \mathcal{H}(\sigma)\mathtt{1} \end{aligned}$$

Thanks to this simple encoding we can represent tuples simply by juxtaposing their values separated by a special character, for example $\mathtt{00}$, because such seuqence can't be generated by $\mathcal{D}$ (the proof of such result can be shown by induction on its first argument).

It is possible to show that the doubling function $\mathcal{D}$ is a $\mathcal{POR}$ function

$$\begin{aligned} \mathcal{D}(\epsilon,\omega) &:= \epsilon\\ \mathcal{D}(y\mathtt{1},\omega) &:= C_{\mathtt{1}}(C_{\mathtt{1}}(\mathcal{D}(y,\omega),\omega))|_{(\mathtt{11})\times(y\mathtt{1})}\\ \mathcal{D}(y\mathtt{0},\omega) &:= C_{\mathtt{0}}(C_{\mathtt{1}}(\mathcal{D}(y,\omega),\omega))|_{(\mathtt{11})\times(y\mathtt{1})} \end{aligned}$$

It is easy to see that, given any string $c_0c_1\dots c_n$, $\mathcal{D}(c_0c_1\dots c_n)=1c_01c_1\dots 1c_n$. The function $\mathcal{H}$ is in $\mathcal{POR}$ as well as the doubling function $\mathcal{D}$. We can define it as follows:

$$\begin{aligned} \mathcal{H}(\epsilon,\omega) &:= \epsilon\\ \mathcal{H}(y\mathtt{0},\omega) &:= concat(\mathcal{H}(y,\omega), If(\mathtt{0},\epsilon, odd(y,\omega),\omega))|y\mathtt{0}\\ \mathcal{H}(y\mathtt{1},\omega) &:= concat(\mathcal{H}(y,\omega), If(\mathtt{1},\epsilon, odd(y,\omega),\omega))|y\mathtt{0} \end{aligned}$$

Lemma 4 ($\mathcal{D}$'s left-inverse)**.** $\forall\sigma\in\{0,1\}^*,\omega,\mathcal{H}(\mathcal{D}(\sigma,\omega),\omega)=\sigma$

Proof. By (right) induction on σ:

ϵ The thesis comes from a trivial rewriting of the two functions' bodies.

τc The thesis is

$$\mathcal{H}(\mathcal{D}(\tau c, \omega), \omega) = \tau c$$
$$\mathcal{H}(\mathcal{D}(\tau, \omega)\mathtt{1}c, \omega) = \tau c$$

By induction on σ we can also prove that $\forall\sigma.\mathit{odd}(\mathcal{D}(\sigma)) = 0$, so we can simplify our claim as follows:

$$\mathcal{H}(\mathcal{D}(\tau, \omega)\mathtt{1}c, \omega) = \tau c$$
$$\mathcal{H}(\mathcal{D}(\tau, \omega)\mathtt{1}, \omega)c = \tau c$$
$$\mathcal{H}(\mathcal{D}(\tau, \omega)\mathtt{1}, \omega) = \tau$$

We have argued that $\forall\sigma.\mathit{odd}(\mathcal{D}(\sigma)) = 0$, so we can state the claim as:

$$\mathcal{H}(\mathcal{D}(\tau, \omega)\mathtt{1}, \omega) = \tau$$
$$\mathcal{H}(\mathcal{D}(\tau, \omega), \omega) = \tau$$

Which is the induction hypothesis, so we proved our lemma.

□

We can finally define the encoding of tuples as follows:

Definition 27 (Tuple Contructors)**.** *We define the family of tuple constructors as the family of function defined as below and indexed by n:*

$$\langle x_0, x_1, \dots, x_n\rangle_\omega \coloneqq 00\mathcal{D}(x_n)00\dots00\mathcal{D}(x_1)00\mathcal{D}(x_n)00\mathcal{D}(\overline{n}_\omega)00$$

We represent tuples of string by encoding all the possible values with sequences of two characters, and using `00` as separators. We now implement a function that allows us to remove the initial separators. The function(s) $\langle\cdot\rangle_\cdot$ is in $\mathcal{POR}$ because t-hey are defined by means of composition of concatenation and $\mathcal{D}$, that are both in $\mathcal{POR}$.

The definition of the tuple's constructor introduces a contable set of function, rather than a single function. Further we will show how to parameterize in $\mathcal{POR}$ such functions.

Remark 8 (tuple's size)**.** *the size of a tuple* $\langle x_0, \dots, x_n\rangle_\omega$ *is* $O(n + \mathit{max}(|x_0|, \dots, |x_n|))$.

Proof. the size of a tuple can be expressed as $\sum_{i=0}^{n} 2\cdot|x_n + 3n$ that is in $O(n + \max(|x_0|, \dots, |x_n|))$. □

Now, it's time to introduce the projectors, we will build them by many step. The first consists in the definition of a function that removes the separators (namely `00`) from the encoding of a tuple.

Definition 28 (Remover of the separator)**.** *We define rmsep, i.e. the function which is intended to remove a separator in a tuple as the double nesting of the pd function, namely:*

$$\mathit{rmsep}(x) \coloneqq \mathit{pd}(\mathit{pd}(x))$$

We start describing how to extract the right-most component. To do so, we define the following functions:

- sz which returns $\mathtt{1}$ if ad only if the rightmost element of its first argument is $\mathtt{0}$, and otherwise it returns $\mathtt{0}$.
- rc' which extracts the rightmost element of a $t$uple, without decoding it.
- rc which is basically the function obtained wrapping rc' with $\mathcal{H}$ in order to decode the tuple's encoding.

Definition 29 (Functions for values' extraction)**.** *The three functions that we have described above are implementable in $\mathcal{POR}$ as follows:*

$$\begin{aligned} sz(x,\omega) &:= eq(lst(x), \mathtt{0}) \\ rc'(\epsilon,\omega) &:= \epsilon \\ rc'(y\mathtt{0},\omega) &:= \mathtt{if}(\epsilon, concat(rc'(y,\omega),\mathtt{0}), sz(y),\omega)|_{y\mathtt{0}} \\ rc'(y\mathtt{1},\omega) &:= concat(rc'(y,\omega),\mathtt{1},\omega))|_{y\mathtt{0}} \\ rc(t,\omega) &:= \mathcal{H}(rc'(rmsep(t),\omega),\omega) \end{aligned}$$

Remark 9 (Correctness of rc)**.** *the following statements are valid:*

- $\forall x_0, x_1, \dots, x_n, \omega. rc'(00\mathcal{D}(x_0,\omega)00\mathcal{D}(x_1,\omega)00\dots00\mathcal{D}(x_n,\omega)) = \mathcal{D}(x_n)$
- $\forall x_0, x_1, \dots, x_n, \omega. rc(00\mathcal{D}(x_0,\omega)00\mathcal{D}(x_1,\omega)00\dots00\mathcal{D}(x_n,\omega)00) = x_n$

The only non-trivial statement is the first, that comes from the fact that $\mathtt{00}$ cannot appear inside $\mathcal{D}(y,\omega)$.

We can define a function which extracts the left sub-tuple of a tuple in a similar fashion to how we defined the rc function. This function is aimed to compute the part of a tuple that isn't returned by rc. Such value isn't actually a tuple, because, our definition records the cardinality of the tuple in its right-most element; so the values returned by lc aren't tuples because their rightmost element doesn't necessairly encode the tuple's cardinality.

The lc function is in $\mathcal{POR}$, indeed:

Definition 30 (Left sub-tuple)**.** *The funtion lc which computes the left sub-tuple of a tuple t is defined as follows:*

$$\begin{aligned} lc'(\epsilon,\omega) &:= \epsilon \\ lc'(y\mathtt{0},\omega) &:= \mathtt{if}(y\mathtt{0}, lc'(y,\omega), sz(y) \wedge_\omega odd(t),\omega)|_y \\ lc'(y\mathtt{1},\omega) &:= lc'(y)|_y \\ lc(t,\omega) &:= lc'(rmsep(t,\omega)) \end{aligned}$$

Please observe that if the function lc is applied on tuples, the condition $odd(t)$ is not required because when reading the encoding of a tuple from right to left, if we find a sequence $\mathtt{00}$, it's due to the presence of a separator. Differently, if we apply lc to that value obtained reversing a tuple, we can find the sequence $\mathtt{000}$, in that case we need to stop after that we have read the first tho $\mathtt{0}$. For doing that, we leverage the odd predicate which is true if the remaining part of the encoding has odd length. More formally we can state the correctness of our definition throughout the following remarks.

Remark 10 (Correctness of lc)**.** *The following statement holds:*

$$\forall \sigma, \tau, x, \omega. lc'(\sigma\mathtt{00}\mathcal{D}(\tau)) = \sigma\mathtt{00}$$

Remark 11 (Left component of the reverse of a tuple)**.**

$$rv(lc(rv(\mathtt{00}\mathcal{D}(x_0)\mathtt{00}\mathcal{D}(x_1)\mathtt{00}\dots\mathtt{00}\mathcal{D}(x_n)\mathtt{00}))) = \mathtt{00}\mathcal{D}(x_1)\mathtt{00}\dots\mathtt{00}\mathcal{D}(x_n)\mathtt{00}$$

For every $n \in \mathbb{N}$, we define the n-th projector of a tuple by nesting n calls to lc, the resulting value will have the n-th element of the staring tuple as its rightmost element, we recall that the values inside a tuple are stored in decreasing left to right order.

Definition 31 (Family of projectors)**.** *we define the family of projectors π_n as below. We also overload the symbol π with the definition of a function which takes $\overline{n}_\omega$ and behaves the same as π_n.*

$$\begin{aligned}
\pi'(t, \epsilon, \omega) &:= t\\
\pi'(t, y\mathtt{0}, \omega) &:= lc(\pi'(t, y, \omega), \omega)|_t\\
\pi'(t, y\mathtt{1}, \omega) &:= lc(\pi'(t, y, \omega), \omega)|_t\\
\\
\pi_n(t, \omega) &:= rc(\pi'(t, pd(\overline{n}_\omega), \omega))\\
\pi(t, x, \omega) &:= rc(\pi'(t, pd(x), \omega))
\end{aligned}$$

We intentionally overloaded the π in order to increase the readability of future definitions.

Remark 12 (Correctness of the tuple's encoding)**.**

$$\begin{gathered}
\pi_n(t, \omega) = rc(lc^n(t, \omega))\\
\forall n, \omega. \forall x_1, \dots, x_{n-1}, x_n. \forall 1 \leq k \leq n. \pi_k(\langle x_1, \dots, x_{n-1}, x_n\rangle_\omega) = x_k\\
\forall n, \omega. \forall x_1, \dots, x_{n-1}, x_n. \forall k > n. \pi_k(\langle x_1, \dots, x_{n-1}, x_n\rangle_\omega) = \epsilon\\
\forall n, \omega. \forall x_1, \dots, x_{n-1}, x_n. \pi_0(\langle x_1, \dots, x_{n-1}, x_n\rangle_\omega) = \overline{n}_\omega\\
\forall t, n, \omega. \pi(t, \overline{n}_\omega, \omega) = \pi_n(t, \omega)\\
\forall n \geq 1, m. \pi_n(\mathtt{1}^m) = \epsilon
\end{gathered}$$

As a corollary of the previous remark, we have can define the function which computes the size of a tuple as follows:

$$|\langle x_0, x_1, \dots, x_n\rangle_\omega|_\omega := \pi_0(\langle x_0, x_1, \dots, x_n\rangle_\omega, \omega)$$

Now we need to define the modifiers that we introduced in 1.2.1; Let us start with the modifiers which removes the right- or the left-most element of a tuple.

Definition 32 (Element Removers)**.** *We define the removers that we introduced in 1.2.1 as follows:*

$$\begin{aligned}
rmr(t, \omega) &:= lc(lc(t))\mathcal{D}(pd(|t|_\omega, \omega), \omega)\mathtt{00}\\
rml(t, \omega) &:= lc(rv(lc(rv(t, \omega), \omega), \omega), \omega)\mathcal{D}(pd(|t|_\omega, \omega), \omega)\mathtt{00}
\end{aligned}$$

The rmr function applies two times lc in order to remove the length of the tuple and the last element, after that, it appends the decreased and re-encoded length of the tuple.

The tuple obtained by removing the leftmost element can be obtained by reversing the tuple, using lc and then reversing the tuple again, as a consequence of remark 11. This part of the task is accomplished thanks to the three inner nested function calls to rc and lv, then the last value (the length) is removed and the length of the tuple is updated.

Similarly we can add a new element to a tuple following the same pattern that we used above: we remove the old length, perform the modification that we need (i.e. we add the element) and then we append the updated length.Finally, we can add an element to the left of a tuple, reversing the tuple, computing the encoding of the value throughout $\mathcal{D}$, appending it to the tuple followed by a new separator and then reversing again the tuple. Then we only need to update the tuple's length.

Definition 33 (Element Appenders)**.** *We define the appenders that we introduced in 1.2.1 as follows:*

$$\begin{aligned} addr(t,x,\omega) &:= lc(t,\omega)\mathcal{D}(x,\omega)\mathtt{00}\mathcal{D}(S(|t|_\omega,\omega),\omega)\mathtt{00} \\ addl(t,x,\omega) &:= lc(rv(rv(t,\omega)rv(\mathcal{D}(x,\omega),\omega)\mathtt{00},\omega),\omega)S(|t|_\omega)\mathtt{00} \end{aligned}$$

Now we can also give an inductive definition of the tuple constructors:

Definition 34 (Inductive definiton of tuple's contructors)**.**

$$\begin{aligned} \langle\rangle_\omega &:= \mathtt{001100} \\ \langle x_0,\dots,x_{n-1},x_n\rangle_\omega &:= addr(\langle x_0,\dots,x_{n-1}\rangle\omega, x_n,\omega) \end{aligned}$$

1.4. **The first inclusion.** Now, all the basic functions introduced in Section 1.2.1 have been defined in $\mathcal{POR}$. Now we should proceed by showing the $\mathcal{POR}$ implementation of the functions in Section 1.2.2.

Lemma 5. *The functions matcht, apply, step and eval as defined in lemma 3 are in $\mathcal{POR}$.*

Proof. We can define the function *matcht* by induction on the cardinality of the tuple. Given t the encoding of the transition function δ by means of tuples, as defined in 9, and the current configuration c, as defined in 11. The function *matcht* analyses all the elements of the encoding of δ and checks:

- Whether the current element of ω is corresponds to the value in the transition.
- Whether the current state matches with the one reported in the transition.
- Whether the value on the main tape corresponds to the value reported in t.

Its implementation under the $\mathcal{POR}$ formalism is the following.

$$\begin{aligned} matcht'(t,\epsilon,c,\omega) &:= \mathtt{0} \\ matcht'(t,yb,c,\omega) &:= \mathtt{if}(\pi(t,yb,\omega), matcht'(t,y,c,\omega), \\ &\qquad eq(\pi_1(\pi(t,yb,\omega)),\pi_2(c,\omega),\omega)\wedge_\omega \\ &\qquad eq(\pi_2(\pi(t,yb,\omega)),\pi_1(\pi_3(c,\omega),\omega),\omega)\wedge_\omega \\ &\qquad eq(query(pd(\pi_4(c,\omega)),\omega),\pi_6(\pi(t,yb,\omega),\omega),\omega),\omega)|_t \\ matcht(t,c,\omega) &:= nextt'(t,|t|_\omega,c,\omega) \end{aligned}$$

The application of a transition to the current configuration acts as follows:

(1) It checks if the transition is $\mathtt{0}$, if it is true, the configuration is unchanged.
(2) It checks if the transition moves the head to the right or to the left by means of the condition $eq(\pi_5(t,\omega),\overline{\mathtt{0}}_\omega)$.
(3) It computes builds a new tuple that encodes the resulting configuration

$$\begin{aligned}apply(t,s,\omega) \coloneqq \mathtt{if}(s, \mathtt{if}(&\langle rmr(\pi_1(s,\omega),\omega),\\ &\pi_3(t,\omega),\\ &addl(addl(\pi_3(s,\omega),\pi_4(t),\omega), rc(lc(\pi_1(s,\omega),\omega),\omega),\omega),\\ &S(\pi_4(s,\omega),\omega)\rangle_\omega,\\ &\langle addr(rmr(\pi_1(s,\omega),\omega),\pi_4(t),\omega),\pi_3(t,\omega), rml(\pi_3(s,\omega)\omega), S(\pi_4(s,\omega),\omega)\rangle_\omega,\\ &eq(\pi_5(t,\omega),\overline{0}_\omega),\omega), eq(t,0,\omega),\omega)\end{aligned}$$

Finally the function *step* of lemma 3 can be easily translated in $\mathcal{POR}$.

$$\begin{aligned}step'(t,s,\epsilon,\omega) &\coloneqq s\\ step'(t,s,yb,\omega) &\coloneqq apply(nextt(t, step'(s,y,\omega),\omega), step'(s,y,\omega),\omega)\end{aligned}$$

$$step(t,s,x,\omega) \coloneqq step(t,s,pd(x,\omega),\omega)$$

Finally we define $eval_\mu(\sigma,\omega)$ as follows

$$eval_{\mu,n}(\sigma,\omega) \coloneqq step(\pi_3(\mu,\omega), \langle\langle\rangle_\omega, \pi_4(\mu,\omega)_\omega, tenc(\sigma), \overline{0}_\omega\rangle_\omega, \overline{n}_\omega,\omega)$$

□

Finally we can prove that that every function that can be expressed by a **SFP** machine can be expressed in $\mathcal{POR}$, too.

Proposition 1. *For each* **SFP** *machine* $M_{\mathbf{SFP}} \coloneqq \langle \mathbb{Q}, \Sigma, \delta_{\mathbf{SFP}}, q_0\rangle$*, there exists a* $\mathcal{POR}$ *function* f *such that* $f(\sigma,\omega) = M_{\mathbf{SFP}}(\sigma,\omega)$*.*

Proof. Summing together lemma 3 and 5, we get the result. □

Lemma 6. *The encoding of the initial state of a* **SFP** *machine* $M(\sigma,\omega)$ *in* $\mathcal{POR}$ *is polynomial in the size* σ*.*

Proof. The size of a tuple is given by the remark 8, and thanks to it we can prove that the encoding of the initial state of the machine is $O(4 + max(6 + |tenc(\sigma)|))$. The size of the representation of $tenc(\sigma)$ in $\mathcal{POR}$ is linear in the size of σ: Σ has a constant number of characters k, so the size of the encoding of a string σ is $O(|\sigma| + 2k)$, that is $O(|\sigma|)$, for that reason the encoding of the initial state of the machine is polynomial in the size of the initial value σ. □

Lemma 7. *The representation in* $\mathcal{POR}$ *of the complexity bound of a* **SFP** *machine* $M(\sigma,\omega)$ *in* $\mathcal{POR}$ *is polynomial in the size* σ*.*

Proof. By definition of **SFP** machine, there exists a polynomial p, that expresses the bound in the size of the encoding of σ, i.e. $|\sigma|$, the size of $\overline{p}(|\sigma|)_\omega$ is still polynomial in $|\sigma|$, because of remark 3. □

1.5. **The other direction.** The first encoding was direct, but maybe too much technical for being appreciated. Contrarily, the proof of its converse, i.e. that each function stat is in $\mathcal{POR}$ can be computed by a **SFP** machine will be indirect, and will use three intermediate formalisms, namely, the $\mathcal{POR}^-$ formalism, the $\mathcal{SIMP}$ formalism, then the Multi-tape Turing Machines' one and finally the formalism of the **SFP** machines.

This section will be organized as follows:

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

1.5.1. *The $\mathcal{POR}^-$ formalism.* This class of functions is obtained by removing the query function f_q from the $\mathcal{POR}$ formalism. The main reason that induced us to define such class is that we will 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 infinitely long sequence of random bits vanishes.

Moreover, proceeding in this way, we will deal with the oracle as with a normal argument of a $\mathcal{POR}^-$ function. So, we won't need to copy such tape in the corresponding **SFP**: differently, we would have had 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 35 (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 $E^-(x,\eta) = \epsilon$*
- *For every $n \in \mathbb{N}$ and for every $1 \leq i \leq n$, the function ${P^-}^n_i$ such that ${P^-}^n_i(x_1,\dots,x_n,\eta) = x_i \quad (1 \leq i \leq n)$*
- *For every $n \in \mathbb{N}$, the function ${H^-}_n$ such that ${H^-}_n(x_1,\dots,x_n,\eta) = \eta$.*
- *For every $b \in \{0,1\}$ the function $C^-_b(x,\eta) = x \cdot b$*
- *The function Q^- defined as*

$$Q^-(x,y,\eta) = \begin{cases} 1 & \textit{if } x \subseteq y \\ 0 & \textit{otherwise} \end{cases}$$

and closed under:

- *Composition. f^- is defined from $g, h_1, \dots, h_k \in \mathcal{POR}^-$ as:*

$$f(x_1,\dots,x_n,\eta) = g\big(h_1(x_1,\dots,x_n,\eta),\dots,h_k(x_1,\dots,x_n,\eta),\eta\big)$$

- *Bounded iteration. f^- is defined from $g, h_0, h_1 \in \mathcal{POR}^-$ with bound t:*

$$\begin{aligned} f(x_1,\dots,x_n,\epsilon,\eta) &= g(x_1,\dots,x_n,\eta) \\ f(x_1,\dots,x_n,y\mathtt{0},\eta) &= h_0\big(x_1,\dots,x_n,y,f(x_1,\dots,x_n,y,\eta)\big)|_{t(x_1,\dots,x_n,y,\eta)} \\ f(x_1,\dots,x_n,y\mathtt{1},\eta) &= h_1\big(x_1,\dots,x_n,y,f(x_1,\dots,x_n,y,\eta)\big)|_{t(x_1,\dots,x_n,y,\eta)} \end{aligned}$$

  For simplicity, we will represent with $ite(g,h_1,h_2,t)$ the function obtained by applying the bounded iteration rule, at the $\mathcal{POR}^-$ functions g ,h_0, h_1 and bound t.

Lemma 8 (Size of $\mathcal{L}_{\mathbb{PW}}$ terms)**.** *The size of a term in $\mathcal{L}_{\mathbb{PW}}$ is polynomial in the size of its variables.*

Proof. By induction on the production of the term.

$\mathtt{0},\mathtt{1}$ If the term is a digit, it has no variables and its size is $\mathtt{1}$, that is a constant all constants are polynomials with no variables, so the claim is proved.

x 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.

$t \frown s$ If the term is the concatenation of two terms $t \frown s$, its size is the sum of the sizes of tits sub terms, which are polynomials by IH. The thesis can be derived by the fact that the sum of polynomials is still a polynomial in the union of the variables of the terms t and s.

$t \times s$ If the term is the product of two terms $t \times s$, we know that 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 a polynomial in the size of the variables in $t \times s$.

□

Lemma 9. $\forall f \in \mathcal{POR}.\forall x_1, \dots, x_n.\forall \omega.\exists p \in POLY.|f(x_1, \dots, x_n, \omega)| \leq p(|x_1|, \dots, |x_n|)$

Proof. By induction on the proof of the fact that $f \in \mathcal{POR}$:

- If f is E, the polynomial that we need to introduce can be the one of rank `0` and `1` as coefficient, i.e. the constant `1`.
- If f is Q, the polynomial that we need to introduce is the constant 1.
- If f is C_0 or C_1, the polynomial that we need to introduce is $|x| + 1$.
- If f is f_q, the polynomial that we need to introduce is the constant 1.
- In the case of projection, the polynomial that we need to introduce is $\sum_{i=0}^{n} |x_i| + 1$, it is easy to see that such value is greater or equal to the size of each input[1].
- In the case of composition, by IH we know that $\forall 1 \leq i \leq k.\exists p_i \in \mathsf{POLY}.|h_i(x_1, \dots, x_n, \omega)| \leq p_i(|x_1|, \dots, |x_n|)$ 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.
- In the case of iteration, by IH we know that the size of g is bounded by a polynomial p_g in its inputs. The we proceed on induction on the string τ that is passed as recursion bound. If such string has length `0`, it is ϵ, so the function f coincides with g and we have shown that has a polynomial bound. Otherwise, the size of the value computed by f is polynomial in its input because it is truncated to the size of a term in $\mathcal{L}_{\mathbb{PW}}$, whose size is polynomial in its variables (that are the inputs of f), by lemma 8.

□

Lemma 10. $\forall f \in \mathcal{POR}^-.\forall x_1, \dots, x_n.\forall \eta.\exists p \in POLY.|f(x_1, \dots, x_n, \eta)| \leq p(|x_1|, \dots, |x_n|, |\eta|)$

Proof. As above, by induction on the proof of the fact that $f \in \mathcal{POR}^-$, but we need to introduce the following case:

- In the case of projection of the oracle H^-, the polynomial that we need to introduce is $|\eta|$, it is easy to see that such value is greater or equal to the size of each input.

□

Now we need to define an order relation between polynomials, and to derive some results on it.

Definition 36 (Pointwise order on polynomials)**.** *Given two polynomials $p : \mathbb{N}^m \to \mathbb{N}$ and $q : \mathbb{N}^m \to \mathbb{N}$ we say that q is* universally greater *than p if and only if $\forall n_1, \dots, n_m \in \mathbb{N}.p(n_1, \dots, n_m) \leq q(n_1, \dots, n_m)$. In this case we write $p \lesssim q$.*

Remark 13 (Partial order of $\lesssim$)**.** *$\lesssim$ is a partial order relation.*

Lemma 11 (Construction of universally greater polynomials)**.** *Given two polynomials $p_1 : \mathbb{N}^m \to \mathbb{N}$ and $p_2 : \mathbb{N}^m \to \mathbb{N}$ there exists a polynomial q so that $p_1 \leq q \wedge p_2 \leq q$*

[1]Actully, we are overkilling the bound: introducing the polynomial $|x_i|$ for each P_i^n would have been sufficent.

Proof. We proceed by induction on the maximum between the order of p_1 and p_2.

- If such value is 0, both the polynomial are constants, so we can choose the greatest of such constants as q.
- If the maximum between the order of p_1 and p_2 is $n+1$ consider the polynomials obtained by p_1 and p_2 removing the all the terms with such order. The resulting polynomial has order lesser than $n+1$, call these polynomials p_1' and p_2'. For IH, we can build a new polynomial q' such that $p_1' \lesssim q \wedge p_2' \lesssim q$. Consider then the polynomials $p_1 - p_1'$ and $p_2 - p_2$. Such polynomials can be expressed respectively as sums of monomials with grade $n+1$, namely $p_1 - p_1' = \sum_i = 1^{m^{n+1}} a_i t_i$ and $p_2 - p_2' = \sum_i = 0^{m^{n+1}} b_i t_i$ some of them can have 0 as coefficient. Now proceed as follows:
$$\forall 0 \leq m^{n+1}.c_i := \begin{cases} a_i & \text{if } a_i \geq b_i \\ b_i & \text{otherwise} \end{cases}$$
The polynomial $q = q' + \sum_{i=0}^{m^{n+1}} c_i t_i$ is such that $p_1 \lesssim q \wedge p_2 \lesssim q_2$.

□

We can conclude that every $\mathcal{POR}$ function queries its oracle function only with strings of polynomial length. Indeed, as we will prove, the length of the sequences described by the $\mathcal{POR}$ functions in polynomially long in their inputs. This allowed us to define the $\mathcal{POR}^-$ class which works with an additional and polynomially long sequence of `0` and `1` bits instead of an oracle function. The definition above is the starting point for the formalization of what we have now stated.

Definition 37 (n-th prefix)**.** *We define with ω_n the n-th prefix of ω. More formally its the sequence:*

$$\omega(\epsilon)\omega(\mathtt{1})\omega(\mathtt{11})\dots\omega(\mathtt{1}^{n-1})$$

Lemma 12. $\forall \omega \in \mathbb{O}.\forall p, q.\forall n_1, \dots, n_m.p \lesssim q \to \omega_{p(n_1,\dots,n_m)} \subseteq \omega_{q(n_1,\dots,n_m)}$.

Proof. By $p \lesssim q$, we know that $p(n_1, \dots, n_m) \leq q(n_1, \dots, n_m)$, so the second prefix is longer than the first, and so the first is a prefix of the second, too. □

The lemma 9 states an important result about the actual usage of the oracle by a $\mathcal{POR}$ function: the only part of an oracle that really affects the result of the computation is its prefix. Moreover, the size of such prefix is polynomially bounded, as stated below:

Lemma 13 (Prefix Lemma)**.** $\forall f \in \mathcal{POR}.\exists p \in \mathsf{POLY}.\forall x_1, \dots, x_k.\forall \omega, \omega'.\omega_{p(|x_1|,\dots,|x_k|)} = \omega'_{p(|x_1|,\dots,|x_k|)} \to f(x_1, \dots, x_k, \omega) = f(x_1, \dots, x_k, \omega')$.

Proof.

- If f is E, C_0, C_1, Q, or a projection, the function doesn't use ω at all, so we can introduce 0.
- If f is f_q, the polynomial that we need to introduce is the linear function in the size of x.
- In the case of composition, by IH we know that $\forall 1 \leq i \leq k.\exists p_i \in \mathsf{POLY}.\omega_{p_i(|x_1|,\dots,|x_n|)} = \omega'_{p_i(|x_1|,\dots,|x_n|)} \to h_i(x_1, \dots, x_n, \omega) = h_i(x_1, \dots, x_n, \omega')$; a similar bound q exists for the external function f, but needs to be composed with the sequence of polynomials obtained from 9. Call such polynomial q'. Now we can apply lemma 11, in order to obtain p. Since $\forall i.p_i \lesssim p \wedge q' \lesssim p$, we have that $\omega_{p(|x_1|,\dots,|x_n|)} = \omega'_{p(|x_1|,\dots,|x_n|)} \to \omega_{p_i(|x_1|,\dots,|x_n|)} = \omega'_{p_i(|x_1|,\dots,|x_n|)}$ by lemma 12, which allows us to use the IH and to conclude the current inductive step.

- In the case of iteration, we can proceed similarly to the case above: by IH we know that the accesses made by g to the oracle are bounded by a polynomial p_g in its input variable. Then, we proceed by inductive hypothesis on h_0 an d h_1 obtaining p_{h_0} and p_{h_1}, we also know that there exists a bound on the size of the function f by lemma 9, that is a polynomial p_f. Similarly to what we did above, we can partially compose the polynomials p_{h_0} and p_{h_1} with p_f obtaining p_0 and p_1. We can build p as described by lemma 11. The polynomial p is greater or equal to each one of the polynomials which bound the accesses to ω, so we can apply lemma 12 and apply the IHs on g, h_0 and h_1. □

The following step requires us to show that each function in $\mathcal{POR}$ can be implemented in $\mathcal{POR}^-$ by means of a function which interprets the queries to the oracle reading from a polynomially bounded prefix of such value. Lemma 13 shows how long should be at least the above mentioned prefix. In order to fully substitute the oracle, we need to perform some other encodings; in particular, we need a $\mathcal{POR}^-$ function that simulates the access to the oracle.

$$\begin{aligned} lft(\epsilon,\eta) &:= \epsilon \\ lft(yb,\eta) &:= y|_y \end{aligned}$$

$$\begin{aligned} nlft(x,\epsilon,\eta) &:= x \\ nlft(x,yb,\eta) &:= lft(nlft(x,y,\eta),\eta)|_y \end{aligned}$$

$$access(x,\eta) := lst(nlft(rv(\eta,\eta),x,\eta),\eta)$$

With a little abuse of notation, we reused the functions lft and rv that we showed being in $\mathcal{POR}$; although we won't show it explicitly, the same construction can be used in order to prove that such functions are in $\mathcal{POR}^-$, too.

Remark 14.

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

Now we can show the main result of this subsection.

Lemma 14 (Implementation of $\mathcal{POR}$ in $\mathcal{POR}^-$)**.**
$\forall f \in \mathcal{POR}.\exists p \in \mathsf{POLY}.\exists g \in \mathcal{POR}^-.\forall q \in \mathsf{POLY}.p \lesssim q.\forall x_1,\dots,x_n,\omega.f(x_1,\dots,x_n,\omega) = g(x_1,\dots,x_n,\omega_{q(x_1,\dots,x_n)})$

Proof. Lemma 13 shows that there is a bound on the size of the portion of the oracle that is actually read by f. We use such bound for p. We proceed by induction on the definition of f.

- If f is E, lemma 13, provides us the bound 0. It is indeed true that $E(x,\omega) = E^-(\omega,\eta)$ where η is any polynomially prefix of ω. We act similarly if f is C_0, C_1, Q, or a projection.
- If f is f_q, its implementation in $\mathcal{POR}^-$ is $access(x,\eta)$. The claim is $f_q(\sigma,\omega) = access(\sigma,\eta)$, where η is any prefix of ω longer than $|\sigma|$.
- In the case of composition, by IH we know that $\forall 1 \leq i \leq k.\exists p_i \in \mathsf{POLY}.\forall q_i \in \mathsf{POLY}.p_i \lesssim q_i \to h_i(x_1,\dots,x_n,\omega) = h'_i(x_1,\dots,x_n,\omega_{q_i(|x_1|,\dots,|x_n|)})$ with $h'_i \in \mathcal{POR}^-$ for each i. Similarly we know that $\exists p_f \in \mathsf{POLY}.\forall q_f \in \mathsf{POLY}.p \lesssim q.f(x_1,\dots,x_k,\omega) = f'(x_1,\dots,x_k,\omega_{q_f(|x_1|,\dots,|x_k|)})$ with $f' \in \mathcal{POR}^-$. We can start introducing from the IHs

the polynomials p_i and p_f. Then we compose p_f with the polynomials that express the size of the arguments of such functions which exist for lemma 9, obtaining p'_f that expresses the actual size of the prefix of ω read by f. Now we can apply lemma 11 in order to obtain a polynomial which is universally greater than the starting ones, call that polynomial p. Finally we have that any polynomial which is universally greater than p is also universally greater than the polynomial p_is and the p_f, so we can apply the IH and obtain the result.

- In the case of iteration, by IH we know that $\exists p_g \in \mathsf{POLY}.\forall q_g \in \mathsf{POLY}.p_q \lesssim q_g \to g(x_1,\ldots,x_k,\omega) = g'(x_1,\ldots,x_k,\omega_{q_g(|x_1|,\ldots,|x_k|)})$ with $g' \in \mathcal{POR}^-$ is bounded by a polynomial p_g in its inputs. Similarly, we know that $\exists p_0, p_1 \in \mathsf{POLY}.\forall q_0, q_1 \in \mathsf{POLY}.p_0 \lesssim q_0 \wedge p_1 \lesssim q_1 \to h_0(x_1,\ldots,x_k,x_{k+1},\omega) = h'_0(x_1,\ldots,x_k,x_{k+1},\omega_{q_0(|x_1|,\ldots,|x_k|,|x_{k+1}|)}) \wedge h_1(x_1,\ldots,x_k,x_{k+1},\omega) = h'_1(x_1,\ldots,x_k,x_{k+1},\omega_{p_1(|x_1|,\ldots,|x_k|,|x_{k+1}|)})$. Similarly to what we did previously, we use lemma 9 in order 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 $x_1,\ldots,x_k$. Call the polynomials that we obtain p'_0 and p'_1. now we can produce an polynomial universally greater that the one that we obtained simply applying 11, obtaining p. This allows us to use all the IHs and to conclude the sub-derivation.

$\square$

At this point, the proof of final result is intuitively concluded. The $\mathcal{POR}^-$ formalism coincides to the Ferreira's $\mathcal{BRS}$ (Ferreira90). The authors of the above mentioned paper claimed such formalism to be polynomially interpretable by a single tape Turing's Machine, which is a particular class of a **SFP** machine. Unfortunately, as far as the authors of this paper know, the proof of such result is unavailable.

For this reason, and for the sake of producing a comprehensive and self-contained work, we decided to prove explicitly that the $\mathcal{POR}^-$ formalism can be interpreted by a single tape Turing's Machine with polynomial complexity.

1.5.2. *The $\mathcal{SIMP}$ formalism.* The **SFP** paradigm, which is a subclass of the Turing Machines' model, is far being a functional paradigm, while $\mathcal{POR}$ is fully functional. For this reason, a direct encoding of the $\mathcal{POR}$ (or even of the $\mathcal{POR}^-$) formalism in **SFP** would be too much complicated because of the radically different natures of the two formalisms.

In order to simplify a little bit the whole encoding, we will pass through an intermediate imperative paradigm, the String's Imperative and Minimal Paradigm $\mathcal{SIMP}$.

The $\mathcal{SIMP}$ paradigm is defined by an enumerable set of correct programs and an operational semantics.

Definition 38 (Correct programs of $\mathcal{SIMP}$)**.** *The language of the $\mathcal{SIMP}$ programs is $\mathcal{L}(\mathsf{Stm})$, i.e. the set of strings produced by the non-terminal symbol* Stm *defined by:*

$$\begin{aligned}
\mathsf{Id} &::= X_i \mid Y_i \mid S_i \mid R \mid Q \mid Z \mid T \qquad i \in \mathbb{N}\\
\mathsf{Exp} &::= \epsilon \mid \mathsf{Exp.0} \mid \mathsf{Exp.1} \mid \mathsf{Id} \mid \mathsf{Exp} \sqsubseteq \mathsf{Exp} \mid \mathsf{Exp} \wedge \mathsf{Exp} \mid \neg \mathsf{Exp}\\
\mathsf{Stm} &::= \mathsf{Id} \leftarrow \mathsf{Exp} \mid \mathsf{Stm}; \mathsf{Stm} \mid \mathbf{while}(\mathsf{Exp})\{\mathsf{Stm}\} \mid \mathbf{skip};
\end{aligned}$$

Definition 39 (Store)**.** *A store is a partial function $\Sigma : \mathsf{Id} \longrightarrow \{\mathsf{0},\mathsf{1}\}^*$.*

Definition 40 (Empty store)**.** *An* empty *store is a store that is undefined on all its domain. We represent such object with* $[]$*.*

Definition 41 (Store updating)**.** *We define the updating of a store Σ with a mapping from $y \in \mathsf{Id}$ to $\tau \in \{\mathsf{0},\mathsf{1}\}^*$ as the store Σ_1 defined as:*

$$\Sigma_1(x) := \begin{cases} \tau & \text{if } x = y \\ \Sigma(x) & \text{otherwise} \end{cases}$$

Definition 42 (semantics of $\mathcal{SIMP}$'s expressions)**.** *The semantics of an expression $E \in \mathcal{L}(\mathsf{Exp})$ is the smallest function $\rightharpoonup$: $\mathcal{L}(\mathsf{Exp}) \times (\mathsf{Id} \longrightarrow \{\mathsf{0},\mathsf{1}\}^*) \longrightarrow \{\mathsf{0},\mathsf{1}\}^*$ closed under the following rules:*

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

$$\frac{\langle e, \Sigma, \rangle \rightharpoonup \sigma \qquad \langle f, \Sigma, \rangle \rightharpoonup \tau \qquad \sigma \subseteq \tau}{\langle e \sqsubseteq f, \Sigma \rangle \rightharpoonup \mathsf{1}} \qquad \frac{\langle e, \Sigma, \rangle \rightharpoonup \sigma \qquad \langle f, \Sigma, \rangle \rightharpoonup \tau \qquad \sigma \nsubseteq \tau}{\langle e \sqsubseteq f, \Sigma \rangle \rightharpoonup \mathsf{0}}$$

$$\frac{\Sigma(Id) = \sigma}{\langle Id, \Sigma \rangle \rightharpoonup \sigma}$$

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

$$\frac{\langle e, \Sigma, \rangle \rightharpoonup \mathsf{1} \qquad \langle f, \Sigma, \rangle \rightharpoonup \mathsf{1}}{\langle e \wedge f, \Sigma \rangle \rightharpoonup \mathsf{1}} \qquad \frac{\langle e, \Sigma, \rangle \rightharpoonup \sigma \qquad \langle f, \Sigma, \rangle \rightharpoonup \tau \qquad \sigma \neq \mathsf{1} \wedge \tau \neq \mathsf{1}}{\langle e \wedge f, \Sigma \rangle \rightharpoonup \mathsf{0}}$$

Definition 43 (Operational semantics of $\mathcal{SIMP}$)**.** *The semantics of a program $P \in \mathcal{L}(\mathsf{Stm})$ is the smallest function $\triangleright : \mathcal{L}(\mathsf{Stm}) \times (\mathsf{Id} \longrightarrow \{\mathsf{0},\mathsf{1}\}^*) \longrightarrow (\mathsf{Id} \longrightarrow \{\mathsf{0},\mathsf{1}\}^*)$ closed under the following rules:*

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

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

Notation 5. *We will use the notation $E \sqsubseteq F$ as a shorthand (syntactic sugar) for $\neg(\neg E.\mathsf{0} \sqsubseteq F \wedge \neg E.\mathsf{1} \sqsubseteq F)$.*

Remark 15. *We will use the pattern $B \leftarrow \epsilon.\mathsf{1}; \mathbf{while}(c \wedge B)\{\mathsf{Stm}; B \leftarrow \epsilon.\mathsf{0}\}$ as an implementation of the if statement: it's indeed true that*

- *The statement* Stm *is executed if and only if c holds.*
- *The statement* Stm *is executed only one time.*

For increasing the readability of our proof, we will introduce the following notation:

Notation 6 (pseudo-procedeure)**.** *A pseudo-procedure is a syntactic sugar for the $\mathcal{SIMP}$'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 parametersand all the* free *names of the body are substituted by fresh names.*

Lemma 15 (Term representation in $\mathcal{SIMP}$)**.** *All the terms of $L_{\mathbb{PW}}$ can be represented in $\mathcal{SIMP}$. Formally: $\forall t \in L_{\mathbb{PW}}. \exists \mathfrak{M} \in \mathcal{L}(\mathsf{Stm}). Vars(t) = \{x_1, \ldots, x_n\} \to \mathfrak{M}(x_1, \ldots, x_n) = t(x_1, \ldots, x_n)$*

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 $\mathfrak{M}$ doesn't write the values it accesses as input.

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

- $\mathfrak{M}(\epsilon) := R \leftarrow \epsilon;$
- $\mathfrak{M}(\mathtt{0}) := R \leftarrow \epsilon.\mathtt{0};$
- $\mathfrak{M}(\mathtt{1}) := R \leftarrow \epsilon.\mathtt{1};$
- $\mathfrak{M}(Id) := R \leftarrow Id;$

For sake of readability, we define the following program that copies the $|Z|$-th bit of S at the end of R, given that Z contains the $|Z|$-th prefix of S.

```
$copyb(Z, S, R) := B \leftarrow \epsilon.\mathtt{1};$
    $\mathbf{while}(Z.0 \sqsubseteq S \wedge B)\{$
        $Z \leftarrow Z.\mathtt{0};$
        $R \leftarrow R.\mathtt{0};$
        $B \leftarrow \epsilon.\mathtt{0};$
    $\}$
    $\mathbf{while}(Z.1 \sqsubseteq S \wedge B)\{$
        $Z \leftarrow Z.\mathtt{1};$
        $R \leftarrow R.\mathtt{1};$
        $B \leftarrow \epsilon.\mathtt{0};$
    $\}$
```

Lemma 16 (Complexity of *copyb*). *The pseudo-procedure copyb requires a number of steps which is a polynomial in the sizes of its arguments.*

Proof. The two **while**( ){ }s are used for implementing an `if` construct as described in Remark 15, so they cause no iteration. Moreover, the two statements are mutually exclusive, so this pseudo-procedure requires at most 5 steps. ☐

Lemma 17 (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's true that $Z.\mathtt{0} \sqsubseteq S \vee Z.\mathtt{1} \sqsubseteq S$. In both case *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 `if`s is executed. The last conclusion comes from the observation that each assignment to Z is followed by a similar assignment to R. ☐

This pseudo-procedure will turn 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**$(Z \sqsubset S)\{$
               $B \leftarrow \epsilon.\mathsf{1};$
               $copyb(Z, S, R)$
               $\}$
```

The encoding of the $\times$ operator is the following:

```
$\mathfrak{M}(t \times s) :=$ $\mathfrak{M}(t)$
            $T \leftarrow R;$
            $\mathfrak{M}(s)$
            $S \leftarrow R;$
            $Z \leftarrow \epsilon;$
            $R \leftarrow \epsilon;$
            $Q \leftarrow \epsilon;$
            **while**$(Z \sqsubset S)\{$
               $B \leftarrow \epsilon.\mathsf{1};$
               **while**$(Z.\mathsf{0} \sqsubseteq S \wedge B)\{$
                  $Z \leftarrow Z.\mathsf{0};$
                  **while**$(Q \sqsubset T)\{$
                     $copyb(Q, T, R)$
                     $\}$
                  $Q \leftarrow \epsilon;$
                  $B \leftarrow \epsilon.\mathsf{0};$
                  $\}$
               **while**$(Z.\mathsf{1} \sqsubseteq S \wedge B)\{$
                  $Z \leftarrow Z.\mathsf{1};$
                  **while**$(Q \sqsubset T)\{$
                     $copyb(Q, T, R)$
                     $\}$
                  $Q \leftarrow \epsilon;$
                  $B \leftarrow \epsilon.\mathsf{0};$
                  $\}$
               $\}$
```

□

Lemma 18 (Complexity of $\mathfrak{M}$)**.** *$\forall t \in L_{\mathbb{PW}}.\mathfrak{M}(t)$ can be computed in number of steps which is polynomial in the size of the variables in t.*

Proof. We proceed by induction on the syntax of t.

- ϵ If the term is ϵ, $\mathfrak{M}(t)$ consists in two steps.
- `0, 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 17 and 16 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 function respects the fact 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.
- $t \times s$ We will distinguish the three levels of **while**( ){ }s by calling them *outer*, *middle* and *inner*. For Lemmas 16 and 17, the inner **while**( ){ }s take at most $|T|$ steps, such value is a polynomial over the free variables of t according to Lemma 8. The value of T is kept constant after its first assignment, for this reason all the inner cycles require a polynomial number of steps. The middle cycles, are an implementation of the `if` construct according to Remark 15, so they are executed only once per each outer cycle. 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 16, the outer cycle takes at most $|S|$ steps which is a polynomial according to 8.

□

Definition 44 (Function described by a $\mathcal{SIMP}$ program)**.** *We say that the value described by a correct $\mathcal{SIMP}$ program P is $\mathcal{F}(P) : \mathcal{L}(Stm) \longrightarrow (\mathbb{S}^n \longrightarrow \mathbb{S})$, where F is defined as above*[2]*:*

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

In order to present in a more compact way the last translation, we need to introduce a pseudo-procedure which truncates a register to the length of another one.

Definition 45 (Truncating pseudo-procedure)**.** *The $trunc(T, R)$ pseudo-procedure is a $\mathcal{SIMP}$ program with free names T and R, defined as follows:*

[2]Instead of the infixed notation for $\triangleright$, we will use its prefixed notation in order to express the store associated to the program and the starting store that are between the curly brackets.

```
$trunc(T, R) :=Q \leftarrow R;$
           $R \leftarrow \epsilon;$
           $Z \leftarrow \epsilon;$
           $Y \leftarrow \epsilon;$
           $\textbf{while}(Z \sqsubset T)\{$
             $B \leftarrow 1;$
             $\textbf{while}(Z.0 \sqsubseteq T \wedge B)\{$
               $copyb(R, Q, Y)$
               $Z \leftarrow Z.0;$
               $B \leftarrow 0;$
               $\}$
             $\textbf{while}(Z.1 \sqsubseteq T \wedge B)\{$
               $B \leftarrow 1;$
               $copyb(R, Q, Y)$
               $Z \leftarrow Z.1;$
               $B \leftarrow 0;$
               $\}$
           $\}$
```

Lemma 19 (Complexity of truncation)**.** *The pseudo-procedure trunc requires a number of steps which is at most polynomial in the sizes of its free names.*

Proof. By Lemma 16 we know that the pseudo-procedure requires a constant number of steps, furthermore, the inner cycles are the implementation of an `if` according to Remark 15, so they are executed only once per outer cycle. 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 20 (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 isn't executed.

σb In this case, only one of the sub-cycles is executed (they are mutually-exclusive), a single more of Q is stored in R according to Lemma 17, 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*.

□

Lemma 21 (Implementation of $\mathcal{POR}^-$ in $\mathcal{SIMP}$)**.** $\forall f \in \mathcal{POR}^-.\exists P \in \mathcal{L}(Stm).$
$\forall x_1, \dots x_n.F(P)(x_1, \dots, x_n) = f(x_1, \dots, x_n)$

Proof. For each function $f \in \mathcal{POR}^-$ we define a program $\mathfrak{L}(f)$ such that $F(\mathfrak{L}(f))(x_1, \dots, x_n) = f(x_1, \dots, x_n)$ by induction on the syntax of f. The correctness of such implementation is given by the following invariant properties:

- The result of the computation is stored in R.
- The inputs and the sub-oracle are stored in the registers of the group X.

- The function $\mathfrak{L}$ doesn't change the values it accesses as input.

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

- $\mathfrak{L}(E^-) \coloneqq R \leftarrow \epsilon; \mathbf{skip};.$
- $\mathfrak{L}(C_0^-) \coloneqq R \leftarrow X_0.0; \mathbf{skip};.$
- $\mathfrak{L}(C_1^-) \coloneqq R \leftarrow X_0.1; \mathbf{skip};.$
- $\mathfrak{L}(P^{-}{}_i^n) \coloneqq R \leftarrow X_i; \mathbf{skip};.$
- $\mathfrak{L}(H_n^-) \coloneqq R \leftarrow X_{n+1}; \mathbf{skip};.$
- $\mathfrak{L}(Q) \coloneqq R \leftarrow X_1 \sqsubseteq X_2; \mathbf{skip};.$

Finally the encoding of the composition and of the bounded iteration:

$$\begin{aligned}
\mathfrak{L}(f(h_1(x_1,\dots,x_n,\eta),\dots h_k(x_1,\dots,x_n,\eta),\eta)) \coloneqq & \mathfrak{L}(h_1) \\
& S_1 \leftarrow R; \\
& \dots \\
& \mathfrak{L}(h_k) \\
& S_k \leftarrow R; \\
& Y_1 \leftarrow X_1; \\
& \dots \\
& Y_{\max(n+1,k+1)} \leftarrow X_{\max(n+1,k+1)}; \\
& X_1 \leftarrow S_1; \\
& \dots \\
& X_k \leftarrow S_k; \\
& X_{k+1} \leftarrow Y_{n+1}; \\
& \mathfrak{L}(f) \\
& X_1 \leftarrow Y_1 \\
& \dots \\
& X_{\max(n+1,k+1)} \leftarrow Y_{\max(n+1,k+1)}; \\
& \mathbf{skip};
\end{aligned}$$

Supposing that g takes n parameters, the bounded iteration is computed as follows:

```
$\mathfrak{L}(ite(g, h_1, h_2, t)) :=Z \leftarrow X_{n+1};$
                        $X_{n+1} \leftarrow \epsilon;$
                        $\mathfrak{L}(g(x_1, \dots, x_n))$
                        $Y \leftarrow X_{n+2};$
                        $\textbf{while}(X_{n+1} \sqsubset Z)\{$
                            $B \leftarrow \epsilon.1;$
                          $\textbf{while}(X_{n+1}.0 \sqsubseteq Z \wedge B)\{$
                            $X_{n+1} \leftarrow X_{n+1}.0;$
                            $\mathfrak{M}(t)$
                            $T \leftarrow R;$
                            $\mathfrak{L}(h_0);$
                            $X_{n+1} \leftarrow R;$
                            $trunc(T, R);$
                            $X_{n+2} \leftarrow R;$
                            $B \leftarrow \epsilon.0;$
                        $\}$  $\textbf{while}(X_{n+1}.1 \sqsubseteq Z \wedge B)\{$
                            $X_{n+1} \leftarrow X_{n+1}.1;$
                            $\mathfrak{M}(t)$
                            $T \leftarrow R;$
                            $\mathfrak{L}(h_0);$
                            $X_{n+1} \leftarrow R;$
                            $trunc(T, R);$
                            $X_{n+2} \leftarrow R;$
                            $B \leftarrow \epsilon.0;$
                       $\}\}X_{n+2} \leftarrow Y;$
                        $\textbf{skip};$
```

□

Proposition 2 (Complexity of $\mathcal{SIMP}$)**.** *$\forall f \in \mathcal{POR}^-.\mathfrak{L}(f)$ takes a number of steps which is polynomial in the size of the arguments of f.*

Proof. We proceed by induction on the proof of the fact that f is indeed in $\mathcal{POR}^-$. All the base cases (E^-, C_0^-,C_1^-, $P^{-n}_{\ i}$, H_n^-,Q) are trivial. The inductive steps follow easily:

- In the case of composition, we know that the thesis holds for all the pseudo-procedures $\mathfrak{L}$. 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 16 for proving that the outer cycle is executed only $|Z|$ times, which is a polynomial in an argument of the encoded function. Moreover, the inner cycles are an implementation of the `if` construct according to Remark 15, so they are executed only once per outer cycle. So the

thesis comes from Lemma 19, from Lemma 18, from the fact that the composition of polynomials is still polynomial and from the inductive hypothesis.

□

Remark 16. *The number of registers used by $\mathfrak{L}(f)$ is finite.*

Proof. Such value can be expressed by the function $\#_r^{\mathfrak{L}}$ described below:

$$\begin{aligned}
\#_r^{\mathfrak{M}}(\epsilon) &\coloneqq 1\\
\#_r^{\mathfrak{M}}(\mathtt{0}) &\coloneqq 1\\
\#_r^{\mathfrak{M}}(\mathtt{1}) &\coloneqq 1\\
\#_r^{\mathfrak{M}}(x) &\coloneqq 2\\
\#_r^{\mathfrak{M}}(t \frown s) &\coloneqq 4 + \#_r^{\mathfrak{M}}(t) + \#_r^{\mathfrak{M}}(s) + 1\\
\#_r^{\mathfrak{M}}(t \times s) &\coloneqq 7 + \#_r^{\mathfrak{M}}(t) + \#_r^{\mathfrak{M}}(s) + 1
\end{aligned}$$

$$\begin{aligned}
\#_r^{\mathfrak{L}}(E^-) &\coloneqq 2\\
\#_r^{\mathfrak{L}}(C_i^-) &\coloneqq 2\\
\#_r^{\mathfrak{L}}(P^{-n}_{\ i}) &\coloneqq 2\\
\#_r^{\mathfrak{L}}(H_n^-) &\coloneqq 2\\
\#_r^{\mathfrak{L}}(Q) &\coloneqq 3\\
\#_r^{\mathfrak{L}}(f(h_1(x_1, \dots, x_n, \eta), \dots h_k(x_1, \dots, x_n, \eta), \eta)) &\coloneqq 2k + \max(k+1, h+1) + \#_r^{\mathfrak{L}}(f)\\
\#_r^{\mathfrak{L}}(ite) &\coloneqq (n+2) + 4 + \#_r^{\mathfrak{M}}(t) + 6 + \#_r^{\mathfrak{L}}(g)
\end{aligned}$$

The inductive cases are correct because as:

$t \frown s$ The value takes account of the inductive calls, the four names used by the function and the register used by *copyb*.

$t \times s$ The value takes account of the inductive calls, the seven names used by the function and the register used by *copyb*.

- The value for the concatenation takes account of the X_i registers, the Y_is, the S_is and of the recursive calls.
- The value for the bounded iteration takes account of the X_i registers, the Y, Z, B, and R register, the registers used by the function $\mathfrak{M}$, the six registers used by the *trunc* pseudo-procedure the S_is and of the recursive calls.

The correctness of the definition can be obtained by induction on the syntax of the function $f \in \mathcal{POR}^-$ that is being translated. The fact that $\#_r^{\mathfrak{L}}$ is finite is trivial by induction on the definition of such function. □

Finally we need to state one of our last results, which now should be intuitive.

Proposition 3. *Each $p \in \mathcal{SIMP}$ program which is polynomial and uses k registers can be simulated with a polynomial complexity on a $k+2$-tape Turing Machine which uses an $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and $*$ as blank character.*

Proof. We won't give a formal proof of such proposition because it would require an extremely complex and almost uninformative construction of the machine, but we will describe its functioning by cases, showing that the overhead is polynomial by induction.

Our machine stores the values of each register in a specific tape, plus two additional tapes for keeping the values of the expressions, e_0 nd e_1, this is why our machine is $k+2$-taped.

Expressions

ϵ This expression can be computed overwriting each 0 or 1 symbol on the e_0 tape with $*$, and finally making the head go back to its starting position. This takes a polynomial number of steps because we know that p is polynomial in the size of its inputs, so the size of each tape is polynomial in those values, too.

Id This expression can be computed by deleting the whole e_0 tape, copying each 0 or 1 of the tape associated to the register Id in the e_0 tape, and finally making the two heads go back to their initial position. This takes a polynomial number of steps because we know that p is polynomial in the size of its inputs, so the size of each tape is polynomial in those values, too.

Exp.0 By induction hypothesis we know that the machine can compute Exp in a polynomial number of steps, and store its value in the e_0 tape, then we need to advance the head to the last bit of such tape, add 0, and make the head go back to its initial position. The cost of this operation is polynomial because the size of any register is, and thanks to the inductive hypothesis.

Exp.1 As above.

$\sqsubseteq$ By induction hypothesis we know that the machine can compute the left and the right Exp in a polynomial number of steps. In this case the machine computes the left Exp, and copies the value from e_0 to e_1 following the same procedure that we described for Id. This requires a polynomial number of steps. Then the machine computes the right expression; according to the inductive hypothesis it requires a polynomial number of steps, too. Now the machine has both the heads at the beginning of e_0 and e_1. Now it proceeds left-to right on both the tapes, until one of the following conditions is met:

(1) A $*$ character is met in e_0.
(2) The value read in e_0 is different from $*$ and is different from the value read on e_1.
(3) A $*$ character is met in e_1 but not in e_0.

Foe each case it behaves as follows:

(1) It overwrites the e_0 tape with $*$, and writes 1.
(2) It overwrites the e_0 tape with $*$, and writes 0.
(3) It overwrites the e_0 tape with $*$, and writes 0.

The overall complexity is polynomial because the overhead is linear in the size of the two starting expression, which are polynomial for induction hypothesis.

$\neg$ By induction hypothesis, we know that the machine can compute the value of Exp in a polynomial number of steps. In this case the machine computes the Exp, then the machine checks whether it is a correct Boolean value (checking its size) and then it negates such value. Now the machine steps back the heads in two a single step. In this case, the complexity is trivially polynomial. If the value isn't a the machine overwrites the tape and writes 0. Also in this case, the overall complexity is polynomial.

$\wedge$ By induction hypothesis we know that the machine can compute the left and the right Exp in a polynomial number of steps. In this case the machine computes the left Exp, and copies the value from e_0 to e_1 following the same procedure that we described for Id. Then it checks whether such value is a Boolean vale; if it is not, the machine deletes the whole tape writes 0, and passes to the next step. This requires a polynomial number of steps. Otherwise, the machine computes the right expression; according to the inductive hypothesis it requires a polynomial number of steps, too. Now the machine checks that this expression is a Boolean value and behaving as above if the condition is not met.

Finally the machine can compute the logical conjunction between the two values and step back the heads in two steps. In this case the complexity is trivially polynomial.

Finally we can describe how our machines implements the statements of $\mathcal{SIMP}$.

Statements

skip; The machine executes the next instruction, if it exists, otherwise it terminates.

; The machine executes the statements in their order. This requires a polynomial complexity for inductive hypothesis and because the sum of polynomials is a polynomial.

$\leftarrow$ The machines computes the values of the expression, than copies in the tape corresponding to the identifier in the same fashion as it copies the value of an identifier in the expression tape. For the same argument that we showed above, these operations require a polynomial number of steps

while The machine behaves in the following way:

(1) It computes the values of the expression.
(2) If such value is `0`, the continues to the next instruction.
(3) If the expression is `1`, the machine evaluates the statement, it goes back to step (1).

These steps have the following complexities:

(1) Polynomial, for induction hypothesis.
(2) Polynomial because p is polynomial and so the size of the register is polynomial, too.
(3) Polynomial, for induction hypothesis.

All the previous steps can be executed at most a polynomial number of times because p's complexity is itself polynomial, so the execution of the while statement requires a polynomial number of steps.

This machine is correct by construction. □

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

Corollary 2. *Each polynomial $\mathcal{SIMP}$ function can be computed by a* **SFP**.

Proof. This result comes from the fact that the **SFP** machines are an extension of the single-tape Turing machines, and that a $\mathcal{SIMP}$ program can be executed on a single-tape Turing machines with a polynomial complexity. □

Proposition 4. *Each $f \in \mathcal{POR}$ can be computed by a* **SFP**.

Proof. By Lemma 14, we know that every function which is in $\mathcal{POR}$ is in $\mathcal{POR}^-$, too. For Lemma 21, we know that every function which is in $\mathcal{POR}$ is in $\mathcal{SIMP}$, too. And finally, by means of Corollary 2, we have the thesis. □

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

Proof. The thesis is a conjunction of Propositions 1 and 4. □