Implicit Complexity over an Arbitrary Structure: Sequential and Parallel Polynomial Time

Olivier Bournez
LORIA
615 rue du Jardin Botanique,BP 101
54602 Villers-lès-Nancy Cedex, Nancy, France
e-mail: `Olivier.Bournez@loria.fr`

Felipe Cucker*
Department of Mathematics
City University of Hong Kong
83 Tat Chee Avenue, Kowloon
HONG KONG
e-mail: `macucker@math.cityu.edu.hk`

Paulin Jacobé de Naurois*
LORIA
615 rue du Jardin Botanique,BP 101
54602 Villers-lès-Nancy Cedex, Nancy, France
e-mail: `Paulin.De-Naurois@loria.fr`

Jean-Yves Marion
LORIA
615 rue du Jardin Botanique,BP 101
54602 Villers-lès-Nancy Cedex, Nancy, France
e-mail: `Jean-Yves.Marion@loria.fr`

Abstract

We provide several machine-independent characterizations of deterministic complexity classes in the model of computation proposed by L. Blum, M. Shub and S. Smale. We provide a characterization of partial recursive functions over any arbitrary structure. We show that polynomial time over an arbitrary structure can be characterized in terms of

*These authors has been supported City University of Hong Kong SRG grant 7001290.

safe recursion. We show that polynomial parallel time over an arbitrary structure can be characterized in terms of safe recursion with substitutions.

1 Introduction

Why are we convinced by the Church Thesis? An answer is that there are many mathematical models, such as partial recursive functions, lambda-calculus, or semi-Thue systems, which are equivalent to the Turing machine, but are also independent from any computational machinery. When computing over arbitrary structures, e.g., over the real numbers, the situation is not so clear. Seeking machine independent characterizations of complexity classes can lend further credence to the importance of the classes and models considered.

We consider here the BSS model of computation over the real numbers introduced by Blum, Shub and Smale in their seminal paper [3]. The model was later on extended to a computational model over any arbitrary logical structure [11, 26]. See the monograph [2] for a general survey about the BSS model.

In this paper we first extend the algebraic definition of partial recursiveness to an arbitrary structure and characterize the resulting class of functions as the BSS-computable functions over this structure. This extends to arbitrary structures a result of [3] proved for the real numbers.

Theorem 1 Over any structure $\mathcal{K} = (\mathbb{K}, op_1, \cdots, op_k, rel_1, \cdots, rel_l, \{c_i\}_{i \in I}, \mathbf{0}, \mathbf{1})$, a function is BSS computable if and only if it can be defined as a partial recursive function over $\mathcal{K}$.

In the last decades interest shifted from computability to complexity issues. Several complexity classes, remarkably P and NP became the object of study. These classes were first defined over strings of bits, i.e. over structure $\mathcal{K} = (\{0, 1\}, =, \mathbf{0}, \mathbf{1})$. Then, in [3] its definition was extended to arbitrary rings and in [26] to arbitrary structures.

In classical complexity theory, several attempts have been done to provide formalisms characterizing complexity classes in a machine independent way. Such attempts include characterizations based on lambda calculus (e.g. [21]), on finite model theory (e.g. [10]), on function algebra (e.g. [5]), or yet one combining the latter two approaches (e.g. [12, 28]). See [4, 9, 15] for more complete references.

Yet another direction is in the recent works of Bellantoni and Cook [1] and of Leivant [20] which suggest a new approach by means of data tiering known as implicit computational complexity. Purely syntactic models of complexity classes are provided which can be applied to analyze program complexity in the study of programming languages [14, 17, 23].

In this paper, following these lines, we establish two "implicit" characterizations of complexity classes. Our characterizations work over arbitrary structures, and subsume previous ones when restricted to booleans or integers.

First, we characterize polynomial time computable BSS functions. This result stems on the safe primitive recursion principle of Bellantoni and Cook [1].

Theorem 2 Over any structure $\mathcal{K} = (\mathbb{K}, op_1, \cdots, op_k, rel_1, \cdots, rel_l, \{c_i\}_{i \in I}, \mathbf{0}, \mathbf{1})$, a function is computed in polynomial time by a BSS machine if and only if it can be defined as a safe recursive function over $\mathcal{K}$.

Second, we capture parallel polynomial time BSS functions based on Leivant and Marion characterization of polynomial space computable functions [22].

Theorem 3 Over any structure $\mathcal{K} = (\mathbb{K}, op_1, \cdots, op_k, rel_1, \cdots, rel_l, \{c_i\}_{i \in I}, \mathbf{0}, \mathbf{1})$, a function is computed in parallel polynomial time by a BSS machine if and only if it can be defined as a safe recursive function with substitutions over $\mathcal{K}$.

Observe that, unlike Leivant and Marion, Theorem 3 characterizes parallel polynomial time and not polynomial space: for classical complexity both classes correspond. However over arbitrary structures, this is not true, since the notion of working space may be meaningless: as pointed out by Michaux [25], on some structures like $(\mathbb{R}, +, -, *, \leq, \mathbf{0}, \mathbf{1})$, any computable function can be computed in constant working space (but in this case we have an exponential increase in the running time; note that there exist structures where there is no running time blowup [26]).

From a programming perspective, a way of understanding all these results is to see computability over arbitrary structures like a programming language with extra data types and operators which come from some external libraries. This observation, and its potential to build methods to automatically derive computational properties of programs, in the lines of [14, 17, 23], is a main motivation on our work.

On the other hand, we believe the BSS computational model provides new insights for understanding complexity theory when dealing with structures over other domains [2]. Several results have been obtained for this model in the last decade, including separation of complexity classes over specific structures, see for example [7, 8, 24].

It is worth mentioning that it is not the first time that the implicit computational complexity community is interested by computations over real numbers. Previous work include the paper of Cook [6] on higher order functionals as well as [16]. A related approach over finite structures can be found in [13]. However this is the first time that implicit characterizations of this type over arbitrary structures are given.

In Section 2, we recall the notion of BSS machine and some induced complexity classes over an arbitrary structure $\mathcal{K}$, such as $\mathrm{P}_{\mathcal{K}}$. In Section 3 we recall the notion of algebraic circuit over $\mathcal{K}$ and we use this notion to define complexity classes for parallel time. In Section 4, we define partial recursive and primitive recursive functions over $\mathcal{K}$ and in Section 5 we do so for safe recursive functions. We prove Theorems 1 and 2 in Section 6. Finally, we prove Theorem 3 in Section 7.

2 Computing over an arbitrary structure

In this section, we briefly introduce computability and complexity over an arbitrary structure. Detailed accounts can be found in [2] —for structures like real and complex numbers— or [26] —for considerations about more general structures.

Definition 4 A *structure* $\mathcal{K} = (\mathbb{K}, op_1, \cdots, op_k, rel_1, \cdots, rel_l, \{c_i\}_{i \in I}, \mathbf{0}, \mathbf{1})$ is given by some underlying set $\mathbb{K}$, a finite number of operators $op_1, \cdots, op_k$ of arity greater than 1, some constants $\{c_i\}_{i\in I}$, and a finite number of relations $rel_1, \dots, rel_l$. Constants correspond to operators of arity 0. While the index set I may be infinite, the number of operators with arity greater than 1 needs to be finite, that is, only symbols for constants may be infinitely many. We will not distinguish between operator and relation symbols and their corresponding interpretations as functions and relations respectively over the underlying set $\mathbb{K}$. We assume that the equality relation $=$ is a relation of the structure, and that there are at least two constant symbols, with different interpretations (denoted by $\mathbf{0}$ and $\mathbf{1}$ in our work) in the structure.

An example of structure is $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, \{c \in \mathbb{R}\})$. Another example, corresponding to classical complexity and computability theory is $\mathcal{K} = (\{0,1\}, =, \mathbf{0}, \mathbf{1})$.

Remark 5 For any structure $\mathcal{K}$ as above, $(\{0,1\}, =, \mathbf{0}, \mathbf{1}) \subseteq \mathcal{K}$.

We denote by $\mathbb{K}^* = \bigcup_{i\in\mathbb{N}} \mathbb{K}^i$ the set of words over the alphabet $\mathbb{K}$. The space $\mathbb{K}^*$ is the analogue to Σ^* the set of all finite sequences of zeros and ones. It provides the inputs for machines over $\mathcal{K}$. For technical reasons we shall also consider the bi-infinite direct sum $\mathbb{K}_*$. Elements of this space have the form

$$(\dots, x_{-2}, x_{-1}, x_0, x_1, x_2, \dots)$$

where $x_i \in \mathbb{K}$ for all $i \in \mathbb{Z}$ and $x_k = 0$ for k sufficiently large in absolute value. The space $\mathbb{K}_*$ has natural shift operations, shift left $\sigma_\ell : \mathbb{K}_* \to \mathbb{K}_*$ and shift right $\sigma_r : \mathbb{K}_* \to \mathbb{K}_*$ where

$$\sigma_\ell(x)_i = x_{i-1} \qquad \text{and} \qquad \sigma_r(x)_i = x_{i+1}.$$

In what follows, words of elements in $\mathbb{K}$ will be represented with overlined letters, while elements in $\mathbb{K}$ will be represented by letters. For instance, $a.\overline{x}$ stands for the word in $\mathbb{K}^*$ whose first letter is a and which ends with the word $\overline{x}$. We denote by ϵ the empty word. The length of a word $\overline{w} \in \mathbb{K}^*$ is denoted by $|\overline{w}|$.

We now define machines over $\mathcal{K}$ following the lines of [2].

Definition 6 A *machine over* $\mathcal{K}$ consists of an input space $\mathcal{I} = \mathbb{K}^*$, an output space $\mathcal{O} = \mathbb{K}^*$, and a register space[1] $\mathcal{S} = \mathbb{K}_*$, together with a connected directed graph whose nodes labelled $0, \dots, N$ correspond to the set of different *instructions* of the machine. These nodes are of one of the five following types: input, output, computation, branching and shift nodes. Let us describe them a bit more.

1. *Input nodes.* There is only one input node and is labelled with 0. Associated with this node there is a next node $\beta(0)$, and the input map $g_I : \mathcal{I} \to \mathcal{S}$.

2. *Output nodes.* There is only one output node which is labelled with 1. It has no next nodes, once it is reached the computation halts, and the output map $g_O : \mathcal{S} \to \mathcal{O}$ places the result of the computation in the output space.

[1]In the original paper by Blum, Shub and Smale, this is called the *state* space. We rename it *register* space to avoid confusions with the notion of 'state' in a Turing machine.

3. *Computation nodes.* Associated with a node m of this type there are a next node $\beta(m)$ and a map $g_m : \mathcal{S} \to \mathcal{S}$. The function g_m replaces the component indexed by 1 of $\mathcal{S}$ by the value $op(w_1, \ldots, w_n)$ where $w_1, w_2, \ldots, w_n$ are components 1 to n of $\mathcal{S}$ and op is some operation of the structure $\mathcal{K}$ of arity n. The other components of $\mathcal{S}$ are left unchanged. When the arity n is zero, m is a constant node. A given machine uses only a finite number of constants, however, in order to compare different machines and to denote the notion of reduction between them and completeness, we need to include all possible constants in the underlying structure $\mathcal{K}$. Thus the possibly infinite index set I.

4. *Branch nodes.* There are two nodes associated with a node m of this type: $\beta^+(m)$ and $\beta^-(m)$. The next node is $\beta^+(m)$ if $rel(w_1, \ldots, w_n)$ is true and $\beta^-(m)$ otherwise. Here $w_1, w_2, \ldots, w_n$ are components 1 to n of $\mathcal{S}$ and rel is some relation of the structure $\mathcal{K}$ of arity n.

5. *Shift nodes.* Associated with a node m of this type there is a next node $\beta(m)$ and a map $\sigma : \mathcal{S} \to \mathcal{S}$. The σ is either a left or a right shift.

Several conventions for the contents of the register space at the beginning of the computation have been used in the literature [2, 3, 26]. We will not dwell on these details but focus on the essential ideas in the proofs to come in the sequel.

Remark 7 A machine over $\mathcal{K}$ is essentially a Turing Machine, which is able to perform the basic operations $\{op_i\}$ and the basic tests $rel_1, \ldots, rel_l$ at unit cost, and whose tape cells can hold arbitrary elements of the underlying set $\mathbb{K}$ [26, 2]. Note that the register space $\mathcal{S}$ above has the function of the tape and that its component with index 1 plays the role of the scanned cell. In what follows we will freely use the common expressions "tape", "scanning head", etc., the translation between these concepts and a shifting register space with a designated 1st position being obvious.

Definition 8 For a given machine M, the function φ_M associating its output to a given input $x \in \mathbb{K}^*$ is called the *input-output function.* We shall say that a function $f : \mathbb{K}^* \to \mathbb{K}^*$ is *computable* when there is a machine M such that $f = \varphi_M$.

Also, a set $A \subseteq \mathbb{K}^*$ is *decided* by a machine M if its characteristic function $\chi_A : \mathbb{K}^* \to \{\mathbf{0}, \mathbf{1}\}$ coincides with φ_M.

A *configuration* of a machine M over $\mathcal{K}$ is given by an instruction q of M along with the position of the head of the machine and two words $w_l, w_r \in \mathbb{K}^*$ that give the contents of the tape at left and right of the head.

We can now define some central complexity classes.

Definition 9 A set $S \subset \mathbb{K}^*$ is in class $\mathrm{P}_{\mathcal{K}}$ (respectively a function $f : \mathbb{K}^* \to \mathbb{K}^*$ is in class FP_K), if there exist a polynomial p and a machine M, so that for all $\overline{w} \in \mathbb{K}^*$, M stops in time $p(|\overline{w}|)$ and M accepts iff $\overline{w} \in S$ (respectively, M computes function $f(\overline{w})$).

This notion of computability corresponds to the classical one for structures over the booleans or the integers, and corresponds to the one of Blum Shub and Smale in [3] over the real numbers.

Proposition 10 **(i)** The class $\mathrm{P}_{\mathcal{K}}$ is the classical P when $\mathcal{K} = (\{0, 1\}, =, \mathbf{0}, \mathbf{1})$.

(ii) The class $\mathrm{P}_{\mathcal{K}}$ is the class $\mathrm{P}_{\mathbb{R}}$ of [3] when $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, \{c \in \mathbb{R}\})$.

3 Computing with circuits

In this section we introduce the notion of circuit over $\mathcal{K}$, and recall some links of this computational device with the BSS model of computation.

3.1 Circuits

Definition 11 A circuit over the structure $\mathcal{K}$ is an acyclic directed graph whose nodes, called *gates*, are labeled either as *input* gates of in-degree 0, *output* gates of out-degree 0, *selection* gates of in-degree 3, or by a relation or an operation of the structure, of in-degree equal to its arity.

The evaluation of a circuit on a given assignment of values of $\mathbb{K}$ to its input gates is defined in a straightforward way, all gates behaving as one would expect. We just note that any selection gate tests whether its first parent is labeled with $\mathbf{1}$, and returns the label of its second parent if this is true or the label of its third parent if not. This evaluation defines a function from $\mathbb{K}^n$ to $\mathbb{K}^m$ where n is the number of input gates and m that of output gates. See [26, 2] for formal details.

We say that a family $\{\mathscr{C}_n \mid n \in \mathbb{N}\}$ of circuits computes a function $f : \mathbb{K}^* \to \mathbb{K}^*$ when the function computed by the nth circuit of the family is the restriction of f to $\mathbb{K}^n$. We say that this family is *P-uniform* when there exist constants $\alpha_1, \dots, \alpha_m \in \mathbb{K}$ and a deterministic Turing machine M satisfying the following. For every $n \in \mathbb{N}$, the constant gates of $\mathscr{C}_n$ have associated constants in the set $\{\alpha_1, \dots, \alpha_m\}$ and M computes a description of the ith gate of the nth circuit in time polynomial in n (if the ith gate is a constant gate with associated constant α_k then M returns k instead of α_k).

Remark 12 It is usually assumed that gates are numbered consecutively with the first gates being the input gates and the last ones being the output gates. In addition, if gate i has parents $j_1, \dots, j_r$ then one must have $j_1, \dots, j_r < i$. Unless otherwise stated we will assume this enumeration applies.

Computations by BSS machines can be done by uniform families of circuits. In [26] the following result is proved.

Proposition 13 Assume M is a BSS machine over $\mathcal{K}$ computing a function f_M. Denote by $\alpha_1, \dots, \alpha_m \in \mathbb{K}$ the constants used by M. Assume moreover that, for all inputs of size n, the computation time of M is bounded by $t(n) \geq n$, and that the length of an output depends only on the size of its input. Then, there exists a family $\{\mathscr{C}_n \mid n \in \mathbb{N}\}$ of circuits such that $\mathscr{C}_n$ has $n + m$ inputs $(x_1, \dots, x_n, y_1, \dots, y_m)$, has size polynomial in $t(n)$, and, for all $\overline{x} = x_1 \dots x_n$, $\mathscr{C}_n(x_1, \dots, x_n, \alpha_1, \dots, \alpha_m)$ equals the output of f_M on input $\overline{x}$. Moreover, there exists a deterministic Turing machine computing a description of the ith gate of the nth circuit in time polynomial in $t(n)$.

The requirements of a homogeneous computation time bound and output size are not too strong: clocking an arbitrary BSS machine, and adding extra iddle characters to its output allows one to build a BSS machine which complies with these requirements.

3.2 A Parallel Model of Computation

The reader can find in [2] the definition of parallel machine over a structure $\mathcal{K}$. We will not give formal definitions here, since we will actually use the alternative characterization given by Proposition 15 below.[2]

Definition 14 $\mathrm{FPAR}_{\mathcal{K}}$ is the class of functions f computable in polynomial time by a parallel machine using an exponentially bounded number of processors and such that $|f(\overline{x})| = |\overline{x}|^{\mathcal{O}(1)}$ for all $\overline{x} \in \mathbb{K}^*$.

Proposition 15 ([2]) A function $f : \mathbb{K}^* \to \mathbb{K}^*$ is in $\mathrm{FPAR}_{\mathcal{K}}$ if and only if $f(\overline{x})$ is computed by a P-uniform family of circuits $\{\mathscr{C}_n(x_1, \dots, x_n) |\ n \in \mathbb{N}\}$ of polynomial depth.

4 Partial Recursive and Primitive Recursive Functions

4.1 Definitions

As in the classical setting, computable functions over an arbitrary structure $\mathcal{K}$ can be characterized algebraically, in terms of the smallest set of functions containing some initial functions and closed by composition, primitive recursion and minimization. In the rest of this section we present such a characterization.

We consider functions $(\mathbb{K}^*)^n \to \mathbb{K}^*$, taking as inputs arrays of words of elements in $\mathbb{K}$, and returning as output a word of elements in $\mathbb{K}$. When the output of a function is undefined, we use the symbol $\perp$.

Definition 16 We call *basic functions* the following four kinds of functions:

(i) Functions making elementary manipulations of words over $\mathbb{K}$. For any $a \in \mathbb{K}, \overline{x}, \overline{x_1}, \overline{x_2} \in \mathbb{K}^*$

$$\begin{array}{rcl} \mathsf{hd}(a.\overline{x}) & = & a \\ \mathsf{hd}(\epsilon) & = & \epsilon \end{array} \qquad \begin{array}{rcl} \mathsf{tl}(a.\overline{x}) & = & \overline{x} \\ \mathsf{tl}(\epsilon) & = & \epsilon \end{array} \qquad \begin{array}{rcl} \mathsf{cons}(a.\overline{x_1}, \overline{x_2}) & = & a.\overline{x_2} \\ \mathsf{cons}(\epsilon, \overline{x_2}) & = & \overline{x_2}. \end{array}$$

(ii) Projections. For any $n \in \mathbb{N}$, $i \leq n$

$$\mathsf{Pr}_i^n(\overline{x_1}, \dots, \overline{x_i}, \dots, \overline{x_n}) = \overline{x_i}.$$

(iii) Functions of structure. For any operator (including the constants treated as operators of arity 0) op_i or relation rel_i of arity n_i we have the following initial functions:

$$\begin{array}{rcl} \mathsf{Op}_i(a_1.\overline{x_1}, \dots, a_{n_i}.\overline{x_{n_i}}) & = & (op_i(a_1, \dots, a_{n_i})).\overline{x_{n_i}} \\ \mathsf{Rel}_i(a_1.\overline{x_1}, \dots, a_{n_i}.\overline{x_{n_i}}) & = & \left\{ \begin{array}{l} \mathbf{1} \text{ if } rel_i(a_1, \dots, a_{n_i}) \\ \epsilon \text{ otherwise.} \end{array} \right. \end{array}$$

[2]The exposition on parallelism in [2, Chapter 18] is for $\mathcal{K}$ the real numbers but the definition of parallel machine as well as the proof of Proposition 15 carry on to arbitrary structures.

(iv) Selection function

$$\mathsf{Selection}(\overline{x},\overline{y},\overline{z}) = \begin{cases} \overline{y} & \text{if } \mathsf{hd}(\overline{x}) = \mathbf{1} \\ \overline{z} & \text{otherwise.} \end{cases}$$

The set of *partial recursive functions* over $\mathcal{K}$ is the smallest set of functions $f : (\mathbb{K}^*)^k \to \mathbb{K}^*$ containing the basic functions and closed under the following operations:

(1) *Composition.* Assume $g : (\mathbb{K}^*)^n \to \mathbb{K}^*$, $h_1, \ldots, h_n : \mathbb{K}^* \to \mathbb{K}^*$ are given partial functions. Then the composition $f : \mathbb{K}^* \to \mathbb{K}^*$ is defined by

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_n(\overline{x})).$$

(2) *Primitive recursion.* Assume $h : \mathbb{K}^* \to \mathbb{K}^*$ and $g : (\mathbb{K}^*)^3 \to \mathbb{K}^*$ are given partial functions. Then we define $f : (\mathbb{K}^*)^2 \to \mathbb{K}^*$

$$\begin{aligned} f(\epsilon, \overline{x}) &= h(\overline{x}) \\ f(a.\overline{y}, \overline{x}) &= \begin{cases} g(\overline{y}, f(\overline{y},\overline{x}), \overline{x}) & \text{if } f(\overline{y},\overline{x}) \neq \perp \\ \perp & \text{otherwise.} \end{cases} \end{aligned}$$

(3) *Minimization.* Assume $g : (\mathbb{K}^*)^2 \to \mathbb{K}^*$ is given. Function $f : \mathbb{K}^* \to \mathbb{K}^*$ is defined by minimization on the first argument of g, written by $f(\overline{y}) = \mu\overline{x}\,(g(\overline{x},\overline{y}))$, if:

$$\mu\overline{x}\,(g(\overline{x},\overline{y})) = \begin{cases} \perp & \text{if } \forall t \in \mathbb{N} \,:\, \mathsf{hd}(g(0^t,\overline{y})) \neq \mathbf{1} \\ \mathbf{1}^k \,:\, k = \min\{t \mid \mathsf{hd}(g(0^t,\overline{y})) = \mathbf{1}\} & \text{otherwise.} \end{cases}$$

Partial recursive functions defined without using the minimization operator are called *primitive recursive.*

Remark 17 **(i)** The formal definition of function tl is actually a primitive recursive definition with no recurrence argument. However, when we introduce the notion of safe recursion in Section 5, this function tl needs to be given as *a priori* functions in order to be applied to safe arguments, and not only to normal arguments. For the sake of coherence, we give it here as an *a priori* function as well.

(ii) Note that primitive recursive functions are total functions whereas partial recursive functions may be partial functions.

(iii) The operation of minimization on the first argument of g returns the smallest word in $\{\mathbf{1}\}^*$ satisfying a given property. The reason why it does not return a smallest word made of *any* letter in $\mathbb{K}$ is to ensure determinism, and therefore computability. On a structure where NP is not decidable, such a non-deterministic minimization may not be computable by a BSS machine, which is in essence deterministic.

(iv) In the definition of composition, primitive recursion, and minimization above we have taken arguments $\overline{x}, \overline{y} \in \mathbb{K}^*$. This is to simplify notations. To be fully formal, we should allow for arguments in $(\mathbb{K}^*)^p$ with $p \geq 1$. We will adopt these simplification all throughout this paper since the proofs for the fully formal case would not be different: just notationally more involved.

(v) In the definition of primitive recursion, the variable a in front of the recurrence argument $a.\overline{y}$ does not appear as argument of the function g. The first reason for this is the need of consistency among argument types: a is a single element in $\mathbb{K}$ whereas all arguments need to be words in $\mathbb{K}^*$. The second reason is that g may still depend on the value of the first element of $\overline{y}$.

(vi) Our definition of primitive recursion and of minimization is slightly different from the one found in [3]. In this paper, the authors introduce a special integer argument for every function, which is used to control recursion and minimization, and consider the other arguments as simple elements in $\mathbb{K}$. Their functions are of type $f : \mathbb{N} \times \mathbb{K}^k \to \mathbb{K}^l$. Therefore, they only capture finite dimensional functions. It is known that over the real numbers with $+, -, *$ operators finite dimensional functions are equivalent to non-finite dimensional functions (see [25]). But this is not true over other structures, for instance $\mathbb{Z}/2\mathbb{Z}$. Our choice is to consider arguments as words of elements in $\mathbb{K}$, and to use the length of the arguments to control recursion and minimization. This allows us to capture non-finite dimensional functions over arbitrary structures.

The following result is immediate.

Proposition 18 The set of partial recursive (resp. primitive recursive) functions over $\{\{0,1\}, =, \mathbf{0}, \mathbf{1}\}$ coincides with the classical partial recursive (resp. primitive recursive) functions.

5 Safe Recursive Functions

In this section we extend to an arbitrary structure $\mathcal{K}$ the notion of safe recursive function over the natural numbers defined by Bellantoni and Cook [1].

Example 19 Consider the following function

$$\exp(\overline{x}) = \mathbf{1}^{2^{|\overline{x}|}}$$

which computes in unary the exponential. It can be easily defined with primitive recursion by

$$\begin{array}{rcl}
\mathsf{Cons}(\epsilon, \overline{y}) & = & \overline{y} \\
\mathsf{Cons}(a.\overline{x}, \overline{y}) & = & \mathsf{cons}(a, \mathsf{Cons}(\overline{x}, \overline{y})) \\
\exp(\epsilon) & = & \mathbf{1} \\
\exp(a.\overline{x}) & = & \mathsf{Cons}(\exp(\overline{x}), \exp(\overline{x})).
\end{array}$$

On the other hand, note that $\exp \notin \mathrm{FP}_{\mathcal{K}}$ since the computed value is exponentially large in the size of its argument. The goal of this section is to introduce a restricted version of recursion not allowing for this exponential growth.

Safe recursive functions are defined in a similar manner as primitive recursive functions. However, following [1], safe recursive functions have two different types of arguments, each of them having different properties and purposes. The first type of argument, called *normal*, is similar to the arguments of the previously defined partial recursive and primitive recursive functions, since it can be used to make basic computation steps or to control recursion. The second type of argument, called *safe*, can not be used to control recursion. In a recursion, the recurrence argument can only be in safe position. We will see that this distinction between safe and normal arguments ensures that safe recursive functions can be computed in polynomial time.

To emphasize the distinction between normal and safe variables we will write $f : N \times S \to R$ where N indicates the domain of the normal arguments and S that of the safe arguments. If all the arguments of f are of one kind, say safe, we will write $\emptyset$ in the place of N. Also, if $\overline{x}$ and $\overline{y}$ are these arguments, we will write $f(\overline{x}; \overline{y})$ separating them by a semicolon ";". Normal arguments are placed at the left of the semicolon and safe arguments at its right.

We define now safe recursive functions. To do so, we consider the set of *basic safe functions* which are the basic functions of Definition 16 with the feature that their arguments are all safe.

Definition 20 The set of *safe recursive functions* over $\mathcal{K}$ is the smallest set of functions $f : (\mathbb{K}^*)^p \times (\mathbb{K}^*)^q \to \mathbb{K}^*$ containing the basic safe functions, and closed under the following operations:

(1) *Safe composition.* Assume $g : (\mathbb{K}^*)^m \times (\mathbb{K}^*)^n \to \mathbb{K}^*$, $h_1, \ldots, h_m : \mathbb{K}^* \times \emptyset \to \mathbb{K}^*$ and $h_{m+1}, \ldots, h_{m+n} : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ are given safe recursive functions. Then their safe composition is the function $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ defined by

$$f(\overline{x}; \overline{y}) = g\left(h_1(\overline{x};), \ldots, h_m(\overline{x};); h_{m+1}(\overline{x}; \overline{y}), \ldots, h_{m+n}(\overline{x}; \overline{y})\right).$$

(2) *Safe recursion.* Assume $h : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ and $g : (\mathbb{K}^*)^2 \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ are given functions. Function $f : (\mathbb{K}^*)^2 \times \mathbb{K}^* \to \mathbb{K}^*$ can then be defined by safe recursion:

$$\begin{aligned} f(\epsilon, \overline{x}; \overline{y}) &= h(\overline{x}; \overline{y}) \\ f(a.\overline{z}, \overline{x}; \overline{y}) &= g(\overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y}) \end{aligned}$$

Remark 21 **(i)** Using safe composition it is possible to "move" an argument from a normal position to a safe position, whereas the reverse is forbidden. For example, assume $g : \mathbb{K}^* \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ is a given function. One can then define with safe composition a function f given by $f(\overline{x}, \overline{y}; \overline{z}) = g(\overline{x}; \overline{y}, \overline{z})$ but a definition like $f(\overline{x}; \overline{y}, \overline{z}) = g(\overline{x}, \overline{y}; \overline{z})$ is not valid.

(ii) Note that it is impossible to turn the definition of exp in Example 19 into a safe recursion scheme. $\mathsf{Cons}(x; y)$ and $\mathsf{Cons}(x, y;)$ can be defined as follows:

$$\begin{aligned} \mathsf{Cons}(\epsilon; \overline{y}) &= \overline{y} \\ \mathsf{Cons}(a.\overline{x}; \overline{y}) &= \mathsf{cons}(; a, \mathsf{Cons}(\overline{x}; \overline{y})) \\ \mathsf{Cons}(\epsilon, \overline{y};) &= \overline{y} \\ \mathsf{Cons}(a.\overline{x}, \overline{y};) &= \mathsf{cons}(; a, \mathsf{Cons}(\overline{x}, \overline{y};)) \end{aligned}$$

However, exp can not be defined, since this would need the use of the recurrence argument $\exp(\overline{x})$ as a normal argument of function Cons, which is forbidden. This obstruction is at the basis of the equivalence between safe recursion and polynomial time computability.

The following result is immediate.

Proposition 22 The set of safe recursive functions over $\{\{0,1\},=,\mathbf{0},\mathbf{1}\}$ coincides with the classical set of safe recursive functions defined by Bellantoni and Cook.

6 BSS Computability and Recursion

This section is devoted to prove Theorems 1 and 2. To do so, we use Proposition 13 to reduce BSS machine computations to circuit evaluation. Recall, the output of this reduction, on an input of size n is a description of a circuit performing the same computation the machine does when restricted to inputs of size n. Since in this proposition we replaced the machine constants by variables, the resulting circuit can be encoded over $\{\mathbf{0},\mathbf{1}\}$ and, as it turns out, the whole reduction can be carried out by a Turing machine. Therefore we can invoke classical results to show that the reduction can be simulated by classical partial recursive or safe recursive functions and, a fortiori, by such kind of functions over an arbitrary structure $\mathcal{K}$. Next, we give a simulation of the circuit given by this reduction by partial recursive or safe recursive functions over an arbitrary structure $\mathcal{K}$. In contrast, when dealing with an arbitrary BSS machine computation, we do not necessarily obtain a P-uniform family of circuits. This is not an issue since the reduction does not need to be polynomial time computable. The reduction as well as the simulation of the family of circuits is done by partial recursive functions. When dealing with a polynomial time BSS machine, the reduction is polynomial in the classical meaning, and we obtain a P-uniform family of circuits. The reduction and the simulation of the family of circuits is done by safe recursive functions. The two cases of simulation of the family of circuits are dealt with in a single result, Lemma 23 below.

Combining these results we obtain proofs for our two theorems.

6.1 Evaluation of Uniform Circuit Families with Recursion

In this section, we make the simulation of a uniform family of circuits by recursive functions explicit.

Lemma 23 Assume $\{\mathscr{C}_n \mid n \in \mathbb{N}\}$ is a family of circuits over $\mathcal{K}$ such that:

- $\mathscr{C}_n$ has $n+m$ input gates $(x_1,\dots,x_n,y_1,\dots,y_m)$,
- there exists a function $t:\mathbb{N}\to\mathbb{N}$ such that, for all $n\in\mathbb{N}$, $|\mathscr{C}_n|\le t(n)$ and $t(n)\ge n$
- there exists a safe recursive (resp. partial recursive) function T such that $T(\overline{x};)=\mathbf{1}^{t(|\overline{x}|)}$

- there exist safe recursive (resp. partial recursive) functions $\mathsf{Gate}, F_1, \ldots F_r$ (here r is the maximum arity of the symbols of $\mathcal{K}$) such that $\mathsf{Gate}(T(\overline{x};), \mathbf{1}^i, \overline{x};)$ describes the i^{th} node of $\mathscr{C}_{|\overline{x}|}$ as follows

$$\mathsf{Gate}(T(\overline{x};), \mathbf{1}^i, \overline{x};) = \begin{cases} \mathbf{0} & \text{if } i \text{ is an input gate} \\ \mathbf{0.0} & \text{if } i \text{ is an output gate} \\ \mathbf{1}^j & \text{if } i \text{ is a gate labeled with } \mathrm{op}_j \\ \mathbf{1}^{k+j} & \text{if } i \text{ is a gate labeled with } \mathrm{rel}_j \\ \mathbf{1}^{k+l+1} & \text{if } i \text{ is a } \textit{selection} \text{ node} \\ \epsilon & \text{otherwise} \end{cases}$$

  and $F_j(T(\overline{x};), \mathbf{1}^i, \overline{x};)$ identifies the j^{th} parent node of the i^{th} node of $\mathscr{C}_{|\overline{x}|}$ as follows

$$F_j(T(\overline{x};), \mathbf{1}^i, \overline{x};) = \mathbf{1}^k \text{ where } k \leq i \text{ is the } j^{th} \text{ parent node of } i.$$

Then, given a set of constants $\overline{\alpha} = \alpha_1, \ldots, \alpha_m \in \mathbb{K}^m$, there exists a safe recursive (resp. partial recursive) function $C^*_{\overline{\alpha}}$ over $\mathcal{K}$ such that, for any $\overline{x} = x_1. \ldots .x_n$, $C^*_{\overline{\alpha}}(\overline{x};)$ equals $\mathscr{C}_n(x_1, \ldots, x_n, \alpha_1, \ldots, \alpha_m)$.

Proof It is easy to define a safe recursive function Pick such that, for any $\overline{z}, \overline{t}, \overline{x} \in \mathbb{K}^*$ satisfying $|\overline{t}| \leq |\overline{x}| \leq |\overline{z}|$,

$$\mathsf{Pick}(\overline{z}; \overline{t}, \overline{x}) = x_{n+1-|\overline{t}|}$$

where $\overline{x} = x_1. \ldots .x_n$. Note that $\overline{z}$ does not occur in the right-hand side of the equality above. It is used to control the recursion and ensures that x can be placed in a safe position. The contents of its components is irrelevant. We only require $|\overline{x}| \leq |\overline{z}|$.

We next want to define a safe recursive function $V_{\overline{\alpha}}$ which, on an input $(\overline{z}, \overline{x};)$, computes the evaluation of all gates numbered from 1 to $|\overline{z}|$ of $\mathscr{C}_{|\overline{x}|}$ on input $(\overline{x}, \overline{\alpha})$ and concatenates the results in one word. For the sake of readability, we denote by $\mathcal{F}_p$ the expression $\mathsf{Pick}(T(\overline{x};); F_p(T(\overline{x};), 1.\overline{z}, \overline{x};), V_{\overline{\alpha}}(\overline{z}, \overline{x}))$. This expression gives the evaluation of the pth parent gate of the current gate (which is the one numbered by $|\overline{z}|+1$). It is obtained by "Picking" it at the right position in the recurrence argument. Denote by $k(i)$ the arity of op_i and by $l(i)$ the arity of rel_i. We may define $V_{\overline{\alpha}}$ as follows (we use definition by cases which can be easily described by combining $\mathsf{Selection}$ operators and simple safe recursive functions) where $G = \mathsf{Gate}(T(\overline{x};), \mathbf{1}.\overline{z}, \overline{x};)$ is the type of the current gate,

$$V_{\overline{\alpha}}(\epsilon, \overline{x};) = \epsilon$$

$$V_{\overline{\alpha}}(c.\overline{z}, \overline{x};) = \begin{cases} \mathsf{cons}(; x_i, V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{0} \text{ and } |\overline{z}| + 1 = i \leq |\overline{x}| \\ \mathsf{cons}(; \alpha_{i-|\overline{x}|}, V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{0} \\ & \text{and } |\overline{z}| + 1 = i \in [|\overline{x}| + 1, |\overline{x}| + m] \\ \mathsf{cons}(; \mathrm{op}_j(; \mathcal{F}_1, \ldots, \mathcal{F}_{k(j)}), V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{1}^j \\ \mathsf{cons}(; \mathrm{rel}_j(; \mathcal{F}_1, \ldots, \mathcal{F}_{l(j)}), V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{1}^{k+j} \\ \mathsf{cons}(; \mathsf{Selection}(; \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3), V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{1}^{k+l+1} \\ \mathsf{cons}(; \mathcal{F}_1, V_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } G = \mathbf{0.0} \\ V_{\overline{\alpha}}(\overline{z}, \overline{x};) & \text{otherwise.} \end{cases}$$

Note that the six first cases in the definition above correspond to the current node being input, constant, operator, relation, selection and output, respectively. The seventh case is to deal with unexpected values of $\overline{z}$, e.g., a too large gate number. We define now a function $R_{\overline{\alpha}}$ which concatenates the values returned by the outputs gates of $\mathscr{C}_{|\overline{x}|}$,

$$\begin{array}{rcl} R_{\overline{\alpha}}(\epsilon, \overline{x};) & = & \epsilon \\ R_{\overline{\alpha}}(c.\overline{z}, \overline{x};) & = & \begin{cases} \mathsf{cons}(; V_{\overline{\alpha}}(\overline{z}, \overline{x};), R_{\overline{\alpha}}(\overline{z}, \overline{x};)) & \text{if } \mathsf{Gate}(T(\overline{x};), \overline{z}, \overline{x};) = \mathbf{0}.\mathbf{0} \\ R_{\overline{\alpha}}(\overline{z}, \overline{x};) & \text{otherwise.} \end{cases} \end{array}$$

The result of the evaluation of $\mathscr{C}_{|\overline{x}|}$ on input $(\overline{x}, \overline{\alpha})$ is given by $R_{\overline{\alpha}}(T(\overline{x};), \overline{x};)$.

The result for partial recursive functions is obtained by replacing all semicolon ";" by commas "," in the proof, and all mentions of "safe recursive" by "partial recursive".

6.2 Proof of Theorem 1

Assume M is a BSS machine over $\mathcal{K}$ computing a function $f_M : \mathbb{K}^* \to \mathbb{K}^*$. Denote by $\alpha_1, \ldots, \alpha_m \in \mathbb{K}$ the constants used by M. A description of the configuration reached by M on input $\overline{x}$ after t iterations can be given by the values of the following four functions:

- $\mathsf{Node}(\overline{x}, \mathbf{0}^t)$, which gives a encoding for the node of M reached after t steps. We assume without loss of generality that the encoding of all output nodes begins with $\mathbf{1}$ and that the encoding of all other nodes begins with $\mathbf{0}$.
- $\mathsf{Length}(\overline{x}, \mathbf{0}^t) = \mathbf{1}^k.\mathbf{0}^{t+|\overline{x}|-k}$, which gives the actual length k of the non-empty part of the tape of M after t steps (the $\mathbf{0}$s are for padding the output so that its length only depends on the length of the input)
- $\mathsf{Tape}(\overline{x}, \mathbf{0}^t) = \overline{y}.\mathbf{0}^{t+|\overline{x}|-k}$, which gives the content $\overline{y}$ of the tape of M after t steps (the $\mathbf{0}$s are for padding as above)
- $\mathsf{Head}(\overline{x}, \mathbf{0}^t)$ which describes the position of the head of M after t steps.

By construction, the length of the output of these four functions depends only on the size of their input. The existence of a universal BSS machine ensures that these functions are BSS computable and, by appropriately clocking the machines computing them, we may assume that the computation time of these machines depends only on the size of their input. Therefore, we can apply Proposition 13 to deduce the existence of four families $\{\mathcal{N}_{n,t}, \mathcal{L}_{n,t}, \mathcal{T}_{n,t}, \mathcal{H}_{n,t} \mid n, t \in \mathbb{N}\}$ of circuits, each of them describable in time $T(n,t)$, polynomial in $n+t$, by a deterministic Turing machine. Using a classical result [18] these machines can be simulated by classical partial recursive functions, which are also partial recursive functions over $\mathcal{K}$ by Remark 5 and Proposition 18. Without loss of generality, one can assume that the descriptions of the circuits above have the form given in the hypotheses of Lemma 23. We can then apply this lemma to conclude that the functions Node, Length, Tape and Head are partial recursive over $\mathcal{K}$. From Length and Tape, it is trivial to build a partial recursive function Result such that $\mathsf{Result}(\overline{x}, \mathbf{1}^t)$ gives exactly the non-empty part of the tape of M on input $\overline{x}$ after t steps. The computation time of M on input $\overline{x}$ is then given by

$$\mathsf{Time}(\overline{x}) = \mu\overline{b}(\mathsf{Node}(\overline{x}, \overline{b}))$$

and the result of this computation by

$$f_M(\overline{x}) = \mathsf{Result}(\overline{x}, \mathsf{Time}(\overline{x})).$$

The computation of M can therefore be simulated by a partial recursive function over $\mathcal{K}$.

The simulation of a partial recursive function by a BSS machine is straightforward.

6.3 Proof of Theorem 2

The first part of the proof is devoted to the simulation of polynomial time BSS machines.

Assume M is a polynomial time BSS machine over $\mathcal{K}$ computing f_M. Denote by $\alpha_1, \dots, \alpha_m \in \mathbb{K}$ the constants used by M. Assume that the computation time is bounded by some polynomial $P(n) = cn^d$. There exist polynomial time BSS machines M' and M'', with the same constants, such that, on input $\overline{x}$, M' returns $\mathbf{1}^{|f_M(\overline{x})|}.\mathbf{0}^{P(|\overline{x}|)-|f_M(\overline{x})|}$, and M'' returns $f_M(\overline{x}).\mathbf{0}^{P(|\overline{x}|)-|f_M(\overline{x})|}$. The strings of $\mathbf{0}$s ensure that the output length depends only on the length of the input. Denote by $f_{M'}$ and $f_{M''}$ the corresponding functions.

By Proposition 13 there exist two P-uniform families of circuits computing $f_{M'}$ and $f_{M''}$.

By Lemma 23, these two families of circuits are simulated by safe recursive functions $F_{M'}$ and $F_{M''}$ over $\mathcal{K}$ respectively. Therefore, $F_{M'}$ and $F_{M''}$ coincide with $f_{M'}$ and $f_{M''}$ respectively. We next define a safe recursive function Hd such that, for $\overline{z}$ of length n and $\overline{y}$ of length at least n, $\mathsf{Hd}(\overline{z}; \overline{y})$ returns the first n characters of $\overline{y}$. In order to do so we introduce Tl such that $\mathsf{Tl}(\overline{z}; \overline{y})$ iterates n times tl on $\overline{y}$, and RevHd wich gives the same result as Hd in the reversed order. We define also a safe recursive function unpad for eliminating all padding $\mathbf{0}$s from the output of $F_{M'}$. These functions are formally defined by

$$\begin{array}{rcl}
\mathsf{Tl}(\epsilon; \overline{y}) & = & \overline{y} \\
\mathsf{Tl}(a.\overline{z}; \overline{y}) & = & \mathsf{tl}(; \mathsf{Tl}(\overline{z}; \overline{y})) \\
\mathsf{RevHd}(\epsilon; \overline{y}) & = & \epsilon \\
\mathsf{RevHd}(a.\overline{z}; \overline{y}) & = & \mathsf{cons}(; \mathsf{Tl}(\overline{z}; \overline{y}), \mathsf{RevHd}(\overline{z}; \overline{y})) \\
\mathsf{Hd}(\overline{z}; \overline{y}) & = & \mathsf{RevHd}(\overline{z}; \mathsf{RevHd}(\overline{z}; \overline{y})) \\
f(\epsilon;) & = & \epsilon \\
f(a.\overline{z};) & = & \left\{ \begin{array}{ll} \mathsf{cons}(; \mathbf{1}, f(\overline{z};)) & \text{if } \mathsf{hd}(; \overline{z}) = \mathbf{1} \\ \epsilon & \text{otherwise} \end{array} \right. \\
\mathsf{unpad}(\overline{z};) & = & \mathsf{cons}(; \mathbf{1}, f(\overline{z};)).
\end{array}$$

Then f_M can be defined by

$$f_M(\overline{x};) = \mathsf{Hd}(\mathsf{unpad}(F_{M'}(\overline{x};);); F_{M''}(\overline{x})).$$

Let us now focus on the time needed to evaluate, with a BSS machine, a safe recursive function f.

We give the proof for $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$. The proof is the same for higher arities (i.e., $f : (\mathbb{K}^*)^p \times (\mathbb{K}^*)^q \to \mathbb{K}^*$).

Let $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ be a safe recursive function over $\mathcal{K}$. We denote by $T[f(\overline{x};\overline{y})]$ the time necessary to compute $f(\overline{x};\overline{y})$ with a BSS machine over $\mathcal{K}$.

Proposition 24 Let $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ be a safe recursive functions. There exists a polynomial p_f such that, for all $(\overline{x},\overline{y}) \in \mathbb{K}^* \times \mathbb{K}^*$,

$$T[f(\overline{x};\overline{y})] \leq p_f(|\overline{x}|).$$

Proof By induction on the depth of the definition tree of f.

If f is a basic safe function then it can be evaluated in constant time. Therefore, the statement is true for basic safe functions.

Assume now f is defined by safe composition from safe recursive functions $g, h_1, \ldots, h_{m+n}$ be as in Definition 20. Then, there exists polynomials $p_g, p_1, \ldots, p_{m+n}$ satisfying the statement for these functions respectively. Without loss of generality, we may assume these polynomials to be nondecreasing. Therefore,

$$\begin{array}{rcl} T[f(\overline{x};\overline{y})] & \leq & T[h_1(\overline{x};\overline{y})] + \cdots + T[h_{m+n}(\overline{x};\overline{y})] \\ & & +T[g(h_1(\overline{x};\overline{y}),\ldots,h_m(\overline{x};\overline{y}); h_{m+1}(\overline{x};\overline{y}),\ldots,h_{m+n}(\overline{x};\overline{y}))] \\ & \leq & p_1(|\overline{x}|) + \cdots + p_{m+n}(|\overline{x}|) + p_g(p_1(|\overline{x}|) + \cdots + p_m(|\overline{x}|)) \end{array}$$

which shows we may take

$$p_f = p_1 + \cdots + p_{m+n} + p_g \circ (p_1 + \cdots + p_m).$$

Finally, assume f is defined by safe recursion and let h, g be as in Definition 20. Then, there exist polynomials p_h, p_g satisfying the statement for these functions respectively. In addition, we may also assume that p_g and p_h are nondecreasing. Therefore,

$$T[f(\epsilon,\overline{x};\overline{y})] = p_h(|\overline{x}|)$$

and, when unfolding the recurrence,

$$\begin{array}{rcl} T[f(a.\overline{z},\overline{x};\overline{y})] & \leq & T[f(\overline{z},\overline{x};\overline{y})] + T[g(\overline{z},\overline{x}; f(\overline{z},\overline{x};\overline{y}),\overline{y})] \\ & \leq & T[f(\overline{z},\overline{x};\overline{y})] + p_g(|\overline{z},\overline{x}|) \\ & \vdots & \\ & \leq & |\overline{z}| p_g(|\overline{z},\overline{x}|) + p_h(|\overline{x}|) \end{array} \tag{1}$$

which shows we may take

$$p_f = \mathrm{Id}\, p_g + p_h.$$

Remark 25 **(i)** Note that in (1) above the evaluation time $T[g(\overline{z},\overline{x}; f(\overline{z},\overline{x};\overline{y}),\overline{y})]$, because of the induction hypothesis, does not depend on $f(\overline{z},\overline{x};\overline{y})$. This independence in the inductive argument in the proof is what keeps the evaluation time polynomially bounded.

(ii) A similar result could be obtained by using the notion of "tier" introduced in [19].

7 A Characterization of the Parallel Class $\mathrm{FPAR}_{\mathcal{K}}$

In this section we prove Theorem 3. We first introduce the notion of safe recursion with substitutions.

7.1 Safe Recursion with Substitutions

Definition 26 The set of functions defined with *safe recursion with substitutions* over $\mathcal{K}$ is the smallest set of functions $f : (\mathbb{K}^*)^p \times (\mathbb{K}^*)^q \to \mathbb{K}^*$, containing the basic safe functions, and closed under safe composition and the following operation:

Safe recursion with substitutions. Let $h : \mathbb{K}^* \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$, $g : (\mathbb{K}^*)^2 \times (\mathbb{K}^*)^{l+1} \to \mathbb{K}^*$, and $\sigma_j : \emptyset \times \mathbb{K}^* \to \mathbb{K}^*$ for $0 < j \leq l$ be safe recursive functions.

Function $f : (\mathbb{K})^2 \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ is defined by safe recursion with substitutions as follows:

$$\begin{array}{rcl} f(\epsilon, \overline{x}; \overline{u}, \overline{y}), & = & h(\overline{x}; \overline{u}, \overline{y}) \\ f(a.\overline{z}, \overline{x}; \overline{u}, \overline{y}) & = & \left\{ \begin{array}{lr} g\left(\overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \sigma_1(;\overline{u}), \overline{y}), \ldots, f(\overline{z}, \overline{x}; \sigma_l(;\overline{u}), \overline{y}), \overline{y}\right) & \\ & \text{if } \forall j \ f(\overline{z}, \overline{x}; \sigma_j(;\overline{u}), \overline{y}) \neq \perp \\ \perp & \text{otherwise.} \end{array} \right. \end{array}$$

The functions σ_j are called *substitution functions.*

7.2 Proof of Theorem 3 ("only if" part)

By hypothesis, the family of circuits we want to simulate here is P-uniform. This means that there exists a machine over $\mathcal{K}$ which, given a pair (n, i), computes the description of the ith gate of the circuit $\mathscr{C}_n$ in time polynomial in n. Let $L(n)$ be the polynomial bounding the number of output gates of $\mathscr{C}_n$ and, for $1 \leq j \leq L(n)$ let $\mathscr{C}_{nj}$ be the circuit induced by $\mathscr{C}_n$ with only one output node (corresponding to the jth output node of $\mathscr{C}_n$).

The idea for writing a function evaluating the circuit $\mathscr{C}_{nj}$ at an input $\overline{x} \in \mathbb{K}^n$ is simple. The function is defined recursively, the evaluation of every node depending on the evaluation of its parent nodes. However, in order to do this with the mechanism of safe recursion with substitutions, we need a way to describe these parents nodes in constant time (actually with a substitution function) and this is not provided by the P-uniformity hypothesis. A way to do so is by describing a parent node simply by its relative position among the possible parent nodes of the node at hand. The iteration of this procedure naturally leads to the notion of path.

A *path* from the output node o to a node w is a sequence of nodes $o = v_0, \ldots, v_\ell = w$ such that v_i is a parent of v_{i-1} for $i = 1, \ldots, \ell$. The node w is the *top* of the path. We describe a path by an element $\overline{y} \in \mathbb{K}^*$ as follows

- The path consisting on the single output node o is represented by $\epsilon \in \mathbb{K}^*$.
- Let r be the maximal arity of a relation or a function of the structure. Then $s = \lceil \log_2(\max\{r, 3\}) \rceil$ is the size necessary to write the binary encoding of any parent for a given node. Assume that $v_0, \ldots, v_{\ell-1}$ is represented by $\overline{y}$. If v_ℓ is the ith parent node of $v_{\ell-1}$ then the path $v_0, \ldots, v_\ell$ is represented by $\overline{a_i}.\overline{y}$,

where $\overline{a_i} \in \{\mathbf{0}, \mathbf{1}\}^*$ represents the binary encoding of i. We add as many $\mathbf{0}$s as needed in front such that $\overline{a_i}$ have length s.

Lemma 27 There exists a safe recursive function Gate such that, for an element $\overline{x} \in \mathbb{K}^n$, a natural $j \leq L(n)$, and a path $\overline{y}$ in $\mathscr{C}_{nj}$,

$$\mathsf{Gate}(\mathbf{1}^j, \overline{x}; \overline{y}) = \begin{cases} \mathbf{0} & \text{if } \mathsf{top}(\overline{y}) \text{ is the } i\text{th input gate} \\ \mathbf{0.0} & \text{if } \mathsf{top}(\overline{y}) \text{ is the output gate} \\ \mathbf{1}^i & \text{if } \mathsf{top}(\overline{y}) \text{ is a gate labeled with } \mathrm{op}_i \\ \mathbf{1}^{k+i} & \text{if } \mathsf{top}(\overline{y}) \text{ is a gate labeled with } \mathrm{rel}_i \\ \mathbf{1}^{k+l+1} & \text{if } \mathsf{top}(\overline{y}) \text{ is a } \textit{selection} \text{ node} \\ \epsilon & \text{otherwise} \end{cases}$$

where $\mathsf{top}(\overline{y})$ denotes the top of $\overline{y}$.

Proof Since the length of $\overline{y}$ is polynomial in n, the function Gate can be computed in polynomial time. Therefore, it is safe recursive by Theorem 2.

Assume $p(n) = cn^d$ is a polynomial bounding the depth of the circuit $\mathscr{C}_n$. The goal now is to describe a function Eval, definable via safe recursion with substitutions, such that $\mathsf{Eval}(\mathbf{1}^j, \overline{t}, \overline{x}; \epsilon)$ is the output of $\mathscr{C}_{nj}$ on input $\overline{x}$ whenever $|\overline{x}| = n$ and $|\overline{t}| = p(n)$.

To do so we will use the substitution functions σ_i defined by $\sigma_i(; \overline{y}) = \overline{a_i}.\overline{y}$ where $\overline{a_i} \in \{\mathbf{0}, \mathbf{1}\}^*$ is the binary encoding of i. Note that σ_i is safe recursive for $i = 1, \ldots, r$. Also, note that a path $\overline{y}$ of length ℓ is easily described by composing ℓ times some of these functions.

Denote by $k(i)$ the arity of op_i, and $l(i)$ the arity of rel_i. Also, denote by $\mathcal{F}_p$ the expression $\mathsf{Eval}(\mathbf{1}^j, \overline{t}, \overline{x}; \sigma_p(; \overline{y}))$ which gives the evaluation of the pth parent gate of the current gate and by $G = \mathsf{Gate}(\mathbf{1}^j, \overline{x}; \overline{y})$ the type of the current gate. The function Eval is then defined as follows,

$$\mathsf{Eval}(\mathbf{1}^j, \epsilon, \overline{x}; \overline{y}) = \epsilon$$

$$\mathsf{Eval}(\mathbf{1}^j, a.\overline{t}, \overline{x}; \overline{y}) = \begin{cases} x_i & \text{if } G = \mathbf{0} \\ \mathrm{op}_i(; \mathcal{F}_1, \ldots, \mathcal{F}_{k(i)}) & \text{if } G = \mathbf{1}^i \\ \mathrm{rel}_i(; \mathcal{F}_1, \ldots, \mathcal{F}_{l(i)}) & \text{if } G = \mathbf{1}^{k+i} \\ \mathsf{Selection}(; \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3) & \text{if } G = \mathbf{1}^{k+l+1} \\ \mathcal{F}_1 & \text{if } G = \mathbf{0.0} \end{cases}$$

All equality and inequality tests are once again easily obtained by combining $\mathsf{Selection}$ operators and simple safe recursive functions.

Note that we use above an enumeration for the gates of $\mathscr{C}_{nj}$ different to the one described in Remark 12. The expressions $\mathcal{F}_p$ evaluating the pth parent of a gate are, consequently, different here and in the proof of Lemma 23.

Theorem 2 ensures the existence of a safe recursive functions P_{cd} such that $|P_{cd}(\overline{x};)| = cn^d$. We use P_{cd} to define with safe recursion over $\mathcal{K}$ the following function

$$\begin{aligned} \mathsf{concat}(\epsilon, \overline{x};) &= \mathsf{Eval}(\mathbf{1}, P_{cd}(\overline{x};), \overline{x}.\overline{z}; \epsilon) \\ \mathsf{concat}(a.\overline{y}, \overline{x};) &= \mathsf{cons}(; \mathsf{Eval}(\overline{y}, P_{cd}(\overline{x};), \overline{x}.\overline{z}; \epsilon), \mathsf{concat}(\overline{y}, \overline{x};)) \end{aligned}$$

such that $\mathsf{concat}(\overline{y}, \overline{x};))$ concatenates the outputs of $\mathscr{C}_{nj}$ for $j = 1, \dots, |\overline{y}|$.

Theorem 2 also ensures the existence of a safe recursive functions P_L such that $P_L(\overline{x}) = \mathbf{1}^{L(n)}$. The function computed by the P-uniform family of circuits $\{\mathscr{C}_n \mid n \in \mathbb{N}\}$ is then defined with safe recursion with substitutions by $\mathsf{Circuit}(\overline{x};) = \mathsf{concat}(P_L(\overline{x};), \overline{x};)$.

7.3 Proof of Theorem 3 ("if" part)

Again, we give the proof for $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$.

Let f be a function defined with safe recursion with substitutions, and denote by f_n the restriction of f to the set of inputs of size n. We need to prove that f can be computed by a P-uniform family of circuits $\{\mathscr{C}_n \mid n \in \mathbb{N}\}$ of polynomial depth.

As in §6.3, for a function $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ defined with safe recursion with substitutions, we denote by $D[f(\overline{x}; \overline{y})]$ the depth of the shortest circuit $\mathscr{C}$ computing $f(\overline{x}; \overline{y})$.

Proposition 28 Let $f : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ be defined by safe recursion with substitutions. There exists a polynomial p_f such that, for all $(\overline{x}, \overline{y}) \in \mathbb{K}^* \times \mathbb{K}^*$,

$$D[f(\overline{x}; \overline{y})] \leq p_f(|\overline{x}|)$$

and

$$|f(\overline{x}; \overline{y})| \leq p_f(|\overline{x}|).$$

Proof It is done by induction on the depth of the definition tree of f, as in the proof of Proposition 24 (to which this proof is actually similar).

If f is a basic safe function then it can be evaluated in constant time. Therefore, the statement is true for basic safe functions.

Assume now f is defined by safe composition from safe recursive functions $g, h_1, \dots, h_{m+n}$ as in Definition 20. Then, there exists polynomials $p_g, p_1, \dots, p_{m+n}$ satisfying the statement for these functions respectively. Without loss of generality, we may assume these polynomials to be nondecreasing. Therefore,

$$\begin{aligned} D[f(\overline{x}; \overline{y})] &\leq \max\{D[h_1(\overline{x}; \overline{y})], \dots, D[h_{m+n}(\overline{x}; \overline{y})]\} \\ &\quad + D[g(h_1(\overline{x}; \overline{y}), \dots, h_m(\overline{x}; \overline{y}); h_{m+1}(\overline{x}; \overline{y}), \dots, h_{m+n}(\overline{x}; \overline{y}))] \\ &\leq \max\{p_1(|\overline{x}|), \dots, p_{m+n}(|\overline{x}|)\} + p_g(p_1(|\overline{x}|) + \cdots + p_m(|\overline{x}|)) \end{aligned}$$

which shows we may take

$$p_f = \max\{p_1, \dots, p_{m+n}\} + p_g \circ (p_1 + \cdots + p_m)$$

where $\max\{p_1, \dots, p_{m+n}\}$ denotes any polynomial bounding above $p_1, \dots, p_{m+n}$.

Finally, assume f is defined by safe recursion with substitutions and let $h_1, \dots, h_k, g_1, \dots, g_k$ and σ_{ij} ($i = 1, \dots, k$ and $j = 1, \dots, l$) as in Definition 26. Then, there exist polynomials $p_{g_1}, \dots, p_{g_k}, p_{h_1}, \dots, p_{h_k}$ satisfying the statement for $g_1, \dots, g_k, h_1, \dots, h_k$ respectively. Again, without loss of generality, we may assume $p_{g_i} = p_g$ for $i = 1, \dots, k$ and $p_{h_i} = p_h$ for $i = 1, \dots, k$. In addition, we also may assume that p_g and p_h are nondecreasing. Therefore, for $i = 1, \dots, k$,

$$D[f_i(\epsilon, \overline{x}; \overline{u}, \overline{y})] = p_h(|\overline{x}|)$$

and

$$\begin{aligned}
D[f_i(a.\overline{z},\overline{x};\overline{u},\overline{y})] &\leq \max_{i,j}\{D[f_i(\overline{z},\overline{x};\sigma_{ij}(;\overline{u}),\overline{y})]\} \\
&\quad +D\big[g_i\big(\overline{z},\overline{x};f_1(\overline{z},\overline{x};\sigma_{11}(;\overline{u}),\overline{y}),\ldots,f_1(\overline{z},\overline{x};\sigma_{1l}(;\overline{u}),\overline{y}), \\
&\qquad\qquad \ldots,f_k(\overline{z},\overline{x};\sigma_{k1}(;\overline{u}),\overline{y}),\ldots,f_k(\overline{z},\overline{x};\sigma_{kl}(;\overline{u}),\overline{y}),\overline{y}\big)\big] \\
&\leq \max_{i,j}\{D[f_i(\overline{z},\overline{x};\sigma_{ij}(;\overline{u}),\overline{y})]\}+p_g(|\overline{z},\overline{x}|).
\end{aligned}$$

Applying the same reasoning for $f_i(\overline{z},\overline{x};\sigma_{ij}(;\overline{u}),\overline{y})$ and denoting $\overline{z'}=\mathsf{tl}(\overline{z})$ we obtain

$$\begin{aligned}
&D[f_i(\overline{z},\overline{x};\sigma_{ij}(;\overline{u}),\overline{y})] \\
\leq\ & D[\sigma_{ij}(;\overline{u})]\}+\max_{i',j'}\{D[f_{i'}(\overline{z},\overline{x};\sigma_{i'j'}(;\sigma_{ij}(;\overline{u})),\overline{y})]\} \\
&+D\big[g_i\big(\overline{z},\overline{x};f_1(\overline{z},\overline{x};;\sigma_{11}(;\sigma_{ij}(;\overline{u})),\overline{y}),\ldots,f_1(\overline{z},\overline{x};\sigma_{1l}(;\sigma_{ij}(;\overline{u})),\overline{y}), \\
&\qquad \ldots,f_k(\overline{z},\overline{x};\sigma_{k1}(;\sigma_{ij}(;\overline{u})),\overline{y}),\ldots,f_k(\overline{z},\overline{x};\sigma_{kl}(;\sigma_{ij}(;\overline{u})),\overline{y}),\overline{y}\big)\big] \\
\leq\ & D[\sigma_{ij}(;\overline{u})]\}+\max_{i',j'}\{D[f_{i'}(\overline{z},\overline{x};\sigma_{i'j'}(;\sigma_{ij}(;\overline{u})),\overline{y})]\}+p_g(|\overline{z'},\overline{x}|).
\end{aligned}$$

Continuing in this way and denoting by $\overline{z_\ell}$ the result of applying ℓ times tl to $\overline{z}$ we obtain

$$\begin{aligned}
D[f_i(a.\overline{z},\overline{x};\overline{u},\overline{y})] &\leq \sum_{\ell=1}^{|a.\overline{z}|}\left(\max_{\substack{i_1,\ldots,i_\ell\leq k\\ j_1,\ldots,j_\ell\leq l}}\left\{D[\sigma_{i_1j_1}(;\sigma_{i_2j_2}(;\ldots\sigma_{i_{\ell-1}j_{\ell-1}}(;u)\ldots))]\right\}+p_g(\overline{z_\ell},\overline{x})\right) \\
&\quad +\max_{\substack{i_1,\ldots,i_\ell\leq k\\ j_1,\ldots,j_\ell\leq l}}\left\{D[h_{i_1}(\overline{x};\sigma_{i_1j_1}(;\sigma_{i_2j_2}(;\ldots\sigma_{i_\ell j_\ell}(;u)\ldots)),\overline{y})]\right\} \\
&\leq |a.\overline{z}|\big(\mathcal{O}(|\overline{z}|)+p_g(|\overline{z},\overline{x}|)\big)+p_h(|\overline{x}|)
\end{aligned}$$

the $\mathcal{O}(|\overline{z}|)$ since each σ_{ij} can be computed in constant time by Proposition 24, and that we compute them sequentially. This yields again a polynomial bound for the depth.

The polynomial bound for the output size is obvious and we can take p_f to be a polynomial bounding both depth and output size.

Proposition 28 shows that every circuit $\mathscr{C}_n$ of the family computing f has depth polynomial in n. The P-uniformity of this family is implicit in the proof of the proposition.

In the classical setting (see [22]), safe recursion with substitution characterizes the class FPSPACE. However, in the general setting, this notion of working space is meaningless, as pointed in [25]: on some structures like $(\mathbb{R},0,1,\leq,+,-,*)$, any computation can be done in constant working space. However, since in the classical setting we have FPAR = FPSPACE, our result extends the classical one from [22].

References

[1] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[2] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer-Verlag, 1998.

[3] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the Amer. Math. Soc.*, 21:1–46, 1989.

[4] P. Clote. Computational models and function algebras. In D. Leivant, editor, *LCC'94*, volume 960 of *Lect. Notes in Comp. Sci.*, pages 98–130. Springer-Verlag, 1995.

[5] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[6] S. A. Cook. Computability and complexity of higher-type functions. In Y. Moschovakis, editor, *Logic from Computer Science*, pages 51–72. Springer-Verlag, New York, 1992.

[7] F. Cucker. $\mathrm{P}_{\mathbb{R}} \neq \mathrm{NC}_{\mathbb{R}}$. *Journal of Complexity*, 8:230–238, 1992.

[8] F. Cucker, M. Shub, and S. Smale. Separation of complexity classes in Koiran's weak model. *Theoretical Computer Science*, 133(1):3–14, 11 October 1994.

[9] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1995.

[10] R. Fagin. Generalized first order spectra and polynomial time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

[11] J. B. Goode. Accessible telephone directories. *Journal for Symbolic Logic*, 59(1):92–105, 1994.

[12] Y. Gurevich. Algebras of feasible functions. In *Twenty Fourth Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[13] Y. Gurevich and E. Grädel. Tailoring recursion for complexity. *Journal for Symbolic Logic*, 60:952–969, 1995.

[14] M. Hofmann. Type systems for polynomial-time computation, 1999. Habilitation.

[15] N. Immerman. *Descriptive Complexity.* Springer-Verlag, 1999.

[16] R. Irwin, B. Kapron, and J. Royer. On characterizations of the basic feasible functionals. *J. of Functional Programming*, 11:117–153, 2001.

[17] N. Jones. The expressive power of higher order types. *J. of Functional Programming*, 11:55–94, 2001.

[18] S.C. Kleene. General Recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[19] D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[20] D. Leivant. Intrinsic theories and computational complexity. In *LCC'94*, volume 960 of *Lect. Notes in Comp. Sci.*, pages 177–194. Springer-Verlag, 1995.

[21] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19:167–184, September 1993.

[22] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL'94*, volume 933 of *Lect. Notes in Comp. Sci.*, pages 369–380, Kazimierz, Poland, 1995. Springer-Verlag.

[23] J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lect. Notes in Comp. Sci.*, pages 25–42. Springer-Verlag, Nov 2000.

[24] K. Meer. A note on a $P \neq NP$ result for a restricted class of real machines. *Journal of Complexity*, 8:451–453, 1992.

[25] C. Michaux. Une remarque à propos des machines sur $\mathbb{R}$ introduites par Blum, Shub et Smale. *C. R. Acad. Sci. Paris*, 309, Série I:435–437, 1989.

[26] B. Poizat. *Les Petits Cailloux.* Aléas, 1995.

[27] H.E. Rose. *Subrecursion.* Oxford Univ. Press, 1984.

[28] V. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 7:319–323, 1980.