Safe Recursion and Calculus over an Arbitrary Structure

Olivier Bournez, Paulin de Naurois *, and Jean-Yves Marion

LORIA/INRIA,
615 rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, Nancy, France.
email: {bournez, denauroi, marionjy}@loria.fr

Abstract. In this paper, we show that the Bellantoni and Cook characterization of polynomial time computable functions in term of safe recursive functions can be transfered to the model of computation over an arbitrary structure developped by L. Blum, M. Shub and S. Smale. Hence, we provide an implicit complexity characterization of functions computable in polynomial time over any arbitrary structure.

1 Introduction

Since the development of classical complexity theory, several attempts have been done to provide nice formalisms to characterize functions computable in polynomial time. An important contribution to this field is the work of Neil Jones, found in [Jon97] and [Jon99], based on some programming languages properties. Another thread of results, called implicit complexity, has its roots in the work of S. Bellantoni and S. Cook, found in [BC92], with a viewpoint more related to the notion of recursion, and formal definition of functions, leading to a hierarchy of subsets of the set of primitive recursive functions.

In their seminal paper [BSS89], Lenore Blum, Mike Shub and Steve Smale introduced a model of computation over the real numbers which was later on extended to a computational model over any arbitrary logical structure [Poi95]. Complexity classes like PTIME and NPTIME can be defined, and complete problems in these classes can be shown to exist [Poi95,BCSS98]. On many aspects, this is an extension of the classical complexity theory since complexity classes correspond to classical complexity classes when dealing with booleans or integers. In addition, this new model provides new insights for understanding complexity theory when dealing with structures over other domains [BCSS98]. Several results have been obtained for this model in the last decade, including separation of complexity classes over specific structures [BCSS98].

Our goal here is to make a junction between these two different fields, yielding some notion of implicit complexity not only over the booleans, or the integers,

* This author has been partially supported by City University of Hong Kong SRG grant 7001290

i.e. in the classical setting, but over any arbitrary structure. This is, to the best of our knowledge, the first attempt to do so.

Indeed, in this paper, we present a characterization of primitive recursive functions over arbitrary structures, and we present a general definition of safe recursion, which gives in the classical case the classical notion of safe recursion over the natural numbers. We then show our main result:

Main Theorem: *Over any structure* $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, 0, 1)$*, the set of safe recursive functions over* $\mathcal{K}$ *is exactly the set of functions computed in polynomial time by a BSS machine over* $\mathcal{K}$*.*

This characterization extends the one from Bellantoni and Cook since its corresponds to the original one of [BC92] when dealing with structures over the booleans or the integers.

In Section 2, we recall what BSS-computability and complexity theory are. In Section 3, we give a characterization of primitive recursive functions over an arbitrary structure. We introduce our notion of safe recursive function in Section 4. We give the proof of the Main Theorem in Section 5. Section 6 is a conclusion.

2 Computing over an Arbitrary Structure

In this section, we introduce computability and complexity over an arbitrary structure. See [BCSS98] for formal details.

A structure $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, rel_1, \ldots, rel_l)$ is given by some underlying set $\mathbb{K}$, some operators $op_1, \ldots, op_k$ with arities, and some relations $rel_1, \ldots, rel_l$ with arities. Constants are given by operators of arity 0. We will not distinguish between operator and relation symbols and their corresponding interpretations as functions and relations respectively over the underlying set $\mathbb{K}$.

Assume that some structure $\mathcal{K}$ is fixed. We assume that equality relation $=$ is one relation of the structure, and that there are at least two different constants 0 and 1 in the structure. A good example for such a structure is $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, 0, 1)$. Another one, corresponding to classical complexity and computability theory is $\mathcal{K} = (\{0, 1\}, \vee, \wedge, =, 0, 1)$.

A BSS-machine over $\mathcal{K}$ is essentially a Turing Machine, which is able to perform the basic operations $op_1, \ldots, op_k$ and the basic tests $rel_1, \ldots, rel_l$ at unit cost, and whose tape cells can hold arbitrary elements of the underlying set $\mathbb{K}$ [Poi95,BCSS98].

More formally.

BSS-machine over $\mathcal{K}$: Let $\mathbb{K}^* = \bigcup_{i \in \mathbb{N}} \mathbb{K}^i$ denote the set of words over alphabet $\mathbb{K}$.

An instantaneous description of a k-tapes BSS-machine is given by some internal state q, belonging to some fixed finite set Q of possible internal states, also called "nodes", and by instantaneous descriptions of the k-tapes of the machine. An instantaneous description of a tape i is given by the position of the

head i of the machine, and by two words $w_l^i, w_r^i \in \mathbb{K}^*$ that give the content of tape i at left and right of the head.

To each node $q \in Q$, is associated some instruction among the following possibilities:

1. *move:* move head left (respectively: right) of tape i and go to node q',
2. *operation:* replace the element in front of the head of tape i by $op(w_1^j, \ldots, w_n^j)$ and go to node q', where $w_r^j = w_1^j.w_2^j.\ldots.w_a^j\ldots$ is the right content of tape j, op is some operation of the structure $\mathcal{K}$, and n its arity.
3. *relation:* test whether $rel(w_1^i, \ldots, w_n^i)$ is true and go to node q' or node q'' accordingly, where $w_r^i = w_1^i.\ldots.w_n^i\ldots$ is the right content of tape i, rel is some relation of the structure $\mathcal{K}$ and n its arity.

Two particular nodes q_0 and q_{acc} are termed as initial and terminal respectively. A language L is a subset of $\mathbb{K}^*$. The language is said to be recognized by the machine if it corresponds to the subset of the words that are accepted: an input $w \in \mathbb{K}^*$ is accepted iff the machine started in node q_0 with w on its first tape, and its other tapes blank, and evolving according to its program, eventually reaches node q_{acc}. The input w is said to be accepted in time T, if the computations requires T instructions. Machine M computes function $f : \mathbb{K}^* \to \mathbb{K}^*$ iff started in node q_0 with $w \in \mathbb{K}^*$ on its first tape, its other tapes blank, and evolving according to its program, it eventually reaches node q_{acc} with its first tape equal to $f(w)$.

Class PTIME: The length of an input $w \in \mathbb{K}^*$, denoted by $|w|$ is the length of word w considered as a word over alphabet $\mathbb{K}$.

A problem $P \subset \mathbb{K}^*$ is in class PTIME (respectively a function $f : \mathbb{K}^* \to \mathbb{K}^*$ is in class FPTIME), if there exists a polynomial p and a machine M, so that for all $w \in \mathbb{K}^*$, M stops in time $p(|w|)$, and for all $w \in \mathbb{K}^*$, M accepts iff $w \in P$ (respectively: M computes function f).

This notion of computability corresponds to the classical one for structures over the boolean or the integers, and corresponds to the one from Blum Shub and Smale over the real numbers.

Proposition 1.

1. *Class* PTIME *is the classical one over structure* $\mathcal{K} = (\{0,1\}, \vee, \wedge, =, 0, 1)$.
2. *Class* PTIME *is the class* PTIME *of Blum Shub and Smale over structure* $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, 0, 1)$.

The reader can refer to [BCSS98] for a monograph presenting the model and some results about complexity in this model for structures over real and complex numbers, or to the monograph [Poi95] for considerations about more general structures.

3 Partial Recursive and Primitive Recursive Functions

As in the classical settings, alternative presentations exists. In particular, computable functions can be characterized algebraically, in terms of the smallest set of functions containing some initial functions and closed by composition, primitive recursion and minimization. This was done for the first time in the original paper [BSS89] for computability over the real numbers.

In this section, we present a characterization that works over any arbitrary structure, and that we think nicer than the original one when applied on real numbers. See comments at the end of this section.

We define a set of functions: $(\mathbb{K}^*)^k \to \mathbb{K}^*$, taking as inputs arrays of words of elements in $\mathbb{K}$, and returning as output a word of elements in $\mathbb{K}$. In our notations, words of elements in $\mathbb{K}$ will be represented with overlined letters, while simple elements in $\mathbb{K}$ will be represented by simple letters. For instance, $a.\overline{x}$ stands for the word in $\mathbb{K}^*$ whose first letter is a and which ends with the word $\overline{x}$. When the output of a function is undefined, we use the symbol $\perp$.

Definition 1. *The set of primitive recursive functions over* $\mathbb{K}$ *is the smallest set of functions:* $(\mathbb{K}^*)^k \to \mathbb{K}^*$*, containing the basic functions, and closed under the operations of*

- *composition*
- *primitive recursion*

Definition 2. *The set of partial recursive functions over* $\mathbb{K}$ *is the smallest set of partial functions:* $(\mathbb{K}^*)^k \to \mathbb{K}^*$*, containing the basic functions, and closed under the operations of*

- *composition*
- *primitive recursion*
- *minimization*

Our basic functions are of three kinds:

- functions making elementary manipulations of words of elements in $\mathbb{K}$.[1] For any $a \in \mathbb{K}, \overline{x}, \overline{x_1}, \overline{x_2} \in \mathbb{K}^*$:

$$\begin{aligned} hd(a.\overline{x}) &= a \\ hd(\emptyset) &= \emptyset \\ tl(a.\overline{x}) &= \overline{x} \\ tl(\emptyset) &= \emptyset \\ cons(a.\overline{x_1}, \overline{x_2}) &= a.\overline{x_2} \end{aligned}$$

[1] The formal definition of functions *hd* and *tl* given here is actually a primitive recursive definition with no recurrence argument. However, when we introduce the notion of safe recursion in section 4, these functions *hd* and *tl* need 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 them here as *a priori* functions as well.

– functions of structure $\mathcal{K}$: for any operator $op_\imath$ or relation $rel_\imath$ of arity $n_\imath$ we have the following initial functions:

$$Op_\imath(a_1.\overline{x_1},\ldots,a_{n_\imath}.\overline{x_{n_\imath}}) = (op_\imath(a_1,\ldots,a_{n_\imath})).\overline{x_{n_\imath}}$$
$$Rel_\imath(a_1.\overline{x_1},\ldots,a_{n_\imath}.\overline{x_{n_\imath}}) = \begin{cases} (1.\overline{x_{n_\imath}}) \text{ if } rel_\imath(a_1,\ldots,a_{n_\imath}) \\ (0.\overline{x_{n_\imath}}) \text{ otherwise} \end{cases}$$

– test function :[2]

$$C(a.\overline{x},\overline{y},\overline{z}) = \begin{cases} \overline{y} & \text{if } a = 1 \\ \overline{z} & \text{otherwise} \end{cases}$$

Operations mentioned above are:

– Composition: Suppose f: $(\mathbb{K}^*)^k \to \mathbb{K}^*$ and $g_1,\ldots,g_k$: $(\mathbb{K}^*)^{l_\imath} \to \mathbb{K}^*$ are given partial functions. Then the composition $g \circ f : (\mathbb{K}^*)^{l_1+\ldots+l_k} \to \mathbb{K}^*$ is defined by
$g \circ f(\overline{x_1},\ldots,\overline{x_{l_1+\ldots+l_k}}) = f(g_1(\overline{x_1},\ldots,\overline{x_{l_1}}),\ldots,g_k(\overline{x_{l_1+\ldots+l_{k-1}+1}},\ldots,\overline{x_{l_k}}))$.
– primitive recursion [3]: Suppose f: $\mathbb{K}^* \to \mathbb{K}^*$ and g: $(\mathbb{K}^*)^3 \to \mathbb{K}^*$ are given partial functions. Then h: $(\mathbb{K}^*)^2 \to \mathbb{K}^*$ is defined with primitive recursion if h satisfies:

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

– minimization: Suppose $f : (\mathbb{K}^*)^2 \to \mathbb{K}^*$. One can define minimization on the first argument of f as a function g: $\mathbb{K}^* \to \mathbb{K}^*$ where we note $g(\overline{y}) = \mu\overline{x}\,(f(\overline{x},\overline{y}))$:

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

[2] One may wonder if we do not need projection and identity. These are actually trivially given by this test function:

$$id(\overline{x}) = C(1,\overline{x},whatever)$$
$$pr_{left}(\overline{x},\overline{y}) = C(1,\overline{x},\overline{y})$$
$$pr_{right}(\overline{x},\overline{y}) = C(0,\overline{x},\overline{y})$$

[3] In this definition, 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 still depends on the value of the first element of $\overline{y}$. Therefore, one can define a recursive function h' such that $h'(0.a.\overline{y},\overline{x})$ gives the expected result.

Note 1. The only operation producing the $\perp$ symbol, used to represent an undefined output, is minimization. Therefore, primitive recursive functions are total functions, whereas partial recursive functions may be partial functions.

Note 2. The operation of minimization on the first argument of f returns the smallest word made only of 0 letters satisfying one property. The reason why it does not return a smallest word made of *any* letter in $\mathbb{K}$ is to ensure determinism, and therefore calculability. On a structure where we have P $\neq$ NP, such a non-deterministic minimization may very well not be computable by a BSS machine, which is in essence deterministic.

Note 3. Our definition of primitive recursion and of minimization is sightly different from the one found in [BSS89]. 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: $\mathbb{N} * \mathbb{K}^k \rightarrow \mathbb{K}^l$. Therefore, they only capture finite dimensional functions. It is known that, on the real numbers with $+, -, *$ operators, finite dimensional functions are equivalent to non-finite dimensional functions (see [Mic89]), but this is not true over other structures, for instance $\mathbb{Z}_2$. 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, thus we consider it to be a more general and natural way to define computable functions, and moreover closer to the way BSS machines over $\mathbb{K}$ really work.

Before going any further, let us state the following technical result:

Proposition 2. *Simultaneous primitive recursion as in the following schema:*

$$
\begin{aligned}
h_1(\emptyset, \overline{x}), \ldots, h_k(\emptyset, \overline{x}) &= f_1(\overline{x}), \ldots, f_k(\overline{x}) \\
h_1(a.\overline{y}, \overline{x}) &= \begin{cases} g_1(\overline{y}, h_1(\overline{y}, \overline{x}), \ldots, h_k(\overline{y}, \overline{x}), \overline{x}) & \text{if } h(\overline{y}, \overline{x}) \neq \perp \\ \perp & \text{otherwise} \end{cases} \\
&\vdots \\
h_k(a.\overline{y}, \overline{x}) &= \begin{cases} g_k(\overline{y}, h_1(\overline{y}, \overline{x}), \ldots, h_k(\overline{y}, \overline{x}), \overline{x}) & \text{if } h(\overline{y}, \overline{x}) \neq \perp \\ \perp & \text{otherwise} \end{cases}
\end{aligned}
$$

is definable with primitive recursive functions.

Now we can state a theorem similar to the one found in [BSS89]:

Theorem 1. *Over any structure $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, 0, 1)$,*
The set of partial recursive functions over $\mathcal{K}$ is exactly the set of functions computed by a BSS machine over $\mathcal{K}$.

The idea of the proof is to build a set of partial recursive functions simulating a BSS machine. This involves the same techniques as the proof of the Main Theorem, and is not developped here. The fact that partial recursive functions are computable by a BSS-machine is clear.

4 Safe Recursive Functions

In this section we define our set of safe recursive functions over $\mathbb{K}$, extending the notion of safe recursive functions over the natural numbers found in [BC92]. Safe recursive functions are defined in a quite similar manner as primitive recursive functions however, for safe recursive functions, we define two different types of arguments, each of which having different properties and different purposes. The first type of argument, called "normal" arguments, is similar to the arguments of our 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 is called "safe", and can not be used to control recursion. This distinction between safe and normal arguments ensures that our safe recursive functions can be computed in polynomial time by a BSS machine. In our notations, the two different types of arguments are separated by a semicolon ";" : on the left part of the argument set we place the normal arguments and on the right part the safe arguments. Let us define now our safe recursive functions:

Definition 3. *The set of safe recursive functions over* $\mathbb{K}$ *is the smallest set of functions:* $(\mathbb{K}^*)^k \to \mathbb{K}^*$*, containing the basic safe functions, and closed under the operations of*

- *safe composition*
- *safe recursion*

Our basic safe functions are the basic functions of section 3, their arguments being all safe.

Operations mentioned above are:

- safe composition: Suppose g: $(\mathbb{K}^*)^2 \to \mathbb{K}^*$, $h_1 : \mathbb{K}^* \to \mathbb{K}^*$ and $h_2 : (\mathbb{K}^*)^2 \to \mathbb{K}^*$ are given functions. Then the composition of g with h_1 and h_2 is the function $f : (\mathbb{K}^*)^2 \to \mathbb{K}^*$:

$$f(\overline{x};\overline{y}) = g(h_1(\overline{x}); h_2(\overline{x};\overline{y}))$$

  Note: it is possible to move an argument from the normal position to the safe position, whereas the reverse is forbidden. Suppose $g : (\mathbb{K}^*)^3 \to \mathbb{K}^*$ is a given function. One can then define with safe composition a function f:

$$f(\overline{x},\overline{y};\overline{z}) = g(\overline{x};\overline{y},\overline{z})$$

  but a definition like the following is not valid:

$$f(\overline{x};\overline{y},\overline{z}) = g(\overline{x},\overline{y};\overline{z})$$

- safe recursion: Suppose $f_1, \ldots, f_k : (\mathbb{K}^*)^2 \to \mathbb{K}^*$ and $g_1, \ldots, g_k : (\mathbb{K}^*)^{k+3} \to \mathbb{K}^*$ are given functions. Functions $h_1, \ldots, h_k : (\mathbb{K}^*)^3 \to \mathbb{K}^*$ can then be defined by safe recursion:

$$h_1(\emptyset, \overline{x}; \overline{y}), \ldots, h_k(\emptyset, \overline{x}; \overline{y}) = f_1(\overline{x};\overline{y}), \ldots, f_k(\overline{x};\overline{y})$$

$$h_1(a.\overline{z},\overline{x};\overline{y}) = \begin{cases} g_1(\overline{z},\overline{x};h_1(\overline{z},\overline{x};\overline{y}),\ldots,h_k(\overline{z},\overline{x};\overline{y}),\overline{y}) & \text{if } \forall \imath\ h_\imath(\overline{z},\overline{x};\overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$h_k(a.\overline{z},\overline{x};\overline{y}) = \begin{cases} g_k(\overline{z},\overline{x};h_1(\overline{z},\overline{x};\overline{y}),\ldots,h_k(\overline{z},\overline{x};\overline{y}),\overline{y}) & \text{if } \forall \imath\ h_\imath(\overline{z},\overline{x};\overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Note 4. The operation of primitive recursion previously defined is a simple recursion, whereas the operation of safe recursion is a simultaneous recursion, in the sense that k different functions are defined simultaneously. As stated by Proposition 2, it is possible, while using only primitive recursive functions, to simulate a simultaneous recursion definition, whereas this does not seem to be true with safe recursion. As shown in the simulation of a BSS machine by safe recursive functions, we need to have a simultaneous recursion able to define simultaneously three functions in order to prove our Main Theorem.

5 Proof of the Main Theorem

For the sake of readability, let us recall the result we want to prove here:

Main Theorem: *Over any structure* $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, 0, 1)$*, the set of safe recursive functions over* $\mathcal{K}$ *is exactly the set of functions computed in polynomial time by a BSS machine over* $\mathcal{K}$*.*

We prove this Main Theorem in two steps. First, we prove that a safe recursive function can be computed by a BSS machine in polynomial time. Second, we prove that all functions computable in polynomial time by a BSS machine over $\mathbb{K}$ can be expressed as safe recursive functions.

5.1 Polynomial time evaluation of a safe recursive function

This is proved by induction on the depth of the definition tree of our safe recursive function. Let f be a safe recursive function:

- If f is a basic safe function, the result is straightforward.
- If g, h_1, h_2 are safe recursive functions computed in polynomial time by a BSS machine (induction hypothesis), and if $f(\overline{x};\overline{y}) = g(h_1(\overline{x});h_2(\overline{x};\overline{y}))$, it is easy to see that f can be computed in polynomial time.
- The non-trivial case is the case of a function f defined with safe recursion. In order to simplify the notations, we write

  $$f(\emptyset,\overline{x};\overline{y}) = h(\overline{x};\overline{y})$$

  instead of

  $$f_1(\emptyset,\overline{x};\overline{y}),\ldots,f_k(\emptyset,\overline{x};\overline{y}) = h_1(\overline{x};\overline{y}),\ldots,h_k(\overline{x};\overline{y})$$

and

$$f(a.\overline{z}, \overline{x}; \overline{y}) = g(\overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y})$$

instead of

$$f_1(a.\overline{z}, \overline{x}; \overline{y}) = \begin{cases} g_1(\overline{z}, \overline{x}; f_1(\overline{z}, \overline{x}; \overline{y}), \ldots, f_k(\overline{z}, \overline{x}; \overline{y}), \overline{y}) & \text{if } \forall \imath\ f_\imath(\overline{z}, \overline{x}; \overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$f_k(a.\overline{z}, \overline{x}; \overline{y}) = \begin{cases} g_k(\overline{z}, \overline{x}; f_1(\overline{z}, \overline{x}; \overline{y}), \ldots, f_k(\overline{z}, \overline{x}; \overline{y}), \overline{y}) & \text{if } \forall \imath\ f_\imath(\overline{z}, \overline{x}; \overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The induction hypothesis states that functions g and h are computed in polynomial time. We can now state the following technical result:

Lemma 1. *Let f be any function defined with safe recursion. If we suppose that $\overline{y}$ is already known (evaluated), the time needed to evaluate $f(\overline{x}; \overline{y})$ does not depend on the value of $\overline{y}$.*

Proof. $\overline{y}$ is a safe argument in the expression $f(\overline{x}; \overline{y})$; therefore, in the syntactic definition tree of $f(\overline{x}; \overline{y})$, strictly no safe recursion on the variable $\overline{y}$ appears. The depth of this syntactic tree does not depend on $\overline{y}$. □

Let us apply this lemma to our function f defined with the safe recursion operation. We get that the time needed to evaluate $f(a.\overline{z}, \overline{x}; \overline{y}) = g(\overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y})$ does not depend on $f(\overline{z}, \overline{x}; \overline{y})$. It is in fact equal to the evaluation time of $g(\overline{z}, \overline{x}; \ldots, \ldots)$, which is, by induction, polynomial. If we write $T^{\ulcorner} f(\ldots)^{\urcorner}$ the evaluation time of $f(\ldots)$, we get:

$$T^{\ulcorner} f(a.\overline{z}, \overline{x}; \overline{y})^{\urcorner} = T^{\ulcorner} g(\overline{z}, \overline{x}; \ldots, \ldots)^{\urcorner} + T^{\ulcorner} f(\overline{z}, \overline{x}; \overline{y})^{\urcorner}$$

We can reasonably suppose, without loss of generality: $T^{\ulcorner} g(a.\overline{z}, \overline{x}; \ldots, \ldots)^{\urcorner} \geq T^{\ulcorner} g(\overline{z}, \overline{x}; \ldots, \ldots)^{\urcorner}$, and then it follows:

$$T^{\ulcorner} f(a.\overline{z}, \overline{x}; \overline{y})^{\urcorner} \leq |a.\overline{z}| T^{\ulcorner} g(\overline{z}, \overline{x}; \ldots, \ldots)^{\urcorner}$$

which means that f can be evaluated in polynomial time.

□

5.2 Simulation of a polynomial time BSS machine

Let M be a BSS machine over the structure $\mathcal{K}$. In order to simplify our exposition we assume, without any loss of generality, that M has a single tape. M computes a partial function f_M from $\mathbb{K}^* = \mathbb{K}^*$. Moreover, we assume that M stops after $c|\overline{y}|^r$ computation steps, where $\overline{y}$ denotes the input of the machine M. Our goal is to prove that f_M can be defined as a safe recursive function.

In what follows, we represent the tape of the machine M by a couple a variables $(\overline{x}, \overline{y})$ in $(\mathbb{K}^*)^2$ such that the non-empty part is given by $\overline{x}^R.\overline{y}$, where

$\overline{x}^R$ is the reversed word of $\overline{x}$, and that the head of the machine is on the first letter of $\overline{y}$.

We also assume that the nodes in M are numbered with natural numbers, node 0 being the initial node and node 1 the terminal node. In the following definitions, node number q will be coded by the variable (word) 0^q of length q.

Let q ($q \in \mathbb{N}$) be a move node. Three functions are associated with this node:

$$\begin{array}{ll} \mathcal{G}_i(;\overline{x},\overline{y}) & = 0^{q'} \\ \mathcal{H}_i(;\overline{x},\overline{y}) & = tl(;\overline{x}) \qquad \text{or } hd(;\overline{y}).\overline{x} \\ \mathcal{I}_i(;\overline{x},\overline{y}) & = hd(;\overline{x}).\overline{y} \ \text{ or } tl(;\overline{y}) \end{array}$$

according if one moves right or left.

Function $\mathcal{G}_i$ returns the encoding of the following node in the computation tree of M, function $\mathcal{H}_i$ returns the encoding of the left part of the tape, and function $\mathcal{I}_i$ returns the encoding of the right part of the tape.

Let q ($q \in \mathbb{N}$) be a node associated to some operation op of arity n of the structure. We also write Op for the corresponding basic operation.

Functions $\mathcal{G}_i$, $\mathcal{H}_i$, $\mathcal{I}_i$ associated with this node are now defined as follows:

$$\begin{array}{ll} \mathcal{G}_i(;\overline{x},\overline{y}) & = 0^{q'} \\ \mathcal{H}_i(;\overline{x},\overline{y}) & = \overline{x} \\ \mathcal{I}_i(;\overline{x},\overline{y}) & = cons(;Op(;hd(;\overline{y}),\ldots,hd(;tl^{(n-1)}(;\overline{y}))),tl^{(n)}(;\overline{y})) \end{array}$$

Let q ($q \in \mathbb{N}$) be a node corresponding to a relation rel of arity n of the structure.

The three functions associated with this node are now:

$$\begin{array}{l} \mathcal{G}_i(;\overline{x},\overline{y}) = C(;rel(;hd(;\overline{x}),\ldots,hd(;tl^{(n-1)}(;\overline{y}))),0^{q'},0^{q''}) \\ \mathcal{H}_i(;\overline{x},\overline{y}) = \overline{x} \\ \mathcal{I}_i(;\overline{x},\overline{y}) = \overline{y} \end{array}$$

It is easy to show that one can define without safe recursion a function $Equal_k$ for any integer k such that:

$$Equal_k(;\overline{x}) = \begin{cases} 1 \text{ if } \overline{x} = 0^k \\ 0 \text{ otherwise} \end{cases}$$

We can now define the safe recursive functions $next_{state}$, $next_{left}$ and $next_{right}$ which, given the encoding of a state and of the tape of the machine, return the encoding of the next state in the computation tree, the encoding of the left part of the tape and the encoding of the right part of the tape:

$$\begin{array}{ll} next_{state}(;\overline{s},\overline{x},\overline{y}) & = C\,(;Equal_0(;\overline{s}),\mathcal{G}_0(;\overline{s},\overline{x},\overline{y}),C\,(;Equal_2(;\overline{s}),\mathcal{G}_2(;\overline{s},\overline{x},\overline{y}), \\ & \quad \ldots C\,(;Equal_{m+n+2}(;\overline{s}),\mathcal{G}_{m+n+2}(;\overline{s},\overline{x},\overline{y}),\emptyset)\ldots)) \\ next_{left}(;\overline{s},\overline{x},\overline{y}) & = C\,(;Equal_0(;\overline{s}),\mathcal{H}_0(;\overline{s},\overline{x},\overline{y}),C\,(;Equal_2(\overline{s}),\mathcal{H}_2(;\overline{s},\overline{x},\overline{y}), \\ & \quad \ldots C\,(;Equal_{m+n+2}(;\overline{s}),\mathcal{H}_{m+n+2}(;\overline{s},\overline{x},\overline{y}),\emptyset)\ldots)) \\ next_{right}(;\overline{s},\overline{x},\overline{y}) & = C\,(;Equal_0(;\overline{s}),\mathcal{I}_0(;\overline{s},\overline{x},\overline{y}),C\,(;Equal_2(;\overline{s}),\mathcal{I}_2(;\overline{s},\overline{x},\overline{y}), \\ & \quad \ldots C\,(;Equal_{m+n+2}(;\overline{s}),\mathcal{I}_{m+n+2}(;\overline{s},\overline{x},\overline{y}),\emptyset)\ldots)) \end{array}$$

From now on, we define with safe recursion the encoding of the state of the machine reached after k computation nodes, where k is encoded by the word $0^k \in \mathbb{K}^*$, and we also define the encoding of the left part and the right part of the tape. All this is done with functions $comp_{state}$, $comp_{left}$ and $comp_{right}$ as follows:

$$\begin{aligned}
comp_{state}(\emptyset;\overline{x},\overline{y}) &= \emptyset \\
comp_{state}(0^{k+1};\overline{x},\overline{y}) &= next_{state}\big(;comp_{state}(0^k;\overline{x},\overline{y}), comp_{left}(0^k;\overline{x},\overline{y}), \\
&\qquad comp_{right}(0^k;\overline{x},\overline{y})\big) \\
comp_{left}(\emptyset;\overline{x},\overline{y}) &= \overline{x} \\
comp_{left}(0^{k+1};\overline{x},\overline{y}) &= next_{left}\big(;comp_{state}(0^k;\overline{x},\overline{y}), comp_{left}(0^k;\overline{x},\overline{y}), \\
&\qquad comp_{right}(0^k;\overline{x},\overline{y})\big) \\
comp_{right}(\emptyset;\overline{x},\overline{y}) &= \overline{y} \\
comp_{right}(0^{k+1};\overline{x},\overline{y}) &= next_{right}\big(;comp_{state}(0^k;\overline{x},\overline{y}), comp_{left}(0^k;\overline{x},\overline{y}), \\
&\qquad comp_{right}(0^k;\overline{x},\overline{y})\big)
\end{aligned}$$

In order to simplify the notations, we write the above as follows:

$$\begin{aligned}
comp(\emptyset;\overline{x},\overline{y}) &= \emptyset \\
comp(0^{k+1};\overline{x},\overline{y}) &= next\big(;comp(0^k;\overline{x},\overline{y})\big)
\end{aligned}$$

On input $\overline{y}$, with the head originally on the first letter of $\overline{y}$, the final state of the computation of M is then reached after t computation steps, where

$$t = c|\overline{y}|^r$$

The reachability of this final state is given by the following lemma:

Lemma 2. *For any $c, d \in \mathbb{N}$, one can write a function $comp_{d,c}$, where $comp_{d,c}(\overline{y})$ gives the encoding of the state and of the tape reached after $c|\overline{y}|^d$ steps of computation.*

The encoding of the tape at the end of computation is then given by $comp_{d,c_{left}}(\overline{y})$ and $comp_{d,c_{right}}(\overline{y})$, ending here our simulation of the BSS machine M.

□

6 Conclusion

In this paper, we presented a recursion-theoric characterization of computable functions and of polynomial time computable functions. Our characterization of recursive functions extends the one from Blum Shub and Smale in [BSS89] from real to arbitrary structures. Our characterization of functions computable in polynomial time extends the characterization from Bellantoni and Cook in [BC92] from the classical settings to arbitrary structures.

We think that this gives a nice setting to define complexity classes over arbitrary structures. Indeed, previous characterizations were either based on machine view, or based on generalization of results of finite model theory, also called descriptive complexity, to structures over the real numbers.

Characterizations based on machines require, like in the classical setting, rather intricate definitions of machines, and assumptions about the equivalence of models, which need to be proved or understood before talking about complexity classes. If this is done and well understood in the classical settings, this is not so clear when talking about machines over arbitrary structures, where for example the usual encoding considerations used in the classical settings are not always possible. Our characterization permits to avoid this details and yields a machine independent view of complexity over arbitrary structures.

Finite model theory presents in the classical setting a nice alternative. The generalization of the classical setting to arbitrary structures have been studied [BCSS98]. However, it requires rather unnatural considerations about types of functions, some of them termed "number terms", other termed "index term", in order to be able to use finiteness considerations over the models even in presence of infinite underlying domains like the field of real numbers. Our characterization does not suffer from the same problems.

Further work includes understanding how to translate other implicit complexity characterizations of complexity classes from the classical setting to arbitrary structures (one difficulty is that there is no valid notion of space complexity over arbitrary structures: see [Mic89]), or if transfer theorems from classical results to that setting may exist.

References

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer Verlag, 1998.

[BSS89] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: Np-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.

[Jon97] Neil Jones. Computability and complexity, from a programming perspective. MIT Press, 1997.

[Jon99] Neil Jones. Logspace and ptime characterized by programming languages. *Theoretical Computer Science*, 228:151–174, 1999.

[Mic89] Christian Michaux. Une remarque à propos des machines sur $\mathbb{R}$ introduites par Blum, Shub et Smale. In *C. R. Acad. Sc. de Paris*, volume 309 of *1*, pages 435–437. 1989.

[Poi95] Bruno Poizat. *Les petits cailloux*. aléas, 1995.