Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 90 (2003) 37–44
www.elsevier.com/locate/entcs

A Refinement of the μ-measure for Stack Programs

Emanuele Covino [1,2]

Dipartimento di Informatica
Universitá di Bari
Bari, Italy

Giovanni Pani[3]

Dipartimento di Informatica
Universitá di Bari
Bari, Italy

Abstract

We analyze the complexity of a programming language operating on stacks, introducing a syntactical measure σ such that to each program P a natural number $\sigma(\mathsf{P})$ is assigned; the measure considers the influence on the complexity of programs of nesting loops and, simultaneously, of sequences of non-size-increasing subprograms. We prove that functions computed by stack programs of σ measure n have a length bound $b \in \mathcal{E}^{n+2}$ (the $n+2$-th Grzegorczyk class), that is $|f(\vec{w})| \leq b(|\vec{w}|)$. This result represents an improvement with respect to the bound obtained via the μ-measure presented in [6].

Keywords: Refinement, μ-measure, stack programs

1 Introduction

In the recent years a number of characterizations of complexity classes has been given (among them, Linspace and Logspace in [5], functions computable

1 We would like to thank the referees for their suggestions and comments to the preliminary drafts of this paper.

2 Email: covino@di.uniba.it
3 Email: pani@di.uniba.it

doi:10.1016/S1571-0661(03)00006-9

within polynomial time in [1], those computable in polynomial space in [8] and [10], the elementary functions in [10], non-size-increasing computations in [2]), showing that all these classes can be captured by means of ramified recursion, without any explicitly bounded scheme of recursion. A different approach can be found in [3], [4], [6], and [7]; starting from the analysis of the syntax of simple programming languages, their properties are investigated, such as complexity, resource utilization, termination.

The properties of a stack programming language over a fixed alphabet are studied in [6]; this language supports loops over stacks, conditionals and concatenation, besides the usual pop and push operations (see section 2 for the detailed semantics). The natural concept of μ-measure is then introduced; it is a syntactical method by which one is able to assign to each program P a number $\mu(\mathsf{P})$. It is proved the following *bounding theorem*: functions computed by stack programs of μ measure n have a length bound $b \in \mathcal{E}^{n+2}$ (the $n+2$-th Grzegorczyk class), that is $|f(\vec{w})| \leq b(|\vec{w}|)$; as a consequence, stack programs of measure 0 have polynomial running time, and programs of measure n compute functions in the $n+2$-th finite level of the Grzegorczyk hierarchy. This result provides a measure of the impact of nesting loops on computational complexity; as reported in [6], this happens because a stack Z is updated into a loop controlled by a stack Y and, afterwards, Y is updated into a loop controlled by Z; this kind of circular reference between stacks is called *top circle*, and when it occurs into an external loop, a blow up in the complexity of the program is produced. The μ-measure is a syntactical way to detect top circles; each time one of them appears in the body of a loop, the μ measure is increased by 1 (see below, definition 2.1).

There are numerous ways of improving measure μ (see [7]), since it is an undecidable problem whether or not a function computed by a given stack program lies in a given complexity class. In this paper we provide a refinement of μ, starting from the following consideration: a program whose structure leads the μ-measure to be equal to n contains n nested top circles, and this implies that, by the bounding theorem, the program has a length bound $b \in \mathcal{E}^{n+2}$. Suppose now that some of the sequences of pop and push (or, in general, some of the subprograms) iterated into the main program leave unchanged the overall space used; since not increasing programs can be iterated without leading to any growth in space, the effective space bound is lower than the bound obtained via the μ measure, and it can be evaluated by an alternative measure σ. While μ grows each time a top circle appears in the body of a loop, σ grows only for *increasing* top circles. In other words, the new measure doesn't take into consideration all the situations in which a number of operations are performed, and their overall balance is negative. We prove a new bounding theorem using the σ-measure, providing a more appropriate bound to the complexity of stacks programs.

2 Stack programs, μ-measure, and Kristiansen and Niggl's result

In this section we briefly recall the main results presented in [6]; the reader is referred to the same paper for the complete set of definitions and proofs, and to [11] for basic facts about the Grzegorczyk hierarchy. We recall that the *principal functions* $E_1, E_2, E_3, \ldots$ are defined by $E_1(x) = x^2+2$ and $E_{n+2}(x) = E^x_{n+1}(2)$ (the x-th iterate of E_{n+1}); and that the n-th Grzegorczyk class $\mathcal{E}^n$ is the least class of functions containing the initial functions zero, successor, projections, maximum and E_{n-1}, and closed under composition and bounded recursion.

Stack programs operate on variables serving as stacks; they contain arbitrary words over a fixed alphabet Σ, and are manipulated by running a program built from imperatives (push(a,X), pop(X), and nil(X)) by sequencing, conditional and loop statements (respectively, P;Q, if top(X)≡a then [P], foreach X [P]). The notation $\{A\}\mathsf{P}\{B\}$ means that if the condition expressed by the sentence A holds before the execution of P, then the condition expressed by the sentence B holds after the execution of P. The intuitive operational semantics of stack programs have the following definition:

- (i) push(a,X) pushes a letter a on the top of the stack X;
- (ii) pop(X) removes the top of X, if any; it leaves X unchanged, otherwise;
- (iii) nil(X) empties the stack X;
- (iv) if top(X)≡a [P] executes P if the top of X is equal to a;
- (v) $\mathsf{P}_1;\ldots;\mathsf{P}_k$ are executed from left to right;
- (vi) foreach X [P] executes P for |X| times.

The only syntactical restriction is that no imperatives over X may occur in the body of a loop foreach X [P] (i.e., in P), and that the loop is executed call-by-value; X is the *control stack* of the loop. A stack program P computes a function $f : (\Sigma^*)^n \to (\Sigma^*)$ if P has an output variable O and n input variables $\bar{\mathsf{X}} = \mathsf{X}_{i_1}, \ldots, \mathsf{X}_{i_n}$ among stacks $\mathsf{X}_1, \ldots, \mathsf{X}_m$ such that $\{\bar{\mathsf{X}} = \vec{w}\}\mathsf{P}\{\mathsf{O} = f(\vec{w})\}$, for all $\vec{w} = w_1, \ldots, w_n \in (\Sigma^*)^n$. For a fixed program P, two sets of variables are considered in [6]:

$$\mathcal{U}(\mathsf{P}) = \{\mathsf{X}|\mathsf{P} \text{ contains an imperative } \mathsf{push(a,X)}\}$$
$$\mathcal{C}(\mathsf{P}) = \{\mathsf{X}|\mathsf{P} \text{ contains a loop } \mathsf{foreach\ X\ [Q]}, \text{ and } \mathcal{U}(\mathsf{Q}) \neq \emptyset\}.$$

Informally, X belongs to $\mathcal{U}(\mathsf{P})$ if it can be changed by a push during a run of P, while X is in $\mathcal{C}(\mathsf{P})$ if it controls a loop in P. The two sets are not disjoint. X *controls* Y in the program P (denoted with $\mathsf{X} \prec_{\mathsf{P}} \mathsf{Y}$) if P contains a loop foreach X [Q], with $\mathsf{Y} \in \mathcal{U}(\mathsf{Q})$; the transitive closure of $\prec_{\mathsf{P}}$ is denoted by $\stackrel{\mathsf{P}}{\rightarrow}$. Starting from this relation, a measure over the set of stack programs is introduced.

Definition 2.1 Let P be a stack program. The *μ-measure of* P (denoted with $\mu(\mathsf{P})$) is defined as follows, inductively:

(i) $\mu(\mathsf{pop}) = \mu(\mathsf{push}) = \mu(\mathsf{nil}) := 0$;

(ii) $\mu(\text{if top(X)}\equiv\text{a [Q]}) := \mu(\mathsf{Q})$;

(iii) $\mu(\mathsf{P};\mathsf{Q}) := \max(\mu(\mathsf{P});\mu(\mathsf{Q}))$;

(iv) $\mu(\text{foreach X [Q]}) := \mu(\mathsf{Q})+1$, if Q is a sequence $\mathsf{Q}_1;\ldots;\mathsf{Q}_l$ with a *top circle* (that is, if there exists Q_i such that $\mu(\mathsf{Q}_i) = \mu(\mathsf{Q})$, some Y controls some Z in Q_i, and Z controls Y in $\mathsf{Q}_1;\ldots;\mathsf{Q}_{i-1};\mathsf{Q}_{i+1};\ldots;\mathsf{Q}_l$); $\mu(\text{foreach X [Q]}) := \mu(\mathsf{Q})$, otherwise.

The core of [6] is the bounding theorem for stack programs.

Lemma 2.2 *Every function f computed by a stack program of μ-measure n has length bound $b \in \mathcal{E}^{n+2}$ satisfying $|f(\vec{w})| \leq b(|\vec{w}|)$, for all $\vec{w}$. In particular, if P computes a function f, and $\mu(P) = 0$, then f has a polynomial length bound, that is, there exists a polynomial p satisfying $|f(\vec{w})| \leq p(|\vec{w}|)$.*

Let $\mathcal{L}^n$ be the class of all functions which can be computed by a stack program of μ-measure $n \geq 0$, and let $\mathcal{G}^n$ be the class of all functions which can be computed by a Turing machine in time $b(|\vec{w}|)$, for some $b \in \mathcal{E}^n$. As a consequence of the bounding lemma, the following result is proved.

Theorem 2.3 For $n \geq 0$: $\mathcal{L}^n = \mathcal{G}^{n+2}$.

3 The σ measure

In the rest of the paper, we denote with imp(Y) an imperative pop(Y), push(a,Y), or nil(Y); we denote with $\mathsf{mod}(\bar{\mathsf{X}})$ a *modifier*, that is a sequence of imperatives operating on the variables occurring in $\bar{\mathsf{X}}$. We introduce a slight modified definition of *circle* which better matches our new measure.

Definition 3.1 Let Q be a sequence in the form $\mathsf{Q}_1;\ldots;\mathsf{Q}_l$. There is a *circle* in Q if there exists a sequence of variables $\mathsf{Z}_1,\mathsf{Z}_2,\ldots,\mathsf{Z}_l$, and a permutation π of $\{1,\ldots,l\}$ such that $\mathsf{Z}_1 \stackrel{\mathsf{Q}_{\pi(1)}}{\rightarrow} \mathsf{Z}_2 \stackrel{\mathsf{Q}_{\pi(2)}}{\rightarrow} \ldots \mathsf{Z}_l \stackrel{\mathsf{Q}_{\pi(l)}}{\rightarrow} \mathsf{Z}_1$. We say that the subprograms $\mathsf{Q}_1,\ldots,\mathsf{Q}_l$ and the variables $\mathsf{Z}_1,\ldots,\mathsf{Z}_l$ are *involved* in the circle.

For sake of simplicity, we will consider $\pi(i) = i$, that is the case $\mathsf{Z}_1 \stackrel{\mathsf{Q}_1}{\to} \mathsf{Z}_2 \stackrel{\mathsf{Q}_2}{\to} \dots \mathsf{Z}_l \stackrel{\mathsf{Q}_l}{\to} \mathsf{Z}_1$; proofs and definitions holds in the general case too.

Definition 3.2 Let P be a stack program and Y a fixed variable. The σ-*measure of* P *with respect to* Y (denoted with $\sigma_{\mathsf{Y}}(\mathsf{P})$) is defined as follows, inductively (with $sg(z) = 1$ if $z \geq 1$, $sg(z) = 0$ otherwise):

(i) $\sigma_{\mathsf{Y}}(\mathsf{mod}(\bar{\mathsf{X}})) := sg(\sum \hat{\sigma}_{\mathsf{Y}}(\mathsf{imp}(\mathsf{Y})))$, for each $\mathsf{imp}(\mathsf{Y}) \in \mathsf{mod}(\bar{\mathsf{X}})$, where

$\hat{\sigma}_{\mathsf{Y}}(\mathsf{push(a,Y)}) := 1$;
$\hat{\sigma}_{\mathsf{Y}}(\mathsf{pop(Y)}) := -1$;
$\hat{\sigma}_{\mathsf{Y}}(\mathsf{nil(Y)}) := -\infty$;
$\hat{\sigma}_{\mathsf{Y}}(\mathsf{imp(X)}) := 0$, with $\mathsf{Y} \neq \mathsf{X}$;

(ii) $\sigma_{\mathsf{Y}}(\mathsf{if\ top\ Z} \equiv \mathsf{a}\ [\mathsf{P}]) := \sigma_{\mathsf{Y}}(\mathsf{P})$;

(iii) $\sigma_{\mathsf{Y}}(\mathsf{P}_1;\mathsf{P}_2) := \max(\sigma_{\mathsf{Y}}(\mathsf{P}_1), \sigma_{\mathsf{Y}}(\mathsf{P}_2))$, with $\mathsf{P}_1;\mathsf{P}_2$ not a modifier;

(iv) $\sigma_{\mathsf{Y}}(\mathsf{foreach\ X}\ [\mathsf{Q}]) := \sigma_{\mathsf{Y}}(\mathsf{Q}) + 1$, if there exists a circle in Q, and a subprogram Q_i s.t.
(a) Y and Q_i are involved in the circle;
(b) $\sigma_{\mathsf{Y}}(\mathsf{Q}) = \sigma_{\mathsf{Y}}(\mathsf{Q}_i)$;
(c) the circle is increasing;
$\sigma_{\mathsf{Y}}(\mathsf{foreach\ X}\ [\mathsf{Q}]) := \sigma_{\mathsf{Y}}(\mathsf{Q})$, otherwise

where the circle is *not increasing* if, denoted with $\mathsf{Q}_1, \mathsf{Q}_2, \dots, \mathsf{Q}_l$ and with $\mathsf{Z}_1, \mathsf{Z}_2, \dots, \mathsf{Z}_l$ the sequences of subprograms and, respectively, of variables involved in the circle, we have that $\sigma_{\mathsf{Z}_i}(\mathsf{Q}_j) = 0$, for each $i := 1 \dots l$ and $j := 1 \dots l$. If the previous condition doesn't hold, we say that the circle is *increasing*.

Note that the σ_{Y}-measure of a modifier (see (i) in the previous definition) is equal to 1 only when, in absence of nil's, the overall number of push's over Y is greater than the number of pop's over the same variable, that is, only when a growth in the length of Y is produced. Moreover, note that the "otherwise" case in (iv) can be split in three different cases. First, there are no circles in which Y is involved. Second, Y is involved, together with a subprogram Q_i, in a circle in Q, but it happens that $\sigma_{\mathsf{Y}}(\mathsf{Q}_i)$ is lower than $\sigma_{\mathsf{Y}}(\mathsf{Q})$ (this means that there is a blow-up in the complexity of Y in $\sigma_{\mathsf{Y}}(\mathsf{Q}_i)$, but this growth is in any case bounded by the complexity of Y in a different subprogram of Q). Third, suppose that Y is involved in some circles in Q, but each of them is not increasing (that is, according to the previous definition, each variable Z_i involved in each circle doesn't produce a growth in the complexity of the subprograms Q_j involved in the same circle). This implies that the space consumed during the execution of the external loop foreach X [Q] is basically the same used by Q (this is not a surprising fact: one can freely iterate a not increasing program without leading an harmful growth). In all three cases the σ-measure must remain unchanged: it is increased when a top circle occurs and when, simultaneously, at least one of the variables involved in that circle causes a growth in the space complexity of the related subprogram (that is, if there exists a p such that $\sigma_{\mathsf{Z}_p}(\mathsf{Q}_p) > 0$). In the following definition, we extend the measure to the whole set of variables occurring in a stack program.

Definition 3.3 Let P be a stack program. The *σ-measure of* P (denoted with $\sigma(\mathsf{P})$) is defined as follows:

(i) $\tilde{\sigma}(\mathsf{mod}(\bar{\mathsf{X}})) := 0$

(ii) $\tilde{\sigma}(\mathsf{if\ top\ Z} \equiv \mathsf{a}\ [\mathsf{Q}]) := \max(\sigma_{\mathsf{Y}}(\mathsf{if\ top\ Z} \equiv \mathsf{a}\ [\mathsf{Q}]))$, for all Y occurring in P;

(iii) $\tilde{\sigma}(\mathsf{P}_1;\mathsf{P}_2) := \max(\sigma_{\mathsf{Y}}(\mathsf{P}_1;\mathsf{P}_2))$, for all Y occurring in P, with $\mathsf{P}_1;\mathsf{P}_2$ not a modifier;

(iv) $\tilde{\sigma}(\mathsf{foreach\ X}\ [\mathsf{Q}]) := \max(\sigma_{\mathsf{Y}}(\mathsf{foreach\ X}\ [\mathsf{Q}]))$, for all Y occurring in P.

(v) $\sigma(\mathsf{P}) := \tilde{\sigma}(\mathsf{P}) \dot{-} 1$

where $\dot{-}$ is the usual cut-off subtraction.

Note that $\sigma(\mathsf{P}) \leq \mu(\mathsf{P})$, for each stack program P. Note also that, roughly speaking, we use $\hat{\sigma}_{\mathsf{Y}}$ to detect all the *increasing* modifiers, for a given variable Y (this is done setting $\hat{\sigma}_{\mathsf{Y}}$ equal to 1); but, once detected, we don't have to consider them in the evaluation of the σ-measure; this is the reason of the "$\dot{-}1$" part in the previous definition. In the rest of the paper we introduce a reduction procedure $\rightsquigarrow$ between stack programs, and we prove a new bounding theorem.

Definition 3.4 P and Q are *space equivalent* if $\{\bar{\mathsf{X}} = \vec{w}\}\mathsf{P}\{|\bar{\mathsf{X}}| = m\}$ implies that $\{\bar{\mathsf{X}} = \vec{w}\}\mathsf{Q}\{|\bar{\mathsf{X}}| = O(m)\}$. This is denoted with $\mathsf{P} \approx_s \mathsf{Q}$.

Definition of $\rightsquigarrow$: let A be a stack program such that $\mu(\mathsf{A}) = n + 1$, and $\sigma(\mathsf{A}) = m$, with $m < n+1$; the program $\rightsquigarrow\mathsf{A}$ is obtained in the following way:

(i) if A=foreach X [R], with $\mu(\mathsf{R}) = \sigma(\mathsf{R}) = n$, and denoted with $C_1, \ldots, C_l$ the top circles in R, and with $\mathsf{A}_{i1}, \ldots, \mathsf{A}_{ip}$ the variables involved in C_i, for each i, we have that $\rightsquigarrow\mathsf{A}$ is the result of changing each $\mathsf{imp}(\mathsf{A}_{ij})$ into $\mathsf{nop}(\mathsf{A}_{ij})$ (a *no-operation* imperative);

(ii) if A=foreach X [R], with $\mu(\mathsf{R}) > \sigma(\mathsf{R})$, , we have that $\rightsquigarrow\mathsf{A}$ is equal to foreach X [$\rightsquigarrow$R];

(iii) if $\mathsf{A}=\mathsf{A}_1;\mathsf{A}_2$ and $\max(\mu(\mathsf{A}_1), \mu(\mathsf{A}_2)) = \mu(\mathsf{A}_1)$, we have that $\rightsquigarrow\mathsf{A}= \rightsquigarrow\mathsf{A}_1;\mathsf{A}_2$; simmetrically, if $\max(\mu(\mathsf{A}_1), \mu(\mathsf{A}_2)) = \mu(\mathsf{A}_2)$, we have that $\rightsquigarrow\mathsf{A}= \mathsf{A}_1;\rightsquigarrow\mathsf{A}_2$; if $\mu(\mathsf{A}_1) = \mu(\mathsf{A}_2)$, we have that $\mathsf{A}=\rightsquigarrow\mathsf{A}_1;\rightsquigarrow\mathsf{A}_2$;

(iv) if A=if top(X)≡a [R], we have that ⇝A=if top(X)≡a [⇝R].

Lemma 3.5 *Given a stack program P, with $\mu(P) = n+1$ and $\sigma(P) = n$, there exists a stack program ⇝P such that $\mu(\rightsquigarrow P) = n$, $\sigma(\rightsquigarrow P) = n$, and P$\approx_s$⇝P.*

Proof. (by induction on n). Base. Let $\mu(\mathsf{P}) = 1$ and $\sigma(\mathsf{P}) = 0$. In the main case, P is in the form foreach X [Q], with a not-incresing circle occurring in Q. Applying ⇝ to P, we obtain a program P′ whose σ-measure is still 0, and whose μ-measure is reduced to 0, because ⇝ has broken off the circle in P that leads μ from 0 to 1 (i.e., in P′, there are no more push's on the variables involved in the circle). Note that P can decrease the length of the stacks involved in the circle, while P′ does not perform any operation in the same circle. Thus, P′ can increase its variables only by a linear factor; indeed, if $\{\bar{\mathsf{X}} = \vec{w}\}\mathsf{P}\{|\bar{\mathsf{X}}| = m\}$ we have that $\{\bar{\mathsf{X}} = \vec{w}\}\mathsf{P}'\{|\bar{\mathsf{X}}| = cm\}$, where c is a constant depending on the structure of P: thus, P$\approx_s$P′.
Step. Let $\mu(\mathsf{P}) = n+2$ and $\sigma(\mathsf{P}) = n+1$. Let P be in the form foreach X [Q], and let C be a top circle occurring in Q, with $\mu(\mathsf{Q}) = n+1$; we have two cases: (1) $\sigma(\mathsf{Q}) = n+1$, or (2) $\sigma(\mathsf{Q}) = n$.
(1) In this case C is a not-increasing circle, because it has been detected by μ, but not by σ. Applying ⇝ to P, we obtain a program P′ such that $\sigma(\mathsf{P}') = n+1$, $\mu(\mathsf{P}') = n+1$, and P$\approx_s$P′.
(2) In this case C is an increasing circle, detected by μ and σ. We have that (by the inductive hypothesis) there exists a program Q′ such that $\mu(\mathsf{Q}') = n$, $\sigma(\mathsf{Q}') = n$, and Q$\approx_s$Q′. Starting from P, we build a new program P′=foreach X [Q'] . We have that $\mu(\mathsf{P}') = \mu(\mathsf{Q}') + 1 = n+1$, $\sigma(\mathsf{P}') = \sigma(\mathsf{Q}') + 1 = n+1$, and P$\approx_s$P′ as expected.
The cases P_1;P_2;... ;P_k and if top(X)≡a [P] can be proved in a similar way. □

Theorem 3.6 *Every function f computed by a stack program P such that $\mu(P) = n$ and $\sigma(P) = m$ has a length bound $b \in \mathcal{E}^{m+2}$ satisfying $|f(\vec{w})| \leq b(|\vec{w}|)$.*

Proof. Let k be $\mu(\mathsf{P}) - \sigma(\mathsf{P})$. Then by k applications of Lemma 3.5, we obtain a sequence $\mathsf{P} =: \mathsf{P}_0, \mathsf{P}_1, \ldots, \mathsf{P}_k$ of stack programs such that for all $i < k$,

$$\mu(\mathsf{P}_{i+1}) = \mu(\mathsf{P}) - i,\ \sigma(\mathsf{P}_i) = \sigma(\mathsf{P}_{i+1}),\ \text{and } \mathsf{P}_i \approx_s \mathsf{P}_{i+1}.$$

By Kristiansen and Niggl's bounding theorem, P_k has a length bound in $\mathcal{E}^{\sigma(\mathsf{P})+2}$, and so does P by transitivity of $\approx_s$. □

References

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

[2] M. Hofmann, *The strength of non-size-increasing computations.* Principles of Programming Languages, POPL'02, Portland, Oregon, January 16-18th, 2002.

[3] N. Jones, *Program analysis for implicit computational complexity.* Third International Workshop on Implicit Computational Complexity (ICC'01), Aarhus.

[4] N. Jones, LOGSPACE *and* PTIME *characterized by programming languages.* Theoretical Computer Science 228(1999)151-174.

[5] Lars Kristiansen, *New recursion-theoretic characterizations of well known complexity classes.* Fourth International Workshop on Implicit Computational Complexity (ICC'02), Copenhagen.

[6] L. Kristiansen and K.-H. Niggl, *On the computational complexity of imperative programming languages.* Theoretical Computer Science, to appear.

[7] L. Kristiansen and K.-H. Niggl, *The garland measure and computational complexity of imperative programs.* Fifth International Workshop on Implicit Computational Complexity, (ICC '03), Ottawa.

[8] D. Leivant and J.-Y. Marion, *Ramified recurrence and computational complexity II: substitution and polyspace*, in J. Tiuryn and L. Pocholsky (eds), Computer Science Logic, LNCS 933(1995) 486-500.

[9] Karl-Heinz Niggl, *Control structures in programs and computational complexity.* Fourth Implicit Computational Complexity Workshop (ICC'02), Copenhagen.

[10] I. Oitavem, *New recursive characterization of the elementary functions and the functions computable in polynomial space*, Revista Matematica de la Universidad Complutense de Madrid, 10.1(1997)109-125.

[11] H. E. Rose, *Subrecursion: functions and hierarchies.* Oxford University Press, Oxford, 1984.