A Note on a Syntactical Measure of the Time-Space Complexity of Stack Programs

Emanuele Covino
Dipartimento di Informatica
Universitá degli Studi di Bari
Bari, Italy
emanuele.covino@uniba.it

***Abstract*—We introduce a programming language operating on stacks and a syntactical measure σ, such that a natural number $\sigma(\mathsf{P})$ is assigned to each program P. The measure considers how the presence of loops defined over a size-increasing (and non-size-increasing) subprogram influences the complexity of the program itself. Functions computed by a program of σ-measure n are exactly those computable by a Turing machine with running time in $\mathcal{E}^{n+2}$ (the $n+2$-th Grzegorczyk class). Programs of σ-measure 0 compute the polynomial-time computable functions. Thus, we have a syntactical characterization of functions belonging to the Grzegorczyk hierarchy; this result represents an improvement with respect to previous similar results. We then extend this approach to the definition of programs with simultaneous time and space bounds in the same hierarchy.**

***Index Terms*—Polynomial-Time Complexity; Grzegorczyk Hierarchy; Imperative Programming Languages; Stack Programs.**

I. Introduction

In this paper, we expand our earlier work [1] on the definition of a programming language operating on stacks, and a syntactical measure σ, such that functions computed by a program of σ-measure n are exactly those computable by a Turing machine with running time in the $n+2$-th Grzegorczyk class $\mathcal{E}^{n+2}$.

The definition of a class of functions with a given complexity is usually provided by introducing an explicit bound on time and/or space resources used by a Turing Machine during the computation of the functions. Other approaches capture complexity classes by means of some form of *limited recursion*; the first characterization of this type has been given by Cobham [2], who shows that the polynomial-time computable functions are exactly those that are definable by *bounded recursion on notation*, starting from a set of simple basic functions. In the recent years, a number of characterizations of complexity classes has been given, showing that they can be captured by means of various forms of *ramified recursion*, without any explicitly bounded scheme of recursion. Initiated by Simmons [3], Bellantoni and Cook [4] and Leivant [5] - [6], one can find functional characterization of linear-space/time computable functions LINSPACE and LOGSPACE [7], polynomial time [8], polynomial space [9] [10], the elementary functions [10] [11], non-size-increasing computations [12], among the others.

A different approach can be found in [13] [14] [15] [16]; more recently, in [17] [18]. The properties of *imperative* programs (such as complexity, resource utilization, termination) are now investigated by analyzing their syntax only. In particular, the properties of a programming language operating on stacks are studied in [15]; the language supports loops over stacks, conditionals and concatenation, besides the usual pop and push operations (see Section II 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 whose time complexity is 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; if a stack Z is updated into a loop controlled by a stack Y and, afterwards, Y is updated into a loop controlled by Z, we have a so called *top circle* in the program; when this circular reference 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 occurs in the body of a loop, the μ measure is increased by 1 (see below, Section III and definition 3.1).

There are various ways of improving the measure μ (for instance, see [16]), 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, by the bounding theorem, that 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 does not consider those situations in which some (potentially harmful) operations are performed, but 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.

Starting from this result, we present in this note a slight modification of the stack programs, the *non-space-increasing* programs. The only general requirement is that each not-increasing program never adds a digit to a stack without erasing another one (or more) from the same or from another stack; thus, a run of a program cannot exceed the overall length of the registers. Together with the σ-measure, this restriction allows us to characterize the set of functions which are computable by a Turing machine with time and space bounds simultaneously imposed. Indeed, we define the class of functions $\mathcal{B}^{m,n}$, that is the class of functions computable by a stack program P;Q (a run of P followed by a run of Q), with $\sigma(\mathsf{P}) = m$, $\sigma(\mathsf{Q}) = n$, and Q a not-increasing stack program. We prove that each function in $\mathcal{B}^{m,n}$ can be simulated by a Turing machine with running time in $\mathcal{E}^{n+2}$, and space not exceeding a bound in $\mathcal{E}^{m+2}$; conversely, each Turing machine with running time bounded by a function in $\mathcal{E}^{n+2}$ and, simultaneously, with space bounded by a function in $\mathcal{E}^{m+2}$ can be simulated by a stack program in $\mathcal{B}^{m,n}$. This result represents an extension to the space complexity case of Kristiansen and Niggl's result (following the problem raised in [12] and [19]), and a generalization to the finite levels of the Grzegorczyk hierarchy of our [11]. A sensible sequel of this note should be the definition of another measure, depending on σ, that should provide a way to evaluate the space complexity of the whole set of stack programs.

In Section II, we recall concepts and definitions of stack programs and of the Grzegorczyk hierarchy. In Section III, we recall the definition of μ-measure. In Section IV, we introduce the definition of the new σ-measure and the new bounding theorem. In Section V, we introduce the definition of the time/space classes and the related bounding theorems. Conclusions and future work can be found in Section VI.

II. Preliminaries on the Grzegorczyk hierarchy and on stack programs

In this section, we recall the definition of the Grzegorczyk hierarchy, and fundamentals on stack programs and their computations; the reader is referred to [15] [19] [20] [21] for complete definitions and properties.

Definition 2.1: Given a unary function f, the k-th *iterate* of f (denoted with f^k) is $f^0(x) = x$ and $f^{k+1}(x) = f(f^k(x))$.

Definition 2.2: The *principal functions* $E_1, E_2, E_3, \ldots$ are $E_1(x) = x^2 + 2$ and $E_{n+2}(x) = E^x_{n+1}(2)$ (the x-th iterate of E_{n+1}).

Definition 2.3: A function f is defined by *bounded recursion* from functions g, h, and b if for all $\vec{x}$, y one has $f(\vec{x}, 0) = g(\vec{x})$, $f(\vec{x}, y) = h(\vec{x}, y, f(\vec{x}))$, and $f(\vec{x}, y) \leq g(\vec{x}, y)$.

Definition 2.4: 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 they are manipulated by running a program built from imperatives push(a,X), pop(X), and nil(X) combined by sequencing, conditional, and loop statements (respectively, P;Q, if top(X)≡a then [P], and foreach X [P]).

Definition 2.5: The operational semantics of stack programs are defined as follows:

1) push(a,X) pushes a letter a on the top of the stack X;
2) pop(X) removes the top of X, if any; it leaves X unchanged, otherwise;
3) nil(X) empties the stack X;
4) if top(X)≡a [P] executes P if the top of the stack X is equal to a;
5) P_1;. . .;P_k are executed from left to right;
6) foreach X [P] executes P for |X| times

with the restriction 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. |X| stands for the length of the word stored in X.

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.

Definition 2.6: 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 defined: $\mathcal{U}(\mathsf{P}) = \{\mathsf{X} | \mathsf{P}$ contains an imperative push(a,X)$\}$ and $\mathcal{C}(\mathsf{P}) = \{\mathsf{X} | \mathsf{P}$ contains a loop foreach X [Q], and $\mathcal{U}(\mathsf{Q}) \neq \emptyset\}$. Variables in $\mathcal{U}(\mathsf{P})$ can be altered by a push during a run of P, while variables in $\mathcal{C}(\mathsf{P})$ control a loop occurring in P. The two sets are not disjoint.

Definition 2.7: X *controls* Y in the program P (denoted with $\mathsf{X} \prec_{\mathsf{P}} \mathsf{Y}$) if P contains a loop foreach X [Q], and $\mathsf{Y} \in \mathcal{U}(\mathsf{Q})$; the transitive closure of $\prec_{\mathsf{P}}$ is denoted by $\overset{\mathsf{P}}{\to}$.

Consider the following program:

P_1:= foreach X_1[. . . foreach X_l [push (a,Y)]]

If words $v_1 \ldots v_l, w$ are stored in $\mathsf{X}_1 \ldots \mathsf{X}_l$, Y, respectively, before P_1 is executed, then Y holds the word $w\mathsf{a}^{|v_1| \cdots |v_l|}$

after the execution of P_1. The depth of loop-nesting is a necessary condition for high computational complexity, but it is not a sufficient condition. Now, consider the following two programs:

```
P₂:= nil(Y); push(a,Y); nil(Z); push(a,Z);
     foreach X [nil(Z); foreach Y [push(a,Z); push(a,Z)];
                nil(Y); foreach Z [push(a,Y)]]

P₃:= nil(Y); push(a,Y); nil(Z);
     foreach X [
         foreach Y [push(a,Z); push(a,Z); push(a,Y)]]
```

Even if both P_2 and P_3 have nesting depth 2, if w is initially stored in X, then Z holds the word $\mathsf{a}^{2^{|w|}}$ after P_2 is executed, while $\mathsf{a}^{|w|(|w|+1)}$ is stored in Z after the execution of P_3. Thus, we see that P_3 runs in polynomial time, whereas P_2 has exponential running time. This happens because of the (control) circle contained inside the outermost loop in P_2: inside the loop governed by X, first Y *controls* Z (in that Z is updated via push(a,Z) inside a loop governed by Y), and then Z *controls* Y in the same sense. In contrast, there is no such circle in P_3. Stack programs where each body of a loop statement is circle-free compute exactly the functions computable within polynomial time, and must be separated from those programs with loops that cause a blow up in running time.

III. The μ-measure on stack programs

Starting from the previous relation $\xrightarrow{\mathsf{P}}$, a measure over the set of stack programs is introduced in [15].

Definition 3.1: Let P be a stack program. The *μ-measure of* P (denoted with μ(P)) is defined as follows, inductively:

1) $\mu(\mathsf{pop}) = \mu(\mathsf{push}) = \mu(\mathsf{nil}) := 0$;
2) $\mu(\text{if top(X)}\equiv\text{a [Q]}) := \mu(\mathsf{Q})$;
3) $\mu(\mathsf{P};\mathsf{Q}) := \max(\mu(\mathsf{P}), \mu(\mathsf{Q}))$;
4) μ(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 $\mathsf{Q}_1;\ldots;\mathsf{Q}_{i-1};\mathsf{Q}_{i+1};\ldots;\mathsf{Q}_l$); μ(foreach X [Q]) $:= \mu(\mathsf{Q})$, otherwise.

To focus on the critical case where P is a loop foreach X [Q], assume that μ(Q) is already determined. Suppose that Q is a sequence Q_1; …;Q_l, in which case μ(Q) is max(Q_1, …,Q_l). Then a blow up in running time can only occur if Q has a top circle, that is, Q has a circle with respect to a control variable Y of some component Q_i of maximal μ-measure μ(Q). In this case, μ(P) is defined as μ(Q)+1. In all other cases, μ(P) is defined as μ(Q). Given that all primitive instructions receive μ-measure 0, one easily verifies for the examples above that $\mu(\mathsf{P}_1)$=$\mu(\mathsf{P}_3)$=0, whereas $\mu(\mathsf{P}_2)$=1.

The core of [15] is the following bounding theorem.

Lemma 3.1: 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 $\mu(\mathsf{P}) = 0$, then f has a polynomial length bound, that is, there exists a polynomial p satisfying $|f(\vec{w})| \leq p(|\vec{w}|)$.

Let $\mathcal{L}^n_\mu$ 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 holds.

Theorem 3.1: For $n \geq 0$: $\mathcal{L}^n_\mu = \mathcal{G}^{n+2}$.

Proof. By mutual simulation. The "$\subseteq$" inclusion starts from an arbitrary stack program Q of μ-measure n. Each imperative program occurring in Q can be simulated on a Turing machine in time $q(|X|)$, with q a polynomial. Let $TIME_P(\vec{w})$ denote the number of steps in a run of P on $\vec{w}$, and let $\mathsf{P}^\sharp$ be the result of replacing in P each imperative imp with imp;push(a,V), for a new variable V. Then the program TIME(P): $\equiv$ nil(V);$\mathsf{P}^\sharp$ has μ measure n and is such that $\{\vec{\mathsf{X}} = \vec{w}\}\mathsf{TIME(P)}\{|\mathsf{V}| = TIME_P(\vec{w})\}$. By the bounding theorem, we obtain a length bound $b \in \mathcal{E}^{n+2}$ satisfying $\{\vec{\mathsf{X}} = \vec{w}\}\mathsf{TIME(P)}\{|\mathsf{V}| \leq b(|\vec{w}|)\}$. Hence, there exists a Turing machine which simulates P within time $q(b(|\vec{w}|)) \cdot b(|\vec{w}|)$.
The opposite "$\supseteq$" inclusion shows that a single-tape Turing machine running in time $b(|\vec{w}|)$ can be simulated by a stack program with μ-measure n, provided that $b \in \mathcal{E}^{n+2}$. The mentioned program has the following form:

```
P:≡  TIME-BOUND(Y);            μ-measure is n
     INITIALIZE(L,Z,R);        μ-measure is 0
     foreach Y [SIM-MOVES];    μ-measure is 0
     OUTPUT(R;O);              μ-measure is 0
```

In order to write the first of the four subprograms, observe that, for each positive n, one can find a sequence LE[n+1] such that μ(LE[n+1]) $= n$ and $\{\mathsf{Y} = w\}\mathsf{LE[n+1]}\{|\mathsf{Y}| = E_{n+1}(|w|)\}$ (that is, LE[n+1] computes the $n+1$-th function of the sequence of principal functions $E_1, E_2, \ldots$, for a given input w); indeed, for some new variable U, LE[n+1] can be defined by:

```
LE[n+2]:≡  nil(U); foreach Y [push(a,U)];
           nil(Y); push(a,Y); push(a;Y);
           foreach U [LE[n+1]]
```

Recalling that there exists a constant c such that $b(x) \leq E^c_{n+1}(x)$, we have

```
TIME-BOUND(Y):≡  nil(Y);
                 foreach X [push(a,Y)]
                 LE[n+1];...;LE[n+1]     (c times).
```

We have that $\{\mathsf{Y} = w\}\mathsf{TIME\text{-}BOUND(Y)}\{\mathsf{X} = w, |\mathsf{Y}| = E^c_{n+1}(|w|)\}$. We omit the details about the definition of INITIALIZE (it sets the registers to the initial configuration of the Turing machine) and OUTPUT (it returns the result of the computation in a fixed register). The program SIM-MOVES is in the form MOVE_1;…;MOVE_k, where MOVE_i simulates the i-th move of the Turing machine, operating on two stacks L and R (one for the left side of the tape, the other for the right side; see [15] for a detailed description of this simulation).

IV. The σ-measure and a new bounding theorem

In the rest of the paper, we denote with $\mathsf{imp(Y)}$ an imperative $\mathsf{pop(Y)}$, $\mathsf{push(a,Y)}$, or $\mathsf{nil(Y)}$; we denote with $\mathsf{mod}(\bar{\mathsf{X}})$ a *modifier*, that is a sequence of imperatives operating on the variables occurring in $\bar{\mathsf{X}} = \mathsf{X}_1, \dots, \mathsf{X}_n$. We introduce a modified definition of *circle*, which better matches our new measure.

Definition 4.1: Let Q be a sequence in the form $\mathsf{Q}_1; \dots; \mathsf{Q}_l$. There is a *circle* in Q if there exists a sequence of variables Z_1,Z_2,...,Z_l, and a permutation π of $\{1, \dots, l\}$ such that $\mathsf{Z}_1 \overset{\mathsf{Q}_{\pi(1)}}{\rightarrow} \mathsf{Z}_2 \overset{\mathsf{Q}_{\pi(2)}}{\rightarrow} \dots \mathsf{Z}_l \overset{\mathsf{Q}_{\pi(l)}}{\rightarrow} \mathsf{Z}_1$. The subprograms $\mathsf{Q}_1, \dots, \mathsf{Q}_l$ and the variables $\mathsf{Z}_1, \dots, \mathsf{Z}_l$ are *involved* in the circle.

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

Definition 4.2: Let P be a stack program and let Y be a given 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):

1) $\sigma_{\mathsf{Y}}(\mathsf{mod}(\bar{\mathsf{X}})) := sg(\sum \hat{\sigma}_{\mathsf{Y}}(\mathsf{imp(Y)}))$, for each $\mathsf{imp(Y)} \in \mathsf{mod}(\bar{\mathsf{X}})$, 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 $\mathsf{Y} \neq \mathsf{X}$;
2) $\sigma_{\mathsf{Y}}(\mathsf{if\ top\ Z \equiv a[P]}) := \sigma_{\mathsf{Y}}(\mathsf{P})$;
3) $\sigma_{\mathsf{Y}}(\mathsf{P}_1;\mathsf{P}_2) := \max(\sigma_{\mathsf{Y}}(\mathsf{P}_1), \sigma_{\mathsf{Y}}(\mathsf{P}_2))$, with $\mathsf{P}_1;\mathsf{P}_2$ not a modifier;
4) $\sigma_{\mathsf{Y}}(\mathsf{foreach\ X\ [Q]}) := \sigma_{\mathsf{Y}}(\mathsf{Q})+1$, 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;
   $\sigma_{\mathsf{Y}}(\mathsf{foreach\ X\ [Q]}) := \sigma_{\mathsf{Y}}(\mathsf{Q})$, otherwise,

where a circle is *not increasing* if, denoted with Q_1,Q_2,...,Q_l and with Z_1,Z_2,...,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 does not hold, we say that the circle is *increasing*.

Note that the σ_{Y}-measure of a modifier (see (1) 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 (4) 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 still bounded by the complexity of Y in a different subprogram of Q). Third, 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 does not produce a growth in the complexity of the subprograms Q_j involved in the same circle). This implies that the space used during the execution of the external loop $\mathsf{foreach\ X\ [Q]}$ 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 only when an increasing top circle occurs and when at least one of the variables involved in that circle causes a growth in the space complexity of the related subprogram, simultaneously (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 4.3: Let P be a stack program. The *σ-measure of* P is $\sigma(\mathsf{P}) := \tilde{\sigma}(\mathsf{P}) \dot{-} 1$, where $\dot{-}$ is the usual cut-off subtraction, and

1) $\tilde{\sigma}(\mathsf{mod}(\bar{\mathsf{X}})) := 0$
2) $\tilde{\sigma}(\mathsf{if\ top\ Z \equiv a\ [P]}) := \max(\sigma_{\mathsf{Y}}(\mathsf{if\ top\ Z \equiv a\ [P]}))$, for all Y occurring in P;
3) $\tilde{\sigma}(\mathsf{P}_1;\mathsf{P}_2) := \max(\sigma_{\mathsf{Y}}(\mathsf{P}_1;\mathsf{P}_2))$, for all Y occurring in P, with $\mathsf{P}_1;\mathsf{P}_2$ not a modifier;
4) $\tilde{\sigma}(\mathsf{foreach\ X\ [Q]}) := \max(\sigma_{\mathsf{Y}}(\mathsf{foreach\ X\ [Q]}))$, for all Y occurring in P.

Note that $\sigma(\mathsf{P}) \leq \mu(\mathsf{P})$, for each stack program P. Note also that we are using the previously defined $\hat{\sigma}_{\mathsf{Y}}$ 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 between stack programs, denoted with $\rightsquigarrow$, and we prove a new bounding theorem.

Definition 4.4: 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 4.5: 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 as follows:

1) if A is $\mathsf{foreach\ X\ [R]}$, with $\mu(\mathsf{R}) = \sigma(\mathsf{R}) = n$, and denoted with $C_1, \dots, C_l$ the top circles in R, and with $\mathsf{A}_{i1}, \dots, \mathsf{A}_{ip}$ 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);
2) if A is $\mathsf{foreach\ X\ [R]}$, with $\mu(\mathsf{R}) > \sigma(\mathsf{R})$, , we have that $\rightsquigarrow\mathsf{A}$ is equal to $\mathsf{foreach\ X\ [} \rightsquigarrow\mathsf{R]}$;
3) if A is A_1;A_2 and $\max(\mu(\mathsf{A}_1), \mu(\mathsf{A}_2)) = \mu(\mathsf{A}_1)$, we have that $\rightsquigarrow\mathsf{A}$ is equal to $\rightsquigarrow\mathsf{A}_1$;$\mathsf{A}_2$;
   simmetrically, if $\max(\mu(\mathsf{A}_1), \mu(\mathsf{A}_2)) = \mu(\mathsf{A}_2)$, we have that $\rightsquigarrow\mathsf{A}$ is equal to A_1;$\rightsquigarrow\mathsf{A}_2$;

if $\mu(\mathsf{A}_1) = \mu(\mathsf{A}_2)$, we have that $\rightsquigarrow\mathsf{A}$ is equal to $\rightsquigarrow\mathsf{A}_1;\rightsquigarrow\mathsf{A}_2$;

4) if A is if top(X)$\equiv$a [R], we have that $\rightsquigarrow\mathsf{A}$ is equal to if top(X)$\equiv$a [$\rightsquigarrow$R].

Lemma 4.1: Given a stack program P, with $\mu(\mathsf{P}) = n+1$ and $\sigma(\mathsf{P}) = n$, there exists a stack program $\rightsquigarrow\mathsf{P}$ such that $\mu(\rightsquigarrow \mathsf{P}) = n$, $\sigma(\rightsquigarrow \mathsf{P}) = n$, and $\mathsf{P}\approx_s\rightsquigarrow\mathsf{P}$.

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 $\rightsquigarrow$ to P, we obtain a program P$'$ whose σ-measure is still 0, and whose μ-measure is reduced to 0, because $\rightsquigarrow$ 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, $\mathsf{P}\approx_s\mathsf{P}'$.

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 $\rightsquigarrow$ to P, we obtain a program P$'$ such that $\sigma(\mathsf{P}') = n+1$, $\mu(\mathsf{P}') = n+1$, and $\mathsf{P}\approx_s\mathsf{P}'$.

(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 $\mathsf{Q}\approx_s\mathsf{Q}'$. 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 $\mathsf{P}\approx_s\mathsf{P}'$ as expected.

The cases P_1;P_2;... ;P_k and if top(X)$\equiv$a [P] can be proved in a similar way.

Theorem 4.1: Every function f computed by a stack program P such that $\mu(\mathsf{P}) = n$ and $\sigma(\mathsf{P}) = m$, with $n > m$, 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 4.1, 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 $\approx_s$.

Let $\mathcal{L}^n_\sigma$ be the class of all functions that can be computed by a stack program of σ-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 Theorem 4.1, and similarly to what has been recalled in Section III, the following result holds.

Theorem 4.2: For $n \geq 0$: $\mathcal{L}^n_\sigma = \mathcal{G}^{n+2}$.

V. Restrictions to time-space complexity

In this section, we introduce some syntactical restrictions to the stack programming language, and we prove that, when combined with the σ-measure, they allow us to evaluate the time and (simultaneously) the space complexity of a program written according to the new syntax. In particular, we define $\mathcal{B}^{m,n}$ as the class of functions computable by a stack program in the form P;Q (a run of P followed by a run of Q), where $\sigma(\mathsf{P}) = m$, $\sigma(\mathsf{Q}) = n$, and where Q is a *not-increasing* stack program (see below for the definition). As we promised in the introduction, we prove that each function in $\mathcal{B}^{m,n}$ can be simulated by a Turing machine with running time in $\mathcal{E}^{n+2}$, and space in $\mathcal{E}^{m+2}$, and, conversely, each Turing machine with running time bounded by a function in $\mathcal{E}^{n+2}$ and, simultaneously, with space bounded by a function in $\mathcal{E}^{m+2}$ can be simulated by stack programs in $\mathcal{B}^{m,n}$.

Definition 5.1:

1) *Given a modifier M and a stack X the* rate of growth *of M with respect to X is the difference between the number of push(a,X) and the number of pop(X) occurring in M.*
2) *A* non-space-increasing stack program *is built from modifiers by sequencing, conditional and loop statements, provided that the rate of growth of each modifier with respect to each stack is negative, or equal to* 0*.*

The following lemma shows that a not-increasing program cannot increase, as expected, the overall length of the registers on which it operates (except for a constant c which depends only on the structure of the modifiers occurring into the program).

Lemma 5.1: Let P be a not-increasing stack program; there exists a constant $c \geq 0$ such that

$$\{\vec{X} = \vec{w}\}P\{|\vec{X}| \leq |\vec{w}| + c\}.$$

Proof. Given the definition of not-increasing programs, it is natural to observe that they cannot add digits to their inputs; there is only one exception to this fact, that is when one or more of the stacks on which a program operates are empty (and thus, some of the pop's occurring into the program have no effect). In this case, only a constant number of digits can be added to some stack, hence the constant c of the theorem; in particular, $c = \sum_{\mathsf{X}} c_{\mathsf{X}}$, with c_{X} the maximum number of push(a,X) occurring into the modifiers of the program. For example, consider P:$\equiv$ pop(X);pop(X);push(a,X);push(a,X), with X equal to the empty word. In this case the first two pop have no effect on X, while the rest of the program pushes two a's into X, returning a value whose length is greater than the input's length, notwithstanding P is still not-increasing. If P occurs into a loop, the number of digits added at the end of the loop's run is still two, since each run of pop(X);pop(X) erases two digits from X, and each run of push(a,X);push(a,X) adds two digits to X.

The proof proceeds by induction on the structure of the program. The base case is obvious, given the definition of modifiers with negative rate of growth. As for the step, let P and Q two not-increasing programs, operating on $\vec{X}$. By the inductive hypothesis, $\{\vec{X} = \vec{w}\}P\{|\vec{X}| \leq |\vec{w}| + c_1\}$, and $\{\vec{X} = \vec{w}\}Q\{|\vec{X}| \leq |\vec{w}| + c_2\}$; the sequence P;Q is such that $\{\vec{X} = \vec{w}\}P;Q\{|\vec{X}| \leq |\vec{w}| + \max\{c_1, c_2\}\}$. The same happens for the conditional and the loop cases.

Definition 5.2: Let h and k be two natural numbers.

1) *$\mathcal{B}^{h,k}$ denotes the class of all functions computable by a stack program in the form P;Q, where $\sigma(P) = h$, $\sigma(Q) = k$, and Q is a non-space-increasing stack program.*
2) *If $k \leq h \leq k+1$, $\mathcal{G}^{h,k}$ denotes the class of functions computable by a Turing machine on input $\vec{w}$ within time $t(|\vec{w}|)$ (with $t \in \mathcal{E}^h$), and in space $s(|\vec{w}|)$ (with $s \in \mathcal{E}^k$), simultaneously.*

The "=" in the following theorem stands for the mutual inclusion between two classes of functions. In this case, between the class of functions computed by our version of stack programs and the class of functions computable by Turing machines with given time and space bounds.

Theorem 5.1: For m and n two natural numbers ($m \leq n \leq m+1$), we have $\mathcal{B}^{m,n} = \mathcal{G}^{n+2,m+2}$.

The result comes from the next two lemmas.

Lemma 5.2: For m and n two natural numbers ($m \leq n \leq m+1$), we have $\mathcal{B}^{m,n} \subseteq \mathcal{G}^{n+2,m+2}$.

Proof. Let S be a stack program in the form P;Q, with $\mu(P) = m$, $\mu(Q) = n$, and Q a not-increasing program. Let TIME$_S(\vec{w})$ denote the number of steps in a run of the program S on $\vec{w}$ (a step is an execution of an imperative), and let SPACE$_S(\vec{w})$ denote the overall number of registers' cells used by S during a run.

By theorem 3.1 one obtains a Turing machine which simulates P on input $\vec{w}$ within time bounded by $b(|\vec{w}|)$, with $b \in \mathcal{E}^{m+2}$; hence, the space used by the Turing machine simulating P is bounded by $b(|\vec{w}|)$. Let $\vec{v}$ the sequence of variables such that $\{\vec{X} = \vec{w}\}P\{\vec{X} = \vec{v}\}$; Q runs on $\vec{v}$. By theorem 3.1 there exists a Turing machine simulating Q on $\vec{v}$, in time $b'(|\vec{v}|)$, with $b' \in \mathcal{E}^{n+2}$. The overall time is TIME$_P$+TIME$_Q$ $\leq b(|\vec{w}|) + b'(|\vec{v}|) \leq b'(|\vec{v}|)$ (being $m \leq n$), with $b' \in \mathcal{E}^{n+2}$. As for the evaluation of the space complexity, note that Q is a not-incresing program, then it uses SPACE$_Q(\vec{v}) \leq |\vec{v}| + c$; hence, the overall space used by S is bounded by $b(|\vec{w}|)$, with $b \in \mathcal{E}^{m+2}$. The sequence of the two Turing machines gives us the desired result.

Lemma 5.3: For m and n two natural numbers ($m \leq n \leq m+1$), we have $\mathcal{G}^{n+2,m+2} \subseteq \mathcal{B}^{m,n}$.

Proof. Let M be an arbitrary Turing machine belonging to $\mathcal{G}^{n+2,m+2}$, that is running (on input $\vec{w}$) in time $t(|\vec{w}|)$, with $t \in \mathcal{E}^{n+2}$, and in space $s(|\vec{w}|)$, with $s \in \mathcal{E}^{m+2}$. M can be simulated by two Turing machines, M_P and M_Q, respectively time-bounded by $s(|\vec{w}|)$ and $t(|\vec{v}|)$; M_P delimits (in $s(|\vec{w}|)$ steps) the space that M_Q will use during its run. Moreover, M_Q does not exceed its input.

By theorem 3.1 there exists a program P which simulates M_P, and such that $\sigma(P) = m$; for the same reason, there exists a program Q which simulates M_Q, with $\sigma(Q) = n$. We cannot use this program in our proof, since it is an increasing program; indeed, according to the second part of theorem 3.1, Q contains a subprogram LE[n+2] which stores into Y the number of times that the SIM-MOVES related to M_Q will be executed. We are looking for an alternative procedure to define, for each n and for each input x, the appropriate sequence of $E_n(x)$ SIM-MOVES needed in order to simulate M_Q correctly. This is done by following the definition of the program LE[n+2] in [15]; the intended output of our procedure is the appropriate number of sequenced SIM-MOVES.

```
$\mathcal{L}E_{(n+2)}$ :≡   for each x do {u:=u+1; SIM-MOVES};
                 x:=2;
                 for each u do $\mathcal{L}E_{(n+1)}$.
```

The sequence of SIM-MOVES we obtain executing c times $\mathcal{L}E_{(n+2)}$ is a not-incresing stack program, since each SIM-MOVES never pushes a digit into a stack without popping another digit from the other one. The reader should note that $\mathcal{L}E$ is not a stack program, but its outputs are. Setting M :≡ SIM-MOVES;...;SIM-MOVES ($E^c_n(x)$ times), and setting Q :≡ INITIALIZE;M;OUTPUT, we have that the stack program P;Q is in $\mathcal{B}^{m,n}$, by definition 5.2.

VI. Conclusions

We have defined a syntactical measure σ that considers how the iteration of imperative stack programs affects the complexity of the programs themselves. In particular, this measure only counts those loops in which programs with a size-increasing effect (w.r.t. the final length of the result) are iterated. Each time such a loop is built over other loops, the σ-measure is increased by 1. Other measures detect potentially harmful loops, but are not able to distinguish between size-increasing and non-size-increasing loops. It is undecidable to know if a function computed by a given stack program lies in a given complexity class, but our measure represents an improvement when compared to previously defined measures. We can assign a function computed by a stack program of σ-measure n to the $n+2-th$ Grzegorczyk class, and this class is lower in the hierarchy, when compared to the class obtained via other measures. We have extended this idea to the classification of programs that computes functions with simultaneous time and space bounds in the same hierarchy.

References

[1] E. Covino, "A Note on a Syntactical Measure of the Complexity of Programs," The Fifteenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking - COMPUTATION TOOLS 2024, April 14, 2024 to April 18, 2024 - Venice, Italy, ISBN: 978-1-68558-158-9, ISSN: 2308-4170.

[2] A. Cobham, "The intrinsic computational difficulty of functions," in Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, pp. 24-30, North-Holland, Amsterdam, 1962.

[3] H. Simmons, "The realm of primitive recursion," Arch. Math. Logic 27 (1988), pp. 177–188.

[4] S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the poly-time functions," Computational Complexity, no. 2, pp. 97-110, 1992.

[5] D. Leivant, "Subrecursion and lambda representation over free algebras," in S. Buss, P. Scott (Eds.), Feasible Mathematics, Perspectives in Computer Science, BirkhLauser, Boston, New York, 1990, pp. 281–291.

[6] D. Leivant, "Stratifed functional programs and computational complexity," in Conf. Record of the 20th Annual ACM Symposium on Principles of Programming Languages, New York, 1993, pp. 325–333.

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

[8] D. Leivant, "Ramifed recurrence and computational complexity I: Word recurrence and poly-time," in P. Clote, J. Remmel (Eds.), Feasible Mathematics II, Perspectives in Computer Science, BirkhLauser, Basel, 1994, pp. 320–343.

[9] 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 no. 933, pp. 486-500, 1995.

[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, no. 10.1, pp. 109-125, 1997.

[11] E. Covino and G. Pani, "Diagonalization and the complexity of programs," The Ninth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, (COMPUTATION TOOLS 2018), February 18-22, 2018, Barcelona, Spain. ISBN: 978-1-61208-394-0. ISSN: 2308-4170

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

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

[14] N. Jones, "LOGSPACE and PTIME characterized by programming languages," Theoretical Computer Science, no. 228, pp. 151-174, 1999.

[15] L. Kristiansen and K.-H. Niggl, "On the computational complexity of imperative programming languages," Theoretical Computer Science, no. 318(1-2), pp. 139–161, 2004.

[16] 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.

[17] D. Leivant, "A generic imperative language for polynomial time," arXiv:1911.04026v2 [cs.LO], 2020.

[18] D. Leivant and J.-Y. Marion, "Primitive recursion in the abstract," Mathematical Structures in Computer Science, Cambridge University Press (CUP), 2020, 30 (1), pp. 33-43. 10.1017/S0960129519000112. hal-02573188.

[19] P. Clote, "Computation models and function algebra," in E. Grivor (Ed.), Handbook of Computability Theory, Elsevier, Amsterdam, 1996.

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

[21] A. Grzegorczyk, "Some classes of recursive functions," Rozprawy Mat., Vol. IV, Warszawa, 1953.