On the complexity of programs with nested loops

E. Covino and G. Pani
Dipartimento di Informatica, Università di Bari, Bari, Italy

Abstract—Given a programming language operating on stacks, we introduce a syntactical measure \(\sigma\) such that, a natural number \(\sigma(P)\) is assigned to each program \(P\). The measure considers how the presence of loops defined over size-increasing (and/or non-size-increasing) subprograms influences the complexity of the program itself. We prove that a generic function \(f\) computed by a stack program with \(\sigma\)-measure \(n\) has length bound \(b \in \mathcal{E}^{n+2}\) (the \(n + 2\)-th Grzegorczyk class), that is \(|f(w)| \leq b(|w|)\). Thus, we have a syntactical characterization of the functions belonging to the Grzegorczyk hierarchy; this result represents an improvement with respect to previous similar results.

Keywords: Complexity classes; hierarchies of functions; stack programs

1. Introduction

The standard definition of a class of functions with a given complexity is made 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 has shown 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 all these classes can be captured by means of ramified recursion, without any explicitly bounded scheme of recursion. Among them, LINSPACE and LOGSPACE in [7], functions computable within polynomial time in [1], those computable in polynomial space in [10] and [12], the elementary functions in [12] and [3], non-size-increasing computations in [4].

A different approach can be found in [5], [6], [8], and [9]; the properties of imperative programs (such as complexity, resource utilization, termination) are investigated by analyzing their syntax only. In particular, the properties of a programming language operating on stacks are studied in [8]; the 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 \(\mu\)-measure is then introduced; it is a syntactical method by which one is able to assign to each program \(P\) a number \(\mu(P)\). It is proved the following bounding theorem: functions computed by stack programs of \(\mu\) measure \(n\) have a length bound \(b \in \mathcal{E}^{n+2}\) (the \(n + 2\)-th Grzegorczyk class), that is \(|f(w)| \leq b(|w|)\); 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; 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 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 \(\mu\)-measure is a syntactical way to detect top circles; each time one of them appears in the body of a loop, the \(\mu\) measure is increased by 1 (see below, definition 3.1).

There are various ways of improving the measure \(\mu\) (see [9]), 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 \(\mu\), starting from the following consideration: a program whose structure leads the \(\mu\)-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 \(\mu\) measure, and it can be evaluated by an alternative measure \(\sigma\). While \(\mu\) grows each time a top circle appears in the body of a loop, \(\sigma\) grows only for increasing top circles. In other words, the new measure doesn’t consider all the situations in which some operations are performed, and their overall balance is negative. We prove a new bounding theorem using the \(\sigma\)-measure, providing a more appropriate bound to the complexity of stacks programs.

2. Stack programs and the Grzegorczyk hierarchy

In this section we recall the definition of the Grzegorczyk hierarchy, and we introduce the fundamental facts about stack programs and their computations; the reader is referred to [13] and [8], respectively, for a complete set of definitions and proofs.

We recall that the principal functions \(E_1, E_2, E_3, \ldots\) are defined by \(E_1(x) = x^2 + 2\) and \(E_{n+2}(x) = E_n(x)^2 + 2\) (the \(x\)-th iterate of \(E_n(x)\)); 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 \(\Sigma\), and are manipulated by running a program built from imperatives \(\text{push}(a,X)\), \(\text{pop}(X)\), and \(\text{nil}(X)\) by sequencing, conditional and loop statements (respectively, \(P;Q\), if \(\text{top}(X)\)\(=\)\(a\) then \(P\), \(\text{foreach} X [P]\)). The notation \(\{A\}P\{B\}\) 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:

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

The only restriction required is that no imperative over \(X\) may occur in the body of a loop \(\text{foreach} X [P]\) (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 \(f : (\Sigma^*)^n \rightarrow (\Sigma^*)^l\) if \(P\) has an output variable \(O\) and \(n\) input variables \(X = X_1,\ldots,X_m\) among stacks \(X_1,\ldots,X_m\) such that \(\{X = \vec{w}\}P\{O = f(\vec{w})\}\), for all \(\vec{w} = w_1,\ldots,w_n \in (\Sigma^*)^n\).

For a fixed program \(P\), two sets of variables are defined:

\[ U(P) = \{X | P\text{ contains a nontrivial \(\text{push}(a,X)\)}\} \]
\[ C(P) = \{X | P\text{ contains a loop }X[Q]\}, \text{ and } U(Q) \neq \emptyset \}. \]

Informally, \(X\) belongs to \(U(P)\) if it can be changed by a push during a run of \(P\), while \(X\) is in \(C(P)\) if it controls a loop in \(P\). The two sets are not disjoint. \(X\) controls \(Y\) in the program \(P\) (denoted with \(X \prec_p Y\)) if \(P\) contains a loop \(X[Q]\), with \(Y \in U(Q)\); the transitive closure of \(\prec_p\) is denoted by \(\overset{*}{\prec}_p\).

3. The \(\mu\)-measure on stack programs

Starting from the previous relation \(\overset{*}{\prec}_p\), a measure over the set of stack programs is introduced in [8].

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

1) \(\mu(\text{pop}) = \mu(\text{push}) = \mu(\text{nil}) = 0\);
2) \(\mu(\text{if top}(X)\equiv a [Q]) = \mu(Q)\);
3) \(\mu(P;Q) = \max(\mu(P);\mu(Q))\);
4) \(\mu(\text{foreach} X [Q]) = \mu(Q) + 1\), if \(Q\) is a sequence \(Q_1;\ldots;Q_i\) with a top circle (that is, if there exists \(Q_i\) such that \(\mu(Q_i) = \mu(Q)\)), some \(Y\) controls some \(Z\) in \(Q_i\), and \(Z\) controls \(Y\) in \(Q_1;\ldots;Q_{i-1};Q_{i+1};\ldots;Q_l\);
5) \(\mu(\text{foreach} X [Q]) = \mu(Q)\), otherwise.

The core of [8] is the least bounding theorem.

**Lemma 3.1:** Every function \(f\) computed by a stack program of \(\mu\)-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 \(\mathcal{L}_\mu^n\) be the class of all functions which can be computed by a stack program of \(\mu\)-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 \(n \geq 0\): \(\mathcal{L}_\mu^n = \mathcal{G}^{n+2}\).

4. The \(\sigma\)-measure and a new bounding theorem

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

**Definition 4.1:** Let \(Q\) be a sequence in the form 
\[Q_1;\ldots;Q_l,\] 
there is a circle \(C\) in \(Q\) if there exists a sequence of variables \(Z_1, Z_2, \ldots, Z_l\), and a permutation \(\pi\) of \([1,\ldots,l]\) such that \(Z_1 \overset{\pi(1)}{\rightarrow} Z_2 \overset{\pi(2)}{\rightarrow} \ldots Z_l \overset{\pi(l)}{\rightarrow} Z_1\). We say that the subprograms \(Q_1,\ldots,Q_l\) and the variables \(Z_1,\ldots,Z_l\) are involved in the circle.

For sake of simplicity, we will consider \(\pi(i) = i\), that is the case \(Z_1 \overset{i}{\rightarrow} Z_2 \overset{i}{\rightarrow} \ldots Z_l \overset{i}{\rightarrow} Z_1\); 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 \(\sigma\)-measure of \(P\) with respect to \(Y\) (denoted with \(\sigma_P(Y)\)) is defined as follows, inductively (with \(sg(z) = 1\) if \(z \geq 1\), \(sg(z) = 0\) otherwise):

1) \(\sigma_P(\text{mod}(X)) := sg(\sum \sigma_P(\text{imp}(Y)))\), for each \(\text{imp}(Y) \in \text{mod}(X)\), where \(\sigma_P(\text{push}(a,Y)) := 1;\)
(\(\sigma_P(\text{pop}(Y)) := -1;\)
(\(\sigma_P(\text{nil}(Y)) := -\infty, \text{ with } Y \neq X;\)
(\(\sigma_P(\text{imp}(X)) := 0, \text{ with } Y = X;\)
2) \(\sigma_P(\text{if top Z} \equiv a[P]) := \sigma_P(P);\)
3) \(\sigma_P(P_1;P_2) := \max(\sigma_P(P_1);\sigma_P(P_2))\), with \(P_1;P_2\) not a modifier;
4) \(\sigma_P(\text{foreach X} [Q]) := \sigma_P(Q) + 1, \text{ if there exists a circle in Q, and a subprogram Q, s.t.}\)
(\(a\) Y and Q are involved in the circle;
(\(b\) \(\sigma_P(Q) = \sigma_P(Q);\)
(\(c\) the circle is increasing;
\(\sigma_P(\text{foreach X} [Q]) := \sigma_P(Q), \text{ otherwise,}\)
where the circle is not increasing if, denoted with \(Q_1, Q_2, \ldots, Q_l\) and with \(Z_1, Z_2, \ldots, Z_l\) the sequences of sub-

programs and, respectively, of variables involved in the

circle, we have that \(\sigma_j(Q_j) = 0\), for each \(i := 1 \ldots l\) and \(j := 1 \ldots l\). If the previous condition doesn’t hold, we say

that the circle is increasing.

Note that the \(\sigma\)-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 \(\sigma(Q_i)\) is lower than \(\sigma(Y)\)

(this means that there is a blow-up in the complexity of \(Y\)

in \(\sigma(Q_i)\), 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 doesn’t 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 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 these cases the \(\sigma\)-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 \(\sigma_{p}(Q_p) > 0\)).

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 \(\sigma\)-measure

of \(P\) is \(\sigma(P) := \bar{\sigma}(P) - 1\), where \(\bar{\sigma}\) is the usual cut-off

subtraction, and

1) \(\bar{\sigma}(\text{mod}(X)) := 0\)

2) \(\bar{\sigma}(\text{fop top Z :=a (Q)})) := \max(\sigma_{\text{fop top Z :=a (Q)}}), \text{ for all Y occurring in P}\)

3) \(\bar{\sigma}(P_1;P_2) := \max(\sigma_{(P_1;P_2)}), \text{ for all Y occurring in P, with P_1;P_2 not a modifier}\)

4) \(\bar{\sigma}(\text{foreach X [Q]})) := \max(\sigma_{(\text{foreach X [Q]}))}, \text{ for all Y occurring in P}\)

Note that \(\sigma(P) \leq \mu(P)\), for each stack program \(P\). Note

also that we are using the previously defined \(\bar{\sigma}\) to detect all

the increasing modifiers, for a given variable \(Y\) (this is done

setting \(\bar{\sigma}_Y\) equal to 1); but, once detected, we don’t have to

consider them in the evaluation of the \(\sigma\)-measure. This is

the reason of the "\(-1\)" 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 4.4:** \(P\) and \(Q\) are space equivalent if \(\{X = w\}P\{X = m\}\) implies \(\{X = w\}Q\{X = O(m)\}\).

This is denoted with \(P \equiv_s Q\).

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

1) if \(A\) is foreach \(X [R]\), with \(\mu(R) = \sigma(R) = n\), and with denoted with \(C_1, \ldots, C_l\) the top circles in \(R\), and with \(A_1, \ldots, A_{ip}\) the variables involved in \(C_i\), for each \(i\), we have that \(\rightsquigarrow A\) is the result of changing each

\(\text{imp}(A_{ij})\) into \(\text{nop}(A_{ij})\) (a no-operation imperative);

2) if \(A\) is foreach \(X [R]\), with \(\mu(R) > \sigma(R)\), we have that \(\rightsquigarrow A\) is equal to foreach \(X [\rightsquigarrow R]\);

3) if \(A\) is \(A_1;A_2\) and \(\max(\mu(A_1),\mu(A_2)) = \mu(A_1)\), we have that \(\rightsquigarrow A\) is equal to \(\rightsquigarrow A_1;A_2\); simmetrically, if \(\max(\mu(A_1),\mu(A_2)) = \mu(A_2)\), we have that \(\rightsquigarrow A\) is equal to \(\rightsquigarrow A_1;\rightsquigarrow A_2\);

4) if \(A\) is if top(X) :=a \([\rightsquigarrow R]\), we have that \(\rightsquigarrow A\) is equal to if top(X) :=a \([\rightsquigarrow R]\).

**Lemma 4.1:** Given a stack program \(P\), with \(\mu(P) = n + 1\) and \(\sigma(P) = n\), there exists a stack program \(\rightsquigarrow P\) such that

\(\mu(\rightsquigarrow P) = n, \sigma(\rightsquigarrow P) = n\), and \(P \equiv_s \rightsquigarrow P\).

**Proof:** (by induction on \(n\)). Base. Let \(P = 1\) and \(\sigma(P) = 0\). In the main case, \(P\) is in the form foreach \(X [Q]\),

with a not-increasing circle occurring in \(Q\). Applying \(\rightsquigarrow\) to

\(P\), we obtain a program \(P'\) whose \(\sigma\)-measure is still 0, and

whose \(\mu\)-measure is reduced to 0, because \(\rightsquigarrow\) has broken off

the circle in \(P\) that leads \(\mu\) 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 \(\{X = \bar{w}\}\{X = m\}\) we have that

\(\{X = \bar{w}\}P'\{X = cm\}\), where \(c\) is a constant depending on

the structure of \(P\); thus, \(P \equiv_s P'\).

Step. Let \(P = n + 2\) and \(\sigma(P) = n + 1\). Let \(P\) be in the

form foreach \(X [Q]\), and let \(C\) be a top circle occurring in

\(Q\), with \(\mu(Q) = n + 1\); we have two cases: (1) \(\sigma(Q) = n + 1\),

or (2) \(\sigma(Q) = n\).

(1) In this case \(C\) is a not-increasing circle, because it has

been detected by \(\mu\), but not by \(\sigma\). Applying \(\rightsquigarrow\) to \(P\), we

obtain a program \(P'\) such that \(\sigma(P') = n + 1, \mu(P') = n + 1,\) and

\(P \equiv_s P'\).

(2) In this case \(C\) is an increasing circle, detected by \(\mu\) and

\(\sigma\). We have that (by the inductive hypothesis) there exists a

program \(Q'\) such that \(\mu(Q') = n, \sigma(Q') = n, \text{ and } Q \equiv_s Q'\).

Starting from \(P\), we build a new program \(P' = \text{foreach X [Q']}\). We have that \(\mu(P') = \mu(Q') + 1 = n + 1, \sigma(P') = \sigma(Q') + 1 = n + 1\), and \(P \equiv_s P'\) as expected.

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

**Theorem 4.1:** Every function \(f\) computed by a stack program \(P\) such that \(\mu(P) = n\) and \(\sigma(P) = m\), with \(n > m\), has a length bound \(b \in E^{m+2}\) satisfying \(|f(\bar{w})| \leq b(|\bar{w}|)\).
Proof: Let $k$ be $\mu(P) - \sigma(P)$. Then by $k$ applications of Lemma 4.1, we obtain a sequence $P = P_0, P_1, \ldots, P_k$ of stack programs such that for all $i < k$,

$$\mu(P_{i+1}) = \mu(P) - i, \sigma(P_i) = \sigma(P_{i+1}), \text{ and } P_i \approx P_{i+1}.$$ 

By Kristiansen and Niggl’s bounding theorem, $P_k$ has a length bound in $E^{\mu(P)+2}$, and so does $P$ by transitivity of $\approx$.

Let $L^n$ be the class of all functions that can be computed by a stack program of $\mu$-measure $n \geq 0$, and let $G^n$ be the class of all functions which can be computed by a Turing machine in time $b(|w|)$, for some $b \in E^\mu$. As a consequence of this new version of the bounding lemma, and similarly to what has been recalled in Section 3, the following result holds.

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

5. Conclusions

We have defined a syntactical measure $\sigma$ 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 $\sigma$-measure is increased by 1. Other measures detect potentially harmful loops, but are not able to distinguish between size-increasing loops and the non-size-increasing one’s. 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 $\sigma$-measure $n$ to the $n+2-th$ Grzegorczyk class, and this class is surely lower in the hierarchy, when compared to the $\mu$-measure.

References