Predicative Recursion, Diagonalization, and Slow-growing Hierarchies of Time-bounded Programs

Emanuele Covino and Giovanni Pani
Dipartimento di Informatica
Università di Bari, Italy
Email: emanuele.covino@uniba.it, giovanni.pani@uniba.it

Abstract—We define a version of predicative recursion, a related programming language, and a hierarchy of classes of programs that represents a resource-free characterization of register machines computing their output within polynomial time $O(n^k)$, for each finite $k$. Then, we introduce an operator of diagonalization, that allows us to extend the previous hierarchy and to capture the register machines with computing time bounded by an exponential limit $O(n^{n^k})$. Finally, by means of a restriction on composition of programs, we characterize the register machines with a polynomial bound imposed over time and space complexity, simultaneously.

Index Terms—Time/space complexity classes; Implicit computational complexity; Predicative recursion; Diagonalization.

I. INTRODUCTION

A complexity class is usually defined by imposing an explicit bound on time and/or space resources used by a Turing Machine (or another equivalent model) during its computation. On the other hand, different approaches use logic and formal methods to provide languages for complexity-bounded computation; they aim at studying computational complexity without referring to external measuring conditions or a to particular machine model, but only by considering language restrictions or logical/computational principles implying complexity properties. In particular, this is achieved by characterizing complexity classes by means of recursive operators with explicit syntactical restrictions on the role of variables. In this paper, we extend the result introduced in [1] by defining a resource-free characterization of register machines computing their output within polynomial time $O(n^k)$, for each finite $k$, and exponential $O(n^{n^k})$; we achieve this result by means of our version of predicative recursion and of a new diagonalization operator, and a related programming language.

One of the first characterization of the polynomial-time computable functions was given by Cobham [2], in which these functions are exactly those generated by bounded recursion on notation. The first predicative definitions of recursion can be found in the work of Simmons [3], Bellantoni and Cook [4], and Leivant [5], [6]: they introduced a ramification principle that does not require that explicit bounds are imposed on the definition of functions, proving that this principle captures the class PTIMEF. Following this approach, several other complexity classes have been characterized by means of unbounded and predicative operators: see, for instance, Leivant and Marion [7] and Oitavem [8] for PSPACEF and the class of the elementary functions; Clote [9] for the definition of a time/space hierarchy between PTIMEF and PSPACE; a theoretical insight has been provided by Leivant [5], [10] and [6]. All these approaches have been dubbed Implicit Computational Complexity: they share the idea that no explicitly bounded schemes are needed to characterize a great number of classes of functions and that, in order to do this, it suffices to distinguish between safe and normal variables (or, following [3], between dormant and normal ones) in the recursion schemes. Roughly speaking, the normal positions are used only for recursion, while the safe positions are used only for substitution. The two main objectives of this area are to find natural implicit characterizations of various complexity classes of functions, thereby illuminating their nature and importance, and to design methods suitable for static verification of program complexity.

Our version of the safe recursion scheme on a binary word algebra is such that $f(x, y, za) = h(f(x, y, z), y, za)$; throughout this paper we will call $x, y$ and $z$ the auxiliary variable, the parameter, and the principal variable of a program defined by recursion, respectively. We do not allow the renaming of variable $z$ as $x$, implying that the step program $h$ cannot assign the value $f(x, y, z)$ of the being-defined program $f$ to the principal variable $z$; in other words, we always know in advance the number of recursive calls of the step program in a recursive definition. We obtain that $z$ is a dormant variable, according to Simmons’ [3], or a safe one, following Bellantoni and Cook [4]. Starting from a natural definition of constructors and destructors over an algebra of lists, we define the hierarchy of classes of programs $T_k$, with $k \in \mathbb{N}$, where programs in $T_1$ can be computed by register machines within linear time, and $T_{k+1}$ are programs obtained by one application of safe recursion to elements in $T_k$; we prove that programs defined in $T_k$ are exactly those programs computable within time $O(n^k)$.

Using the definition of structured ordinals as given in [11], we introduce an operator of constructive diagonalization, and we extend the previous hierarchy to $T_\lambda$, with $\omega \leq \lambda \leq \omega^\omega$. Programs in $T_{\lambda+1}$ are obtained by one application of safe recursion to elements in $T_\lambda$; if $\lambda$ is a limit ordinal, and $\lambda_1, \ldots, \lambda_k, \ldots$ is the associated fundamental
sequence, programs in $T_k$ are obtained by diagonalization on the previously defined sequence of classes $T_{\lambda_1}, \ldots, T_{\lambda_k}, \ldots$. This allows us to harmonize in a single hierarchy of classes of programs all the register machines with computing time bounded by polynomial time $O(n^k)$ and exponential time $O(n^{\omega(k)})$, for each finite $k$. Similar results, achieved with different approaches, can be found in [12], [13], [14], and [15].

Then, we restrict $T_k$ to the hierarchy $S_k$, whose elements are the programs computable by a register machine in linear space. By means of a restricted form of composition between programs, we define a polytime-space hierarchy $T S_{qp}$, such that each program in $T S_{qp}$ can be computed by a register machine within time $O(n^p)$ and space $O(n^q)$, simultaneously. Similar results are in [16] and [7], and are a preliminary step for an implicit classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [9].

The paper is organized as follows: in Section II, we introduce the basic instructions of our language, the notion of composition of programs, and the classes of programs $T_0$ and $T_1$; in Section III, we recall the definition of register machine, the model of computing underlying our characterization, and we prove that programs in $T_1$ capture exactly the computations performed by a register machine within linear time; in Section IV, we define the finite hierarchy $T_k$, with $k \geq 1$, and we prove that programs in this hierarchy capture the computations performed within polynomial time; in Section V, we introduce the diagonalization operator, and we extend the finite hierarchy in order to capture the computations with time-complexity up to exponential time; in Section VI, we redefine the notion of composition, and we give a characterization of classes of programs with time and space polynomial bound.

II. BASIC INSTRUCTIONS AND DEFINITION SCHEMES

In this section, we introduce the basic operators of our programming language and the first two classes on which our hierarchy is based. The language is defined over a binary word alphabet, with the restriction that words are packed into lists, with the symbol @ acting as a separator between each word. This allows us to handle a sequence of words as a single object. The basic instructions allow us to manipulate lists of words, with some restrictions on the renaming of variables; the language is completely defined adding our version of recursion and composition of programs.

A. Recursion-free programs and class $T_0$

$B$ is the binary alphabet \{0, 1\}, $a, b, a_1, \ldots$ denote elements of $B$, and $U, V, \ldots, Y$ denote words over $B$. $p, q, \ldots, s, \ldots$ stand for lists in the form $Y_1 \oplus Y_2 \oplus \ldots \oplus Y_{n-1} \oplus Y_n$. $\epsilon$ is the empty word. The ith component $(s)_i$ of $s = Y_1 \oplus Y_2 \oplus \ldots \oplus Y_{n-1} \oplus Y_n$ is $Y_i$. $|s|$ is the length of the list $s$, that is the number of letters occurring in $s$. We write $x, y, z$ for the variables used in a program, and we write $u$ for one among $x, y, z$. Programs are denoted with the letters $f, g, h$, and we write $f(x, y, z)$ for the application of the program $f$ to variables $x, y, z$, where some among them may be absent.

Definition 2.1: The basic instructions are:

1) the identity $i(u)$ that returns the value $s$ assigned to $u$;
2) the constructors $C_0^i(s)$ that add the digit $a$ at the right of the last digit of $(s)_i$, with $u = 0, 1$ and $i \geq 1$;
3) the destructors $D_i(s)$ that erase the rightmost digit of $(s)_i$, with $i \geq 1$.

Constructors $C_0^i(s)$ and destructors $D_i(s)$ leave the input $s$ unchanged if it has less than $i$ components. Example 2.1: Given the word $s = 011000$, we have that $|s| = 9$, and $(s)_2 = 11$. We also have $C_1^1(011) = 01111$, $D_2(0000) = 0000$, $D_2(0000) = 0000$.

Definition 2.2: Given the programs $g$ and $h$, $f$ is defined by simple schemes if it is obtained by:

1) renaming of $x$ as $y$ in $g$, that is, $f$ is the result of the substitution of the value of $y$ to all occurrences of $x$ into $g$. Notation: $f = RNM_{x/y}(g)$;
2) renaming of $z$ as $y$ in $g$, that is, $f$ is the result of the substitution of the value of $y$ to all occurrences of $z$ into $g$. Notation: $f = RNM_{z/y}(g)$;
3) selection in $g$ and $h$, when for all $s, t, r$ we have

$$f(s, t, r) = \begin{cases} 
  g(s, t, r) & \text{if the rightmost digit of } (s)_i \text{ is } b \\
  h(s, t, r) & \text{otherwise},
\end{cases}$$

with $i \geq 1$ and $b = 0, 1$. Notation: $f = SEL_i^b(g, h)$.

Simple schemes are denoted with SIMPLE.

Example 2.2: if $f$ is defined by $RNM_{x/y}(g)$ we have that $f(t, r) = g(t, t, t)$. Similarly, $f$ defined by $RNM_{z/y}(g)$ implies that $f(s, t) = g(s, t, t)$. Let $s$ be the word 00$\oplus$1010, and $f = SEL_2^0(g, h)$; we have that $f(s, t, r) = g(s, t, r)$, since the rightmost digit of $(s)_2$ is 0.

Definition 2.3: Given the programs $g$ and $h$, $f$ is defined by safe composition of $h$ and $g$ in the variable $u$ if it is obtained by the substitution of $h$ to $u$ in $g$, if $u = x$ or $u = y$; the variable $x$ must be absent in $h$, if $u = z$. Notation: $f = SCMP_u(h, g)$.

The reason of this particular form of composition of programs will be clear in the following section, where we will show to the reader how to combine composition and recursion in order to obtain new time-bounded programs.

Definition 2.4: A modifier is obtained by the safe composition of a sequence of constructors and a sequence of destructors.

Definition 2.5: $T_0$ is the class of programs defined by closure of modifiers under selection and safe composition. Notation: $T_0 = (\text{modifier} \ast \text{SCMP} \ast \text{SEL})$.

All programs in $T_0$ modify their inputs according to the result of some test performed over a fixed number of digits.
B. Safe recursion and class $T_1$

In what follows we introduce the definition of our form of recursion and iteration on notation (see [4] and [10]).

Definition 2.6: Given the programs $g(x, y)$ and $h(x, y, z)$, $f(x, y, z)$ is defined by safe recursion in the basis $g$ and in the step $h$ if for all $s, t, r$ we have

$$
\begin{align*}
    f(s, t, a) &= g(s, t) \\
    f(s, t, ra) &= h(f(s, t, r), t, ra),
\end{align*}
$$

with $a \in B$. Notation: $f = \text{srec}(g, h)$.

In particular, $f(x, z)$ is defined by iteration of $h(x)$ if for all $s, r$ we have

$$
\begin{align*}
    f(s, a) &= s \\
    f(s, ra) &= h(f(s, r)).
\end{align*}
$$

with $a \in B$. Notation: $f = \text{iter}(h)$. We write $h^{\lfloor r \rfloor}(s)$ for iter$(h)(s, r)$ (i.e., the $|r|$-th iteration of $h$ on $s$).

Definition 2.7: $T_1$ is the class defined by closure under simple schemes and safe composition of programs in $T_0$ and programs obtained by one application of iter to $T_0$ (denoted with iter$(T_0)$).

Notation: $T_1 = \text{iter}(T_0); \text{SCMP}; \text{SIMPLE}$.

As we have already stated in the Introduction, we call $x, y$ and $z$ the auxiliary variable, the parameter, and the principal variable of a program obtained by means of the previous recursion scheme. That the renaming of $z$ as $x$ is not allowed (see definition 2.2), and if the step program of a recursion is defined itself by safe composition of programs $p$ and $q$, no variable $x$ (i.e., no potential recursive calls) can occur in the function $p$, when $p$ is substituted into the principal variable $z$ of $q$ (see definition 2.3). These two restrictions implies that the step program of a recursive definition never assigns the recursive call to the principal variable. This is the key of the polynomial-time complexity bound intrinsic to our programs.

Definition 2.8: 1) Given $f \in T_1$, the number of components of $f$ is $\max\{i|D_i \text{ or } c_i \in \text{Sel}_f \text{ occurs in } f\}$. Notation: $\#(f)$.

2) Given a program $f$, its length is the number of constructors, destructors and defining schemes occurring in its definition. Notation: $lh(f)$.

III. Computation by register machines

In this section, we recall the definition of register machine as presented in [6], and we give the definition of computation within a given time and/or space bound. We prove that programs in $T_1$ are exactly those computable within linear time.

Definition 3.1: Given a free algebra $A$ generated from constructors $c_1, \ldots, c_n$ (with $\text{arity}(c_i) = r_i$), a register machine over $A$ is a computational device $M$ having the following components:

1) a finite set of states $S = \{s_0, \ldots, s_n\}$;

2) a finite set of registers $\Phi = \{\pi_0, \ldots, \pi_m\}$;

3) a collection of commands, where a command may be:

   a branching $s_1 \pi_j s_1 \ldots s_k$, such that when $M$ is in the state $s_i$, switches to state $s_i, \ldots, s_k$ according to whether the main constructor (i.e., the leftmost) of the term stored in register $\pi_j$ is $c_1, \ldots, c_k$;

   a constructor $s_1 \pi_j \ldots \pi_{j_r} c_i \pi r$, such that when $M$ is in the state $s_i$, store in $\pi$ the result of the application of the constructor $c_i$ to the values stored in $\pi_j \ldots \pi_{j_r}$, and switches to $s_r$;

   a p-destructor $s_1 \pi_j \pi r$, $p \leq \text{max}(r_i)$, such that when $M$ is in the state $s_i$, store in $\pi$ the $p$-th subterm of the term in $\pi_j$, if it exists; otherwise, store the term in $\pi_j$. Then it switched to $s_r$.

A configuration of $M$ is a pair $(s, F)$, where $s \in S$ and $F : \Phi \rightarrow A$. $M$ induces a transition relation $\Gamma_M$ on configurations, where $\kappa \Gamma_M \kappa'$ holds if there is a command of $\kappa'$ whose execution converts the configuration $\kappa$ to $\kappa'$.

A computation of $M$ on input $X = X_1, \ldots, X_p$ with output $Y = Y_1, \ldots, Y_q$ is a sequence of configurations, starting with $(s_0, F_0)$, and ending with $(s_1, F_1)$ such that:

1) $F_0(\pi_j(i)) = X_i$, for $1 \leq i \leq p$ and $j'$ a permutation of the $p$ registers;

2) $F_1(\pi_j''(i)) = Y_i$, for $1 \leq i \leq q$ and $j''$ a permutation of the $q$ registers;

3) each configuration is related to its successor by $\Gamma_M$;

4) the last configuration has no successor by $\Gamma_M$.

Definition 3.2: A register machine $M$ computes the program $f$ if, for all $s, t, r$, we have that $f(s, t, r) = q$ implies that $M$ computes $(q)_{\text{j_1}}, \ldots, (q)_{\#(f)}$ on input $(s)_{\text{j_1}}, \ldots, (s)_{\#(f)}$, $(t)_{\text{j_1}}, \ldots, (t)_{\#(f)}$, $(r)_{\text{j_1}}, \ldots, (r)_{\#(f)}$.

Definition 3.3: Given a register machine $M$ and the polynomials $p(n)$ and $q(n)$, for each input $X$ (with $|X| = n$),

1) $M$ computes its output within time $O(p(n))$ if its computation runs through $O(p(n))$ configurations;

2) $M$ computes its output in space $O(q(n))$ if, during the computation, the global length of the contents of its registers is $O(q(n))$;

3) $M$ computes its output with time $O(p(n))$ and space $O(q(n))$ if the two bounds occur simultaneously, during the same computation.

Note that the number of registers needed by $M$ to compute a program $f$ has to be fixed a priori (otherwise, we should have to define a family of register machines for each program to be computed, with each element of the family associated to an input of a given length). According to definitions 2.8 and 3.2, $M$ uses a number of registers which linearly depends on the highest component’s index that $f$ can manipulate or access with one of its constructors, destructors or selections; and which depends on the number of times a variable is used by $f$, that is, on the total number of different copies of the registers that $M$ needs during the computation. Both these numbers are constant values, and can be detected before the
computation occurs.

Unlike the usual operators cons, head and tail over Lisp-like lists, our constructors and destructors can have direct access to any component of a list, according to definition 2.1. Hence, their computation by means of a register machine requires constant time, but it requires an amount of time which is linear in the length of the input, when performed by a Turing machine.

**Codes.** We write $s_i \otimes F_j(\pi_0) \otimes \ldots \otimes F_j(\pi_k)$ for the word that encodes a configuration $(s_i, F_j)$ of $M$, where each component is a binary word over $\{0, 1\}$.

**Lemma 3.1:** if $T_1$ and only if $f$ is computable by a register machine within time $O(n)$.

**Proof:** To prove the first implication we show (by induction on the structure of $f$) that each $f \in T_1$ can be computed by a register machine $M_f$ in time $cn$, where $c$ is a constant which depends on the construction of $f$, and $n$ is the length of the input.

Base. $f \in T_0$. This means that $f$ is obtained by closure of a number of modifiers under selection and safe composition; each modifier $g$ can be computed within time bounded by $lh(g)$, the overall number of basic instructions and definition schemes of $g$, i.e., by a machine running over a constant number of configurations; the result follows, since the selection can be simulated by a branching, and the safe composition can be simulated by a sequence of register machines, one for each modifier.

Step. Case 1. $f = iter(g)$, with $g \in T_0$. We have that $f(s, r) = g|^r|(s)$. A register machine $M_f$ can be defined as follows: $(s_i)$, is stored in the register $\pi_i (i = 1 \ldots |f|)$ and $(r_j)$ is stored in the register $\pi_j (j = |f| + 1 \ldots |f| + 1)$; $M_f$ runs $M_g$ (within time bounded by $lh(g)$) for $|r|$ times. Each time $M_g$ is called, $M_f$ deletes one digit from one of the registers $\pi_{g(|f|+1)} \ldots \pi_{g(|f|+1)}$, starting from the first; the computation stops, returning the final result, when they are all empty. Thus, $M_f$ computes $f(s, r)$ within time $|r|lh(g)$.

Case 2. Let $f$ be defined by simple schemes or safe composition. The result follows by direct simulation of the schemes.

In order to prove the second implication, we show that the behaviour of a $k$-register machine $M$, which operates in time $cn$ can be simulated by a program in $T_1$. Let $nxt_M$ be a program in $T_0$, such that $nxt_M$ operates on input $s = s_i \otimes F_j(\pi_0) \otimes \ldots \otimes F_j(\pi_k)$ and it has the semantic if state$[i](s)$ then $E_i$, where state$[i](s)$ is a test that is true if the state of $M$ is $s_i$, and $E_i$ is a modifier that updates the code of the state and the code of one among the registers, according to the definition of $M$. By means of $c - 1$ safe compositions, we define $nxt_M^c$ in $T_0$, which applies $nxt_M$ to the word that encodes a configuration of $M$ for $c$ times. We define in $T_1$

$$\begin{cases} 
    linsim_M(x, a) = x \\
    linsim_M(x, 2a) = nxt_M^c(linsim_M(x, z))
\end{cases}$$

$linsim_M(s, r)$ iterates $nxt_M(s)$ for $|c| \cdot |r|$ times, returning the code of the configuration that contains the final result of $M$.

**IV. The time hierarchy**

In this section, we recall the definition of the class of programs $T_i$; we define our hierarchy of classes of programs, and we prove the relation with the classes of register machines, which compute their output within a polynomially-bounded amount of time.

**Definition 4.1:** 1) $iter(T_0)$ denotes the class of programs obtained by one application of iteration to programs in $T_0$;

2) $T_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $T_0$ and programs in $iter(T_0)$;

Notation: $T_1 = (T_0, iter(T_0); scmp, simple)$;

3) $T_{k+1}$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $T_k$ and programs in $srec(T_k)$, with $k \geq 1$;

Notation: $T_{k+1} = (T_k, srec(T_k); scmp, simple)$.

**Lemma 4.1:** Each $f(s, t, r) \in T_0$ can be computed by a register machine within time bounded by $|s| + lh(f)(|t| + |r|)^p$, with $k \geq 1$.

**Proof:** Base. $f \in T_1$. The relevant case is when $f$ is in the form $iter(h)$, with $h \in T_0$. In lemma 3.1 (step, case 1) we have proved that $f(s, r)$ can be computed within time $|r|lh(h)$; hence, we have the thesis.

Step. $f \in T_{p+1}$. The most significant case is when $f = srec(g, h)$. By the inductive hypothesis there exist two register machines $M_g$ and $M_h$ which compute $g$ and $h$ within the required time. Let $r$ be the word $a_1 \ldots a_{|r|}$; recalling that $f(s, t, ra) = h(f(s, t, r, t, ra)$, we define a register machine $M_f$ that calls $M_g$ on input $s, t$, and calls $M_h$ for $|r|$ times on input stored into the appropriate set of registers (in particular, the result of the previous recursive step has to be stored always in the same register). By inductive hypothesis, $M_g$ needs time $|s| + lh(g)(|t|)^p$ in order to compute $g$; for the first computation of the step program $h$, $M_h$ needs time $|g(s, t) + lh(h)(|t| + |a_{|r|} - a_{|r|}|)^p$.

After $|r|$ calls of $M_h$, the final configuration is obtained within overall time $|s| + max(lh(g), lh(h))(|t| + |r|)^p + 1 \leq |s| + lh(f)(|t| + |r|)^{p+1}$.

**Lemma 4.2:** The behaviour of a register machine which computes its output within time $O(n^k)$ can be simulated by a program $f$ in $T_k$, with $k \geq 1$.

**Proof:** Let $M$ be a register machine respecting the hypothesis. As we have already seen, there exists $nxt_M \in T_0$ such that, for input the code of a configuration of $M$, it returns the code of the configuration induced by the relation $\vdash_M$. Given a fixed $i$, we write the program $\sigma_i$ by means of $i$ safe recursions nested over $nxt_M$, such that it iterates $nxt_M$ on input $s$ for $n^i$ times, with $n$ the length of the input:

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org
\( \sigma_0 := \text{ITER}(nxt_M) \) and
\( \sigma_{n+1} := \text{RNM}_{z=y}(\gamma_{n+1}) \), where \( \gamma_{n+1} := \text{SREC}(\sigma_n, \sigma_n) \).
We have that
\[
\sigma_0(s, t) = \text{nxt}\|t\|_M(s), \quad \sigma_{n+1}(s, t, t)
\]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
for each \( i \), the program \( f(x, y) \) is defined by diagonalization at \( \lambda \) if for all \( s, t \) if
\[ f(s, t) = \text{ITER }|t|(q(|t|))(s, t) \]
where
\[
\begin{align*}
ITER^1(p)(s, t) &= ITER(p)(s, t) \\
ITER^{k+1}(p)(s, t) &= ITER(ITER^k(p))(s, t).
\end{align*}
\]
and \( f_{\lambda_i} \) belongs to a previously defined class \( C_{\lambda_i} \), for each \( i \). Notation: \( f = \text{DIAG}(\lambda) \).

Note that the previous definition requires that \( f_{\lambda_i} \in C_{\lambda_i} \), but no other requirements are made on how the \( C \)'s classes are built. In what follows, we introduce the transfinite hierarchy of programs, with an important restriction on the definition of the \( C \)'s.

**Definition 5.2**: Given \( \lambda < \omega^\omega \), \( T_{\lambda} \) is the class of programs obtained by

1. closure under safe composition and simple schemes of programs in \( T_\alpha \) and programs in \( \text{SREC}(T_\alpha) \), if \( \lambda = \alpha + 1 \);
   Notation: \( T_{\alpha + 1} = (T_\alpha, \text{SREC}(T_\alpha), \text{SCMP}, \text{SIMPLE}) \).
2. closure under simple schemes of programs obtained by one application of diagonalization at \( \lambda \), if \( \lambda \) is a limit ordinal, with \( f_{\lambda_i} \in T_{\lambda_i} \), for each \( \lambda_i \) in the fundamental sequence of \( \lambda \).
   Notation: \( T_\lambda = (\text{DIAG}(\lambda); \text{SIMPLE}) \).

**Lemma 5.1**: Each \( f(s, t, r) \) in \( T_\lambda \) (\( \lambda < \omega^\omega \)) can be computed by a register machine within time \( B_{\lambda}(n) \).

Proof: By induction on \( \lambda \) with three cases:

1. \( \lambda \) is a finite number; we note that \( B_\lambda(n) = n^s \), and the proof follows from Lemma 4.1.
2. \( \lambda = \beta + 1 \); this implies that \( f \in T_{\beta + 1} \), and the relevant subcase is when \( f = \text{SREC}(g, h) \), with both \( g \) and \( h \) belonging to \( T_\beta \). By the inductive hypothesis, there exist the register machines \( M_g \) and \( M_h \) computing \( g \) and \( h \), respectively, within time bounded by \( B_\beta(n) \). A register machine \( M_f \) can be defined, such that it calls \( M_g \) on input \( s, t \), and calls \( M_h \) for \( |r| \) times on input stored into the appropriate set of registers. \( M_f \) needs time \( B_\beta(n) + |r|B_\beta(n) \) to perform this computation; thus, the overall time is bounded by \( B_{\beta + 1}(n) \), by definition of \( B \).
3. \( \lambda \) is a limit ordinal; this means that \( f \) is defined by \( \text{DIAG}(\lambda) \), that is \( f(s, t) = \text{ITER}^{|t|}(g(\lambda_{|t|}))(s, t) \), with \( \lambda_{|t|} \) the fundamental sequence of \( \lambda \) and \( g(\lambda_{|t|}) = f_{\lambda_{|t|}} \in T_{\lambda_{|t|}} \).
   By induction on the length of the input, we have that \( f(s, a) = \text{ITER}^{|a|}(g(\lambda_{|a|}))(s, a) = s \); obviously, there exists a register machine computing the result within time \( B_{\lambda_{|a|}}(n) \). As for the step case we have that
   \[
   f(s, ta) = \text{ITER}^{|a|}(g(\lambda_{|ta|}))(s, ta) = \text{ITER}(\text{ITER}^{|t|}(g(\lambda_{|ta|}))(s, t));
   \]
   by inductive hypothesis, there exist a sequence of register machines \( M_{\lambda_{|ta|}} \) computing the programs \( g(\lambda_{|ta|}) \)'s within time \( B_{\lambda_{|ta|}}(n) \). We define a register machine \( M_f \) such that, on input \( s, t \) iterates \( |t| \) times \( M_{\lambda_{|ta|}} \), within time \( B_{\lambda_{|ta|}}(n) \leq B_\lambda(n) \).

**Lemma 5.2**: The behaviour of a register machine which computes its output within time \( O(B_{\lambda}(n)) \) can be simulated by a program \( f \) in \( T_\lambda \).

Proof: Given a register machine \( M \) respecting the hypothesis, we have already seen that there exists a program \( n x t_M \in T_\emptyset \) such that, for input the code of a configuration of \( M \), it returns the code of the configuration induced by the relation \( r_M \). We have three cases:

1. \( \lambda \) is a finite number; the proof follows from Lemma 4.2.
2. \( \lambda \) is in the form \( \beta + 1 \); in this case, we define the program \( \sigma_{\lambda} \) as follows:
   \[
   \sigma_{\beta + 1}(s, t) = \sigma_{\beta}(\sigma_{\beta + 1}(s, t, t), t),
   \]
   In particular, we have
   \[
   \sigma_{\beta + 1}(s, t) = \sigma_{\beta}(\sigma_{\beta}(\ldots \sigma_{\beta}(s, t), \ldots), t) \]
   By induction we see that \( \sigma_{\beta} \) iterates \( n x t_M \) on its input \( s \) for \( B_\beta(|t|) \) times, and that it belongs to \( T_\emptyset \). The result follows observing that \( \sigma_{\beta + 1} \) iterates \( n x t_M \) for \( |t|B_\beta(|t|) = B_{\beta + 1}(|t|) \) times.
3. \( \lambda \) is a limit ordinal. Let \( \lambda_1, \ldots, \lambda_n, \ldots \) the fundamental sequence associated to \( \lambda \), and \( \sigma_{\lambda_1}, \ldots, \sigma_{\lambda_n}, \ldots \) the sequence of programs enumerated by \( g \), such that \( g(\lambda_i) = \sigma_{\lambda_i} \in T_{\lambda_i} \). We define \( \gamma_{\lambda} \) by diagonalization at \( \lambda \). With a fixed input \( s, t \), we have that \( \gamma_{\lambda}(s, t) = \text{ITER}^{|t|}(g(\lambda_{|t|}))(s, t) = \text{ITER}^{|t|}(\sigma_{\lambda_{|t|}})(s, t) \).
   The programs \( g(\lambda_{|t|}) \) are defined in \( T_{\lambda_{|t|}} \), and they iterate the program \( n x t_M \) on its input for \( B_{\lambda_{|t|}}(|t|) \) times; this implies that \( \gamma_{\lambda} \) iterates \( n x t_M \) for \( B_{\lambda_{|t|}}(|t|) = B_{\lambda}(|t|) \), for each \( t \).

**Theorem 5.1**: A program \( f \) belongs to \( T_{\emptyset} \) if and only if \( f \) is computable by a register machine within time \( O(B_{\emptyset}(n)) \), with \( \omega < \omega^\omega \).

Proof: By lemma 5.1 and lemma 5.2.

**VI. THE TIME-SPACE HIERARCHY**

In this section, we introduce a restricted version of the previously defined time-hierarchy of recursive programs, and we prove the equivalence with the classes of register machines, which compute their output with a simultaneous bound on time and space, following the work of [2].

A. Recursion-free programs and class \( S_0 \)

The reader should refer to Section II for the definitions of basic instruction (the identity \( I(n) \), the constructors \( C^s_n(s) \), and the destructors \( D^s_l(s) \)); simple schemes (the renaming \( \text{RMN}_{\pi/\gamma}(g) \) and \( \text{RMN}_{\pi/\gamma}(g) \), and the selection \( \text{SEL}^b_h(g, h) \)); and safe composition \( \text{SCMP}_w(h, g) \). In particular, a modifier is
obtained by the safe composition of a sequence of constructors and a sequence of destructors; according to definition 2.5, the class $T_0$ is the class of programs defined by closure of modifiers under SEL and SCMP.

Definition 6.1: Given $f \in T_0$, the rate of growth $\text{rog}(f)$ is such that

1) if $f$ is a modifier, $\text{rog}(f)$ is the difference between the number of constructors and the number of destructors occurring in its definition;
2) if $f=\text{SEL}_0^m(g, h)$, then $\text{rog}(f) = \max(\text{rog}(g), \text{rog}(h))$;
3) if $f=\text{SCMP}_a(h, g)$, then $\text{rog}(f) = \max(\text{rog}(g), \text{rog}(h))$.

Definition 6.2: $S_0$ is the class of programs in $T_0$ with non-positive rate of growth, that is $S_0 = \{ f \in T_0 | \text{rog}(f) \leq 0 \}$. Note that all programs in $S_0$ modify their inputs according to the result of some test performed over a fixed number of digits and, moreover, they cannot return values longer than their input.

B. Safe recursion and class $S_1$

As written in section 2.1, a program $f(x, y, z)$ is defined by safe recursion in the basis $g(x, y)$ and in the step $h(x, y, z)$ if for all $s, t, r$ we have

$$\begin{cases} f(s, t, a) = g(s, t) \\ f(s, t, ra) = h(f(s, t, r), t, ra). \end{cases}$$

In this case, $f$ is denoted with $\text{SREC}(g, h)$. In particular, $f(x, z)$ is defined by iteration of $h(x)$ if for all $s, r$ we have

$$\begin{cases} f(s, a) = s \\ f(s, ra) = h(f(s, r)). \end{cases}$$

In this case, $f$ is denoted with $\text{ITER}(h)$, and we write $h^{|r|}(s)$ for $\text{ITER}(h)(s, r)$.

Definition 6.3: 1) $\text{ITER}(S_0)$ denotes the class of programs obtained by one application of iteration to programs in $S_0$;
2) $S_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $S_0$ and programs in $\text{ITER}(S_0)$;
Notation: $S_1 = (S_0, \text{ITER}(S_0))$; SCMP, SIMPLE);
3) $S_{k+1}$ is the class of programs obtained by closure under simple schemes of programs in $S_k$ and programs in $\text{SREC}(S_k)$.
Notation: $S_{k+1} = (S_k, \text{SREC}(S_k); \text{SIMPLE})$.

Hence, hierarchy $S_k$, with $k \in \mathbb{N}$, is a version of $T_k$ in which each program returns a result whose length is exactly bounded by the length of the input; this does not happen if we allow the closure of $S_k$ under SCMP. We will use this result to evaluate the space complexity of our programs.

Definition 6.4: Given the programs $g$ and $h$, $f$ is obtained by weak composition of $h$ in $g$ if $f(x, y, z) = g(h(x, y, z), y, z)$. Notation: $f = \text{WCMP}(h, g)$.

In the weak form of composition the program $h$ can be substituted only in the variable $x$, while in the safe composition the substitution is possible in all variables.

Definition 6.5: For all $p, q \geq 1$, $TS_{ap}$ is the class of programs obtained by weak composition of $h$ in $g$, with $h \in T_q, g \in S_q$, and $q \leq p$.

Lemma 6.1: For all $f$ in $S_p$, we have $|f(s, t, r)| \leq \max(|s|, |t|, |r|)$.

Proof: By induction on $p$. The relevant case is when $f \in S_1$ and $f$ is defined by iteration of $g$ in $S_0$ (that is, $\text{rog}(g) \leq 0$). By induction on $r$, we have that $|f(s, a)| = |s|$, and $|f(s, ra)| = |g(f(s, r))| \leq |f(s, r)| \leq \max(|s|, |r|)$.

Step. Given $f \in S_{p+1}$, defined by $\text{SREC}$ in $g$ and in $h$ in $S_p$, we have

$$|f(s, t, a)| = |g(s, t)| \quad \text{by definition of } f$$
$$\leq \max(|s|, |t|) \quad \text{by inductive hypothesis.}$$

by definition of $f$, inductive hypothesis on $h$ and induction on $r$.

Lemma 6.2: Each $f$ in $TS_{ap}$ (with $p, q \geq 1$) can be computed by a register machine within space $O(n^p)$ and space $O(n^q)$.

Proof: Let $f$ be in $TS_{ap}$. By definition 6.5, $f$ is defined by weak composition of $h \in T_q$ into $g \in S_p$, that is, $f(s, t, r) = g(h(s, t, r), t, r)$. The theorem 5.1 states that there exists a register machine $M_h$, which computes $h$ within time $n^q$, and there exists another register machine $M_g$, which computes $g$ within time $n^p$. Since $g$ belongs to $S_p$, lemma 6.1 holds for $g$; hence, the space needed by $M_g$ is at most $n$.

We define now a machine $M_f$ that, by input $s, t, r$, performs the following steps:

1) it calls $M_h$ on input $s, t, r$;
2) it calls $M_g$ on input $h(s, t, r), t, r$, stored in the appropriate registers.

According to lemma 4.2, $M_h$ needs time equal to $|s| + lh(h)(|t| + |r|)^q$ to compute $h$, and $M_g$ needs $|h(s, t, r)| + lh(g)(|t| + |r|)^p$ to compute $g$.

This happens because lemma 4.2 shows, in general, that the time used by a register machine to compute a program is bounded by a polynomial in the length of its inputs, but, more precisely, it shows that the time complexity is linear in $|s|$. Moreover, since in our language there is no kind of identification of $x$ as $z$, $M_f$ never moves the content of a register associated to $h(s, t, r)$ into another register and, in particular, into a register whose value plays the role of recursive variable. Thus, the overall time bound is $|s| + lh(h)(|t| + |r|)^q + lh(g)(|t| + |r|)^p$ which can be reduced to $n^p$, being $q \leq p$.

$M_h$ requires space $n^q$ to compute the value of $h$ on input $s, t, r$; as we noted above, the space needed by $M_g$ for the
computation of $g$ is linear in the length of the input, and thus the overall space needed by $M_f$ is still $n^q$.

Lemma 6.3: A register machine which computes its output within time $O(n^p)$ and space $O(n^q)$ can be simulated by a program $f \in TS_{qp}$.

Proof: Let $M$ be a register machine, whose computation is time-bounded by $n^p$ and, simultaneously, is space-bounded by $n^q$. $M$ can be simulated by the composition of two machines, $M_h$ (time-bounded by $n^p$), and $M_g$ (time-bounded by $n^p$ and, simultaneously, space-bounded by $n^q$); the former delimits (within $n^q$ steps) the space that the latter will successively use in order to simulate $M$.

By theorem 5.1 there exists $h \in T_q$ that simulates the behaviour of $M_h$, and there exists $g \in T_p$ that simulates the behaviour of $M_g$; this is done by means of $nxt_g$, which belongs to $S_0$, since it never adds a digit to the description of $M_g$ without erasing another one.

According to the proof of lemma 4.1, we are able to define $\sigma_{n-1} \in S_0$, such that $\sigma_{n-1}(s,t) = \text{nxt}_g(s,t)$. The result follows defining $\text{sim}(s) = \sigma_{p-1}(h(s), s) \in TS_{qp}$.

Theorem 6.1: $f$ belongs to $TS_{qp}$ if and only if $f$ is computable by a register machine within time $O(n^p)$ and space $O(n^q)$.

Proof: By lemma 6.2 and lemma 6.3.

VII. CONCLUSIONS AND FURTHER WORK

In this paper, we have introduced a version of safe recursion, together with constructive diagonalization; by means of these two operators, we’ve been able to define a hierarchy of classes of programs $T_\lambda$, with $0 \leq \lambda < \omega^\omega$. Each finite level of the hierarchy characterizes the register machines computing their output within time $O(n^k)$; using the natural definition of structured ordinals, and combining it with the diagonalization operator, we have that the transfinite levels of the hierarchy characterize the classes of register machine computing their output within time bounded by the slow-growing function $B_\lambda(n)$, up to the machines with exponential time complexity. In the last section, we have defined a hierarchy of programs with simultaneous time and space bound.

While the safe recursion scheme has been studied thoroughly, we feel that the diagonalization operator as presented in this work, or as in Marion’s approach (see [12]), deserves a more accurate analysis. In particular, we believe that it can be considered as predicative as the safe recursion, and that it could be used to stretch the hierarchy of programs in order to capture the low Grzegorczyk classes (see [17] for a non-constructive approach).

REFERENCES


