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 PSPACEF; 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 $\mathcal{T}_k$, with $k \in \mathbb{N}$, where programs in $\mathcal{T}_1$ can be computed by register machines within linear time, and $\mathcal{T}_{k+1}$ are programs obtained by one application of safe recursion to elements in $\mathcal{T}_k$; we prove that programs defined in $\mathcal{T}_k$ 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 $\mathcal{T}_\lambda$, with $\omega \leq \lambda \leq \omega^\omega$. Programs in $\mathcal{T}_{\alpha+1}$ are obtained by one application of safe recursion to elements in $\mathcal{T}_\alpha$; if λ is a limit ordinal, and $\lambda_1, \ldots, \lambda_k, \ldots$ is the associated fundamental

sequence, programs in $\mathcal{T}_\lambda$ are obtained by diagonalization on the previously defined sequence of classes $\mathcal{T}_{\lambda_1}, \ldots, \mathcal{T}_{\lambda_k}, \ldots$. 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 $O(n^{n^k})$, for each finite k. Similar results, achieved with different approaches, can be found in [12], [13], [14], and [15].

Then, we restrict $\mathcal{T}_k$ to the hierarchy $\mathcal{S}_k$, 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 $\mathcal{TS}_{qp}$, such that each program in $\mathcal{TS}_{qp}$ 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 $\mathcal{T}_0$ and $\mathcal{T}_1$; in Section III, we recall the definition of register machine, the model of computation underlying our characterization, and we prove that programs in $\mathcal{T}_1$ capture exactly the computations performed by a register machine within linear time; in Section IV, we define the finite hierarchy $\mathcal{T}_k$, 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 algebra, with the restriction that words are packed into lists, with the symbol ©acting as a separator between each word. This allow 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 $\mathcal{T}_0$

$\mathbf{B}$ is the binary alphabet $\{0,1\}$. $a, b, a_1, \ldots$ denote elements of $\mathbf{B}$, and $U, V, \ldots, Y$ denote words over $\mathbf{B}$. $p, q, \ldots, s, \ldots$ stand for lists in the form $Y_1©Y_2©\ldots©Y_{n-1}©Y_n$. ϵ is the empty word. The *i-th component* $(s)_i$ of $s = Y_1©Y_2©\ldots©Y_{n-1}©Y_n$ 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* $\mathrm{I}(u)$ that returns the value s assigned to u;
2) the *constructors* $\mathrm{C}_i^a(s)$ that add the digit a at the right of the last digit of $(s)_i$, with $a = 0, 1$ and $i \geq 1$;
3) the *destructors* $\mathrm{D}_i(s)$ that erase the rightmost digit of $(s)_i$, with $i \geq 1$.

Constructors $\mathrm{C}_i^a(s)$ and destructors $\mathrm{D}_i(s)$ leave the input s unchanged if it has less than i components.

Example 2.1: Given the word $s = 01©11©©00$, we have that $|s| = 9$, and $(s)_2 = 11$. We also have $\mathrm{C}_1^1(01©11) = 011©11$, $\mathrm{D}_2(0©0©) = 0©©$, $\mathrm{D}_2(0©©) = 0©©$.

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: $f = \mathrm{RNM}_{x/y}(g)$;
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: $f = \mathrm{RMN}_{z/y}(g)$;
3) *selection* in g and h, when for all s, t, r we have

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

with $i \geq 1$ and $b = 0, 1$. Notation: $f = \mathrm{SEL}_i^b(g, h)$.

Simple schemes are denoted with SIMPLE.

Example 2.2: if f is defined by $\mathrm{RNM}_{x/y}(g)$ we have that $f(t, r) = g(t, t, r)$. Similarly, f defined by $\mathrm{RNM}_{z/y}(g)$ implies that $f(s, t) = g(s, t, t)$. Let s be the word 00©1010, and $f = \mathrm{SEL}_2^0(g, h)$; 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: $f = \mathrm{SCMP}_u(h, g)$.

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: $\mathcal{T}_0$ is the class of programs defined by closure of modifiers under selection and safe composition.
Notation: $\mathcal{T}_0$=($modifier$; SCMP, SEL).

All programs in $\mathcal{T}_0$ modify their inputs according to the result of some test performed over a fixed number of digits.

B. Safe recursion and class $\mathcal{T}_1$

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

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

with $a \in \mathbf{B}$. 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

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

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

Definition 2.7: $\mathcal{T}_1$ is the class defined by closure under simple schemes and safe composition of programs in $\mathcal{T}_0$ and programs obtained by one application of ITER to $\mathcal{T}_0$ (denoted with $\text{ITER}(\mathcal{T}_0)$).
Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, $\text{ITER}(\mathcal{T}_0)$; SCMP, 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. Note 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 $f \in \mathcal{T}_1$, the *number of components* of f is $max\{i | \text{D}_i$ or C_i^a or SEL_i^b 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 $\mathcal{T}_1$ are exactly those computable within linear time.

Definition 3.1: Given a free algebra $\mathbf{A}$ generated from constructors $\mathbf{c_1},\ldots,\mathbf{c_n}$ (with $arity(\mathbf{c_i}) = r_i$), a *register machine* over $\mathbf{A}$ 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** $s_i\pi_j s_{i_1}\ldots s_{i_k}$, such that when M is in the state s_i, switches to state $s_{i_1},\ldots,s_{i_k}$ according to whether the main constructor (i.e., the leftmost) of the term stored in register π_j is $\mathbf{c}_1,\ldots,\mathbf{c}_k$;
   a **constructor** $s_i\pi_{j_1}\ldots\pi_{j_{r_i}}\mathbf{c}_i\pi_l s_r$, such that when M is in the state s_i, store in π_l the result of the application of the constructor $\mathbf{c}_i$ to the values stored in $\pi_{j_1}\ldots\pi_{j_{r_i}}$, and switches to s_r;
   a **p-destructor** $s_i\pi_j\pi_l s_r$ ($p \leq max(r_i)_{i=1\ldots k}$), such that when M is in the state s_i, store in π_l the p-th subterm of the term in π_j, if it exists; otherwise, store the term in π_j. Then it switched to s_r.

A *configuration* of M is a pair (s,F), where $s \in S$ and $F : \Phi \to \mathbf{A}$. M induces a transition relation $\vdash_M$ on configurations, where $\kappa \vdash_M \kappa'$ holds if there is a command of M whose execution converts the configuration κ in κ'. A *computation* of M on input $\vec{X} = X_1,\ldots,X_p$ with output $\vec{Y} = Y_1,\ldots,Y_q$ 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 $\vdash_M$;
4) the last configuration has no successor by $\vdash_M$.

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 $(q)_1,\ \ldots,\ (q)_{\#(f)}$ on input $(s)_1,\ \ldots,\ (s)_{\#(f)}$, $(t)_1,\ \ldots,\ (t)_{\#(f)}$, $(r)_1,\ \ldots,\ (r)_{\#(f)}$.

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

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 $s_i©F_j(\pi_0)©\ldots©F_j(\pi_k)$ 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: f belongs to $\mathcal{T}_1$ if 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 $f \in \mathcal{T}_1$ 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. $f \in \mathcal{T}_0$. 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. $f =\text{ITER}(g)$, with $g \in \mathcal{T}_0$. 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 π_i $(i = 1\ldots\#(f))$ and $(r)_j$ is stored in the register π_j $(j = \#(f)+1\ldots 2\#(f))$; 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 $\pi_{\#(f)+1}\ldots\pi_{2\#(f)}$, 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 $\mathcal{T}_1$. Let nxt_M be a program in $\mathcal{T}_0$, such that nxt_M operates on input $s = s_i©F_j(\pi_0)©\ldots©F_j(\pi_k)$ 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 $\mathcal{T}_0$, which applies nxt_M to the word that encodes a configuration of M for c times. We define in $\mathcal{T}_1$

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

$linsim_M(s,r)$ iterates $nxt_M(s)$ for $c|r|$ 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 $\mathcal{T}_1$; 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($\mathcal{T}_0$) denotes the class of programs obtained by one application of iteration to programs in $\mathcal{T}_0$;
2) $\mathcal{T}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_0$ and programs in ITER($\mathcal{T}_0$);
   Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE);
3) $\mathcal{T}_{k+1}$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_k$ and programs in SREC($\mathcal{T}_k$), with $k \geq 1$;
   Notation: $\mathcal{T}_{k+1}$=($\mathcal{T}_k$, SREC($\mathcal{T}_k$); SCMP, SIMPLE).

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

Proof: Base. $f \in \mathcal{T}_1$. The relevant case is when f is in the form ITER(h), with $h \in \mathcal{T}_0$. 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. $f \in \mathcal{T}_{p+1}$. The most significant case is when $f =\text{SREC}(g,h)$. 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 $|g(s,t)| + lh(h)(|t| + |a_{|r|-1}a_{|r|}|)^p$.
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 $\mathcal{T}_k$, with $k \geq 1$.

Proof: Let M be a register machine respecting the hypothesis. As we have already seen, there exists $nxt_M \in \mathcal{T}_0$ 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 σ_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:

$\sigma_0 :=\text{ITER}(nxt_M)$ and
$\sigma_{n+1} :=\text{RNM}_{z/y}(\gamma_{n+1})$, where $\gamma_{n+1} :=\text{SREC}(\sigma_n, \sigma_n)$.
We have that
$\sigma_0(s,t) = nxt_M^{|t|}(s)$, $\sigma_{n+1}(s,t) = \gamma_{n+1}(s,t,t)$, and

$$\begin{cases} \gamma_{n+1}(s,t,a) & = \sigma_n(s,t) \\ \gamma_{n+1}(s,t,ra) & = \sigma_n(\gamma_{n+1}(s,t,r),t) \\ & = \gamma_n(\gamma_{n+1}(s,t,r),t,t) \end{cases}$$

In particular, we have

$$\sigma_1(s,t) = \quad \gamma_1(s,t,t) = \quad \underbrace{\sigma_0(\sigma_0(\ldots\sigma_0(s,t)\ldots))}_{|t|\ times} = nxt_M^{|t|^2}$$

$$\sigma_2(s,t) = \quad \gamma_2(s,t,t) = \quad \underbrace{\sigma_1(\sigma_1(\ldots\sigma_1(s,t)\ldots))}_{|t|\ times} = nxt_M^{|t|^3}$$

By induction, we see that σ_{k-1} iterates nxt_M on input s for $|t|^k$ times, and that it belongs to $\mathcal{T}_k$. The result follows defining $f(t) = \sigma_{k-1}(t,t)$, with t the code of an initial configuration of M. ∎

Theorem 4.1: A program f belongs to $\mathcal{T}_k$ if and only if f is computable by a register machine within time $O(n^k)$, with $k \geq 1$.

Proof: By lemma 4.2 and lemma 4.1. ∎

We recall that register machines are polytime reducible to Turing machines; thus, the sequence of classes $\mathcal{T}_k$ captures PTIMEF (see [6] and [12] for similar characterization of this complexity class).

V. Extending the polynomial-time hierarchy to transfinite

In this section, we extend the definition of the classes of programs $\mathcal{T}_k$, with $k \geq 1$, to a transfinite hierarchy of classes; in order to do this, we recall the definition of structured ordinals and of hierarchies of slow/fast growing functions, as reported in [11]. Then, we introduce a natural slow growing function B, and we give the definition of *diagonalization* at a given limit ordinal λ, based on the sequence of classes $\mathcal{T}_{\lambda_1}, \ldots, \mathcal{T}_{\lambda_n}, \ldots$ associated with the fundamental sequence of λ. A similar constructive operator can be found in [15] and [12]. We prove that this transfinite hierarchy of programs characterize the classes of register machines computing their output within time between $O(n^k)$ and $O(n^{n^k})$ (with $k \geq 1$ and n the length of the input), that is, the computations with time complexity between polynomial and exponential time.

A. Structured ordinals and hierarchies

Following [11], we denote limit ordinals with greek small letters α, β, λ, ..., and we denote with λ_i the i-th element of the fundamental sequence assigned to λ. For example, ω is the limit ordinal of the fundamental sequence $1, 2, \ldots$; and ω^2 is the limit ordinal of the fundamental sequence $\omega, \omega 2, \omega 3, \ldots$, with $(\omega^2)_k = \omega k$.

The *slow-growing functions* $G_\alpha : \mathbb{N} \to \mathbb{N}$ are defined by the recursion

$$\begin{cases} G_0(n) & = & 0 \\ G_{\alpha+1}(n) & = & G_\alpha(n) \\ G_\lambda(n) & = & G_{\lambda_n}(n). \end{cases}$$

The *fast-growing functions* $F_\alpha : \mathbb{N} \to \mathbb{N}$ are defined by the recursion

$$\begin{cases} F_0(n) & = & n+1 \\ F_{\alpha+1}(n) & = & F_\alpha^{n+1}(n) \\ F_\lambda(n) & = & F_{\lambda_n}(n). \end{cases}$$

We define the *slow-growing functions* $B_\alpha : \mathbb{N} \to \mathbb{N}$ by means of the recursion

$$\begin{cases} B_0(n) & = & 1 \\ B_{\alpha+1}(n) & = & nB_\alpha(n) \\ B_\lambda(n) & = & B_{\lambda_n}(n). \end{cases}$$

Note that $B_k(n) = n^k$, $B_\omega(n) = n^n$, $B_{\omega+k}(n) = n^{n+k}$, $B_{\omega k}(n) = n^{n \cdot k}$, $B_{\omega^k}(n) = n^{n^k}$, and $B_{\omega^\omega}(n) = n^{n^n}$; moreover, we have that $B_{\alpha+\beta}(n) = B_\alpha(n) \cdot B_\beta(n)$, and that $G_{\omega^\alpha}(n) = n^{G_\alpha(n)} = B_\alpha(n)$.

B. Diagonalization and transfinite hierarchy

The finite hierarchy $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k, \ldots$, captures the register machines that compute their output with time in $O(1), O(n), O(n^2), \ldots, O(n^k), \ldots$, respectively. Jumping out of the hierarchy requires something more than safe recursion.

A possible approach consist in defining a kind of ranking function that counts the number of nested recursion violating our "no-bad-renaming" rule or, in general, not respecting the predicative definition of a program. A class of time-bounded register machines is associated to each level of violation. This idea was introduced in [17].

On the other hand, given a limit ordinal λ, we propose a new operator that *diagonalizes* at level λ over the classes $\mathcal{T}_{\lambda_i}$, that is, that selects programs in a previously defined class according to the length of the input. There is no circularity in a program defined by diagonalization, and we believe that this program isn't less predicative than a program defined by safe recursion. For instance, at level ω, we are able to select programs in the sequence $\mathcal{T}_i$, where the value of i depends on the length of the input; thus, this level of diagonalization captures the class of all register machines whose computation is bounded by a polynomial. Extending this approach to next levels of structured ordinals, we are able to reach the machines computing within exponential time.

Definition 5.1: Given a limit ordinal λ with the fundamental sequence $\lambda_0, \ldots, \lambda_k, \ldots$, and given an enumerator program q such that $q(\lambda_i) = f_{\lambda_i}$, for each i, the program $f(x,y)$ is defined by *diagonalization* at λ if for all s, t if

$$f(s,t) =\text{ITER}^{|t|}(q(\lambda_{|t|}))(s,t)$$

where

$$\begin{cases} \text{ITER}^1(p)(s,t) & = & \text{ITER}(p)(s,t) \\ \text{ITER}^{k+1}(p)(s,t) & = & \text{ITER}(\text{ITER}^k(p))(s,t). \end{cases}$$

and f_{λ_i} belongs to a previously defined class $\mathcal{C}_{\lambda_i}$, for each i. Notation: $f = \text{DIAG}(\lambda)$.

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

Definition 5.2: Given $\lambda < \omega^\omega$, $\mathcal{T}_\lambda$ is the class of programs obtained by

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

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

Proof: By induction on λ. We have three cases:
(1) λ is a finite number; we note that $B_k(n) = n^k$, and the proof follows from Lemma 4.1.
(2) $\lambda = \beta + 1$; this implies that $f \in \mathcal{T}_{\beta+1}$, and the relevant subcase is when $f = \text{SREC}(g,h)$, with both g and h belonging to $\mathcal{T}_\beta$. 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) λ 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 λ_i the fundamental sequence of λ and $g(\lambda_i)$=$f_{\lambda_i} \in \mathcal{T}_{\lambda_i}$. 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

$$\begin{aligned} f(s,ta) &= \text{ITER}^{|ta|}(g(\lambda_{|ta|}))(s,ta) \\ &= \text{ITER}(\text{ITER}^{|t|}(g(\lambda_{|ta|})))(s,t); \end{aligned}$$

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 $\mathcal{T}_\lambda$.

Proof: Given a register machine M respecting the hypothesis, we have already seen that there exists a program $nxt_M \in \mathcal{T}_0$ such that, for input the code of a configuration of M, it returns the code of the configuration induced by the relation $\vdash_M$. We have three cases:
(1) λ is a finite number; the proof follows from Lemma 4.2.
(2) λ is in the form $\beta+1$; in this case, we define the program σ_λ as follows:
$\sigma_{\beta+1}$:=$\text{RNM}_{z/y}(\gamma_{\beta+1})$, where $\gamma_{\beta+1}$:=$\text{SREC}(\sigma_\beta, \sigma_\beta)$.
We have that
$\sigma_{\beta+1}(s,t) = \gamma_{\beta+1}(s,t,t)$, and

$$\begin{cases} \gamma_{\beta+1}(s,t,a) & = \sigma_\beta(s,t) \\ \gamma_{\beta+1}(s,t,ra) & = \sigma_\beta(\gamma_{\beta+1}(s,t,r),t) \\ & = \gamma_\beta(\gamma_{\beta+1}(s,t,r),t,t) \end{cases}$$

In particular, we have

$$\sigma_{\beta+1}(s,t) = \underbrace{\gamma_{\beta+1}(s,t,t) = \sigma_\beta(\sigma_\beta(\ldots\sigma_\beta(s,t)\ldots,t),t)}_{|t|times}$$

By induction we see that σ_β iterates nxt_M on its input s for $B_\beta(|t|)$ times, and that it belongs to $\mathcal{T}_\beta$. The result follows observing that $\sigma_{\beta+1}$ iterates nxt_M for $|t|B_\beta(|t|) = B_{\beta+1}(|t|)$ times.
(3) λ is a limit ordinal. Let $\lambda_1, \ldots, \lambda_n, \ldots$ the fundamental sequence associated to λ, and $\sigma_{\lambda_1}, \ldots, \sigma_{\lambda_n}, \ldots$ the sequence of programs enumerated by g, such that $g(\lambda_i) = \sigma_{\lambda_i} \in \mathcal{T}_{\lambda_i}$. We define γ_λ by diagonalization at λ. 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 $\mathcal{T}_{\lambda_{|t|}}$, and they iterate the program nxt_M on its input for $B_{\lambda_{|t|}}(|t|)$ times; this implies that γ_λ iterates nxt_M for $B_{\lambda_{|t|}}(|t|) = B_\lambda(|t|)$, for each t. ∎

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

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 $\mathcal{S}_0$

The reader should refer to Section II for the definitions of basic instruction (the *identity* $\text{I}(u)$, the *constructors* $\text{C}_i^a(s)$, and the *destructors* $\text{D}_i(s)$); simple schemes (the *renaming* $\text{RNM}_{x/y}(g)$ and $\text{RMN}_{z/y}(g)$, and the *selection* $\text{SEL}_i^b(g,h)$); and safe composition $\text{SCMP}_u(h,g)$. 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 $\mathcal{T}_0$ is the class of programs defined by closure of modifiers under SEL and SCMP.

Definition 6.1: Given $f \in \mathcal{T}_0$, the *rate of growth* $rog(f)$ is such that

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

Definition 6.2: $\mathcal{S}_0$ is the class of programs in $\mathcal{T}_0$ with non-positive rate of growth, that is $\mathcal{S}_0 = \{f \in \mathcal{T}_0 | rog(f) \leq 0\}$. Note that all programs in $\mathcal{S}_0$ 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 $\mathcal{S}_1$

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 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 ITER(h), and we write $h^{|r|}(s)$ for ITER$(h)(s,r)$.

Definition 6.3:
1) ITER$(\mathcal{S}_0)$ denotes the class of programs obtained by one application of iteration to programs in $\mathcal{S}_0$;
2) $\mathcal{S}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{S}_0$ and programs in ITER$(\mathcal{S}_0)$;
   Notation: $\mathcal{S}_1$=($\mathcal{S}_0$, ITER$(\mathcal{S}_0)$; SCMP, SIMPLE);
3) $\mathcal{S}_{k+1}$ is the class of programs obtained by closure under simple schemes of programs in $\mathcal{S}_k$ and programs in SREC$(\mathcal{S}_k)$.
   Notation: $\mathcal{S}_{k+1}$=($\mathcal{S}_k$, SREC$(\mathcal{S}_k)$; SIMPLE).

Hence, hierarchy $\mathcal{S}_k$, with $k \in \mathbb{N}$, is a version of $\mathcal{T}_k$ 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 $\mathcal{S}_k$ 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 =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 $p,q \geq 1$, $\mathcal{TS}_{qp}$ is the class of programs obtained by weak composition of h in g, with $h \in \mathcal{T}_q$, $g \in \mathcal{S}_p$ and $q \leq p$.

Lemma 6.1: For all f in $\mathcal{S}_p$, we have $|f(s,t,r)| \leq \max(|s|,|t|,|r|)$.

Proof: By induction on p. Base. The relevant case is when $f \in \mathcal{S}_1$ and f is defined by iteration of g in $\mathcal{S}_0$ (that is, $rog(g) \leq 0$). 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 $f \in \mathcal{S}_{p+1}$, defined by SREC in g and h in $\mathcal{S}_p$, we have

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

and

$$\begin{aligned} |f(s,t,ra)| &= |h(f(s,t,r),t,ra)| \\ &\leq |\max(|f(s,t,r)|,|t|,|ra|)| \\ &\leq |\max(\max(|s|,|t|,|r|),|t|,|ra|)| \\ &\leq |\max(|s|,|t|,|ra|)|. \end{aligned}$$

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

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

Proof: Let f be in $\mathcal{TS}_{qp}$. By definition 6.5, f is defined by weak composition of $h \in \mathcal{T}_q$ into $g \in \mathcal{S}_p$, 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 $\mathcal{S}_p$, 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 $f \in \mathcal{TS}_{qp}$.

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^q), and M_g (time-bounded by n^p and, simultaneously, space-bounded by n): 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 $h \in \mathcal{T}_q$ that simulates the behaviour of M_h, and there exists $g \in \mathcal{T}_p$ that simulates the behaviour of M_g; this is done by means of nxt_g, which belongs to $\mathcal{S}_0$, 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 $\sigma_{n-1} \in \mathcal{S}_n$, such that $\sigma_{n-1}(s,t) = nxt_g^{|t|^n}$. The result follows defining $sim(s) = \sigma_{p-1}(h(s), s) \in \mathcal{TS}_{qp}$. ■

Theorem 6.1: f belongs to $\mathcal{TS}_{qp}$ 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 $\mathcal{T}_\lambda$, 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

[1] E. Covino and G. Pani, "A Specialized Recursive Language for Capturing Time-Space Complexity Classes," in The Sixth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, (COMPUTATION TOOLS 2015), Nice, France, 2015, pp. 8–13.

[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, Amsterdam. North-Holland, 1962, pp. 24–30.

[3] H. Simmons, "The realm of primitive recursion," Arch.Math. Logic, vol. 27, 1988, pp. 177–121 885.

[4] S. Bellantoni and S. Cook, "A New Recursion-Theoretic Characterization Of The Polytime Functions," Computational Complexity, vol. 2, 1992, pp. 97–110.

[5] D. Leivant, "A foundational delineation of computational feasibility," in Proceedings of the 6th Annual Symposium on Logic in Computer Science, (LICS'91), Amsterdam. IEEE Computer Society Press, 1991, pp. 2–18.

[6] ——, Predicative recurrence and computational complexity I: word recurrence and polytime. Birkauser, 1994, pp. 320–343.

[7] D. Leivan and J.-Y. Marion, "Ramified recurrence and computational complexity II: substitution and polyspace," in J.Tiuryn and L.Pocholsky (eds), Computer Science Logic, LNCS 933, Amsterdam. Springer Berlin Heidelberg, 1995, pp. 486–500.

[8] I. Oitavem, "New recursive characterization of the elementary functions and the functions computable in polynomial space," Revista Matematica de la Univaersidad Complutense de Madrid, vol. 10, 1997, pp. 109–125.

[9] P. Clote, "A time-space hierarchy between polynomial time and polynomial space," Math. Sys. The., vol. 25, 1992, pp. 77–92.

[10] D. Leivant, "Stratified functional programs and computational complexity," in Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL'93), Charleston. ACM, 1993, pp. 325–333.

[11] M. Fairtlough and S. Weiner, Hierarchies of provably recursive functions. Elsevier, Amsterdam, 1998, chapter 3, pp. 149–207.

[12] J. Marion, "On tiered small jump operators," Logical Methods in Computer Science, vol. 5, no. 1, 2009.

[13] T. Arai and N. Eguchi, "A new function algebra of EXPTIME functions by safe nested recursion," ACM Transactions on Computational Logic, vol. 10, 2009, pp. 1–19.

[14] D. Leivant, "Ramified recurrence and computational complexity III: higher type recurrence and elementary complexity," Annals of pure and applied logic, vol. 96, 1999, pp. 209–229.

[15] S. Caporaso, G. Pani, and E. Covino, "A predicative approach to the classification problem," Journal of Functional Programming, vol. 11, 2001, pp. 95–116.

[16] M. Hofmann, "Linear types and non-size-increasing polynomial time computation," in Proceedings of the 14th Symposium on Logic in Computer Science (LICS'99), Trento. IEEE Computer Society Press, 1999, pp. 464–473.

[17] S. Bellantoni and K. Niggl, "Ranking primitive recursion: the low Grzegorczyk classes revisited," SIAM Journal on Computing, vol. 29, 1999, pp. 401–4015.