Diagonalization and Elementary Complexity

Emanuele Covino and Giovanni Pani
Dipartimento di Informatica, Università di Bari, Italy

Abstract—By means of the definition of predicative recursion, we introduce a programming language that provides a resource-free characterization of register machines computing their output within polynomial time $O(n^k)$, for each finite $k$. We extend the language with a constructive diagonalization scheme, and we define a transfinite hierarchy of programs with exponential-time complexity (at level $\omega^n$), and with elementary complexity (at level $\epsilon_0$). This is achieved by means of predicative operators, contrasting to previous results. We discuss the feasibility and the complexity of our diagonalization operator.

Keywords: Predicative recursion; Constructive diagonalization; Hierarchies of complexity classes; Polynomial-time computable functions; Elementary functions.

1. Introduction

The Grzegorczyk’s classes of functions $E^n$ ($n = 0, 1, \ldots$), have been introduced in [10], with the property that $\bigcup_{n<\omega} E^n$ is the class of primitive recursive functions. In order to define these classes, the hierarchy functions $E_n$ are defined as follows: $E_0(x, y) = x + y$, $E_1(x) = x^2 + 2$, $E_{n+2}(0) = 2$, $E_{n+2}(x + 1) = E_{n+1}(E_{n+2}(x))$; they are repeated iterations of the successor function, essentially. Grzegorczyk’s hierarchy can be defined inductively: $E^0$ is the class whose initial functions are the zero, successor and the projection functions, and that is closed under composition and limited recursion; $E^{n+1}$ is defined similarly, except that the function $E_n$ is added to the list of the initial functions. $E^n$ is called the $n$-th Grzegorczyk class. Other sequences of hierarchy functions have been used in literature, for instance the Ackermann function. The reader can find detailed definitions, properties, and theorems in [18]; we recall that the class $E^3$ is the class of the elementary functions $E'$ (that is, the class of functions containing the successor, projections, zero, addition, multiplication, subtraction functions, and closed under composition and bounded sum and product).

Harmonizations of significant complexity classes with the Grzegorczyk hierarchy have been obtained by Leivant [12], Niggl [16], and Bellantoni and Niggl [3]. In these papers, the class of polynomial-time computable functions is characterized by means of different definitions of predicative recursion [2] or ramified recurrence [11], [13], and starting from a set of initial functions. A predicative definition of a recursive function is based on the idea that functions have two kind of variables: those whose values are known entirely (and which can be recursed upon, for instance), and those whose values are still being computed (and are accessible in a more restricted way, on the least significant digits, for instance). These two types of variables are called safe and normal in [2] (dormant and normal in [19]); normal variables are used only for recursive calls, while safe variables are used only for substitution. This allows to discard explicitly bounded schemes (i.e., the limited recursion) to characterize classes of functions.

An alternative approach to look at ramification can be found in [3], where the classes $E^n$ are captured by closure under composition of functions, and by counting the number of infringements to the predicative principle made into the recursive definitions. Rather than controlling the type of the variables, and how they are used in the definition of a recursive function, first a function is defined, and then one examines it in order to count how many levels of impredicative definitions are used. This approach represents a detailed analysis of the effects of nesting recursive definitions, but it appears evident that counting the number of violations to the predicative principle is as impredicative as using limited recursion, or as adding a hierarchy function to the list of initial functions.

In a previous paper [5], we introduced our version of predicative recursion, together with a constructive diagonalization operator; they allow us to define a hierarchy of programs $T_k$ ($k = 0, 1, \ldots$), such that each program defined in $T_k$ is computable by a register machine within time bounded by a polynomial $n^k$; and to extend this hierarchy up to the programs computable within exponential-time bound. In this contribution we prove that our approach can be extended further, and that our hierarchy reaches the level 3 of the Grzegorczyk hierarchy, that is, the class of the elementary functions.

The paper is organized as follows. In Section II, we recall the definition of our programming language, and the results that hold on the finite levels of our hierarchy of programs. In Section III, we introduce the definition of diagonalization, we discuss its feasibility, and we extend the hierarchy of programs to transfinite levels; then, we state the result on these levels, up to the elementary level. In Section IV, proofs of the results introduced in the previous sections are provided. Conclusions are in Section V.
2. Basic instructions, composition of programs and recursion

In this section, we introduce the basic instructions of our programming language, together with the definition schemes of composition and recursion of programs. We then give the definition of a finite hierarchy of programs, which captures the polynomial-time computable functions.

The language is built over lists of binary words, with the symbol © acting as a separator between each word. B denotes the alphabet \{0, 1\}, and \(a, b, a_1, \ldots\) denote elements of B; \(U, V, \ldots, Y\) denote words over B. \(r, s, \ldots\) stand for lists in the form \(Y_1\otimes Y_2\otimes \ldots \otimes Y_n\), © is the empty word. The \(i\)-th component \((s)_i\) of a list \(s = Y_1\otimes Y_2\otimes \ldots \otimes Y_n\) is \(Y_i\), \(|s|\) is the length of the list \(s\), that is the overall number of symbols occurring in \(s\).

We write \(x, y, z\) for the whole set of variables used in programs, and we write \(u\) for one among \(x, y, z\). Programs are denoted with 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. In what follows, the length \(lh(f)\) of a program \(f\) is the number of basic instructions and defining schemes occurring in its definition.

The basic instructions allow us to manipulate lists of words, adding digits to (or erasing parts of) each component; the simple schemes allow us to change the name of some among the variables and to select between programs, according to the value of the variables.

**Definition 2.1:** The basic instructions are:

1) the identity \(1\) that returns the value \(s\) assigned to \(u\);
2) the constructors \(C^a_i(s)\) 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 \(D_i(s)\) that erase the rightmost digit of \((s)_i\), with \(i \geq 1\).

Constructors \(C^a_i(s)\) and destructors \(D_i(s)\) leave the input \(s\) unchanged if it has less than \(i\) components.

For instance, for \(s = 01©11©000\), we have that \(|s| = 9\), and \((s)_2 = 11\); we also have \(c_1(01©11) = 011©11,\)
\(d_2(0©0©0) = ©0©,\) and \(d_2(0©©) = ©©©\).

**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: \(f = RNM_{x/y}(g)\);
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: \(f = SEL^b_i(g, h)\).

Simple schemes are denoted with SIMPLE.

For instance, if \(f\) is defined by \(RNM_{z/y}(g)\) we have that \(f(t, r) = g(t, t, r)\). Similarly, \(f\) defined by \(RNM_{z/y}(g)\) implies that \(f(s, t) = g(s, t, t)\). For \(s = 00©1010\), and \(f = SEL^0_i(g, h)\), 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 \(h\), the program \(f\) 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 rationale behind this definition and the definition of renaming will be clear as soon as we will introduce the safe recursion scheme.

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

**Definition 2.5:** The class \(T_0\) is defined by closure of modifiers under safe composition and selection. Notation: \(T_0 = \langle\text{modifier}; \text{SCMP}, \text{SEL}\rangle\).
All programs in \(T_0\) modify their inputs according to the result of some test performed on a fixed number of digits.

**Definition 2.6:** Given the programs \(g(x, y)\) and \(h(x, y, z)\), the program \(f(x, y, z)\) 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: \(f = SREC(g, h)\).

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 \(h|^{r\text{-th}}(s)\) for \(\text{ITER}(h)(s, r)\) (i.e., the \(r\)-th iteration of \(h\) on \(s\)).

We recall that \(x, y\) and \(z\) are the auxiliary variable, the parameter, and the principal variable of a program obtained by means of a recursion scheme, respectively. Note also that, according to the previous definitions, the renaming of \(z\) as \(x\) is not allowed, and if the step program of a recursion is defined itself by safe composition of programs \(p\) and \(q\), no variable \(i.e., \) no recursive calls) can occur in the function \(p\), when \(p\) is substituted into the principal variable \(z\) of \(q\). These two restrictions imply that the step program of a recursive definition never assigns a recursive call to the principal variable. This limits the growth of the result of a program, and it’s the key to the polynomial-time complexity bound intrinsic to our programs; moreover, these definitions fulfill the predicative criteria.

Given the previous basic instructions and definition schemes, we are able to define the hierarchy of classes of programs \(T_k\), with \(k < \omega\), as follows:

**Definition 2.7:** 1) \(\text{ITER}(T_0)\) denotes the class of programs obtained by one application of iteration to programs in \(T_0\);
2) $T_k$ 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: $T_1=\langle T_0, \text{ITER}(T_0); \text{SCMP}, \text{SIMPLE} \rangle$; 
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 $\text{SREC}(T_k)$, with $k \geq 1$; 
   Notation: $T_{k+1}=\langle T_k, \text{SREC}(T_k); \text{SCMP}, \text{SIMPLE} \rangle$.

Definition 2.8: Given $f \in T_k$, the number of components of $f$ is $\#(f)=\max\{i|d_i \in C^i \text{ or } C^i \text{ or } C^i \text{ occurs in } f\}$.

In [4] and [5] we proved the following two theorems using as a model of computation the register machines introduced by Leivant [13]. The reader can find definitions and proofs in Sections 4.1, 4.2, 4.3.

Lemma 2.1: Each program $f(s,t,r)$ defined in $T_k$ can be computed by a register machine within time bounded by the polynomial $|s|+lh(f)(|t|+|r|)^k$, with $k \geq 1$;

Lemma 2.2: A register machine which computes its output within time $O(n^k)$ can be simulated by a program $f$ in $T_k$, with $k \geq 1$.

Together, they allow us to prove the following

Theorem 2.1: A program $f$ belongs to $T_k$ if and only if $f$ is computable by a register machine within time in $O(n^k)$, with $k \geq 1$.

We recall that register machines are polytime reducible to Turing machines; thus, the class $\bigcup_{k<\omega} T_k$ captures PTIMEF (see [2] and [13] for similar characterizations of this complexity class).

3. Diagonalization, exponential-time and elementary computable programs

In what follows, we recall the definition of structured ordinals and of hierarchies of slow growing functions. Then, we give the definition of diagonalization at a given limit ordinal $\lambda$, starting from a sequence of classes of programs $T_{\beta_1}, \ldots, T_{\beta_n}, \ldots$, which have been previously defined, and which are associated with the fundamental sequence of $\lambda$. A similar approach can be found in [6], and an alternative approach is in [15]. Using safe recursion and diagonalization, we are able to define a transfinite hierarchy of programs characterizing the class 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; moreover, we can extend the hierarchy to the ordinal $\varepsilon_0$, with $\varepsilon_0=\omega^{\omega^\omega} = \sup\{\omega, \omega^{\omega}, \omega^{\omega^\omega}, \ldots \}$; in particular, $T_{\varepsilon_0}$ is the class of programs computable within time $O(n^{\varepsilon_0})$. Given that a function $f(n)$ is elementary if and only if it is computable in time bounded by $n^{\varepsilon_0}$ (see [8]), we have that $\bigcup_{k<\varepsilon_0} T_k$ characterizes the class of the elementary functions.

3.1 Structured ordinals and hierarchy functions

Following [9], we denote limit ordinals with greek small letters $\alpha, \beta, \lambda, \ldots$, and we denote with $\lambda_i$ the $i$-th element of the fundamental sequence assigned to $\lambda$. For example, $\omega$ is the limit ordinal of the fundamental sequence $1, 2, \ldots$. and $\omega^2$ is the limit ordinal of the fundamental sequence $\omega, \omega^2, \omega^3, \ldots$, with $(\omega^2)_k = \omega k$.

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

\[
\begin{align*}
G_0(n) &= 0 \\
G_{\alpha+1}(n) &= G_\alpha(n) + 1 \\
G_\lambda(n) &= G_{\lambda}(n).
\end{align*}
\]

We slightly change the previous definition, and we define the slow-growing functions $B_\alpha : \mathbb{N} \rightarrow \mathbb{N}$ by the recursion

\[
\begin{align*}
B_0(n) &= 1 \\
B_{\alpha+1}(n) &= nB_\alpha(n) \\
B_\lambda(n) &= B_{\lambda}(n).
\end{align*}
\]

Note that $B_k(n)=n^k$, $B_\omega(n)=n^n$, $B_{\omega+k}(n)=n^{n+k}$, $B_{\omega^2}(n)=n^{n^2}$, $B_{\omega^3}(n)=n^{n^3}$, and $B_{\omega^k}(n)=n^{n^{n^k}}$; moreover, we have that $B_{\alpha+1}(n)=B_\alpha(n) \cdot B_\beta(n)$, and that $G_\omega(n)=n^{G_\alpha(n)}=B_\alpha(n)$.

3.2 Defining programs by diagonalization

The finite hierarchy $T_0, T_1, T_2, \ldots, T_\lambda, \ldots$, captures the register machines that compute their output with time-bound 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. We already discussed in the Introduction the approach presented in [3], that is, to define a ranking function that counts the number of nested recursions infringing the predicative definition of a program; a class of time-bounded register machines can be associated to each level of the ranking. On the other hand, given a limit ordinal $\lambda$, we proposed in [5] a new operator that diagonalizes at level $\lambda$ over the classes $T_{\lambda_i}$, that is, that selects and iterates programs in a previously defined class $T_{\lambda_i}$, with $i$ the length of the input. There is no circularity in a program defined by diagonalization, and this program is as predicative as one defined by safe recursion. For instance, at level $\omega$, we select (and iterate $i$ times) programs in the classes $T_i$, where $i$ is the length of the input; thus, the first level of diagonalization $T_{\omega}$ captures the class of all register machines whose computation is bounded by a polynomial. By extending this approach to the next levels of structured ordinals, we were able to characterize the machines computing their output within exponential time $O(n^{n^k})$ by means of programs defined in $T_{\omega^k}$.

Definition 3.1: Given a limit ordinal $\lambda$ 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 constructive diagonalization at $\lambda$ if, for all $s,t$

$$f(s,t) = \text{ITER}^{[1]}(q(\lambda_{|s|}))(s,t)$$

where

\[
\begin{align*}
\text{ITER}^1(p)(s,t) &= \text{ITER}(p)(s,t) \\
\text{ITER}^{k+1}(p)(s,t) &= \text{ITER}(\text{ITER}^k(p))(s,t).
\end{align*}
\]

and $f_\lambda$ belongs to a previously defined class of programs $C_\lambda$, for each $i$. Notation: $f = \text{DIAG}(\lambda)$.

Note that the previous definition requires that $f_\lambda \in C_\lambda$, but there are no other requirements on the classes $C$'s are built. In what follows, we introduce our transfinite hierarchy of programs, with an important restriction on the definition of the $C$'s.

**Definition 3.2:** Given $\lambda < \epsilon_0$, $T_\lambda$ is the class of programs obtained by

1) closure under safe composition and simple schemes of programs in $T_\alpha$ and programs in 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 $f_\lambda \in T_\lambda$, for each $\lambda$ in the fundamental sequence of $\lambda$. Notation: $T_\lambda = (\text{DIAG}(\lambda); \text{SIMPLE})$.

### 3.3 Capturing exponential and elementary programs

Starting from the results in [5], we prove that

**Lemma 3.1:** Each program $f(s,t,r)$ defined in $T_\lambda$ can be computed by a register machine within time $B_\lambda(n)$, with $\lambda < \epsilon_0$.

**Lemma 3.2:** A register machine which computes its output within time $O(B_\lambda(n))$ can be simulated by a program $f$ in $T_\lambda$, with $\lambda < \epsilon_0$.

**Lemma 3.1** is proved by structural induction on the ordinal $\lambda$, that can be a finite number, an ordinal $\beta + 1$, or a limit ordinal: in each case we build the register machine that computes the program $f$ at level $\lambda$ using the machines provided by the inductive hypothesis, and we compute the overall time consumption, showing that it respects the bound $B_\lambda(n)$. Lemma 3.2 is proved for each time-bounded register machines showing that, given a program $nxt_{\lambda} \in T_\lambda$ that simulates the transition between the machine’s configurations, a program in the appropriate class $T_\beta$ can be built as the repeated iteration of $nxt_{\lambda}$, in order to simulate the overall computation performed by the register machine itself. Both proofs can be found in Section 4.4.

Using the previous results, we then have that

**Theorem 3.1:** A program $f$ belongs to $T_\alpha$, if and only if $f$ is computable by a register machine within time $O(B_\alpha(n))$, with $\alpha < \epsilon_0$.

Two intertwined questions (not addressed in [5]) could be raised about the enumerator $q(\lambda_i)$. First, to which class does the enumerator belongs? And what are its results? For complexity reasons, it appears clear that the enumerator should be defined into the same hierarchy of classes that we are using to diagonalize; in particular, it can be defined into the first class $T_\lambda$ of every sequence $T_{\lambda_1}, \ldots, T_{\lambda_n}$, because it only has to write sequences of SREC’s and DIAG’s according to the definition on the ordinal $\lambda_i$, in order to write the programs $f_{\lambda_i} \in T_{\lambda_i}$. This leads us to the second question: the results of any program in our language are lists of binary words; they are not programs. This means that the enumerator $q$ returns the code of a program in $T_{\lambda_i}$, and not the program itself. Defining a code for every element of our language is straightforward, but we must underline that every time we diagonalize over a sequence of classes, we should see the $f_{\lambda_i}$’s as codes of programs; this implies that an interpreter is concealed in the definition of diagonalization.

Our operators of predicative recursion and constructive diagonalization can be used to provide a fine hierarchy of classes of programs between $T_\alpha$ and $T_{\alpha+1}$, with time complexity between PTIME and $\mathcal{E}^3$, respectively. As far as we know, this is the only characterization of these classes built "from below", by means of constructive operators. Other characterizations, like Oitavem [17], Arai and Eguchi [1], and Marion [15], capture the elementary functions alone, using various forms of predicative recursion and composition. Leviant [14], captures $\mathcal{E}^3$ using an extension to higher types of ramified recurrence.

### 4. Proofs of the results

#### 4.1 Computation by register machines

In this section, we recall the definition of a register machine as presented in [13], and we give the definition of computation within a given time bound. We prove that programs in $T_1$ are exactly those computable within linear time $O(n)$.

**Definition 4.1:** Given a free algebra $A$ generated from constructors $c_1, \ldots, c_n$ (with 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** $s_i\pi_j s_{i_1} \ldots s_{i_k}$, 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 $\pi_j$ is $c_1, \ldots, c_k$;
   - a **constructor** $s_i\pi_j \ldots \pi_{j_n} c_i s_{r_i}$, such that when $M$ is in the state $s_i$, store in $\pi_j$ the result of the application of the constructor $c_i$ to the values stored in $\pi_j, \ldots, \pi_{j_n}$, and switches to $s_{r_i}$;
   - a **destructor** $s_i\pi_j s_{r_p} (p \leq \max(r_j) = 1, \ldots, k)$, such that when $M$ is in the state $s_i$, store in $\pi_j$ the $p$-th subterm of the term in $\pi_j$, if it exists; otherwise, store the term in $\pi_j$. Then it switched to $s_{r_p}$. [4]
A configuration of $M$ is a pair $(s, F)$, where $s \in S$ and $F : \Phi \rightarrow A$. $M$ induces a transition relation $\rightarrow_M$ on configurations, where $\kappa \rightarrow_M \kappa'$ holds if there is a command of $M$ whose execution converts the configuration $\kappa$ in $\kappa'$. 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. $F_0(\pi_j(s_0)) = X_i$, for $1 \leq i \leq p$ and $j'$ a permutation of the $p$ registers;
2. $F_1(\pi_{j''}(s_1)) = Y_i$, for $1 \leq i \leq q$ and $j''$ a permutation of the $q$ registers;
3. each configuration is related to its successor by $\rightarrow_M$;
4. the last configuration has no successor by $\rightarrow_M$.

Definition 4.2: 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 4.3: Given a register machine $M$ and the polynomial $p(n)$, for each input $\vec{X}$ (with $|\vec{X}| = n$), $M$ computes its output within time $O(p(n))$ if its computation runs through $O(p(n))$ configurations.

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 4.2, $M$ uses a number of registers which linearly depends on the highest component’s index of $f$ that $M$ 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 $s_i \circ F_j(\pi_0) \cdots \circ F_j(\pi_k)$ for the word that encodes a configuration $(s_i, F_j)$ of $M$, where each component is a binary word over $\{0, 1\}$.

4.2 Linear time computability

Lemma 4.1: $f$ belongs to $T_1$ 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 $f \in T_1$ 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. $f = \text{iter}(g)$, with $g \in T_0$. We have that $f(s, r) = g^{\#(f)}(s)$. 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 $\pi_j$ ($j = \#(f) + 1, \ldots, 2\#(f)$); $M_f$ operates on input $lh(g)$ 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 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 $s = s_i \circ F_j(\pi_0) \cdots \circ F_j(\pi_k)$ and it has the semantic if $\text{state}[i](s)$ then $E_i$, where $\text{state}[i](s)$ 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 $nxt_M$ in $T_0$, which applies $nxt_M$ to the word that encodes a configuration of $M$ for $c$ times. We define in $T_1$

$$
\begin{align*}
\text{lin}_M(x, a) &= x \\
\text{lin}_M(x, za) &= nxt_M(\text{lin}_M(x, z))
\end{align*}
$$

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

4.3 Polynomial time computability

Lemma 4.2: Each program $f(s, t, r)$ defined in $T_k$ can be computed by a register machine within time bounded by the polynomial $|s| + lh(f)(|t| + |r|)^k$, with $k \geq 1$.

Proof: Base. $f \in T_1$. The relevant case is when $f$ is in the form $\text{iter}(h)$, with $h \in T_0$. In lemma 4.1 (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 $f = \text{srec}(g, h)$. 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 $|s| + h_B(g)(|t|)p$ in order to compute $g$; for the first computation of the step program $h$, $M_h$ needs time $|g(s, t)| + h_B(h)(|t|) + |a_{g_{r-1}}a_{r_{fr}}||p^p$. After $|r|$ calls of $M_h$, the final configuration is obtained within overall time $|s| + \max(h(g), h(h))(|t| + |r|)p^{p+1}$.

**Lemma 4.3:** 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 $\text{next}_M \in T_0$ such that, for input the code of a configuration of $M$, it returns the code of the configuration induced by the relation $\Gamma_M$. Given a fixed $i$, we write the program $\sigma_i$ by means of $i$ safe recursions nested over $\text{next}_M$, such that it iterates $\text{next}_M$ on input $s$ for $n^i$ times, with $n$ the length of the input:

$$\sigma_0 := \text{ITER}(\text{next}_M)$$

$$\sigma_{n+1} := \text{RNM}_{\beta}(\gamma_{n+1}^i), \text{ where } \gamma_{n+1} := \text{SREC}(\sigma_n, \sigma_n).$$

We have that

$$\sigma_0(s, t) = \text{next}_M(s, \sigma_0) = \sigma_0(s, t, t), \text{ and }$$

$$\gamma_{n+1}(s, t, a) = \sigma_n(s, t),$$

$$\gamma_{n+1}(s, t, r) = \sigma_n(s, t, t).$$

In particular, we have

$$\gamma_1(s, t) = \gamma_0(s, t) = \sigma_0(s, t, t),$$

$$\gamma_2(s, t) = \gamma_1(s, t) = \sigma_1(s, t, t).$$

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

### 4.4 Exponential time and elementary computability

**Lemma 4.4:** Each $f(s, t, r) \in T_\lambda$ can be computed by a register machine within time $B_\lambda(n)$, with $\lambda < \epsilon_0$.

**Proof:** By induction on $\lambda$. We have three cases:

1. $\lambda$ is a finite number; we note that $B_{\lambda}(n) = n^\lambda$, and the proof follows from Lemma 2.1.
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 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 $f(s, t) = \text{ITER}_B(\gamma_{\lambda(|t|)})(s, t)$, with $\lambda_i$ the fundamental sequence of $\lambda$ and $g(\lambda_i) = f_{\lambda_i} \in T_\lambda$.

By induction on the length of the input, we have that $f(s, a) = \text{ITER}_B(\gamma_{\lambda(|a|)})(s, a) = \gamma_0$; obviously, there exists a register machine computing the result within time $B_{\lambda+1}(n)$.

As for the step case we have that

$$\gamma_0(s, t) = \gamma_1(s, t),$$

$$\gamma_2(s, t) = \gamma_1(s, t).$$

By inductive hypothesis, there exist a sequence of register machines $M_{\lambda, \lambda}$ computing the programs $g(\lambda_i)(n)$ within time $B_{\lambda, \lambda}(n)$. We define a register machine $M_f$ such that, on input $s, t$ iterates $|t|$ times $M_{\lambda, \lambda}(n)$, within time $B_{\lambda, \lambda}(n)$. The result follows from Lemma 2.2.

**Lemma 4.5:** The behaviour of a register machine which computes its output within time $O(B_\lambda(n))$ can be simulated by a program $f$ in $T_\lambda$, with $\lambda < \epsilon_0$.

**Proof:** Given a register machine $M$ respecting the hypothesis, we have already seen that there exists a program $\text{next}_M \in T_0$ such that, for input the code of a configuration of $M$, it returns the code of the configuration induced by the relation $\Gamma_M$. We have three cases:

1. $\lambda$ is a finite number; the proof follows from Lemma 2.2.
2. $\lambda$ is in the form $\beta + 1$; in this case, we define the program $\sigma_\lambda$ as follows:

$$\sigma_{\beta+1} := \text{RNM}_{\beta}(\gamma_{\beta+1}^i), \text{ where } \gamma_{\beta+1} := \text{SREC}(\sigma_{\beta}, \sigma_{\beta}).$$

We have that

$$\sigma_{\beta+1}(s, t) = \gamma_{\beta+1}(s, t, t),$$

$$\gamma_{\beta+1}(s, t, a) = \sigma_{\beta}(s, t),$$

$$\gamma_{\beta+1}(s, t, r) = \sigma_{\beta}(\gamma_{\beta+1}(s, t, t).$$

In particular, we have

$$\sigma_{\beta+1}(s, t) = \gamma_{\beta+1}(s, t, t),$$

$$\gamma_{\beta+1}(s, t, a) = \sigma_{\beta}(s, t),$$

$$\gamma_{\beta+1}(s, t, r) = \sigma_{\beta}(\gamma_{\beta+1}(s, t, t).$$

By induction we see that $\sigma_{\beta}$ iterates $\text{next}_M$ on input $s$ for $|t|^\beta$ times, and that it belongs to $T_\beta$. The result follows defining $f(t) = \sigma_{\beta}(t, t)$, with $t$ the code of an initial configuration of $M$. The programs $g(\lambda_i)(n)$ are defined in $T_{\lambda_i}$ and they iterate the program $\text{next}_M$ on its input for $B_{\lambda_i}(n)$ times; this implies that $\gamma_\lambda$ iterates $\text{next}_M$ for $B_{\lambda_\lambda}(n)$ times, for each $t$.  

5. Conclusions and further work

In this work, we have used a version of safe recursion and constructive diagonalization to define a hierarchy of classes of programs $T_{\lambda}$, with $0 \leq \lambda < \epsilon_0$. Each finite level $k$ 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, the transfinite levels of the hierarchy characterize the classes of register machines computing their output within time bounded by the slow-growing function $\Omega(\lambda^\varepsilon_0)$. Among the others, we are able to write programs with exponential-time complexity (at level $\omega^\omega$), and programs with elementary complexity (at level $\epsilon_0$), that is the class $E^3$ of the Grzegorczyk hierarchy.

We believe possible to investigate higher levels of the hierarchy related to higher levels of ordinals, beyond $\epsilon_0$; in particular, the levels $E^{n+3}$ (with $n \geq 0$) of the Grzegorczyk hierarchy could be characterized by means of classes of programs $T_{\varepsilon_n}$, with $\varepsilon_{n+1} = \sup\{\varepsilon_n, \varepsilon_n^\varepsilon, \varepsilon_n^{\varepsilon_n}, \ldots\}$. At the Feferman-Schütte ordinal $\Gamma_0$ (the first impredicative ordinal) we should be able to capture the class of primitive recursive functions.

References