Diagonalization and the Complexity of Programs

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

***Abstract*—Starting from the definitions of *predicative recursion* and *constructive diagonalization*, we recall our specialized programming language that provides a resource-free characterization of register machines computing their output within polynomial time $O(n^k)$, and exponential time $O(n^{n^k})$, for each finite k. We discuss the possibility of extending this characterization to a transfinite hierarchy of programs that captures the Grzegorczyk hierarchy of functions at elementary level. This is done by means of predicative operators, contrasting to previous results. We discuss the feasibility and the complexity of our diagonalization operator.**

***Index Terms*—Hierarchies of complexity classes; Predicative recursion; Constructive diagonalization.**

I. Introduction and position of the problem

In [1], the Grzegorczyk's classes of functions $\mathcal{E}^n$ ($n = 0, 1, \ldots$), have been introduced, with the property that $\bigcup_{n<\omega} \mathcal{E}^n$ is the class of primitive recursive functions. In order to define these classes, the hierarchy functions E_n have to be introduced. They are essentially repeated iterations of the successor function, i.e., $E_0(x, y) = x + y$, $E_1(x) = x^2 + 2$, $E_{n+2}(0) = 2$, $E_{n+2}(x + 1) = E_{n+1}(E_{n+2}(x))$. Grzegorczyk's classes can be defined as follows. $\mathcal{E}^0$ is the class whose initial functions are the zero, successor and the projection functions, and is closed under composition and limited recursion. $\mathcal{E}^{n+1}$ is defined similarly, except that the function E_n is added to the list of the initial functions. $\mathcal{E}^n$ 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 properties and theorems in [2]; we recall that the class $\mathcal{E}^3$ is the class of the elementary functions $\mathcal{E}'$ (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 classes have been obtained by Leivant [3], Niggl [4], and Bellantoni and Niggl [5]. In these papers, the class of polynomial-time computable functions is characterized by means of different definitions of *predicative recursion* [6] or *ramified recurrence* [7], and starting from a set of initial functions. Note that 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 [6] (dormant and normal in [8]); roughly speaking, normal variables are used only for recursive calls, while safe variables are used only for substitution. This allows to discard explicitly bounded schemes (like the limited recursion) to characterize classes of functions.

In [5], the classes $\mathcal{E}^n$ are captured by closure under composition of functions, and by counting the number of infringements to the predicative principle made into the recursive definitions. This is an alternative approach to look at ramification: rather than controlling the type of the variables, and how they are used in the definition of a recursive function, first a definition of a function is given, and then one examines it in order to see, or to count, how many levels of impredicative definitions are used. Even if this approach represents a detailed analysis of the effects of nesting recursive definitions, we believe 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 [9], we introduced our version of predicative recursion, together with a *constructive diagonalization* operator; they allow us to define a hierarchy of programs $\mathcal{T}_k$ ($k = 0, 1, \ldots$), such that each program defined in $\mathcal{T}_k$ 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 claim that our approach can be extended further, and that our hierarchy reaches the level 3 of the Grzegorczyk hierarchy. We address questions raised about the complexity and feasibility of the diagonalization, and we compare it to a recent different approach [10].

In Section II, we recall the definition of our programming language, and the results holding on the finite levels of our hierarchy of programs. In Section III, we introduce the definition of diagonalization, we discuss its feasibility, and we recall the result on programs with exponential-time complexity. In Section IV, we extend the hierarchy of programs up to the elementary level of the Grzegorczyk hierarchy. In Section V, we compare our approach to Marion's [10]. Conclusions and further work are in Section VI.

II. Basic instructions, composition of programs and recursion

In this section, we recall the basic instructions of our programming language, together with the definition schemes of composition and recursion over programs we introduced in [9]. We then recall the definition of our 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. $\mathbf{B}$ denotes the alphabet $\{0,1\}$, and $a, b, a_1, \ldots$ denote elements of $\mathbf{B}$; $U, V, \ldots, Y$ denote words over $\mathbf{B}$. $r, s, \ldots$ stand for lists in the form $Y_1 © Y_2 © \ldots © Y_n$. ϵ is the empty word. The *i-th component* $(s)_i$ of a list $s = Y_1 © Y_2 © \ldots © Y_n$ 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 variables used in a program, 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 so-called 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. The *basic instructions* are:

1) the *identity* $\textsc{i}(u)$ that returns the value s assigned to u;
2) the *constructors* $\textsc{c}^a_i(s)$ 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* $\textsc{d}_i(s)$ that erase the rightmost digit of $(s)_i$, with $i \geq 1$.

Constructors $\textsc{c}^a_i(s)$ and destructors $\textsc{d}_i(s)$ leave the input s unchanged if it has less than i components. For instance, for $s = 01©11©©00$, we have that $|s| = 9$, and $(s)_2 = 11$; we also have $\textsc{c}^1_1(01©11) = 011©11$, $\textsc{d}_2(0©0©) = 0©©$, and $\textsc{d}_2(0©©) = 0©©$.

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

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

with $i \geq 1$ and $b = 0, 1$. Notation: $f = \textsc{sel}^b_i(g, h)$.

Simple schemes are denoted with SIMPLE. For instance, if f is defined by $\textsc{rnm}_{x/y}(g)$ we have that $f(t, r) = g(t, t, r)$. Similarly, f defined by $\textsc{rnm}_{z/y}(g)$ implies that $f(s, t) = g(s, t, t)$. For $s = 00©1010$, and $f = \textsc{sel}^0_2(g, h)$, we have that $f(s, t, r) = g(s, t, r)$, since the rightmost digit of $(s)_2$ is 0.

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: $f = \textsc{scmp}_u(h, g)$. The rationale behind this definition will be clear as soon as we will define the safe recursion scheme.

A *modifier* is obtained by the safe composition of a sequence of constructors and a sequence of destructors, and the class $\mathcal{T}_0$ is 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.

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

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

with $a \in \mathbf{B}$. Notation: $f = \textsc{srec}(g, h)$.
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: $f = \textsc{iter}(h)$. We write $h^{|r|}(s)$ for $\textsc{iter}(h)(s, r)$ (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 the previous 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 x (i.e., no potential 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 the recursive call to the principal variable. This is the key to the polynomial-time complexity bound intrinsic to our programs, and fulfills the predicative criteria.

Given the previous basic instructions and definition schemes, we are able to define the hierarchy of classes of programs $\mathcal{T}_k$, with $k < \omega$, as follows:

1) $\textsc{iter}(\mathcal{T}_0)$ 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 $\textsc{iter}(\mathcal{T}_0)$;
   Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, $\textsc{iter}(\mathcal{T}_0)$; 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).

In [11] and [9] we proved the following two theorems using as a model of computation the register machines introduced by Leivant [7] . We have that

1) each program $f(s, t, r)$ defined in $\mathcal{T}_k$ can be computed by a register machine within time bounded by the polynomial $|s| + lh(f)(|t| + |r|)^k$, with $k \geq 1$;
2) a register machine which computes its output within time $O(n^k)$ can be simulated by a program f in $\mathcal{T}_k$, with $k \geq 1$.

The previous result allowed us to prove the following

Theorem 2.1: 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 $k \geq 1$.

We recall that register machines are polytime reducible to Turing machines; thus, the sequence of classes $\mathcal{T}_k$ captures PTIMEF (see [6] and [7] for similar characterizations of this complexity class).

III. DIAGONALIZATION AND EXPONENTIAL-TIME COMPUTABLE PROGRAMS

We recall the definition of structured ordinals and of hierarchies of slow growing functions, as reported in [12]. Then, we give the definition of *diagonalization* at a given limit ordinal λ, based on the sequence of classes $\mathcal{T}_{\lambda_1}, \dots, \mathcal{T}_{\lambda_n}, \dots$ associated with the fundamental sequence of λ. A similar operator can be found in [13], and we will discuss later the relation between our diagonalization and its analogue in [10]. Using safe recursion and diagonalization, we are able to define a transfinite hierarchy of programs characterizing 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.

Following [12], 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 $1, 2, \dots$; and ω^2 is the limit ordinal of the fundamental sequence $\omega, \omega 2, \omega 3, \dots$, with $(\omega^2)_k = \omega k$.

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

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

We slightly change the previous definition, and we define the *slow-growing functions* $B_\alpha : \mathbb{N} \to \mathbb{N}$ by 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 $G_{\omega^\alpha}(n) = n^{G_\alpha(n)} = B_\alpha(n)$.

The finite hierarchy $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_k, \dots$, captures the register machines that compute their output with time in $O(1), O(n), O(n^2), \dots, O(n^k), \dots$, respectively. Jumping out of the hierarchy requires something more than safe recursion. We already discussed in the Introduction the approach presented in [5], 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 λ, we proposed in [9] a new operator that *diagonalizes* at level λ over the classes $\mathcal{T}_{\lambda_i}$, that is, that selects and iterates programs in a previously defined class $\mathcal{T}_{\lambda_i}$ 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 select (and iterate i times) programs in the classes $\mathcal{T}_i$, where i is the length of the input; thus, the first level of diagonalization 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 reach the machines computing their output within exponential time n^{n^k}.

Given a limit ordinal λ with the fundamental sequence $\lambda_0, \dots, \lambda_k, \dots$, 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

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

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 =DIAG(λ). Note that the previous definition requires that $f_{\lambda_i} \in \mathcal{C}_{\lambda_i}$, but there aren't other requirements on how the $\mathcal{C}$'s classes are built. In what follows, we introduce our transfinite hierarchy of programs, with an important restriction on the definition of the $\mathcal{C}$'s.

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 SREC($\mathcal{T}_\alpha$), if $\lambda = \alpha+1$; Notation: $\mathcal{T}_{\alpha+1}$=($\mathcal{T}_\alpha$, 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);

In [9] we proved that

1) each program $f(s, t, r)$ defined in $\mathcal{T}_\lambda$ ($\lambda < \omega^\omega$) can be computed by a register machine within time $B_\lambda(n)$;

2) a register machine which computes its output within time $O(B_\lambda(n))$ can be simulated by a program f in $\mathcal{T}_\lambda$.

We then have that

Theorem 3.1: 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 $\alpha < \omega^\omega$.

Two intertwined questions (not addressed in [9]) 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 $\mathcal{T}_{\lambda_1}$ of every sequence $\mathcal{T}_{\lambda_1}, \ldots, \mathcal{T}_{\lambda_n}, \ldots$, because it only has to write sequences of SREC's and DIAG's according to the definition on the ordinal λ_i, in order to write down the definition of $f_{\lambda_i} \in \mathcal{T}_{\lambda_i}$. This leads us to the second question: the results of any program in our language are lists of binary words, and they aren't other programs. This means that the enumerator must return the code of a program in $\mathcal{T}_{\lambda_i}$, 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_{λ_i}'s as codes of programs; this implies that an interpreter is concealed in the definition of diagonalization.

IV. Extending the hierarchy to the elementary functions

In [9], the classes $\mathcal{T}_\lambda$ have been defined between level 1 and ω^ω, reaching the programs computable within exponential time. This hierarchy can be extended up to the ordinal ϵ_0, with $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}} = sup\{\omega, \omega^\omega, \omega^{\omega^\omega}, \ldots\}$; in particular, $\mathcal{T}_{\epsilon_0}$ is the class of programs computable within time $O(B_{\epsilon_0}(n)) = O(n^{n^{n^{\cdots}}})$. Given that a function $f(n)$ is elementary if and only if it is computable in time bounded by $n^{n^{n^{\cdots}}}$ (see [14]), we have that $\mathcal{T}_{\epsilon_0}$ characterizes the class of the elementary functions. The proof of this result is an extension of the proof introduced in [9]; given $\lambda < \epsilon_0$, we can prove that

1) each program $f(s,t,r)$ in $\mathcal{T}_\lambda$ can be computed by a register machine within time in $O(B_\lambda(n))$;
2) every register machine computing its output within time $O(B_\lambda(n))$ can be simulated by a program f in $\mathcal{T}_\lambda$.

Lemma (1) is proved by structural induction on the ordinal λ, 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 λ 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 (2) is proved for each time-bounded register machines showing that, given a program $nxt_M \in \mathcal{T}_0$ that simulates the transition between the machine's configurations, a program in the appropriate class $\mathcal{T}_\lambda$ can be built as the iteration of nxt_M, in order to simulate the overall computation performed by the register machine itself. By (1) and (2) we have that

Theorem 4.1: 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 $\alpha < \epsilon_0$.

Our operators of predicative recursion and constructive diagonalization can be used to provide a fine hierarchy of classes between PTIME and $\mathcal{E}^3$. 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 [15], and Arai and Eguchi [16], capture the elementary functions alone, and not in a hierarchy of classes, using various forms of predicative recursion. Leivant [17], captures $\mathcal{E}^3$ using an extension to higher types of ramified recurrence.

V. Marion's diagonalization operator

In [10], the classes of functions $\mathcal{I}_k$ $(k = 0, 1, \ldots)$ are defined on the cartesian product of natural numbers. The class $\mathcal{I}_0$ is defined starting from a finite set of linear functions and closing it by composition and *flat recursion*, defined as follows:

$$\begin{cases} f(0,t) & = & g(t) \\ f(s+1,t) & = & h(s,t) \end{cases}$$

The class $\mathcal{I}_{k+1}$ contains all the functions defined in $\mathcal{I}_k$ and is closed by flat recursion and by (a version of) predicative recursion and composition. It is proved that each class $\mathcal{I}_k$ is the class of the functions computable within polynomial-time bound n^k; thus, $\bigcup_{k<\omega} \mathcal{I}_k$ is the class of polynomial time computable functions. The proof works by induction: if a function g belongs to $\mathcal{I}_0$, then a function F_k computing g^{n^k} can be defined as follows:

$$\begin{cases} F_0(0,x,y) & = & g(y) \\ F_{k+1}(0,x,y) & = & y \\ F_{k+1}(s+1,x,y) & = & F_k(x,x,F_{k+1}(s,x,y)) \end{cases}$$

F_k belongs to $\mathcal{I}_k$, for all k. Note how the variable x is used in order to jump from each class $\mathcal{I}_k$ to class $\mathcal{I}_{k+1}$. In the last line of the definition, the number of computations of the function F_{k+1} is set by using the first occurrence of x, and the information about how many times all the remaining functions $F_k, \ldots, F_0$ have to be computed is given by the second occurrence of x itself. Now the *small jump operator* is defined, in order to jump from every class to the next one:

$$\begin{cases} \Delta[F_k](0,x,y) & = & y \\ \Delta[F_k](r+1,x,y) & = & F_k(x,x,\Delta[F_k](r,x,y)) \end{cases}$$

It is proved in [10] that $\Delta[F_k](r,x,y) = F_{k+1}(r,x,y)$. If the parameter k assumes the role of a variable, Δ^ω is defined as follows:

$$\begin{cases} \Delta^\omega[g](0,n,x,y) & = & g(y) \\ \Delta^\omega[g](r+1,0,x,y) & = & y \\ \Delta^\omega[g](r+1,n+1,x,y) & = & \\ \quad = \Delta^\omega[g](r,x,x,\Delta^\omega[g](r+1,n,x,y)) & & \end{cases}$$

The operator Δ^ω, which is not a polynomial function, computes all the polynomial-time computable functions. Contrasting with our approach, no enumerating functions are used; starting from a function in I_0, the small jump operator defines a chaining of "growing" functions (w.r.t. the time complexity) and this chaining is used to jump out of polynomial class. The problem of how to define functions with higher time-complexity is not addressed.

VI. Conclusions and further work

In our previous work, we have used a version of safe recursion and constructive diagonalization to define a hierarchy of classes of programs $\mathcal{T}_\lambda$, with $0 \le \lambda < \omega^\omega$. 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, 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 this paper, we have given a hint of how to extend our hierarchy further, reaching the class $\mathcal{E}^3$ at level ϵ_0.

While predicative recursion has been studied thoroughly, we feel that the diagonalization operator as presented in this work deserves a more accurate analysis. In particular, we believe that it is as predicative as the recursion, and that it could be used to stretch the hierarchy of programs in order to capture the low Grzegorczyk classes above the elementary level.

References

[1] A. Grzegorczyk, Some classes of recursive functions. Rozprawy Matematyczne, vol. IV, 1953.

[2] H. E. Rose, Subrecursion: functions and hierarchies. Clarendon press, Oxford, 1984.

[3] 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, 1993, pp. 325–333.

[4] K.-H. Niggl, “The μ-measure as a tool for classifying computational complexity,” Archive for Mathematical Logic, vol. 39, no. 7, 2000, pp. 515–539.

[5] S. Bellantoni and K.-H. Niggl, “Ranking primitive recursion: the low Grzegorczyk classes revisited,” SIAM Journal on Computing, vol. 29, no. 2, 1999, pp. 401–4015.

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

[7] D. Leivant, Predicative recurrence and computational complexity I: word recurrence and polytime, in Feasible Mathematics II, P.Clote and J.Remmel (eds). Birkauser, 1994, pp. 320–343.

[8] H. Simmons, “The realm of primitive recursion,” Arch.Math. Logic, vol. 27, no. 2, 1988, pp. 177–188.

[9] E. Covino and G. Pani, A Slow-growing Hierarchy of Time-bounded Programs, in Advances in Intelligent Systems: Reviews’ Book Series, Vol. 1. IFSA Publishing, Barcelona, Spain, 2017, pp. 151–171.

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

[11] 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.

[12] M. Fairtlough and S. Weiner, Hierarchies of provably recursive functions, in Handbook of Proof theory, B. Samuel (ed), Studies in logic and the foundations of mathematics, vol. 137. Elsevier, Amsterdam, 1998, chapter 3, pp. 149–207.

[13] S. Caporaso, G. Pani, and E. Covino, “A predicative approach to the classification problem,” Journal of Functional Programming, vol. 11, no. 1, 2001, pp. 95–116.

[14] N. Cutland, Computability: an introduction to recursive function theory. Cambridge university press, Cambridge, 1980.

[15] 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, no. 1, 1997, pp. 109–125.

[16] T. Arai and N. Eguchi, “A new function algebra of EXPTIME functions by safe nested recursion,” ACM Transactions on Computational Logic, vol. 10, no. 4, 2009, pp. 1–19.

[17] D. Leivant, “Ramified recurrence and computational complexity III: higher type recurrence and elementary complexity,” Annals of Pure and Applied Logic, vol. 96, no. 1–3, 1999, pp. 209–229.