A Specialized Recursive Language for Capturing Time-Space Complexity Classes

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

***Abstract*—We provide a resource-free characterization of register machines that computes their output within polynomial time $O(n^k)$, by defining our version of *predicative recursion* and a related recursive programming language. Then, by means of some restriction on composition of programs, we define a programming language that characterize the register machines with a polynomial bound imposed over time and space complexity, simultaneously. A simple syntactical analysis allows us to evaluate the complexity of a program written in these languages.**

Keywords–Specialized computation languages, time-space classes, implicit computational complexity, predicative recursion.

I. INTRODUCTION

The definition of a complexity class is usually made 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 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.

The first characterization of this type was given by Cobham [4], in which the polynomial-time computable functions are exactly those functions generated by bounded recursion on notation; Leivan [8] and Bellantoni and Cook [1] gave other characterizations of PTIMEF. Several other complexity classes have been characterized by means of unlimited operators: see, for instance, Leivant and Marion [9] and Oitavem [10] for PSPACEF and the class of the elementary functions; Clote [3] for the definition of a time/space hierarchy between PTIMEF and PSPACEF; Leivant [6], [7] and [8] for a theoretical insight. 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 *unsafe* variables (or, following Simmons [11], between *dormant* and *normal* ones) in the recursion schemes. This distinction yields many forms of *predicative* recursion, in which the being-defined function cannot be used as counter into the defining one. 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. This approach represent a bridge between the complexity theory and the programming language theory; a mere syntactical inspection allows us to evaluate the complexity of a given program written in one of the previous mentioned specialized languages.

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 don't allow the renaming of variable z as x, and this implies that the step program h cannot assign the previous value 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' approach, or a *safe* one, following Bellantoni and Cook.

In Section II, starting from a natural definition of constructors and destructors over an algebra of lists, we give our definition of recursion-free programs and of the safe recursion scheme. In section III, we recall the definition of computation by register machines as provided by [8]. In section IV, 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 they are computable within time $O(n^k)$. We then 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, in Section V, 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. We have that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME. Even though this is a well-known result (see [8]), we use it to prove the second one (see also [5] and [9] for other approaches to the characterization of joint time-space classes). Both results are a preliminary step for an implicit classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [3]. In Section VI we summarize the results and give some hints about future work.

 ISBN: 978-1-61208-394-0

II. BASIC INSTRUCTIONS AND DEFINITION SCHEMES

In this section we introduce the basic operators of our programming language (for a different approach see [2]). The language is defined over a binary word algebra, with the only restriction that words are packed into lists, with the symbol © acting as a separator between words. In this way, we are able to handle a sequence of words as a single object.

A. Recursion-free programs and classes $\mathcal{T}_0$ and $\mathcal{S}_0$

$\mathbf{B}$ is the binary alphabet $\{0,1\}$. $a, b, a_1, \ldots$ denotes elements of $\mathbf{B}$, and $U, V, \ldots, Y$ denotes words over $\mathbf{B}$. $p, q, \ldots, s, \ldots$ denotes 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 $s = Y_1 0 Y_2 0 \ldots 0 Y_{n-1} 0 Y_n$ 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 will be denoted with f, g, h, and they will have the form $f(x, y, z)$, where some among the variables 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 adds 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 erases the rightmost digit of $(s)_i$, with $a = 0, 1$ and $i \geq 1$.

Constructors $\mathrm{C}_i^a(s)$ and destructors $\mathrm{D}_i(s)$ leave 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: $f = \mathrm{RNM}_{z/y}(g)$;
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: $f = \mathrm{SEL}_i^b(g, h)$.

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: f is obtained by *safe composition* of h and g in the variable u if it is obtained by substitution of h to u in g; if $u = z$, then x must be absent in h. Notation: $f = \mathrm{SCMP}_u(h, g)$.

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 SEL and SCMP.

Definition 2.6: 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 $f = \mathrm{SEL}_i^b(g, h)$, then $rog(f)$ is $\max(rog(g), rog(h))$;
3) if $f = \mathrm{SCMP}_u(h, g)$, then $rog(f)$ is $\max(rog(g), rog(h))$.

Definition 2.7: $\mathcal{S}_0$ 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 elements in $\mathcal{T}_0$ and in $\mathcal{S}_0$ modify their inputs according to the result of some test performed over a fixed number of digits. Moreover, elements in $\mathcal{S}_0$ cannot return values longer than their input.

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

Definition 2.8: 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

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

Notation: $f = \mathrm{SREC}(g, h)$.

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

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

Notation: $f = \mathrm{ITER}(h)$. We write $h^{|r|}(s)$ for $\mathrm{ITER}(h)(s, r)$ (i.e., the $|r|$-*th* iteration of h on s).

Definition 2.9: $\mathcal{T}_1$ (respectively, $\mathcal{S}_1$) is the class defined by closure under simple schemes and SCMP of programs in $\mathcal{T}_0$ (resp., $\mathcal{S}_0$) and programs obtained by one application of ITER to $\mathcal{T}_0$ (resp., $\mathcal{S}_0$).
Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE)
(resp., $\mathcal{S}_1$=($\mathcal{S}_0$, ITER($\mathcal{S}_0$); 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. The renaming of z as x is not allowed (see definitions 2.2 and 2.3), implying that the step program of a recursive definition cannot assign the recursive

 ISBN: 978-1-61208-394-0

call to the principal variable. This is the key of the time-complexity bound intrinsic into our programs, together with the limitations imposed to the renaming of variables.

Definition 2.10: 1) Given $f \in \mathcal{T}_1$, the *number of components* of f is $max\{i|\mathrm{D}_i$ 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 (see [8]), and we give the definition of computation within a given time (or space) bound.

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: 1) 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; M computes its output in space $O(q(n))$ if, during the whole computation, the global length of the contents of its registers is $O(q(n))$.
2) For each input $\vec{X}$ (with $|\vec{X}| = n$), M needs 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 given 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 definition 2.10 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.
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 $s_i \copyright F_j(\pi_0) \copyright \ldots \copyright F_j(\pi_k)$ 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 simple schemes; 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 safe composition and the selection can be simulated by our model of computation.
Step. Case 1. $f = \mathrm{ITER}(g)$, 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 $lh(g)$) 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 SCMP. 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 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 form $if\ state[i](s)\ then\ E_i$, where $state[i](s)$ is a test which is true if the state of M is s_i, and E_i is a modifier which updates the code of the state and the code of one among the registers, according to the definition of M. By means of $c-1$ SCMP's we define nxt_M^c in $\mathcal{T}_0$, which applies c times nxt_M to the word that encodes a configuration of M. We define in $\mathcal{T}_1$

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

$linsim_M(s,t)$ iterates $nxt_M(s)$ for $c|t|$ times, returning the code of the configuration which contains the final result of M. ■

IV. THE TIME HIERARCHY

In this section, we recall the definition of classes of programs $\mathcal{T}_1$ and $\mathcal{S}_1$; we define our hierarchy of classes of programs, and we state the relation with the classes of register machines, which compute their output within a polynomially-bounded amount of time.

Definition 4.1: ITER($\mathcal{T}_0$) denotes the class of programs obtained by one application of iteration to programs in $\mathcal{T}_0$. $\mathcal{T}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_0$ and programs in ITER($\mathcal{T}_0$).
$\mathcal{T}_{k+1}$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_k$ and programs in SREC($\mathcal{T}_k$).
Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE).
$\mathcal{T}_{k+1}$=($\mathcal{T}_k$, SREC($\mathcal{T}_k$); SCMP, SIMPLE).

Definition 4.2: ITER($\mathcal{S}_0$) denotes the class of programs obtained by one application of iteration to programs in $\mathcal{S}_0$. $\mathcal{S}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{S}_0$ and programs in ITER($\mathcal{S}_0$).
$\mathcal{S}_{k+1}$ 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: $\mathcal{S}_1$=($\mathcal{S}_0$, ITER($\mathcal{S}_0$); SCMP, SIMPLE).
$\mathcal{S}_{k+1}$=($\mathcal{S}_k$, 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 doesn't 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.

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

Proof: Base. $f \in \mathcal{T}_1$. The relevant case is when f is in the form ITER(h), with h a modifier in $\mathcal{T}_0$. In lemma 3.1 (case 1 of the step) 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 =SREC(g,h). The inductive hypothesis gives 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 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 (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 an f in $\mathcal{T}_k$.

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:

σ_0 :=ITER(nxt_M) and
σ_{n+1} :=IDT$_{z/y}(\gamma_{n+1})$, where γ_{n+1} :=SREC(σ_n, σ_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

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

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

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: f belongs to $\mathcal{T}_k$ if and only if f is computable by a register machine within time $O(n^k)$.

Proof: By lemma 4.1 and lemma 4.2. ■

We recall that register machines are polytime reducible to Turing machines; this implies that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME (see [8]).

V. THE TIME-SPACE HIERARCHY

In this section, we define a time-space hierarchy of recursive programs (see [4]), and we state the equivalence with the classes of register machines which compute their output with a simultaneous bound on time and space.

Definition 5.1: Given the programs g and h, f is obtained by *weak composition* of h in g if $f(x,y,z) = g(h(x,y,z),y,z)$. Notation: $f = \text{WCMP}(h,g)$.

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

Definition 5.2: 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 5.1: 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

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

and

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

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

Lemma 5.2: 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 5.2, 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 4.1 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 5.1 holds for g; hence, the space needed by M_g is at most n.
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 4.1, 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 4.1 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 counter. 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 5.3: A register machine which computes its output within time $O(n^p)$ and space $O(n^q)$ can be simulated by an $f \in \mathcal{TS}_{qp}$.

Proof: Let M be a register machine, whose computation is time-bounded by n^p and, simultaneously, it 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 4.1 there exists $h \in \mathcal{T}_q$ which simulates the behaviour of M_h, and there exists $g \in \mathcal{T}_p$ which 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 4.2, 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 5.1: 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 5.2 and lemma 5.3. ■

VI. CONCLUSIONS

We have provided a resource-free characterization of register machines that computes their output within polynomial time, and we have extended it to register machines that computes their output with a polynomial bound imposed on time and space, simultaneously; this is made by a version of predicative recursion and a related recursive programming language. A program written in this languages can be analyzed, and the complexity of the program is evaluated by means of this simple analysis. This result represents a preliminary step for a resource-free classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [3].

 ISBN: 978-1-61208-394-0

REFERENCES

[1] S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the poly-time functions," Computational Complexity, vol. 2, 1992, pp. 97-110.

[2] S. Caporaso, G. Pani, E. Covino, "A Predicative Approach to the Classification Problem," Journal of Functional Programming, vol. 11(1), Jan. 2001, pp. 95-116.

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

[4] A. Cobham, "The intrinsic computational difficulty of functions," in Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science. North-Holland, Amsterdam, 1962, pp. 24-30, Y. Bar-Hillel Ed.

[5] M. Hofmann, "Linear types and non-size-increasing polynomial time computation," Information and Computation, vol. 183(1), May 2003, pp. 57-85.

[6] D. Leivant, "A foundational delineation of computational feasibility," in Proceedings of the Sixth Annual IEEE symposium on Logic in Computer Science (LICS'91). IEEE Computer Society Press, 1991, pp. 2-11.

[7] 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). ACM, New York, 1993, pp. 325-333.

[8] D. Leivant, Ramified recurrence and computational complexity I: word recurrence and polytime. Birkauser, 1994, pp. 320-343, in Feasible Mathematics II, P. Clote and J. Remmel Eds.

[9] D. Leivant and J.-Y. Marion, "Ramified recurrence and computational complexity II: substitution and polyspace," Computer Science Logic, LNCS vol. 933, 1995, pp. 486-500, J. Tiuryn and L. Pocholsky Eds.

[10] 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 10.1, 1997, pp. 109-125.

[11] H. Simmons, "The realm of primitive recursion," Arch.Math. Logic, vol. 27, 1988, pp. 177-188.

 ISBN: 978-1-61208-394-0