Journal of Universal Computer Science, vol. 8, no. 1 (2002), 75-84
submitted:8/9/00, accepted: 30/4/01, appeared: 28/1/02 © J.UCS

An Implicit Recursive Language for the Polynomial Time-space Complexity Classes

Emanuele Covino [1]
(Dipartimento di Informatica dell'Università di Bari, Italy
covino@di.uniba.it)
Giovanni Pani [2]
(Dipartimento di Informatica dell'Università di Bari, Italy
pani@di.uniba.it)

Abstract: We define a language over an algebra of words by means of a version of *predicative recursion*, and we prove that it represents a resource-free characterization of the computations performed by a register machine in time $O(n^k)$, for each finite k; starting from this result, and by means of a restricted form of composition, we give a characterization of the computations of a register machine with a polynomial bound simultaneously imposed over time and space complexity.
Keywords: time-space classes, implicit computational complexity, predicative recursion.
Category: F.1.3

1 Introduction

In the recent years, Leivant, Bellantoni & Cook and many others have given different characterization of PTIME, showing that this complexity class can be captured by means of a *ramified* (or *safe*, or *unbounded*) form of recursion (see [Leivant, 94] and [Bellantoni, Cook, 92]). Starting from this result, several other complexity classes have been characterized (see [Simmons, 88], [Leivant, 91], [Leivant, 93] and [Leivant, 94] 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 appropriately specify the role of the variables in the recursion scheme, pointing out the difference between *safe* and *unsafe* variables (or, following [Simmons, 88], between *dormant* and *normal* ones). The distinction yields many forms of *predicative recurrence*, in which the being-defined function cannot be used as counter into the defining one. This approach should represent a bridge between the complexity theory and the programming language theory; the mere syntactical inspection of a program allows us to evaluate the complexity of that program.

Our version of the *safe recursion* scheme on a binary word algebra is such that $f(x, y, za) = h(f(x, y, z), y, za)$, where x, y, z are respectively devoted to the role of auxiliary variable, parameter and recursion variable of f; no other type of variables can be used, and the identification of z with x is not allowed.

[1] This author was supported in part by the "Giovani ricercatori" research project and by the "Centro Interdipartimentale Logica e Applicazioni", University of Bari.
[2] This author was supported in part by the "Centro Interdipartimentale Logica e Applicazioni", University of Bari.

We define the hierarchy of classes of programs $\{\mathcal{T}_k\}_{k<\omega}$, where $\mathcal{T}_1$ is a characterization of the class of register machines which compute their output in linear time, and $\mathcal{T}_{k+1}$ is the class of programs obtained by exactly one application of safe recursion to elements in $\mathcal{T}_k$. We then restrict $\{\mathcal{T}_k\}_{k<\omega}$ to the hierarchy $\{\mathcal{S}_k\}_{k<\omega}$, whose elements are the classes of programs computable by a register machine in at most linear space; this classes are similar to the classes of non-size-increasing functions showed in [Hofmann, 99], but we obtain them without the use of types. By means of a restricted form of composition, we define a polytime-space hierarchy $\{\mathcal{TS}_{qp}\}_{qp<\omega}$, such that, for all $k, q, p \geq 1$, we have:

1. $\mathcal{T}_k$ is the class of programs computable by a register machine within time $O(n^k)$;
2. $\mathcal{TS}_{qp}$ is the class of programs computable by a register machine within time $O(n^p)$ and, simultaneously, space $O(n^q)$.

A corollary of the first result is that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME. Even though this is a well-known result (see [Leivant, 94]), we use it to show the second one (see also [Hofmann, 99] and [Leivant, Marion, 00] for other approaches to the characterization of joint time-space classes). We feel that this is a preliminary step in the implicit classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [Clote, 92].

2 Basic instructions and definition schemes

We define in this section the language we will use to provide our characterizations (for a similar approach see [Caporaso, Pani, Covino]). It is a natural language over a binary word algebra, with the only restriction that words are packed into lists, in which each word is divided from the following one by means of the digit 0. In this way, we are able to handle a sequence of words as a single object. The reader should consider this ternary word as a list (like those in Lisp) in which each zero plays the role of a comma.

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

T is the ternary alphabet $\{0, 1, 2\}$. $p, q, \ldots, s, \ldots$ are the word 0 or words over **T** not beginning and not ending with 0; ϵ is the empty word. **B** is the binary alphabet $\{1, 2\}$. $U, V, \ldots, Y$ are words over **B**. $a, b, a_1, \ldots$ are letters of **T** or **B**. The *i-th component* $(s)_i$ of a word s in the form $Y_1 0 Y_2 0 \ldots 0 Y_{n-1} 0 Y_n$ is Y_i. $|s|$ is the length of the word s, that is the number of letters occurring in s.
We write x, y, z for the variables used in our programs, denoted with f, g, h, and we write u for one among x, y, z. The objects of our language will have the form $f(x, y, z)$, where some among the variables may be absent.

Definition 1. The *basic instructions* are:

1. the *identity* $\mathrm{I}(u)$, which returns the value s assigned to u;
2. the *constructors* $\mathrm{C}_i^a(u)$, which, when s is assigned to u, adds the digit a at the right of the last digit of $(s)_i$, with $a = 1, 2$ and $i \geq 1$;
3. the *destructors* $\mathrm{D}_i(u)$, which, when s is assigned to u, erases the rightmost digit of $(s)_i$, with $a = 1, 2$ 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; see the section 3 for a note on the complexity of constructors and destructors.

Example 1. The word $s = 120220011$ had to be understood as the sequence $12, 22, \epsilon, 11$; in this case, $|s| = 9$ and, for example, $(s)_2 = 22$. We also have $\mathrm{C}_1^1(12022) = 121022$, $\mathrm{D}_2(1010) = 100$, $\mathrm{D}_2(100) = 100$.

Definition 2. Given the programs g and h, f is obtained by *simple schemes* if it is obtained by:

1. *identification* of x as y in g, that is, f is the result of the assignment of the value of y to all occurrences of x into g. Notation: $f = \mathrm{IDT}_{x/y}(g)$;
2. *identification* of z as y in g, that is, f is the result of the assignment of the value of z to all occurrences of x into g. Notation: $f = \mathrm{IDT}_{z/y}(g)$;
3. *branching* in g and h, when for all s, t, r assigned to x, y, z 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 $b = 1, 2$. Notation: $f = \mathrm{BRANCH}_i^b(g,h)$.

Example 2. $f = \mathrm{IDT}_{x/y}(g)$ implies $f(t,r) = g(t,t,r)$. Similarly, $f = \mathrm{IDT}_{z/y}(g)$ implies $f(s,t) = g(s,t,t)$. Let s be the word 1102121, and $f = \mathrm{BRANCH}_2^1(g,h)$; we have $f(s,t,r) = g(s,t,r)$, since the rightmost digit of $(s)_2$ is 1.

Definition 3. Let $\mathcal{C}$ a class of programs. f is obtained by *safe composition* of h and g (with respect to $\mathcal{C}$) in the variable u if it is obtained by substitution of h to u in g, with g or h in the class $\mathcal{C}$ and, when $u = z$, with x absent in h. Notation: $f = \mathrm{SCMP}_u(h,g)$.

Definition 4. A *modifier* is obtained by the safe composition of a sequence of constructors and a sequence of destructors.

Definition 5. $\mathcal{T}_0$ is the class of programs defined by closure of modifiers under BRANCH and SCMP.

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

Definition 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.

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

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

Definition 9. $\mathcal{T}_1$ (respectively, $\mathcal{S}_1$) is the class defined by closure under simple schemes and SCMP of programs obtained by one application of ITER to $\mathcal{T}_0$ (resp. $\mathcal{S}_0$).

Throughout this paper we will call x, y and z the auxiliary variable, the parameter, and the principal variable of a program obtained by means of the previous recursion scheme. Note that since identification of z as x is not allowed (see definitions 2 and 3), the step h cannot assign the previous value of the object being defined by SREC to the recursion variable: hence, we always know in advance the number of recursive calls of the step. We obtain that z is a *dormant* variable, according to the Simmons' approach, or a *safe* one, following Bellantoni&Cook.

Definition 10.
1. Given $f \in \mathcal{T}_1$, the *number of components* of f is $max\{i|\text{D}_i$ or C_i^a or BRANCH_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: $lh(f)$.

3 Computation by register machines

We recall in this section the definition of register machine (see [Leivant, 94]), and we give the definition of computation with a given time (or space) bound.

Definition 11. 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) a **branching** $s_i \pi_j s_{i_1} \ldots s_{i_k}$, such that when M is in the state s_i, switch 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 $\mathbf{c}_1, \ldots, \mathbf{c}_k$;

(b) a **constructor** $s_i \pi_{j_1} \dots \pi_{j_{r_i}} \mathbf{c}_i \pi_l s_r$, 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 $\pi_{j_1} \dots \pi_{j_{r_i}}$, and switch to s_r;

(c) a **p-destructor** $s_i \pi_j \pi_l s_r$ $(p \leq max(r_i)_{i=1 \dots k})$, 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.

A *configuration* of M is a pair (s, F), where $s \in S$ and $F : \Phi \rightarrow \mathbf{A}$. 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 $\mathbf{X} = X_1, \dots, X_p$ with output $\mathbf{Y} = Y_1, \dots, Y_q$ 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 12. 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 $(q)_1, \dots, (q)_{\#(f)}$ on input $(s)_1, \dots, (s)_{\#(f)}, (t)_1, \dots, (t)_{\#(f)}, (r)_1, \dots, (r)_{\#(f)}$.

Definition 13. 1. For each input $\mathbf{X}$ (with $|\mathbf{X}| = n$), M computes its output within time $O(p(n))$ if its computation on t 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 $\mathbf{X}$ (with $|\mathbf{X}| = n$), 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 10 and 12, 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 branchings (that is, depends on $\#(f)$, a constant value); 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.
Unlike the usual operators *cons*, *head* and *tail* over Lisp-like lists, our constructors and destructors can have access to any component of a list, according to definition 1. Hence, their computation by means of the previously defined 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 $\langle s_i, F_j(\pi_0), \dots, F_j(\pi_k) \rangle$ for the word that encodes a configuration (s_i, F_j) of M, where each comma is a zero and each component is a binary word over $\{1, 2\}$.

Lemma 14. Let f be a program; the following are equivalent:

1. f belongs to $\mathcal{T}_1$.

2. 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 f.
Base. $f \in \mathcal{T}_0$. We note that a modifier g can be computed in time $lh(g)$, i.e. by a machine running over a constant number of configurations; the result follows, since the safe composition and the branching can be easily simulated by our model of computation.
Step. Case 1. $f = $ITER$(g)$, with $g \in \mathcal{T}_0$. We have $f(s,r) = g^{|r|}(s)$. A register machine M_f can be defined as follows: $(s)_i$ is stored in the register π_i $(i = 1 \dots \#(f))$ and $(r)_j$ is stored in the register π_j $(j = \#(f)+1 \dots 2\#(f))$; it computes g (in time $lh(g)$) for $|r|$ times. Each time g is computed, M_f erases one digit from one of the registers $\pi_{\#(f)+1} \dots \pi_{2\#(f)}$; 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 $\mathcal{T}_1$. Define a program $nxt_M \in \mathcal{T}_0$, such that nxt_M operates on input $s = \langle s_i, F_j(\pi_0), \dots, F_j(\pi_k)\rangle$ and it has the form $if\ state[i](s)\ then\ E_i$, where $state[i](s)$ is a test which is true iff 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^c_M in $\mathcal{T}_0$, which applies c times nxt_M to the word that encodes a configuration of M. We define now in $\mathcal{T}_1$

$$\begin{cases} linsim_M(x,a) = x \\ linsim_M(x,za) = nxt^c_M(linsim_M(x,z)) \end{cases}$$

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

4 The time hierarchy

In this section we define our time-hierarchy and we state the relation between this hierarchy and the classes of register machines which compute their output within a given amount of time. We write:

1. $\Sigma(\mathcal{C})$ for the class of programs obtained by one application of the scheme Σ to the class $\mathcal{C}$;
2. $(\mathcal{C};\Sigma)$ for the closure of $\mathcal{C}$ under Σ.

Writing SIMPLE for the set of the simple schemes of definition 2, we have

Definition 15. 1. $\mathcal{T}_1 = ($ITER$(\mathcal{T}_0)$; SCMP, SIMPLE$)$;
2. $\mathcal{T}_{k+1} = ($SREC$(\mathcal{T}_k)$; SCMP, SIMPLE$)$.

This means that $\mathcal{T}_{k+1}$ is obtained by one application of the safe recursion scheme to the previously defined class $\mathcal{T}_k$, and by closure under safe composition and simple schemes.

Definition 16. 1. $\mathcal{S}_1 = (\text{ITER}(\mathcal{S}_0); \text{SCMP}, \text{SIMPLE})$;
2. $\mathcal{S}_{k+1} = (\text{SREC}(\mathcal{S}_k); \text{SIMPLE})$.

This means that $\mathcal{S}_{k+1}$ is obtained by one application of the safe recursion scheme to the previously defined class $\mathcal{S}_k$, and by closure under the simple schemes. Hierarchy $\{\mathcal{S}_k\}_{k<\omega}$ is a version of $\{\mathcal{T}_k\}_{k<\omega}$, in which each program returns a result whose length is *exactly* bounded by the length of the input; this doesn't happen if we close the class $\mathcal{S}_k$ under SCMP. We will use this result to evaluate the space complexity of our programs.

Lemma 17. 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 $\text{ITER}(h)$, with h a modifier in $\mathcal{T}_0$. In lemma 14 (case 1 of the step) we have proved that $f(s,r)$ is computed within time $|r|lh(g)$; hence, we have the thesis.
Step. $f \in \mathcal{T}_{p+1}$. The most significant case is when $f = \text{SREC}(g,h)$. 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 $a_1 \dots a_{|r|}$; 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 (i.e., the result of the previous recursive step has to be stored always in the same register). By inductive hypothesis, M_f needs time $|s| + lh(g)(|t|)^p$ in order to compute g; for the computation of the first step of h, it 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 18. 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:
$\sigma_0 := \text{ITER}(nxt_M)$ and $\sigma_{n+1} := \text{IDT}_{z/y}(\gamma_{n+1})$, where $\gamma_{n+1} := \text{SREC}(\sigma_n, \sigma_n)$.
We have that

$$\sigma_0(s,t) = nxt_M^{|t|}(s),$$

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

and

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

In particular we have

$$\sigma_1(s,t) = \gamma_1(s,t,t) = \underbrace{\sigma_0(\sigma_0(\ldots\sigma_0(s,t)\ldots))}_{|t|\ times} = nxt_M^{|t|^2}$$
$$\sigma_2(s,t) = \gamma_2(s,t,t) = \underbrace{\sigma_1(\sigma_1(\ldots\sigma_1(s,t)\ldots))}_{|t|\ times} = nxt_M^{|t|^3}$$

By simple 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 19. Let f be a program; the following are equivalent:

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

Proof. By lemma 17 and lemma 18.

We recall that register machines are polytime reducible to Turing machines; this implies that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME (see [Leivant, 94]).

5 The time-space hierarchy

In this section we define a time-space hierarchy (see [Cobham, 62]), and we state the equivalence with the classes of register machines which compute their output within a bounded amount of time and space.

Definition 20. Given the programs g and h, f is obtained by *weak composition* of h in g if $f(x,y,z) = g(h(x,y,z),y,z)$. Notation: $f =$WCMP(h,g).

Note that the difference between the *weak* form of composition and the *safe* one is that the former is applied to programs which can be in any class, while the latter applies only when g or h belongs to $\mathcal{T}_0$.

Definition 21. For all p,q, $\mathcal{TS}_{qp}$ 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 22. For all f in $\mathcal{S}_p$, we have $|f(s,t,r)| \leq \max(|s|,|t|,|r|)$.

Proof. By induction on p. Base. $f \in \mathcal{S}_1$ and f is defined by iteration of g in $\mathcal{S}_0$ (that is, $rog(g) \leq 0$); in all other cases the result trivially follows. We have, by induction on r, $|f(s,a)| = |s|$, and $|f(s,ra)| = |g(f(s,r))| \leq |f(s,r)| \leq \max(|s|,|r|)$.
Step. Given $f \in \mathcal{S}_{p+1}$, defined by SREC in g and h in $\mathcal{S}_p$, we have

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

and

$$\begin{aligned} |f(s,t,ra)| &= |h(f(s,t,r),t,ra)| && \text{by definition of } f \\ &\leq |\max(|f(s,t,r)|,|t|,|ra|)| && \text{by inductive hypothesis on } h \\ &\leq |\max(\max(|s|,|t|,|r|),|t|,|ra|)| && \text{by induction on } r \\ &\leq |\max(|s|,|t|,|ra|)|. \end{aligned}$$

Lemma 23. 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 21, 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 19 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 22 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 17, 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 17 shows, in general, that the time used by a register machine to compute an element of our language is 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 24. 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 19 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 18, we write $\sigma_{n-1} \in \mathcal{S}_n$, 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 25. Let f be a program; the following are equivalent:

1. f belongs to $\mathcal{TS}_{qp}$.
2. f is computable by a register machine within time $O(n^p)$ and space $O(n^q)$.

Proof. By lemma 23 and lemma 24.

References

[Bellantoni, Cook, 92] S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the poly-time functions.* Computational Complexity 2(1992)97-110.

[Caporaso, Pani, Covino] S. Caporaso, G. Pani, E. Covino, *Predicative Recursion, Constructive Diagonalization and the Elementary Functions*, Workshop in Implicit Computational Complexity (ICC'99), Trento.

[Clote, 92] P. Clote, *A time-space hierarchy between polynomial time and polynomial space.* Math. Sys. The. 25(1992)77-92.

[Cobham, 62] A. Cobham, *The intrinsic computational difficulty of functions.* Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, pages 24-30, North-Holland, Amsterdam, 1962.

[Hofmann, 99] M. Hofmann, *Linear types and non-size-increasing polynomial time computation.* Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99), 464-473.

[Leivant, 91] D. Leivant, *A foundational delineation of computational feasibility.* Proc. of the 6th Annual IEEE symposium on Logic in Computer Science, (IEEE Computer Society Press, 1991), 2-18.

[Leivant, 93] D. Leivant, *Stratified functional programs and computational complexity.* Conference Records of the 20th Annual ACM Symposium on Principles of Programming Languages, New York, 1993, ACM.

[Leivant, 94] D. Leivant, *Ramified recurrence and computational complexity I: word recurrence and polytime.* P.Clote and J.Remmel (eds), Feasible Mathematics II (Birkauser, 1994), 320-343.

[Leivant, Marion, 00] D. Leivant and J.-Y. Marion, *A characterization of alternating log time by ramified recurrence.* Theoretical Computer Science, 236(2000).

[Simmons, 88] H. Simmons, *The realm of primitive recursion.* Arch.Math. Logic, 27(1988)177-188.