Extending the Implicit Computational Complexity approach to the sub-elementary time-space classes

Emanuele Covino, Giovanni Pani, and Salvatore Caporaso

Dipartimento di Informatica dell'Università di Bari
(covino|pani|caporaso)@di.uniba.it

Abstract. A resource-free characterization of some complexity classes is given by means of the *predicative recursion* and *constructive diagonalization* schemes, and of restrictions to substitution. Among other classes, we predicatively harmonize in the same hierarchy PTIMEF, the class $\mathcal{E}$ of the elementary functions, and classes DTIMESPACEF(n^p, n^q).
Keywords: time-space classes, implicit computational complexity, elementary functions.

1 Introduction

Position of the problem. The standard definition of a complexity class involves the definition of a bound imposed on time and/or space resources used by a Turing Machine during its computation; a different approach characterizes complexity classes by means of limited recursive operators. The first characterization of this type of a small complexity class was given by Cobham [8], who showed that the polytime functions are exactly those functions generated by *bounded recursion on notation*; however, a *smash* initial function $2^{|x||y|}$ was used to provide space enough.
Leivant [12] and Bellantoni & Cook [2] gave the characterizations of PTIMEF; several other complexity classes have been characterized by means of unlimited operators (see [13, 5] for PSPACEF, [4] for PTIME, PSPACE (languages), PH and its elements, [1, 3] for $\mathcal{NP}$, [14] for PSPACEF and the class of the elementary functions, [7] for the definition of a time-space hierarchy between PTIMEF and PSPACEF). 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 [17], between *dormant* and *normal* ones) in the recursion schemes. This distinction yields many forms of *predicative recurrence*, in which the being-defined function cannot be used as counter into the defining one.

Statement of the result. We define a *safe recursion* scheme SREC on a ternary word algebra, such that $f(x, y, za) = h(f(x, y, z), y, za)$, where x, y, z are, respectively, the auxiliary variable, the parameter and the recursion variable; no

other type of variables can be used, and the identification of z with x is not allowed. We also define a *constructive diagonalization* scheme CDIAG, such that $f(n) = \{e(n)\}(n)$, where $\{m\}$ is the standard Klenee's notation for the function coded by m and e is a given enumerator.
Starting from a characterization $\mathcal{T}_1$ of LINTIMEF, and from an assignment of fundamental sequences λ_n for ordinals $\lambda < \epsilon_0$ (see [16] for further details on ordinals), we define the hierarchy $\{\mathcal{T}_\alpha\}_{\alpha<\epsilon_0}$ as follows:

1. at each successor ordinal, $\mathcal{T}_{\alpha+1}$ is the class of all functions obtained by one application of safe recursion to functions in $\mathcal{T}_\alpha$;
2. at each limit ordinal λ, $\mathcal{T}_\lambda$ is the class of all functions obtained by one application of constructive diagonalization in an enumerator $e \in \mathcal{T}_{\lambda_1}$, such that $\{e(n)\} \in \mathcal{T}_{\lambda_n}$.

Given an ordinal α in Cantor normal form, $B_\alpha(n)$ is the $\max(2,n)^{clps(\alpha,n)}$, where $clps(\alpha, n)$ is the result of replacing ω by n in α. We have that:

1. for all finite k, $\mathcal{T}_k =$DTIMEF(n^k);
2. for all $\omega < \alpha < \epsilon_0$, DTIMEF$(B_\alpha(n)) \subseteq \mathcal{T}_\alpha \subseteq$ DTIMEF$(B_\alpha(n + O(1)))$.

Thus, $\bigcup_{\beta<\omega} \mathcal{T}_\beta =$PTIMEF and $\bigcup_{\beta<\epsilon_0} \mathcal{T}_\beta = \mathcal{E}$, the elementary functions.
In analogy with $\{\mathcal{T}_\alpha\}_{\alpha<\epsilon_0}$ we define a hierarchy $\{\mathcal{S}_\alpha\}_{\alpha<\epsilon_0}$ of *not space-increasing* functions and, by means of a restricted form of substitution, we define a time-space hierarchy $\{\mathcal{TS}_{qp}\}_{qp<\omega}$, such that

3. $\mathcal{TS}_{qp} =$DTIMESPACEF(n^p, n^q).

2 Costants, basic functions and definition schemes

2.1 Recursion free functions

T is the ternary alphabet $\{0, 1, 2\}$. $p, q, \ldots, s, \ldots$ are the word 0 or words over **T** not beginning 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 of the form $Y_n 0 Y_{n-1} 0 \ldots 0 Y_2 0 Y_1$ is Y_i. The rationale of this definition is that ternary words are actually handled as tuples of binary words, with the zeroes playing the role of commas. $|s|$ is the length of the word s.
We denote with x, y, z variables used as, respectively, auxiliary, parameter and principal in the construction of the current function. $f(s, t, r)$ is the result of assigning words s, t, r to x, y, z. By a notation like $f(x, y, z)$ we always allow some among the indicated variables to be absent.

Definition 1. Given $i \geq 1$, $a = 1, 2$ and $u = x, y, z$, the *initial functions* are the following unary functions:

1. the *identity* I(u), which returns the value s assigned to u; sometimes we write s instead of I(s);

2. the *constructor* $\mathrm{C}_i^a(u)$, which, when s is assigned to u, adds the digit a at the right of the last digit of $(s)_i$, ; it leaves s unchanged if $(s)_i$ is a single letter;
3. the *destructor* $\mathrm{D}_i(s)$, which, when s is assigned to u: (a) erases the rightmost digit of $(s)_i$, if $(s)_i$ is not a single digit; (b) returns 0, otherwise. Constructors and destructors leave s unchanged if it has less than i components.

Example 1. $\mathrm{C}_1^1(12022) = 120221$; $\mathrm{D}_2(1010) = 100$; $\mathrm{D}_2(100) = 100$.

Definition 2. Given $i \geq 1$ and $b = 1, 2$, we have the following *simple schemes*:

1. $f = \mathrm{IDT}_x(g)$ is the result of the *identification* of x as y in g;
2. $f = \mathrm{IDT}_z(g)$ is the result of the *identification* of z as y in g;
3. $f = \mathrm{ASG}_u(s, g)$ is the result of the *assignment* of s to the variable u in g;
4. $f = \mathrm{BRANCH}_i^b(g, h)$ is defined by *branching* in g and h if for all s, t, r we have $f(s, t, r) =$ *if* the rightmost digit of $(s)_i$ is b *then* $g(s, t, r)$ *else* $h(s, t, r)$.

Example 2. $f = \mathrm{IDT}_x(g)$ implies $f(t, r) = g(t, t, r)$. Similarly, $f = \mathrm{IDT}_z(g)$ implies $f(s, t) = g(s, t, t)$. Let s be the word 110212, 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. A *modifier* is the sequence composition of n constructors and m destructors.

Definition 4. Class $\mathcal{T}_0$ is the closure of modifiers under BRANCH_i^b and composition.

Definition 5. 1. The *rate of growth* $rog(g)$ of a modifier g is $n - m$, where n and m are respectively the number of constructors and destructors occurring in g.
2. For all $f \in \mathcal{T}_0$ built-up by means of some branchings from modifiers $g_1, \ldots, g_k$, we have $rog(f) := \max_{i \leq k} rog(g_i)$.

Definition 6. Class $\mathcal{S}_0$ is the class of functions in $\mathcal{T}_0$ with non-positive rate of growth, that is $\mathcal{S}_0 = \{f \in \mathcal{T}_0 | rog(f) \leq 0\}$.

Notice that all functions in $\mathcal{T}_0$ are unary, and they modify their inputs according to the result of some test performed over a fixed number of digits. Functions in $\mathcal{S}_0$, or their iteration, cannot return values longer than their input.

2.2 Safe recursion and diagonalization

Definition 7. $f = \mathrm{SREC}(g, h)$ is defined by *safe recursion* in the *basis function* $g(x, y)$ and in the *step function* $h(x, y, z)$ if for all s, t, r we have

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

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

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

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

Definition 8. f =CDIAG(e) is defined by *constructive diagonalization* in the *enumerator* e if for all s, t, r we have

$$f(s,t,r) = \{e(r)\}(s,t,r)$$

where $\{m\}$ denotes the function coded by m.

Definition 9. f =CMP(h,g) is defined by *composition* of h and g if $f(u) = g(h(u))$, with h or g in $\mathcal{T}_0$.

Definition 10. Class $\mathcal{T}_1$ (resp. $\mathcal{S}_1$) is the closure under simple schemes and CMP of functions obtained by one application of ITER to $\mathcal{T}_0$ (resp. $\mathcal{S}_0$).

Note that since identification of z as x is not allowed (see definition 2), the step function cannot assign the previous value of the function being defined by SREC to the recursion variable. Thus, we obtain that z is a *dormant* variable, according to the Simmons' approach (see [17]), or a *safe* one, following Bellantoni&Cook: we always know in advance the number of recursive calls of the step function, and this number will never be affected by the previous values of f.

Example 3. By a sequence of SREC's we now define a sequence of functions g_n which, at m, *compute in unary* m^n, that is, such that $|g_n(a,t)| \leq |t|^n$. We then use the fact that the generation of the g_n's is uniform in n to define by CDIAG a function f_ω which computes in unary m^m. That is:

$$g_1 := \text{CMP}(\text{ITER}(\text{C}_1^1), \text{D}_1); \quad f_{n+1} := \text{SREC}(g_n, g_n);\ g_{n+1} := \text{IDT}_z(f_{n+1}).$$

We have

$$\begin{cases} g_1(s,a) &= g_n(s,t) \\ g_1(s,ra) &= \text{C}_1^1(g_1(s,r)) \end{cases} ; \begin{cases} f_{n+1}(s,t,a) &= g_n(s,t) \\ f_{n+1}(s,t,ra) &= g_n(f_{n+1}(s,t,r),t). \end{cases}$$

By induction on n and r one sees that we have $|f_{n+1}(s,t,r)| = |s| + |t|^n|r|$ and, therefore, $|g_n(a,t)| \leq |t|^n$. Assume now defined a function $e \in \mathcal{T}_1$ such that $e(r) = \ulcorner g_{|r|} \urcorner$. If we define f_ω :=CDIAG(e), we have $|f_\omega(a,t,t)| = g_{|t|}(a,t) = |t|^{|t|}$.

2.3 Standard computability

As model of computation we use the *push-down* Turing machine, and we give the definition of inclusion between classes of TM's and classes of functions.

Definition 11. A *binary push-down* Turing machine M is defined as follows:

– M has k push-down tapes over the alphabet $\mathbf{B}$ and $m+1$ states (0 denotes the final state, 1 the initial state);
– the description of M consists of m rows of the type

$$R_i = (i, j(i), i_1, j_1, I_1, i_2, j_2, I_2, i_3),$$

(one for each non-final state) where:
(i) i, i_1, i_2, i_3 are states, with $i \neq 0$;
(ii) $j(i), j_1, j_2$ are tapes, with j a given function;
(iii) I_1, I_2 are defined on the set of instructions $\{pop, push\ 1, push\ 2\}$
– each row of M should be intended as
if the current state is i then
    if $top(j(i)) = 1$ then enter i_1
        apply I_1 to j_1;
    if $top(j(i)) = 2$ then enter i_2
        apply I_2 to j_2;
    if $(j(i))$ is empty then enter i_3.

Given a push-down TM M with k tapes, $T_i = X$ means that the content of tape T_i is the binary word X.
Note that the previous model and the ordinary Turing machine model are equivalent, with respect to the order of time needed to compute a given function. In fact, let M be an ordinary TM, with n tapes unlimited to the left, alphabet $\mathbf{B}$ and a fixed set of states. M can be simulated by a push-down TM N with $2n$ tapes and the same number of states, which stores the content of the i-th tape of M at the left of the observed symbol in its tape $2i$-1, and the part at the right in its tape $2i$. If M moves left on tape i, N pops a symbol from tape $2i$-1 and pushes it into tape $2i$; similarly, if M moves right on tape i, N pops a symbol from tape $2i$ and pushes it into tape $2i$-1.

Definition 12. 1. A push-down Turing machine M, by input $s = X_1 0 \ldots 0 X_n$, *standard computes* $q = Y_1 0 \ldots 0 Y_m$ ($M(s) =_{sc} q$) if starts with $T_i = X_i$ $(1 \leq i \leq n)$ and stops with $T_j = Y_j$ $(1 \leq j \leq m)$.
2. M *standard computes* the function f ($M =_{sc} f$) if $f(s) = q$ implies that $M(s) =_{sc} q$.

It is natural to observe that the number of tapes of the Turing machine which standard computes a function must be independent from the number of components of its input; following the previous definition, a new Turing machine should be defined for each possible number of components of the input of f. However, when we are defining a TM that standard computes a function f, we need a number of tapes that depends only on the maximum number of components of the input that f can manipulate or check with a constructor, a destructor or a branching.

Definition 13. 1. Given $f \in \mathcal{T}_1$, we define the *number of components* of f (denoted with $\#(f)$) as $max\{i | \mathrm{D}_i$ or a C_i^a or a BRANCH_i^b occurs in $f\}$.

2. Given a function f, we define the *length* of f (denoted with $lh(f)$) as the number of destructors, destructors and defining schemes occurring in its construction.

Definition 14. Given the class of time-bounded push-down TM's DTIME$(p(n))$ and given $\mathcal{C}$ a class of functions, we say that

1. DTIME$(p(n)) \subseteq \mathcal{C}$ if for all $M \in$DTIME$(p(n))$ there exists a function $f \in \mathcal{C}$ such that $M(s) =_{sc} f(s)$;
2. $\mathcal{C} \subseteq$DTIME$(p(n))$ if for all $f \in \mathcal{C}$ there exists a push-down TM M with $\#(f)$ tapes such that, for all s, t, r, M returns $f(s,t,r)$ in time $p(|s|+|t|+|r|)$.

Let $\mathbf{U}$ be the finite alphabet that we use to write our functions; the code of a function f is obtained by concatenation of the codes for the letters of $\mathbf{U}$ which compose f; the *arity* associated with each letter ensures unique parsing.

Definition 15. The code $\lceil L \rceil$ of the i-th letter L of $\mathbf{U}$ is $2^{i+1}1$. If the arity of $L \in \mathbf{U}$ is n then $\lceil E_n \rceil \ldots \lceil E_1 \rceil \lceil L \rceil$ codes the expression $LE_1 \ldots E_n$. We write $\langle E_1, \ldots, E_n \rangle$ for $\lceil E_n \rceil \ldots \lceil E_1 \rceil$.

In the same way we can define the code of a push-down TM.

Definition 16. Let M be a binary push-down TM with k tapes and $m+1$ states.

1. The code of the row $R_i = (i, j(i), i_1, j_1, I_1, i_2, j_2, I_2, i_3)$ $(1 \leq i \leq m)$ is the word
$$\langle i, j(i), i_1, j_1, I_1, i_2, j_2, I_2, i_3 \rangle.$$
2. The code of M is the word $\langle R_1, \ldots, R_m \rangle$.
3. An instantaneous description of M is coded by the word $X_1, \ldots, X_k state$, where each X_i encodes the i-th tape of M, and *state* encodes the current state.

Lemma 1. $\mathcal{T}_1 =$DTIMEF(n).

Proof. We first show (by induction on the construction of f) that each function $f \in \mathcal{T}_1$ can be computed by a push-down TM in time $lh(f)n$. Base. $f \in \mathcal{T}_0$. The result follows from the definition of initial functions and CMP (see definition 4).
Step. Case 1. $f =$ITER(g), with $g \in \mathcal{T}_0$. We have $f(s,r) = g^{|r|}(s)$. A push-down TM M_f with $\#(f)+1$ tapes can be defined as follows: with $(s)_i$ on tape i $(1 \leq i \leq \#(f))$ it computes g (in time $lh(g)$) and, after $|r|$ repetitions, it stops returning the final result. Thus, M_f standard computes $f(s,r)$, within time $|r|lh(g)$.
Case 2. Let f be defined by branching or CMP. The result follows by direct simulation of the schemes.
In order to prove the second inclusion, we show that the behaviour of an m-tape TM M (with linear time bound cn) can be simulated by a function in $\mathcal{T}_1$. Indeed a function nxt_M can be defined in $\mathcal{T}_0$, which uses two components for each tape

of M, one for the part at the left of the observed symbol, one for the part at the right (read in reverse order); the internal state is stored in the $(2m+1)$-*th* component. $nxt_M(s)$ has the form $if\ state[i](s)\ and\ top[b](j(s))\ then\ E_{ib}$, where: (a) $state[i](s)$ is a test which is true iff the state of M is coded by i; (b) $top[b](j(s))$ is a test which is true iff the observed character on tape $j(s)$ is b; (c) E_{ib} is a sequence of modifiers which update the code of the state and part of the tape, according to the definition of M. By means of $c-1$ CMP's we define in $\mathcal{T}_0$ the function nxt^c_M, which applies c times the functions nxt_M to the word that encodes an istantaneous description of M. 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 ID of M which contains the final result.

3 The hierarchies

3.1 Ordinals

In this section we define a hierarchy of functions $\{\mathcal{T}_\alpha\}_{\alpha<\epsilon_0}$, starting from $\mathcal{T}_1$, by means of closures under safe recursions and diagonalizations, such that $\mathcal{T}_{\alpha+1}$ is defined by one application of safe recursion to $\mathcal{T}_\alpha$, and $\mathcal{T}_\lambda$, for the limit ordinal λ, is obtained by diagonalization on classes $\mathcal{T}_{\lambda_n}$.
In the rest of this paper greek small letters are ordinal numbers, with λ, μ limit ordinals; λ_n is the n-th element of the fundamental sequence assigned to λ. Recalling the definition of the standard assignment of fundamental sequences for all $\lambda \leq \epsilon_0$ (cfr. [16], page 78), we introduce a slightly modified assignment.

$$\lambda_n = \begin{cases} n & \text{if } \lambda < \omega^2 \\ \omega^{\mu_n} & \text{if Cantor normal form for } \lambda \text{ is } \omega^\mu \\ \omega^\alpha n & \text{if Cantor normal form for } \lambda \text{ is } \omega^{\alpha+1} \\ \mu + (\omega^\alpha)_n & \text{if Cantor normal form for } \lambda \text{ is } \mu + \omega^\alpha \end{cases}$$

We now define a hierarchy of functions $B_\alpha(n) := \max(2,n)^{clps(\alpha,n)}$, where $clps(\alpha,n)$ is the result of replacing ω by n in the Cantor normal form of α. By simple computation one can see that $B_m(n) = n^m$, $B_{\omega c}(n) = n^{cn}$, $B_{\omega^c}(n) = n^{n^c}$, $B_{\omega_c}(n) = n^{n^{\cdots n}}$ (c times).

Definition 17. Given $1 \leq \alpha \leq \epsilon_0$ and λ a limit ordinal,

1. $\mathcal{T}_{\alpha+1}$ is the closure under the simple schemes and CMP of the class of functions obtained by one application of SREC to $\mathcal{T}_\alpha$.
2. $\mathcal{T}_\lambda$ is the closure under the simple schemes and CMP of the class of functions obtained by CDIAG(e), where $e \in \mathcal{T}_{\lambda_1}$, $\{e(r)\} \in \mathcal{T}_{\lambda_{|r|}}$.

Example 4. In the example 3 we have defined a sequence of functions g_n, with $n \geq 1$; according to definition 17 one can easily verify that $g_n \in \mathcal{T}_n$, and that $f_\omega \in \mathcal{T}_\omega$.

Definition 18. Given $1 \leq \alpha < \epsilon_0$ and λ a limit ordinal,

1. $\mathcal{S}_{\alpha+1}$ is the closure under the simple schemes of the class of functions obtained by one application of SREC to $\mathcal{S}_\alpha$.
2. $\mathcal{S}_\lambda$ is the closure under the simple schemes of the class of functions obtained by CDIAG(e), where $e \in \mathcal{S}_{\lambda_1}$, $\{e(r)\} \in \mathcal{S}_{\lambda_{|r|}}$.

Definition 19. f =WSBST(h, g) is defined by *weak substitution* of h in g if $f(x, y, z) = g(h(x, y, z), y, z)$.

Definition 20. For all positive p, q, $\mathcal{TS}_{qp}$ is the class of functions defined by weak substitution of h in g, with $h \in \mathcal{T}_q$, $g \in \mathcal{S}_p$ and $q < p$.

Theorem 1. 1. For all finite k, $\mathcal{T}_k$ =DTIMEF(n^k).
2. For all $\omega < \alpha < \epsilon_0$, DTIMEF$(B_\alpha(n)) \subseteq \mathcal{T}_\alpha \subseteq$ DTIMEF$(B_\alpha(n+4))$.
3. $\mathcal{TS}_{qp}$ =DTIMESPACEF(n^p, n^q).

Proof. 1. Let $f \in \mathcal{T}_k$. In lemma 2 a TM which interprets the function f is defined, and its runtime is proved to be in DTIMEF(n^k). Let M be a TMin DTIMEF(n^k). By lemma 3 we have that the iteration n^k times of nxt_M can be defined in $\mathcal{T}_k$; this function, starting from the code of the initial configuration of M, returns the code of its final configuration.
2. The first inclusion follows by lemma 3, and the second by lemma 2
3. The two inclusions follow by lemma 6 and lemma 5

The following results immediatly follow from theorem 1.

Corollary 1. 1. DTIMEF$(n^n) \subseteq \mathcal{T}_\omega \subseteq$DTIMEF$((n+4)^{(n+4)})$.
2. $\bigcup_{\beta<\omega} \mathcal{T}_\beta$ =PTIMEF.
3. $\bigcup_{\beta<\epsilon_0} \mathcal{T}_\beta = \mathcal{E}$.

4 Proofs

4.1 Simulation of functions by TM's

Lemma 2. For all $\alpha \geq \omega$ we have $\mathcal{T}_\alpha \subseteq$DTIMEF$(B_\alpha(n+4))$.
For all finite m, we have $\mathcal{T}_m \subseteq$DTIMEF$(B_m(n))$.

Proof. In what follows we will define a TM INT, which interprets the input $\lceil f \rceil, s, t, r$, returning the value of f applied to its arguments. Given a function $f \in \mathcal{T}_\alpha$, we need to know, while designing INT, the number of tapes it uses. In order to do this, define d such that (a) $|s| + |t| + |r| \leq d$ or (b) no CDIAG occurs in f and $\#(g) \leq d$, for each $g \in \mathcal{T}_1$ occurring in the definition of f. This means that the parts of the arguments which can be modified (and thus the number of tapes of INT) depend on d. At the end of this proof we reduce INT to a two-tape TM with a logarithmic increment in the time bound.

The interpreter INT uses the following stacks:
(a) T^x, T^y, T^z, to store the values of x, y, z during the computation; each of them consists of d tapes, one for each of the modifiable parts of the value assigned to x; their initial values are, respectively, s, t, r;
(b) T^u, to store the value of the principal variable of the current recursion;
(c) T_f, to store (the codes of) some sub-functions of f; its initial value is the code of f;

INT repeats, until T_f is not empty, the following cycle:
- it pops a function k from the top of T_f, and un-nests the outermost sub-function j of k;
- according to the form of j, it carries-out different actions on the stacks;
- if the form of j is $\text{ITER}(g)$ with $g \in \mathcal{T}_0$, it calls an interpreter $ITER$ for $\mathcal{T}_1$ which simulates g on T^x for $|t|$ times, where t is on the top of T^y;
- in all other cases, it pushes into T_f an information of the form $j\ MARK\ k$, where $MARK$ informs about the outermost scheme used to define j.

Thus we define

$INT(\lceil f \rceil, s, t, r)$:=
$T_f := \lceil f \rceil$; $T^x := s$; $T^y := t$; $T^z := r$;
while T_f not empty *do* $A :=$ last record(s) of T_f;
case

$A = CMP(g, h)$	*then* push $g\,h$ into T_f
$A = ASG_X(p, g)$	*then* push g into T^f; copy p into T^X
$A = IDT_X(h)$	*then* push h in T_f; copy last record of T^X into T^y
$A = BRANCH_i^b(g, h)$	*then* pop T_f; if $top((T^x)_i) = b$ then push g into T_f else push h into T_f
$A = DIAG(h)$	*then* push $DG\ h$ into T_f; copy last record of T^x into T^u
$A = DG$	*then* pop T_f; pop last record of T^x and push it into T_f; pop last record of T^u and push it into T^x
$A = SREC(g, h)$	*then* push $A\ RC\ g$ into T_f; copy last record of T^z into T^u; push last digit of T^u into T^z
$A = SREC(g, h)\ RC$	*then* if $T^u = T^z$ then pop T_f; pop T^u; pop T^z else push h into T_f; push last digit of T^u into T^z
$A = ITER(g)$	*then* call $ITER$.

end *case*;
end *while*.

We now show that for all f, s, t, r respecting the imposed condition over d we have

$$f \in \mathcal{T}_\alpha \to INT(\lceil f \rceil, s, t, r) = f(s, t, r) \text{ within time } |s| + |\lceil f \rceil| B_\alpha(|t| + |r| + 1_\alpha)$$

where $1_\alpha = 0$ if $\alpha < \omega$ and $1_\alpha = 1$ otherwise. The result follows, since every function $f \in \mathcal{T}_\alpha$ is then computed in $\text{DTIMEF}(B_\alpha(n + 1_\alpha))$ by the composition of the constant-time TM writing the code of f with INT.

Define $m := |s|,\ n := |t| + |r|,\ c := |\lceil f \rceil|$. We show that, for all $f \in \mathcal{T}_\alpha$, INT moves within $m + cB_\alpha(n + 1_\alpha)$ steps from an istantaneous description of the form

$$T_f = Z\lceil f \rceil;\ T^x = s_0 s;\ T^y = t_0 t;\ T^z = r_0 r;\ T^u = q,$$

to a new istantaneous description of the form

$$T_f = Z;\ T^x = s_0 f(s,t,r);\ T^y = t_0 t;\ T^z = r_0 r;\ T^u = q.$$

Induction on the construction of f. Basis. Let $f \in \mathcal{T}_1$. We have $1_\alpha = 0$. The complexity of $ITER$ is obviously bounded by $m + cn$.
Step. Case 1. $f = \text{SREC}(g,h)$. We have $\alpha = \beta + 1$; let r be the word $a_{|r|} \dots a_1$. By the inductive hypothesis, INT needs time $\leq m + |\lceil g \rceil| B_\beta(n + 1_\alpha)$ to produce the istantaneous description

$$T_f = Z\lceil f \rceil\ RC;\ T^x = s_0\ g(s,t,a_1);\ T^y = t_0 t;\ T^z = r_0 r a_1;\ T^u = qr.$$

If $|r| > 1$ then INT puts $T_f := Z\ \lceil f \rceil\ RC\ \lceil h \rceil$ and $T^z := r_0 r a_2 a_1$, and calls itself in order to compute h and the next value of f. By the inductive hypothesis we have that INT needs time $\leq |g(s,t,a_1)| + |\lceil h \rceil| B_\beta(n + 1_\alpha)$ to produce an istantaneous description of the form

$$T_f = Z\lceil f \rceil\ RC;\ T^u = qr;$$

$$T^x = s_0\ (h(g(s,t,a_1),t,a_2 a_1));\ T^y = t_0 t;\ T^z = r_0 r a_2 a_1.$$

After $|r|$ simulations of h we obtain the promised istantaneous description within an amount of time

$$m + |r| \max(|\lceil g \rceil|, |\lceil h \rceil|) B_\beta(n + 1_\alpha) \leq m + |r| c B_\beta(n + 1_\alpha) \leq m + cB_\alpha(n + 1_\alpha)$$

where, since $\alpha \geq 2$, in these evaluations we may compensate the quadratic amount of time needed to copy r and its digits with the difference between c and $\max(|\lceil g \rceil|, |\lceil h \rceil|)$.
Case 2. $f = \text{CDIAG}(h) \in \mathcal{T}_\lambda$. We have $h \in \mathcal{T}_{\lambda_1}$ and (recall that $\lambda_1 = 1$ when $\lambda < \omega^2$)

$$B_{\lambda_1}(n+1) + B_{\lambda_n}(n+1) \leq B_{\lambda_{n+1}}(n+1) = B_\lambda(n+1). \tag{1}$$

INT computes $h(r)$, understands from the mark DG that the result is the code for the function to be computed, and, accordingly, pushes it into T_f.
To compute $h(r)$ and $\{h(r)\}(s,t,r)$ the interpreter INT needs, by the inductive hypothesis, time $m + |\lceil h \rceil| B_{\lambda_1}(|r| + 1) + |\{h(r)\}| B_{\lambda_{|r|}}(n+1) \leq$ (by (1))$m + |\lceil h \rceil| B_\lambda(n+1) \leq m + cB_\lambda(n+1)$.
Reduction to two tapes. The interpreter we have just defined uses a number of tapes d that depends on the construction of the simulated function. We use now the general procedure showed in [9] to reduce a k-tape TM bounded by $T(n)$ to a 2-tape TM bounded by $kT(n) \log T(n)$; thus, we obtain a 2-tape interpreter bounded by

$$T(n) = kB_\alpha(n+1) \log B_\alpha(n+1) \leq kB_\alpha(n+2) \log(n+1) \leq B_\alpha(n+4)$$

4.2 Simulation of TM's by functions

Lemma 3. For all $1 \leq \alpha < \epsilon_0$, $\text{DTIMEF}(B_\alpha(n)) \subseteq \mathcal{T}_\alpha$.

Proof. Let M be a TM in $\text{DTIMEF}(B_\alpha(n))$. There exists a function $nxt_M \in \mathcal{T}_0$ such that, for input the code of an istantaneous description of M, nxt_M returns the code of the next description. We define the following function:

$$\sigma(s, \lceil \alpha \rceil) = \begin{cases} s \lceil nxt_M \rceil & \text{if } \alpha = 0 \\ \sigma(s, \lceil \beta \rceil)\sigma(s, \lceil \beta \rceil)\lceil \text{SREC} \rceil & \text{if } \alpha = \beta + 1 \\ \sigma(s, \lceil \alpha_1 \rceil)\sigma(s, \lceil \alpha_{|r|} \rceil)\lceil \text{SREC} \rceil \lceil \text{CDIAG} \rceil & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

We prove by induction on α that the function whose code is generated by σ is in $\mathcal{T}_\alpha$.
Base. $\alpha = 0$. We have that $nxt_M \in \mathcal{T}_0$, by hypothesis.
Step. Case 1. $\alpha = \beta + 1$. We have that $\{\sigma(s, \lceil \alpha \rceil)\} = \text{SREC}(\sigma(s, \lceil \beta \rceil), \sigma(s, \lceil \beta \rceil))$. The function is in $\mathcal{T}_\alpha$, by induction on α and definition 17.
Case 2. $\alpha = \lambda$. We have that $\{\sigma(s, \lceil \lambda \rceil)\} = \text{CDIAG}(\text{SREC}(\sigma(s, \lceil \lambda_1 \rceil), \sigma(s, \lceil \lambda_{|s|} \rceil)))$. This function is in $\mathcal{T}_\lambda$, since $\{\sigma(s, \lceil \lambda_1 \rceil)\} \in \mathcal{T}_{\lambda_1}$ and, for all s, $\{\sigma(s, \lceil \lambda_{|s|} \rceil)\} \in \mathcal{T}_{\lambda_{|s|}}$.
The function σ writes the code of a function which iterates nxt_M for $B_\alpha(n)$ times; by input the code of the initial configuration of M, this function returns the codes of the final configuration.
Note that, given the code of a limit ordinal λ, we need at most a quadratic amount of time to return the code of λ_n.

4.3 Time-space classes

Lemma 4. For all f in $\mathcal{S}_p$, we have $|f(s, t, r)| \leq \max(|s|, |t|, |r|)$.

Proof. By induction on p. Base. $f \in \mathcal{S}_1$.
Case 1. f is defined by iteration of a function g in $\mathcal{S}_0$; we have, by induction on r, $|f(s, a)| = |s|$, and $|f(s, ra)| = |g(f(s, r))| \leq |f(s, r)| \leq \max(|s|, |r|)$.
Case 2. f is defined by simple scheme or CMP. The result follows by the inductive hypothesis.
Step. Given $f \in \mathcal{S}_{p+1}$, defined by SREC in functions 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 5. $\mathcal{TS}_{qp} \subseteq$DTIMESPACEF$(n^p, n^q)$.

Proof. Let f be a function in $\mathcal{TS}_{qp}$. By definition 20, f is defined by weak substitution of a function $h \in \mathcal{T}_q$ into a function $g \in \mathcal{S}_p$, that is, $f(s,t,r) = g(h(s,t,r),t,r)$. The theorem 1 states that there exists an interpreter INT computing the values of h within time n^q, and computing the values of g within time n^p. The lemma 4 holds for g, since g belongs to $\mathcal{S}_p$; thus, the space needed by INT to compute g is at most n.
Define now a TM M that, by input $\lceil f \rceil, s, t$, and r performs the following steps (recall lemma 2 for the definition of INT):
(1) it pushes $\lceil g \rceil \lceil h \rceil$ into the tape T^f of INT, which contains the codes of the functions that the interpreter will compute;
(2) it calls INT on input $\lceil g \rceil \lceil h \rceil, s, t, r$.
The time complexity of (1) is linear in the length of $\lceil f \rceil$; in (2), INT needs time equal to n^q to compute h, and needs only n^p (and not n^{q^p}) to compute g. This happens because $h(s,t,r)$ is computed in the safe position, and this implies that its length does not affect the number of steps performed by the second call to INT. In fact, INT never moves the content of a safe position into the tapes whose values play the role of recursive counters; they depend only on n, the length of the original input. Thus, the overall time bound is $n^q + n^p$, which can be reduced to n^p, being $q < p$.
INT requires space n^q to compute the value of h on input s,t,r; as we noted above, the space needed for the computation of g is linear in the length of the input, and thus the overall space needed by M is still n^q.

Lemma 6. DTIMESPACEF$(n^p, n^q) \subseteq \mathcal{TS}_{qp}$

Proof. Let M be a TM in DTIMESPACEF(n^p, n^q). This means that the computation of M is time-bounded by n^q and, simultaneously, it is space-bounded by n^p. M can be simulated by the composition of two TM's, M_g and M_h, with $M_h \in$DTIMEF(n^q) and $M_g \in$DTIMESPACEF(n^p, n): the former constructs (within polynomial time) the space that the latter will successively use in order to simulate M.
By theorem 1 there exists a function $h \in \mathcal{T}_q$ which simulates the behaviour of M_h, and there exists a function $g \in \mathcal{S}_q$ which simulates the behaviour of M_g; in particular, there exists the function $nxt_g \in \mathcal{T}_0$. Note that nxt_g belongs to $\mathcal{S}_0$, since it never adds a digit to the description of M_g without erasing another one.
We define the function $\sigma_0 :=$ITER(nxt_g), and the sequence $\gamma_{n+1} :=$SREC(σ_n, σ_n), with $\sigma_{n+1} :=$IDT$_z(\gamma_{n+1})$.
We have that

$$\begin{cases} \gamma_1(s,t,a) = \text{ITER}(nxt_g)(s,t) \\ \gamma_1(s,t,ra) = \text{ITER}(nxt_g)(\gamma_1(s,t,r),t) \end{cases}$$

and

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

We can easily see that $\sigma_0 \in \mathcal{S}_1$, by definition of this class and, again by definition 18, that $\sigma_n \in \mathcal{S}_{n+1}$.

Given the code s of the initial id of M, we can define $sim_M(s) = \sigma_{p-1}(h(s), s)$, which simulates the behaviour of M. This function is defined in $\mathcal{TS}_{qp}$.

References

1. S.J. Bellantoni, *Predicative recursion and the polytime hierarchy*, in P.Clote and J.Remmel (eds), Feasible Mathematics II (Birkauser, 1994), 320-343.
2. S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the polytime functions*, Computational Complexity 2(1992)97-110.
3. S. Caporaso, *Safe Turing machines, Grzegorczyc classes and polytime*, Intern. J. Found. Comp. Sc., 7.3(1996)241-252.
4. S. Caporaso, N. Galesi, M. Zito, *A predicative and decidable characterization of the polytime classes of languages*, to appear in Theoretical Comp. Sc.
5. S. Caporaso, M. Zito, N. Galesi, E. Covino *Syntactic characterization in Lisp of the polynomial complexity classes and hierarchies*, in G. Bongiovanni, D.P. Bovet, G. Di Battista (eds), Algorithms and Complexity, LNCS 1203(1997)61-73.
6. S. Caporaso, G. Pani, E. Covino, *Predicative recursion, constructive diagonalization and the elementary functions*, Workshop on Implicit Computational Complexity (ICC99), affiliated with LICS99, Trento, 1999.
7. P. Clote, *A time-space hierarchy between polynomial time and polynomial space*, Math. Sys. The. 25(1992)77-92.
8. A. Cobham, *The intrinsic computational difficulty of functions*, in Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, pages 24-30, North-Holland, Amsterdam, 1962.
9. F.C.Hennie and R.E.Stearns, *Two-tape simulation of multi-tapes TM's*, Journal of ACM 13.4(1966)533-546.
10. 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.
11. D. Leivant, *Stratified functional programs and computational complexity*, in Conference Records of the 20th Annual ACM Symposium on Principles of Programming Languages, New York, 1993, ACM.
12. D. Leivant, *Ramified recurrence and computational complexity I: word recurrence and polytime*, in P.Clote and J.Remmel (eds), Feasible Mathematics II (Birkauser, 1994), 320-343.
13. D. Leivant and J.-Y. Marion, *Ramified recurrence and computational complexity II: substitution and polyspace*, in J. Tiuryn and L. Pocholsky (eds), Computer Science Logic, LNCS 933(1995) 486-500.
14. I. Oitavem, *New recursive characterization of the elementary functions and the functions computable in polynomial space*, Revista Matematica de la Universidad Complutense de Madrid, 10.1(1997)109-125.
15. R.W. Ritchie, *Classes of predictable computable functions*, Transactions of A.M.S., 106, 1993.
16. H.E. Rose, *Subrecursion: Functions and hierarchies*, Oxford University Press, Oxford, 1984.
17. H. Simmons, *The realm of primitive recursion*, Arch.Math. Logic, 27(1988)177-188.