SYNTACTIC CHARACTERIZATION IN LISP OF THE POLYNOMIAL COMPLEXITY CLASSES AND HIERARCHY

Salvatore Caporaso[1], Michele Zito[1 2] Nicola Galesi[1 3] and Emanuele Covino[1 4]

[1] Univ di Bari, Dip. di Informatica, v.Amendola 173, I-70126, logica@gauss.uniba.it
[2] University of Warwick Coventry, CV4 7AL, UK, M.Zito@dcs.warwick.ac.uk
[3] Universitat Politecnica de Catalunya, Barcelona, Galesi@goliat.up.es
[4] Abt. th. Informatik, Universität Ulm, Covino@theorie.informatik.uni-ulm.de

Abstract. The definition of a class $\mathcal{C}$ of functions is *syntactic* if membership to $\mathcal{C}$ can be decided from the construction of its elements. Syntactic characterizations of PTIMEF, of PSPACEF, of the polynomial hierarchy PH, and of its subclasses Δ_n^p are presented. They are obtained by progressive restrictions of recursion in Lisp, and may be regarded as *predicative* according to a foundational point raised by Leivant.

1 Introduction

At least since 1965 [6] people think to complexity in terms of *TM's plus clock or meter*. However, understanding a complexity class may be easier if we define it by means of *operators* instead of *resources*. Different forms of *limited recursion* have been used to this purpose. After the well-known characterizations of LINSPACEF [15] and PTIMEF [5], further work in this direction has been produced (see, for example, [11], [8], [4]).

Both approaches (resources and limited operators) are not *syntactic*, in the sense that membership to a given class cannot be decided from the construction of its elements (for example, if f is primitive recursive (PR) in g and h, we cannot decide whether f is actually bounded above by a third function k). And both approaches may be criticized on foundational grounds. The definition of an entity E is *impredicative* (see Poincaré [14], p. 307) if it uses a variable defined on a domain including E. Examples of impredicative definitions are $\sqrt{2} =_{df} \max\ z(z^2 \leq 2)$ and $\mathrm{Pow}(x) =_{df} \{y | y \subseteq x\}$. The definition of, say, PTIMEF, by means of the (predicative) class of all T-computable functions, might be regarded as impredicative too. For a better position of the problem, and for a remarkable solution in proof-theoretic terms, see Leivant [9].

The first purely syntactic definition of PTIMEF, based on a form of unlimited PR on binary numerals is in [1]. Further characterizations of the same class are in [12] and [10], using finite automata and, respectively, λ-calculus. A syntactic definition of PTIMEF and LINTIMEF, by a tortuous variant of TM's, is in [2]. PSPACEF has been studied less. We are not aware of any recursive characterization (even impredicative) of the polynomial hierarchy PH.

In this paper we define a number of fragments of Lisp, by means of a progressive sequence of restrictions to (unlimited) recursion; and we show the equivalence between these fragments and the polynomial classes. Lisp has been chosen, instead of other models of computation, because it offers the obvious advantages of a richer data type and of a higher-level language, and because it fits traditional mathematical methods of investigation, like induction on the construction of functions and arguments. A preliminary validation of this choice is discussed in the last section of this paper, together with perspectives and other aspects of our work.

We now outline the adopted recursion schemes. A function $f[\mathbf{x}; y]$ is defined by *course-of-values recursion* if its value depends on a pre-assigned number n of values $f[\mathbf{x}; y_i]$ for n previous values of y. What makes the difference is the meaning of *previous*. For PSPACEF and PH we mean any z such that $|z| < |y|$ (that is we may choose n values among $O(2^{c|y|})$ previous values). For PTIMEF we mean any subexpression of y (n among $O(|y|)$ values). The restriction of PSPACE to PH is obtained by asking that the *invariant* function of the recursion be in the form $f[\mathbf{x}; y_1]$ *or* $\ldots$ *or* $f[\mathbf{x}; y_n]$. Classes Δ_k are defined by counting in the most obvious way the levels of nesting of this form of recursion.

A rather extreme formulation of an aspect of the work presented here is that it allows a position of some celebrated problems in terms of comparison between similar operators, of an apparently increasing strength, instead than in terms of contrast between heterogeneous resources.

2 Recursion Free Lisp

An *atom* is a sequence of capital letters and decimal digits. A special role is assigned to atoms $T (F)$, associated with the truth-values *true (false)*, and NIL. An (S-)*expression* is an atom, or a *dotted couple* $(x \cdot y)$, where x and y are expressions. $\omega, \omega_1, \ldots$ are (variables defined on the) atoms; $s, \ldots, z, s_1, \ldots$ are S-expressions. $\mathbf{s}, \ldots, \mathbf{z}$ are *tuples of expressions* of the form $x_1; \ldots; x_n$ $(n \geq 0)$. An *(S-)function* f takes a tuple of arguments $\mathbf{x}$ into an expression $f[\mathbf{x}]$; d, e, f, g, h are functions, and $\mathbf{d}, \mathbf{d}^{\,1} \ldots$ are tuples of functions. If a tuple of syntactical entities has been introduced by means of a notation of the form $\mathbf{E}$, we denote by E_i its i-th member (for example x_i, z_i^j are the i-th expression of $\mathbf{x}, \mathbf{z}^{\,j}$; and f_i^j is the i-th function of $\mathbf{f}^{\,j}$).

A *list* is an expression of the particular form we now describe: atom NIL is the *empty list*, also denoted by (); all other lists x are in the form $(x_n \cdot (\ldots \cdot (x_1 \cdot \mathrm{NIL}) \ldots)$, and are shown as $(x_n, \ldots, x_1)$; $(x)_i = x_i$ is the *i-th component of* x, and $\#(x) = n \geq 0$ is its *number of components.*

Sometimes, along a computation, we mark an (occurrence of an) expression x by a superscript $\tau = A, B, AB$, and we say that x^τ is of type τ; when x has not been marked, we say that it is of type 0, and we write x^0. (Thus marked S-expressions are the actual constants of our language.) The type of all atoms is 0. The type of all non-atomic sub-expressions of x^τ is τ. A relation of compatibility is established by stating that:

1. all expressions are compatible with those of type 0;
2. all expressions of type $\tau \neq 0$ are incompatible with those of their same type τ and with those of type AB.

$\mathbf{x}^{\tau}$ ($\mathbf{x}^{\neq\tau}$) is a tuple of variables of the same type τ (of type $\neq \tau$). Types are not specified in the definition of a function, when they don't change (cf. 2.2.2) or when they don't affect the result (cf. 2.2.1).

2.1 Basic functions

The class $\mathcal{B}$ of the *basic functions* consists of:

1. *predicates at* and *eq*, such that $at[x] = T(F)$ if x is (not) an atom, and $eq[x;y] = T(F)$ if x and y are (not) the same atom;
2. the *conditional* $cond[x;y;z] = y$ if $x = T$, and $= z$ if $x \neq T$; $cond[x_1;y_1;\ldots;$ $cond[x_n;\, y_n;z]\ldots]$ is usually displayed as $[x_1 \to y_1;\ldots;x_n \to y_n;T \to z]$.
3. the *selectors* $sel^n_j[\mathbf{x}] = x_j$, and, for every atom ω, the *constant functions* $\omega[\mathbf{x}] = \omega$; we often let these functions be denoted by their results; *id* is the *identity* sel^1_1;
4. the *predecessors car* and *cdr*, such that: $car[\omega] = cdr[\omega] = \omega$; $car[(y \cdot z)] = y$ and $cdr[(y \cdot z)] = z$; sometimes we write x' for $car[x]$ and x'' for $cdr[x]$;
5. the *constructor*
$$cons[x^{\tau_1};y^{\tau_2}]=\begin{cases}(\text{NIL})^{AB} & \text{if the arguments are incompatible}\\ (x \cdot y)^0 & \text{if both } \tau_i \text{ are } 0\\ (x \cdot y)^{\tau} & \text{if one of the } \tau_i \text{ is } \tau \text{ and the other is } 0\\ (x \cdot y)^{AB} & \text{if one of the } \tau_i \text{ is A and the other is B;}\end{cases}$$
6. functions α, β, ζ, which leave un-changed the atoms, and such, otherwise, that $\alpha[x^0] = x^A, \beta[x^0] = x^B$, $\alpha[x^{\neq 0}] = \beta[x^{\neq 0}] =$NIL; $\zeta[x^{\tau}] = x^0$;
7. function *unite*, which leaves its argument x un-changed if x' or x'' are not lists; and takes $((x_1,\ldots,x_m),x_{m+1},\ldots,x_{m+n})$ into $(x_1,\ldots,x_{m+n})$ otherwise; for example $unite[((A,B,C),D,E)] = (A,B,C,D,E)$.

These basic functions differ from those of pure Lisp for a few changes, adopted to handle the types and to exclude marginal cases of undefined functions.

2.2 Substitutions

By a composite notation like $f[\mathbf{x}]$, we mean that all arguments of f occur (not necessarily once) in $\mathbf{x}$, but we don't imply that every x_i is an actual argument of f.

A main difference with pure Lisp is that we renounce to its λ's to show substitutions (SBST) explicitly, by replacing the substituted variable with the substituend function. This rudimental way allows simpler definitions and space-complexity evaluations, at the price of a systematic ambiguity between functions and values. Thus, deciding for example whether $car[x]$ and $car[y]$ are *the same thing* is left to context. A SBST to an absent variable has no effect; all occurrences of the substituted variable are replaced by the substituend function. No kind of disjunction between original and new variables is assumed.

We write $\mathbf{f}[\mathbf{x}]$ for $f_1[\mathbf{x}]; \ldots; f_n[\mathbf{x}]$. Given n functions $\mathbf{h}[\mathbf{u}]$, and given $g[\mathbf{x}; \mathbf{z}]$, we write $g[\mathbf{x}; \mathbf{h}[\mathbf{u}]]$ for the *simultaneous SBST* of $\mathbf{h}[\mathbf{u}]$ to the n variables $\mathbf{z}$ in g. The special form of substitution we now introduce allows to by-pass the type-restrictions on the *cons*'s one should otherwise handle, in order to re-assemble the parts of the argument, after processing them separately.

Definition 1. The unary function f is defined by *internal substitution* (INSBST) in $g_1, \ldots, g_k$ if we have

$$f[x] = \begin{cases} \text{NIL} & \text{if } \#(x) < k \\ (g_1[(x)_1], \ldots, g_k[(x)_k], (x)_{k+1}, \ldots, (x)_{\#(x)}) & \text{otherwise,} \end{cases}$$

or

$$\begin{cases} f[\omega] & = \text{NIL} \\ f[(u \cdot w)] & = (g_1[u] \cdot g_2[w]); \end{cases}$$

Notation: $f = \pi(\mathbf{g})$. Functions $\mathbf{g}$ are the *scope* of the *INSBST*.

Given a class $\mathcal{C}$ of functions, we denote by $\mathcal{C}^*$ its closure under SBST and INSBST. For example, the class of all *recursion-free* functions is $\mathcal{B}^*$.

2.3 Lengths

The *length* $|z|$ of z is the number of atoms and dots occurring in (the value assigned to) z. $|\mathbf{x}|$ and $\max(\mathbf{x})$ are respectively $\sum_i |x_i|$ and $\max_i(|x_i|)$.

$|f[\mathbf{x}]|$ is the length of the value of $f[\mathbf{x}]$ when a system of values is assigned to $\mathbf{x}$; $|\mathbf{f}[\mathbf{x}]|$ is $\sum_i |f_i[\mathbf{x}]|$. For example $|cons[x; x]| = 2|x| + 1$; $|y''| \leq \max(1, |y| - 2)$. We say that $f[\mathbf{x}]$ is *limited* by the numerical function ϕ (possibly a constant) if for all $\mathbf{x}$ we have $|f[\mathbf{x}]| \leq \phi(|\mathbf{x}|)$.

Define $lh_c(f)$ to be $2n + 1$, where n is the number of *cons* occurring in the construction of f.

The idea of next lemma is rather simple: types allow cobbling together, without any limitation, the arguments of type 0; but at most one A and/or one B may contribute to the function being computed.

Lemma 2. *For all recursion-free function f in which ζ doesn't occur, we have*

$$|f[\mathbf{x}^{\,0}; \mathbf{s}^{\,1A}; \mathbf{s}^{\,2B}]| \leq lh_c(f) \max(1, |\mathbf{x}|) + \max(\mathbf{s}^{\,1}) + \max(\mathbf{s}^{\,2}).$$

Proof. Let us write m for $lh_c(f) \max(1, |\mathbf{x}|)$, and M_i for $\max(\mathbf{s}^{\,i})$. We show that $z^\tau = f[\mathbf{x}; \mathbf{s}^{\,1}; \mathbf{s}^{\,2}]$ implies $|z| \leq m + n$, where:

$\tau = 0$ (case 1) implies $n = 0$;

$\tau = A$ ($\tau = B$) (case 2) implies $n \leq M_1$ ($n \leq M_2$); and

$\tau = AB$ (case 3) implies $n \leq M_1 + M_2$.

Induction on the construction of f. Base. Assume that f is *cons*, since else the result is trivial. We have, for example, $|cons[t^A; t^B]| = 2|t^{AB}| + 1 \leq 3 + 2|t|$. Etc.

Step. (1) Let us first assume that f begins by a basic function. Then we may assume further that the form of f is $cons[g_1[\mathbf{x};\mathbf{s}^{1A};\mathbf{s}^{2B}];g_2[\mathbf{x};\mathbf{s}^{1A};\mathbf{s}^{2B}]]$, since the lemma is an immediate consequence of the ind. hyp. for all other basic functions. Let $g_i[\mathbf{x};\mathbf{s}^1;\mathbf{s}^2]=z_i^{\tau_i},\ i=1,2$; thus $z^\tau = cons[z_1;z_2]$. Let us write m_i for $lh_c(g_i)\max(1,|\mathbf{x}|)$ Cases 1-3 as above.
Case 1. We have $\tau_1=\tau_2=0$ The ind. hyp. gives $|z_i|\le m_i$. The result follows, since $lh_c(f)=lh_c(g_1)+lh_c(g_2)+1$.
Case 2. We have $\tau=A$ or $\tau=B$, one of the τ_i is τ, and the other is 0; let for example $\tau_1=B$. The ind. hyp. gives $|g_1|\le m_1+M_2$, $|g_2|\le m_2$. The result follows by immediate computations.
Subcase 3.1. One of the τ_i, say τ_1, is A, and the other is B. The ind. hyp. gives $|z_i|\le m_i+M_i$ and the result follows immediately, since $m_1+m_2\le m$.
Subcase 3.2. One of the τ_i is AB, and the other is 0. Similarly.
(2) The possibility remains that the form of f is $\pi(g_1,\ldots,g_k)[h[\mathbf{x};\mathbf{s}^1;\mathbf{s}^2]]$. Let $h[\ldots]=y^\tau$. Assume $\#(y)=k$. Case 1. $\tau\neq 0$. Then, since all components of y are of the same type τ, the ind. hyp. gives $|f|\le\sum_i(|(y)_i|+lh_c(g_i))+k-1\le |y|+\sum_i(lh_c(g_i))$, and the result follows by the ind. hyp. applied to h, since $lh_c(f)\ge lh_c(h)+\sum_i lh_c(g_i)$. Case 2. $\tau=0$. Immediately from the ind. hyp., applied to h and to the g's.

2.4 Some classes of recursion-free functions

(1) A *proper cut of order n* is a composition of $n\ge 0$ predecessors. We regard the identity as an *improper cut* of order 0. Two cuts g_1,g_2 are *disjoint* if they don't return two overlapping sub-expressions of their argument. In syntactic terms, the g's are disjoint if for no g_i there is a cut h, such that $g_i[x]=h[g_{3-i}[x]]$. A *fully disjoint tuple (of cuts) C* is a tuple $\mathbf{e}$, such that: (a) every e_i is either a cut g_i, or is in the form $\pi(\mathbf{h}^i)$, where every h^i_j is a cut; and (b) all couples g_i,h^i_j are disjoint.

Define the cuts $1st, 2d, 3d, \ldots, i\text{-}th$, such that $i\le\#(x)$ implies $i\text{-}th[x]=(x)_i$; any tuple of cuts of this form is an example of fully disjoint tuple.
(2) For every list $y=(\omega_1,\ldots,\omega_n)$, we call *unary append (of order n)*, and we denote by $app(y)$, function $cons[\omega_1;[\ldots;cons[\omega_n;x]\ldots]]$; if x is a list, it appends its components to those of y. For example, for $y=(A,B)$ and for every list $x=(x_1,\ldots,x_n)$, we have $app(A,B)[x]=(A,B,x_1,\ldots,x_n)$.
Define $list[x]=cons[x;\mathrm{NIL}[x]]$; for all n define $list[x_{n+1};\ldots;x_1]=cons[x_{n+1};list[x_n;\ldots;x_1]]$.

For example, we have $list[()]=(())$; $list[A,(A),((A))]=(A,(A),((A)))$.
(3) Define the *sentential connectives not, or, and* from

$$not[x]=[x\to F;T\to T],\quad x\ or\ y\ =[x\to T;y\to T;T\to F].$$

A *simple boolean* function is built-up from eq, at and the connectives. A *boolean* function is obtained by substitution of some cuts to some variables in a simple boolean function.
(4) For all list of atoms q,s,t define functions $g(q,s,t)$ by (see proof of Lemma 3 for their use) $g(q,s,t)[x]=\pi(app(s),q,t,cdr)$; we have

$$g(q,s,t)[((x_1,\ldots,x_n),u,w,(y_1,\ldots,y_m))] = ((s,x_1,\ldots,x_n),q,t,(y_2,\ldots,y_m)).$$

(5) A function is *trivially decreasing* if is a proper cut; or if it is in the form $\pi(g_1,\ldots,g_m)$, and: (a) every g_i is a cut, or a unary *app*; and (b) the sum of the orders of all cuts is higher than the sum of the orders of all unary *app*'s. For example, $\pi(app(T), 3d, id)$ is trivially decreasing. If $f = \pi(g_1,\ldots,g_m)$ is trivially decreasing, and if no $g_i[y]$ is an atom, then $|f[y]| < |y|$.

3 Recursion schemes

An obvious condition to ensure that a recursion scheme defines total functions is that its recursive calls refer to values of the recursion variable, which preceed, according to some (partial) order, its current value. In the Conclusion, doubts are expressed about closure of the polynomial classes under recursion schemes based on an order isomorphic to the natural numbers. Hence our first restriction is to the order $x < y$ iff $|x| < |y|$.

Definition 3. Given (1) m *parameters* $\mathbf{x}$, a *principal variable* y, and n *auxiliary variables* $\mathbf{s}$;
(2) an n-ple $\mathbf{d}$ of trivially decreasing functions, together with a *terminating* boolean function $g^*[y]$, depending on the form of the d's in some trival way that we don't specify here;
(3) an *initial* function $g[\mathbf{x}; y]$ and an *invariant* function $h[\mathbf{x}; y; \mathbf{s}]$;
function f is defined by *course-of-values recursion* (CVR) *in* g, h if we have

$$f[\mathbf{x}; y] \begin{cases} g[\mathbf{x}; y] & \text{if } g^*[y] = T \\ h[\mathbf{x}; y; f[\mathbf{x}; d_1[y]]; \ldots; f[\mathbf{x}; d_n[y]]]] & \text{otherwise.} \end{cases}$$

The following example shows that an exponential space complexity may easily be reached with very poor means: no nesting, and a single recursive call to the most obvious sub-expression of the recursion variable. Thus restrictions to the invariant h have to be adopted. We have two alternatives: either we drastically impose that h be boolean, or we use the types machinery to rule its growth.

$$\begin{cases} ex[x;\omega] = cons[x;x] \\ ex[x;y] \;= cons[ex[x;y'']; ex[x;y'']]. \end{cases}$$

Definition 4. 1. Function f is *(recursively) boolean* if is boolean and recursion-free, or if is defined by CVR with boolean invariant function.
2. Function $f[\mathbf{x}; y]$ is defined by *short CVR* (SCV) if it is defined by CVR, if the initial function g is in the class $\mathcal{PL}$ defined below, and if the invariant is boolean.
3. Function f is defined by *or-SCV* (OR-CV) if it is defined by SCV, and the form of its invariant is

$$h[\mathbf{x}; \mathbf{y}; \mathbf{s}] = s_1 \text{ } or \text{ } s_2 \text{ } or \text{ } \ldots \text{ } or \text{ } s_n.$$

4. Function f is defined by *fast CVR* (FCV) if: is defined by SCV; the decreasing functions form a fully disjoint tuple of cuts; and the invariant h is
   (a) either boolean, or
   (b) is recursion-free, and there is a function h^*, in which ζ doesn't occur, and a tuple $\mathbf{e}$ of α's and β's, such that
$$h[\mathbf{x}; y; \mathbf{s}] = \zeta[h^*[\mathbf{x}; y; e_1[s_1]; \ldots; e_n[s_n]]].$$

The sense of clause (b) above is that, if $z_1, \ldots, z_n$ are the previous values of f, then h is not allowed to *cons* any z_i with itself, though it may *cons* at most one of the z's in the scope of an α with at most one of those in the scope of a β.

Examples of FCV. Define the *numeral* $num(m)$ for m to be the list whose $m+1$ components are all 0. A function *mult*, such that $mult[num(h); num(k)] = num(hk)$ may be obtained from function $mult_0$ below, by some trivial changes

$$mult_0[x; y] = \begin{cases} x & \text{if } y \text{ is an atom} \\ \zeta[unite[cons[x; \alpha[mult_0[x; cdr[y'']]]]]] & \text{otherwise.} \end{cases}$$

Thus FCV, with *cdr* as decreasing function, may be regarded as an analogue of number-theoretic PR. Next example shows that, with car, cdr as decreasing functions, FCV is the analogue of the form of recursion known in Literature as *tree PR*. In the concluding section the advantages of taking less trivial cuts as decreasing functions are discussed. The following function lh computes $num(|y|)$

$$\begin{cases} lh[\omega] = (0) \\ lh[y] \;= \zeta[cons[0; unite[list[\alpha[lh[y']]; \beta[lh[y'']]]]]]. \end{cases}$$

Define the equality by $x = y := eqc[cons[x; y]]$, where eqc is defined by FCV, with $d_1 = \pi(car, car)$ and $d_2 = \pi(cdr, cdr)$, by

$$eqc[y] = \begin{cases} eq[y'; y''] & \text{if } at[y'] \text{ } or \text{ } at[y''] = T \\ eqc[d_1[y]] \text{ } and \text{ } eqc[d_2[y]] & otherwise. \end{cases}$$

Example of OR-CV: SAT. Assume defined function $true[(v, u, w, z)]$, which, if v is a list of atoms and z is (the code for) a sentential formula: (a) assigns true (false) to the i-th literal of z if the i-th component of v is (not) T; (b) returns $T(F)$ if z is true (false) under this truth-assignment. Define by OR-CV, with decreasing tuples

$$d_1 = \pi(app(T), cdr, cdr, id) \quad d_2 = \pi(app(F), cdr, cdr, id)$$

$$st[y] = \begin{cases} true[y] & \text{if } at[(y)_2] \\ st[d_1[y]] \text{ } or \text{ } st[d_2[y]] & \text{otherwise.} \end{cases}$$

Satisfiability is decided by $sat[x] = list[(); lh[x]; lh[x]; x]$.

Example of SCV : QBF. We show that thePSPACE-complete language QBF is accepted by a function qbf definable in $\mathcal{PSL}$. Let $b, b_1, \ldots$ be *(boolean) literals*, and let ϕ, χ be *quantified boolean formulas*. Let $num_2(i)$ be the binary numeral for i, and define the code ϕ^* for ϕ by

$0^* = T$; $1^* = F$; $b_i^* = (VAR, num_2(i))$; $(\neg\phi)^* = (NOT, \phi^*)$; $(\forall b\phi)^* = (ALL, b^*, \phi^*)$;

$(\exists b\phi)^* = (EX, b^*, \phi^*)$; $(\chi \wedge \psi)^* = (AND, \chi^*, \psi^*)$; $(\chi \vee \psi)^* = (OR, \chi^*, \psi^*)$.

We associate each occurrence $\hat{b}$ of literal b in formula ϕ with a list $AV(\hat{b}, \phi)$, to be used as *address and truth-assignment*, and defined by

1. let ϕ be $\chi \vee \psi$ or $\phi = \chi \wedge \psi$; if $\hat{b}$ is in χ (is in ψ) then $AV(\hat{b}, \phi)$ is $(L, AV(\hat{b}, \chi))$ (is $(R, AV(\hat{b}, \psi))$); it says that $\hat{b}$ is in the left (right) principal sub-formula of ϕ;
2. if ϕ is $\forall(\exists) b_i \chi$, and we wish to assign T, F to the occurrences of b_i in the scope of the indicated quantifier, then $AV(\hat{b}, \phi) = (T, AV(\hat{b}, \chi))$ or, respectively, $(F, AV(\hat{b}, \chi))$.

A function $val[(x, u, z)]$ can be defined in $\mathcal{PL}$, which, by an input of the form $(AV(\hat{b}, \phi), u, \phi^*)$ returns $T(F)$ if $AV(\hat{b}, \phi)$ is the address and truth-assignment of an occurrence in ϕ of a true (false) literal. Define

$$qb[y] = \begin{cases} val[y] & \text{if } at[(y)'_2] \\ [(y)'_2 = AND \rightarrow qb[d_{11}[y]] \textit{ and } qb[d_{12}[y]]; & \\ (y)'_2 = OR \quad \rightarrow qb[d_{11}[y]] \textit{ or } qb[d_{12}[y]]; & \\ (y)'_2 = ALL \ \rightarrow qb[d_{21}[y]] \textit{ and } qb[d_{22}[y]]; & \\ (y)'_2 = EX \quad \rightarrow qb[d_{21}[y]] \textit{ or } qb[d_{22}[y]]; & \\ (y)'_2 = NOT \rightarrow not[qb[d_3[y]]]; & \\ (y)'_2 = VAR \rightarrow qb[d_3[y]]] & \text{otherwise;} \end{cases}$$

function qb is defined by SCV, with the following trivially decreasing tuples

$d_{11} = \pi(app(L), 2d, id)$ $d_{12} = \pi(app(R), 3d, id)$

$d_{21} = \pi(app(T), 3d, id)$ $d_{22} = \pi(app(F), 3d, id)$ $d_3 = \pi(id, 2d, id)$

We can now define $qbf[x] = qb[list[(); x; x]]$.

4 Characterization

Given an operator O taking functions to functions, and a class $\mathcal{C}$ of functions, we write $O(\mathcal{C})$, for the class of all functions obtained by at most one application of O to the elements of $\mathcal{C}$; $O^*(\mathcal{C})$ is the closure of $\mathcal{C}$ under O. Thus, $O(\mathcal{C})^*$ and $O^*(\mathcal{C})^*$ are the closures of $O(\mathcal{C})$ and $O^*(\mathcal{C})$ under substitution.

Definition 5. Define

POLYTIMEF LISP($\mathcal{PL}$, also $\Delta_1^p\mathcal{L}$) $= \text{FCV}^*(\mathcal{B}^*)^*$;

$\Delta_{n+2}^p\mathcal{L}$ $= \text{OR-SCV}(\Delta_{n+1}^p\mathcal{L})^*$;

POLYNOMIAL HIERARCHY LISP($\mathcal{PHL}$) $= \text{OR-SCV}^*(\mathcal{PL})^*$.

POLYSPACEF LISP($\mathcal{PSL}$) $= \text{SCV}^*(\mathcal{PL})^*$.

Theorem 6. *All Lisp classes above are equivalent to the complexity classes their names suggest.*

Proof. We have POLYTIMEF$\subseteq \mathcal{PL}$ by lemma 8. By lemma 7, all functions in $\mathcal{PL}$ are limited by a polynomial; hence, by lemma 9, $\mathcal{PL}$ $\subseteq$POLYTIMEF. By the same lemma, since the invariant in definitions by SCV is boolean, we have $\mathcal{PSL}$ $\subseteq$PSPACEF. We have PSPACEF$\subseteq \mathcal{PSL}$, since, by the example above, the PSPACE-complete set QBF can be decided in $\mathcal{PSL}$, and since $\mathcal{PL} \subseteq \mathcal{PSL}$. Lemma 10 shows the equivalence of the two hierarchies.

5 Equivalence

Lemma 7. 1 *If $f[\mathbf{x}; y]$ is FCV in g and h, with recursion variable y, then there is a constant m such that*

$$|f[\mathbf{x}; y]| \leq m|\mathbf{x}; y| \times |y|.$$

2 *Every function definable in $\mathcal{PL}$ is limited by a polynomial.*

Proof. 1 Notations like under definition 4(4). Assume that h is not boolean, and define $M = \max(lh_c(g), lh_c(h))$. Induction on $|y|$. Base. Immediately by lemma 1 (with $\mathbf{s}$ absent). Step. Assume $N := |\mathbf{x}| \geq 1$. Let $\mathbf{s}^{\,1A}$ denote the tuple of expressions such that $e_j = \alpha$ and $s_j^{1A} = e_j[f[\mathbf{x}; d_j[y]]]$ for some j; similarly for $\mathbf{s}^{\,2B}$. By lemma 1, since $lh_c(h) \leq M$, we have

$$|f[\mathbf{x}; y]| \leq M(N + |y|) + \max(\mathbf{s}^{\,1A}) + \max(\mathbf{s}^{\,2B}).$$

Since $\mathbf{d}$ is fully disjoint, there exist two sub-expressions u and w, such that $\max(\mathbf{s}^{\,1A}) = |f[\mathbf{x}; u]|$, $\max(\mathbf{s}^{\,2B}) = |f[\mathbf{x}; w]|$ and $|u| + |w| < |y|$. By the ind. hyp. we then have

$$|f[\mathbf{x}; y]| \leq M(N+|y|)+M(N+|u|)|u|+M(N+|w|)|w| \leq m(n+|y|)(1+|u|+|w|).$$

2 Induction on the construction of f. Step. If f is defined by FCV, part 1 applies. If $f[\mathbf{x}]$ is defined by SBST in $g_1[\mathbf{x}; u]$ of $g_2[\mathbf{x}]$ to u, by the ind. hyp. there are k_1, k_2, such that g_i is limited by $\lambda n.m_i n^{m_i} + m_i$, with $m_i = 2^{k_i}$; f is then limited by $\lambda n.mn^m + m$, with $m = 2^{k_1(k_2+1)}$. If $f[y] = \pi(\mathbf{g})[y]$, the result follows immediately by the ind. hyp. applied to the g's.

5.1 Simulation of TM's

Lemma 8. *All functions computable in polynomial time are definable in $\mathcal{PL}$.*

Proof. We restrict ourselves to TM's with a single semitape, that conclude their operations by entering an endless loop. *Productions* are in the form $(q_i S_j \Rightarrow q_{ij} I_{ij})$ $(i \leq s,\ j \leq t)$ where $q_i,\ q_{ij}$ are states, S_j is a tape symbol, and *instruction* I_{ij} is a new symbol or $\in$ {*right,left*}. We use the same notations for states (tape symbols) and for their codes, which are lists of s (t) atoms. *Instantaneous*

descriptions (i.d.) are coded by a list of the form (l, q, o, r), where: q and o are the state and the observed symbol; the j-th component of list r (list l) is the list of t atoms coding the j-th symbol at the right (left) of the observed symbol. A recursion-free function $next_M$ can be defined that takes an i.d. of a given TM M into the next one. Its form is
$[eql(q_1)[2d[x]] \to [eql(S_1)[3d[x]] \to exec_{11}; \ldots; eql(S_t)[3d[x]] \to exec_{1t}[x]];$
$\ldots;$
$[eql(q_s)[2d[x]] \to [eql(S_1)[3d[x]] \to exec_{s1}; \ldots; eql(S_t)[3d[x]] \to exec_{st}[x]];,$
where, for all lists of atoms p, predicate $eql(p)[x]$ is true iff $x = p$, and where $exec_{ij}$ is the function that executes instruction I_{ij}. For example, if q_{ij} is q, and I_{ij} is right, then $exec_{ij}$ is

$[eql(S_1)[car[4th[x]] \to ex_{ij1}[x]; \ldots; eql(S_t)[car[4th[x]] \to ex_{ijt}[x]],$

where ex_{ijk} is obtained from functions $g(q_i, S_j, S_k)$ in 2.5(4), by replacing (in order to add a *blank symbol* BL when M moves right to visit for the first time a new cell) the indicated *cdr* by

$[eq[\mathrm{NIL}; cdr[u]] \to (BL); T \to cdr[u]].$

Let a TM M be given, together with an input (coded by) x, and with a polynomial bound of the form $\lambda n.(h+n)^k$. From functions *mult* and *lh* of Section 3, a function p_{hk} can be defined which takes x into $num((h+|x|)^k)$; a function *start* can be defined, which takes x into the initial i.d. $(x, q_1, BL, (BL))$, where BL is the code for M's blank symbol. The following function s_M, by input x and $y = num(h)$, simulates the behaviour of M for h steps

$$\begin{cases} s_M[x; \omega] = x \\ s_M[x; y] \ = next_M[s_M[x; y'']]; \end{cases}$$

the required simulation is performed by $sim_M[x] = s_M[start[x]; p_{hk}[x]]$.

5.2 Simulation of CVR by TM's

Lemma 9. *If f is defined by CVR (FCV) and is limited by a polynomial, if its initial function is in* POLYTIMEF*, then f is in* POLYSPACEF (POLYTIMEF)*.*

Proof. Outline of the simulation. Let f be defined by CVR with notations of Definition 2. Let $g, g^*, h, \mathbf{d}$ be simulated by the TM's G, G^*, H, D_i. Assume that f is limited by a polynomial p. A TM F simulating f can be defined, which behaves in the following way.

Let θ be a n-ary tree of height $\leq |y|$, whose root is (labelled by) y, and such that: every internal node z has n children $d_1[z], \ldots, d_n[z]$; and every leaf satisfies the terminating condition decided by G^*. F visits θ in the mode known as *post-order*. It records in a stack Σ_1 the sequence of recursive calls; and it stores in a second stack Σ_2 the values $f[\mathbf{x}; d_j[z]]$ which are needed to compute $h[\mathbf{x}; z; f[\mathbf{x}; d_1[z]]; \ldots; f[\mathbf{x}; d_n[z]]]$.
Space complexity. In addition to space used by G and H, F needs space for the stacks; the amount for Σ_1 is linear in $|y|$, since we have to store $\leq |y|$ objects, each $\leq n$. When in Σ_1 there are r numbers j_q, in Σ_2 there are $\sum_{q=1}^{r}(n - j_q) \leq n|y|$ values of f; thus space for $|\Sigma_2|$ is linear in $p(|\mathbf{x}; y|) \cdot |y|$.

Time complexity. Let f be defined by FCV in g, h. Since $\mathbf{d}$ is fully disjoint, the tree θ has $\leq |y|$ nodes, and, therefore, G, H are applied less than $|y|$ times. The result follows, since their input is bounded above by p.

5.3 Equivalence of PH and PHL

Lemma 10. *For all n we have $\Delta_n^p = \Delta_n^p\mathcal{L}$.*

Proof. (Outline) $\subseteq$. Induction on n. Step. Let language $L \in \Delta_n^p$ over alphabet $\Gamma = \{S_1, \ldots, S_q\}$ be given. Let atom ω_i code S_i, and let word $w = S_{i(1)}, \ldots, S_{i(n)} \in \Gamma^*$ be coded by the list of atoms $X = (\omega_{i(1)}, \ldots, \omega_{i(n)})$. Let $g[\mathbf{x}; u] \in \Delta_n^p\mathcal{L}$ be the characteristic function of L, which is granted by the ind. hyp. We show that the characteristic function f of

$$L' = \{(X_1, \ldots, X_m, Y) : \exists U (|U| \leq |Y| \wedge (X_1, \ldots, X_m, U) \in L)\}$$

is in $\Delta_{n+1}^p\mathcal{L}$. With decreasing tuples $\pi(app(\omega_i), cdr, cdr)$, define by OR-SCV

$$f^*[\mathbf{x}; y] = \begin{cases} g[\mathbf{x}; y] & \text{if at[3d[y]]} \\ f^*[\mathbf{x}; d_1[y]] \textit{ or } \ \ldots \ \textit{ or } f^*[\mathbf{x}; d_q[y]] & \text{otherwise.} \end{cases}$$

Language L' is accepted by $f[\mathbf{x}; y] = f^*[\mathbf{x}; (); y; y]$.
$\supseteq$. Induction on n and on the construction of function $f \in \mathcal{PHL}$ to be simulated. Assume that $f[\mathbf{x}; y]$ is defined by OR-SCV in $g \in \Delta_n^p\mathcal{L}$ and h, with decreasing functions d_j (since else the result is an immediate consequence of the induction on f or of the fact that PTIMEF$= \Delta_1^p\mathcal{L}$). Let g decide language L. A nondeterministic TM M_f with oracle L can be defined, which: (1) at each call to h, iterates an invariant cycle, including, at each *or* of h, the choice of a j and the simulation of d_j; (2) at each g, queries the oracle. The time complexity of M_f is quadratic ($\leq |\mathbf{y}|$ applications of the TM's simulating functions d_j).

6 Conclusions

Normal form From proof of Lemma 8 (from the example on QBF), we see that only one level of nesting of FCV (SCV) is actually needed to compute POLYTIMEF (POLYSPACEF). This may be used to give an analogue for these classes of Kleene's *normal form theorem* for PR functions.

Classes DTIMEF(n^k). The fact above implies that to characterize these classes we have to rule the number and quality of the SBST's. For example, let $\mathcal{PL}_3$ be the Lisp class which is obtained from FCV$(\mathcal{B}^*)^*$ by excluding substitutions of the arguments of a recursive function by other recursive functions; and let $\mathcal{PL}_2$ be the further retriction of $\mathcal{PL}_3$ to recusively boolean functions; it can be proved that $\mathcal{PL}_3 \subseteq$DTIMEF(n^3), that $\mathcal{PL}_2 \subseteq$DTIME(n^2); and that if f is recursively boolean in functions in DTIMEF(n^k), then it is in DTIME(n^{k+1}). A classification of all classes DTIMESPACEF(n^k, n^m) can be easily obtained by following this approach.

Validation By scanning [7], §51,57 we see that all algorithms for the first Gödel theorem and for predicate T (a universal function) are written in a language quite close to our $\mathcal{PL}$ (besides notations, we have just to replace all bounded quantifiers by a program for search of sub-expressions). This is not surprising, since Kleene's arithmetization methods are based on his *generalized arithmetic* ([7], §50) which, in turn, may be regarded as a form of *primitive recursive Lisp*. This might point out a certain adequacy of our dialects of Lisp to represent algorithms. It might then be sensible to show the time/space complexity of an algorithm by just describing it in the language of $\mathcal{PL}$, and then checking to which element of the classification above does it belong.

Improvements to the language Types are only an apparent burden for concrete use of $\mathcal{PL}$, since we may forget them, and just watch that the previous values of the function under definition by FCV be not *cons-ed* together by the invariant, if not boolean. A more serious obstacle is that we are free to nest any number of boolean FCV's above at most one not-boolean FCV. We plan to remove this limitation by means of a re-definition of the types.

A point dividing these authors We have defined only the Δ-subclasses of PH, and not the Σ's and Π's, like NP, co-NP, etc. Some among us believe that class OR-SCV($\mathcal{PL}$) characterizes NP, while others maintain that it is *too large*. To discuss this point, let us say that language L is *accepted* by f when we have $x \in L$ iff $f[x] = T$. Indeed $\overline{SAT}$ is accepted by function $not[sat[x]]$, and this function is in OR-SCV$(\mathcal{PL})^*$, and not in OR-SCV($\mathcal{PL}$), since is defined by substitution of $sat[x]$ in function $not[x]$. Thus, from a strictly syntactic point-of-view, we might pretend that classes OR-SCV($\Delta^p_k\mathcal{L}$) are characterizations of Σ^p_{k+1}, and define $\Pi^p_k\mathcal{L}$ to be the class of all functions of the form $not[f[x]]$, $f \in \Sigma^p_k\mathcal{L}$. But perhaps we should look at more substantial facts than mere syntax: it is undeniable that, so to say, *sat knows* $\overline{SAT}$; thus one is entitled to say that OR-SCV($\mathcal{PL}$) is not well-defined with respect to resources, and is not an acceptable characterization of NP.

Stronger forms of recursion. Let $<_S$ be a total order of the S-expressions. Let us say that f is defined by *n-strong* CVR if $f[y]$ depends on n values $f[y_i]$, such that, for all i, we have $y_i <_S y$. It can be easily proved that POLYSPACEF is closed under 1-strong CVR. Apparently ([3]), it can be proved that POLYTIMEF is not closed under 2-strong CVR; and that if POLYSPACEF is closed under 2-strong CVR, then POLYSPACE = EXPTIME. The proof of this result fails when relativized to oracle-TM's.

References

References

1. S. Bellantoni and S. Cook, A new recursive characterization of the polytime functions, in 24th Annual ACM STOC (1992) 283-293.

2. S. Caporaso, Safe Turing machines, Grzegorczyk classes and Polytime (to appear on Intern. J. Found. Comp. Sc.).
3. S. Caporaso and M. Zito, On a relation between uniform coding and problems of the form DTIME($\mathcal{F}$) =? DSPACE($\mathcal{F}$), submitted to Acta Informatica.
4. P. Clote, A time-space hierarchy between polynomial time and polynomial space, Math. Systems Theory, 25(1992) 77-92.
5. A. Cobham, The intrinsic computational complexity of functions, in Y. Bar Hillel, ed., Proc. Int. Conf. Logic, Methodology and Philosophy Sci. (North-Holland, Amsterdam, 1965) 24-30.
6. J. Hartmanis and R.E. Stearns, On the computational complexity of algorithms. Trans. A.M.S. 117(1965)285-306.
7. S.C. Kleene, Introduction to metamathematics (North-Holland, Amsterdam, 1952).
8. M. Kutylowski, A generalyzed Grzegorczyk hierarchy and low complexity classes, Information and Computation, 72.2(1987) 133-149.
9. D. Leivant, A foundational delineation of computational feasibility. 6th Annual IEEE symposium on Logic in computer science (1991).
10. D.Leivant and J.Y. Marion, Lambda calculus characterization of Poly-time (to appear on Fundamenta Informaticae).
11. M. Liskiewicz, K. Lorys, and M. Piotrow, The characterization of some complexity classes by recursion schemata, in Colloquia Mathematica Societatis János Bolyai, 44, (Pécs, 1984) 313-322.
12. Mecca and Bonner, Sequences, Datalog and transducers, PODS 1995 (reference to be revs'd in the final version of this paper).
13. C.H. Papadimitriou, A note on expressive power of PROLOG, Bull. EATCS, 26(1985) 21-23.
14. H. Poincaré Les mathématiques et la logique, Revue de Métaphisique et de Morale, 14(1906)297-317.
15. R.W. Ritchie, Classes of predictably computable functions, Trans. Am. Math. Soc. 106(1963) 139-173.
16. H.E. Rose, Subrecursion: Functions and hierarchies, (Oxford Press, Oxford, 1984).