POLYNOMIAL AND ABSTRACT SUBRECURSIVE CLASSES*

Kurt Mehlhorn

TR 74-198

February 1974

Computer Science Department
Cornell University
Ithaca, New York 14850

*The research presented in this paper was supported in part by the NSF under Grant GJ-579.

POLYNOMIAL AND ABSTRACT SUBRECURSIVE CLASSES*

Kurt Mehlhorn
Department of Computer Science
Cornell University
Ithaca, New York
14850

Abstract:

We define polynomial time computable operator. Our definition generalizes Cook's definition to arbitrary function inputs. Polynomial classes are defined in terms of these operators; the properties of these classes are investigated. Honest polynomial classes are generated by running time. They posses a modified Ritchie-Cobham property. A polynomial class is a complexity class iff it is honest.

Starting from the observation that many results about subrecursive classes hold for all reducibility relations (e.g. primitive recursive in, elementary recursive in), which were studied so far, we define abstract subrecursive reducibility relation. Many results hold for all abstract subrecursive reducibilities.

I. Introduction:

Subrecursive reducibility relations allow us to classify all computable functions into subrecursive classes. Several such relations (e.g. elementary recursive in, primitive recursive in, doubly recursive in, ...) are mentioned in the literature [1,10,11,12,15,16]. All these relations are rather coarse; in particular all functions, the complexity of which is not greater than $\lambda x\,[2^x]$, belong to the same class. In the case of decision problems (sets) the situation is different. Cook [5] defined the relation 'polynomial computable in' for decision problems; it is a proper refinement of the relation 'elementary recursive in' and splits the class of sets, the complexity of which is subexponential, into different classes. It is desirable to generalize the notion 'polynomial computable in', such that it is defined between arbitrary computable functions. Constable [3] started research in this direction; we continue his work.

In section II we define polynomial time computable operator. We give two equivalent definitions for this class of operators: one is in terms of resource bounded oracle Turing machines, the other one is a syntactic definition. The polynomial class generated by a function f is the class of functions, which are obtained by application of the polynomial operators to f; it is always a subset of the class of functions, which are elementary recursive in f.

In section III we study the properties of polynomial classes. For example, we show that every countable partial ordering can be embedded into the polynomial classes $\mathcal{C}$, such that $\mathcal{A} \subset \mathcal{C} \subset \mathcal{B}$ (we assume $\mathcal{A} \subset \mathcal{B}$).

In section IV we study the relation between polynomial classes and computational complexity. Polynomial classes classify computable functions according to their complexity; i.e. if f is polynomial computable in g ($f \leq_{pol} g$), then for every running time T_g of g there is a running time T_f of f such that $T_f \leq_{pol} T_g$. Honest polynomial classes are polynomial classes, which are generated by running time. Every generator of a honest class is honest; honest classes satisfy a modified Ritchie-Cobham property. A polynomial class is honest iff it is a complexity class. Finally we consider density properties of honest polynomial classes.

Machtey [12] notes that several properties of the primitive recursive reducibility hold true for several other reducibilities including elementary, doubly recursive, and multiply recursive reducibilities. Ladner [10] shows that the range of these results is even wider. The results are true of a wide variety of Turing machine space or time definable reducibilities (if we restrict our attention to decision problems). We remove the restriction to reducibilities, which are definable in terms of resource bounded oracle Turing machines. We define abstract subrecursive reducibility; the definition is in the spirit of Strong's definition [20] of BRFTs. An abstract subrecursive re-

*The research presented in this paper was supported in part by the NSF under Grant GJ-579.

ducibility is defined by a set of general recursive operators which contains the operator APPLY and the constant operators, which is closed under composition, definition by cases and forming finite variants, and which possesses a rudimentary simulation feature.

II. Polynomial Operators

In this section we define polynomial time computable operator; we denote the class of polynomial time computable operators by $\mathcal{P}ol(\ )$. $\mathcal{P}ol(f)$ is the polynomial class generated by f; it is the result of applying the operators in $\mathcal{P}ol(\ )$ to f. Cook [5] gave a definition for $\mathcal{P}ol(f)$ in the case that f is a characteristic function. A very simple definition suffices in this case: $g \in \mathcal{P}ol(f)$ if g can be computed by an oracle Turing machine, the running time of which is bounded by a polynomial, using an oracle for f. Constable [3] gave a definition for $\mathcal{P}ol(\ )$ in the general case. His definition is also in terms of resource bounded oracle machines. Since the bounds are considerably more complex than in the simple case of set inputs, a complexity-theoretic definition does not provide us with an efficient and transparent definition of $\mathcal{P}ol(\ )$. Because of this Constable also proposed a syntactic definition of $\mathcal{P}ol(\ )$, which he called $\mathcal{K}(\ )$. By syntactic definition we mean a characterization as closure class: simple operations on operators together with a small set of simple basic operators generate the entire class. Unfortunately $\mathcal{P}ol(f) = \mathcal{K}(f)$ holds only for nondecreasing functions f. Our definition is a variant of Constable's definition; it agrees with his definition in the case of nondecreasing functions and it agrees with Cook's definition in the case of characteristic functions. For our definition we are able to prove the equality between $\mathcal{P}ol(\ )$ and the syntacticly defined class of operators.

$\mathcal{P}ol$ is the class of functions which can be computed in polynomial time on a Turing machine. Several investigators [2,21] gave a syntactic definition for $\mathcal{P}ol$. Weihrauch defined the class $\mathcal{E}_2^\Sigma = [S_1, S_2, \ldots, S_r, A_2; Os, R_<^N]$; it is the smallest class of functions, which contains the generalized successor functions $S_i(x) = xa_i$ (for all $x \in \Sigma^*$, $\Sigma = \{a_1, \ldots, a_r\}$, $r \geq 2$), the length bounded exponentiation function $A_2(x,y) = a_1^{|x| \cdot |y|}$ ($|x|$ is the length of the string x) and is closed under the operations of substitution [4] and limited recursion on notation. f is defined from g, $h_1, \ldots, h_r$ and b by limited recursion on notation if for some $n \geq 0$

$$g: (\Sigma^*)^n \to \Sigma$$

$$h_i: (\Sigma^*)^{n+2} \to \Sigma \qquad (1 \leq i \leq r)$$

$$b: (\Sigma^*)^{n+1} \to \Sigma$$

$$f: (\Sigma^*)^{n+1} \to \Sigma$$

$$f(\vec{x}, \varepsilon) = g(\vec{x})$$

$$f(\vec{x}, ya_i) = h_i(\vec{x}, y, f(\vec{x}, y))$$

$$|f(\vec{x}, y)| \leq |b(\vec{x}, y)|$$

for all $\vec{x} \in (\Sigma^*)^n$ and $y \in \Sigma^*$.

Fact: [2,21] $\mathcal{P}ol = \mathcal{E}_2^\Sigma$

The following simple observation leads in a natural way to our definition.

Observation: Let $g \in \mathcal{E}_2^\Sigma$, $g: (\Sigma^*)^n \to \Sigma$. Then there is a polynomial $p: \mathbb{N} \to \mathbb{N}$ such that for all $\vec{x} \in (\Sigma^*)^n$

$$|g(\vec{x})| \leq p(\max\{|x_i|;\ 1 \leq i \leq n\})$$

Conversely let p be any polynomial. Then there is a function $g \in \mathcal{E}_2^\Sigma$, $g: \Sigma^* \to \Sigma^*$, such that for all $x \in \Sigma^*$

$$p(|x|) \leq |g(x)|$$

Using this observation we can give an alternate definition of $\mathcal{P}ol$

$$\mathcal{P}ol = \{f;\ \text{there is a TM M computing f such that the running time of M is bounded by some } h \in |\mathcal{E}_2^\Sigma|\}^*$$

The following equation summarizes the discussion up to this point.

$$\mathcal{E}_2^\Sigma = \{f;\ f \text{ can be computed within time bound } h \text{ for some } h \in |\mathcal{E}_2^\Sigma|\}$$

In the following we extend this equality to the operator level. As machine model we use oracle Turing machines [11,19].

Def 2.1: $\mathcal{E}_2^\Sigma(f) = [S_1, S_2, \ldots, S_r, A_2, f;\ Os, R^N_{\leq}]$ is the smallest class of functions, which contains the base functions $S_1, S_2, \ldots, S_r, A_2$ and f and is closed under the operations of substitution and limited recursion on notation. If we consider f as an uninterpreted function symbol, then $\mathcal{E}_2^\Sigma(f)$ is a class of operators. We denote it by $\mathcal{E}_2^\Sigma(\ )$.

Def 2.2: $\mathcal{P}ol(\ ) = \{\phi[\ ];$ there is an oracle machine M, which computes the operator $\phi[\ ]$, and a bounding expression $G \in |\mathcal{E}_2^\Sigma(\ )|$, such that for all f and x : the running time of M on f and $\vec{x}$ is bounded by $G[f](\vec{x})\}$

Example: APPLY[ ] (APPLY[f](x) = f(x)) is a polynomial time computable operator. We describe an oracle machine M, which computes APPLY[ ]. M copies the input from the input tapes on its oracle input tapes ($|x_1| + |x_2| + \ldots + |x_n|$ steps), calls the oracle (1 step) and finally copies the result of the oracle call from the oracle output tape on its output tape ($|f(x)|$ steps). Let G be the following operator in $\mathcal{E}_2^\Sigma(\ )$

$$G[f](\vec{x}) = a_1\, x_1\, x_2 \ldots x_n\, f(\vec{x})$$

$|G[\ ]|$ is a bounding expression for M.

Theorem 2.1: $\mathcal{P}ol(\ ) = \mathcal{E}_2^\Sigma(\ )$

Proof:

a) $\mathcal{E}_2^\Sigma(\ ) \subseteq \mathcal{P}ol(\ )$: For every $H \in \mathcal{E}_2^\Sigma(\ )$ we construct an oracle machine M and a bounding expression $G \in \mathcal{E}_2^\Sigma(\ )$ such that for all f and $x \in (\Sigma^*)^n$

$$M[f](\vec{x}) = H[f](\vec{x})$$
$$T_M[f](\vec{x}) \leq |G[f](\vec{x})|$$

($T_M[f](\vec{x})$ is the running time of M on f and x). We proceed by induction on the structure of H. The base step and part of the induction step (the operations of substitution) are trivial and left to the reader. It remains to consider limited recursion on notation.

Let $H_0, H_1, \ldots, H_r$ and B be in $\mathcal{E}_2^\Sigma(\ )$ and let H be defined from them by limited recursion on notation.

$$H[f](\vec{x}, \varepsilon) = H_0[f](\vec{x})$$
$$H[f](\vec{x}, ya_i) = H_i[f](x, y, H[f](\vec{x}, y))$$
$$|H[f](\vec{x}, y)| \leq |B[f](\vec{x}, y)|$$

Let oracle machine M_i compute H_i and let G_i be a bounding expression for M_i. We describe M.

---

*Let $\mathcal{A}$ be a set of functions which map $(\Sigma^*)^n$ into Σ. Then $|\mathcal{A}|$ is a set of functions which map $(\Sigma^*)^n$ into $\mathbb{N}$. $h \in |\mathcal{A}|$ if there is a $g \in \mathcal{A}$ such that $h(\vec{x}) = |g(\vec{x})|$.

Step 1: Copy $\vec{x}$ on the input tapes of the M_i's ($1 \leq i \leq r$) and reset the heads ($2(|x_1|+\ldots+|x_n|)$ steps).

Step 2: Compute $H_0[f](x)$ using M_0 ($T_{M_0}[f](\vec{x})$ steps) and reset the head on the output tape ($|H_0[f](x)|$ steps). Let the current output tape be the output tape of M_0.

Step 3: Copy the next symbol of y (say a_j) on the appropriate square of the $n+1^{st}$ input tape of M_i ($1 \leq i \leq r$) and reset the heads of all input tapes ($2|y|+|x_1|+\ldots|x_n|$ steps).

If there is no next symbol then copy the content of the current output tape on the output tape ($|H[f](\vec{x},y)|$ steps) and halt.

Step 4: Compute $H_j[f](x,y_1\ldots y_k,H[f](x,y_1\ldots y_k))$ using M_j; M_j uses the current output tape as its $n + 2^{nd}$ input tape ($T_{M_j}[f](\vec{x},y_1\ldots y_k, H[f](\vec{x},y_1\ldots y_k))$ steps). Let the current output tape be M_j's output tape. Goto step 3.

The running time of M is bounded by

$$T_M[f](\vec{x},y) \leq 2(|x_1|+\ldots+|x_n|) + T_{M_0}[f](\vec{x}) + |H_0[f](\vec{x})|$$

$$+ \sum_{k=0}^{|y|-1} (2|y| + |x_1| +\ldots+ |x_n| + T_{M_{j_k}}[f](x,y_1\ldots y_k,H[f](\vec{x},y_1\ldots y_k)))$$

$$+ |H[f](\vec{x},y)|$$

$$\leq |A_2(x_1\ldots x_n,a_1a_1y)A_2(y,y)\ A_2\ (y,y)\ G_0[f](\vec{x})\ H_0[f](\vec{x})\ H[f](\vec{x},y)$$

$$\prod_{k=0}^{|y|-1} G_{M_{j_k}}[f](\vec{x},y_1\ldots y_k,\ H[f](\vec{x},y_1\ldots y_k))|$$

In order to show that the last expression is in $\mathcal{E}_2^{\Sigma}(\ )$ it is sufficient to show that $\mathcal{E}_2^{\Sigma}(\ )$ is closed under length bounded concatenation.

Lemma 2.1: If $G[\ ](\vec{x},y) \in \mathcal{E}_2^{\Sigma}(\ )$ then

$$\prod_{k=0}^{|y|} G[\ ](\vec{x},y_1\ldots y_k) \in \mathcal{E}_2^{\Sigma}(\ ).$$

Proof: The product is defined by limited recursion on notation. In order to get a bound on the length of the product we determine first the longest operand of the product.

$$L[f](\vec{x},\varepsilon) = \varepsilon$$
$$L[f](\vec{x},ya_i) = \underline{\text{if}}\ |G[f](\vec{x},ya_i)| < |G[f](\vec{x},L[f](\vec{x},y))|\ \underline{\text{then}}\ ya_i\ \text{else}\ L[f](\vec{x},y)$$
$$|L[f]\ (\vec{x},y)| \leq |y|$$

and

$$\prod_{k=0}^{|\varepsilon|} G[f](\vec{x},y_1\ldots y_k) = G[f](\vec{x},\varepsilon)$$

$$\prod_{k=0}^{|ya_i|} G[f](\vec{x},y_1\ldots y_k) = \prod_{k=0}^{|y|} G[f](\vec{x},y_1\ldots y_k)\ \ G[f](\vec{x},ya_i)$$

$$\left|\prod_{k=0}^{|y|} G[f](\vec{x},y_1\ldots y_k)\right| \leq |A_2(a_1y,G[f](\vec{x},L[f](\vec{x},y)))|$$

b) $\mathcal{P}ol(\ ) \subseteq \mathcal{E}_2^{\Sigma}(\ )$: For every operator $Op[\ ] \in \mathcal{P}ol$ (computed by oracle machine M and bounding expression $G[\ ] \in |\mathcal{E}_2^{\Sigma}(\ )|$) we have to construct an operator $H[\ ] \in \mathcal{E}_2^{\Sigma}(\ )$, such that $Op[f](\vec{x}) = H[f](\vec{x})$ for all functions f and inputs $\vec{x} \in (\Sigma^*)^n$. H is constructed by arithmetization and simulation of oracle machines [18]. From M we define functions Init $(\vec{x})$, Decode(y) and an operator Yield[ ]. Init $(\vec{x})$ is the Godel number of the initial instantaneous description of M. Yield[f](y) is the Godel number of the instantaneous description, which is the result of applying M (with oracle f) to the instantaneous description, whose Godel number is y. Decode(y) is the content of the output tape of the instantaneous description y. Init, Decode are in $\mathcal{E}_2^{\Sigma}$, Yield[ ] is in $\mathcal{E}_2^{\Sigma}(\ )$. Now we define a functional U[ ] by limited recursion on notation.

$$U[f](\vec{x},\varepsilon) = \text{Init}\ (x)$$
$$U[f](\vec{x},ya_i) = \text{Yield}[f](U[f](\vec{x},y)$$
$$|U[f](\vec{x},y)| \leq ?$$

In order to exhibit a bound for the length of $U[f](\vec{x},y)$, we have to derive a relation between the length of the non-blank portions of the tapes and the length of the computation. Such a relation is easily established for all tapes except the oracle output tape. In order to establish the relation for the oracle output tape, one only has to note, that though the oracle machine is charged only a nominal amount for calling the oracle (1 step) it is charged in a reasonable way for reading the output of the oracle call. (Compare the example preceeding the proof). A complete proof can be found in the full paper.

Def 2.3: $\mathcal{P}ol(f)$ is the polynomial class generated by f. A function g is polynomial computable in f (write $g \leq_{pol} f$) if $f \in \mathcal{P}ol(g)$.

Theorem 2.2: (Basic properties of $\leq_{pol}$)

a) Our definition of $\mathcal{P}ol(f)$ agrees with Cook's definition [5] in the case of characteristic functions.

b) Our definition of $\mathcal{P}ol(f)$ agrees with Constable's definition [3], in the case of non-decreasing functions.

c) $\leq_{pol}$ is transitive and reflexive

d) $\mathcal{P}ol(\ )$ is recursively enumerable

e) $\mathcal{P}ol(f) \subseteq \mathcal{E}(f)$ (= the set of functions elementary recursive in f [12,16]) for all f. If $|f(x)| \geq 2^{|x|}$ for all x, then $\mathcal{P}ol(f) = \mathcal{E}(f)$.

Remark 1: Polynomial classes are always subclasses of the elementary classes (Thm 2.2.e), but not always proper subclasses. This fact might be surprising, however, it is not counterintuitive. Intuitively speaking polynomial operations are length-bounded (length-bounded search, length-bounded summation $\sum_{i=0}^{|x|}$, length-bounded product $\prod_{i=0}^{|x|}$,...), elementary operations are value-bounded (value-bounded summation $\sum_{i=0}^{x}$, value-bounded product $\prod_{i=0}^{x}$,...). In the presence of exponentiation the difference disappears.

Remark 2: By 2.2.d $\mathcal{P}ol(\ )$ is recursively enumerable. For the following we assume a fixed enumeration $\{Op_i[\ ]\}_{i=0}^{\infty}$.

III. On the structure of polynomial classes

In this section we consider the relation $\leq_{pol}$ in greater detail. First we consider the density properties of the relation. We state only the most simple density result and give an informal proof for it. By doing so we hope to build up enough intuition for the abstract treatment of density in section V.

Theorem (3.1): (see also [10]) Let g be a computable function such that $0 <_{pol} g$. There is a computable function h such that $0 <_{pol} h <_{pol} g$.

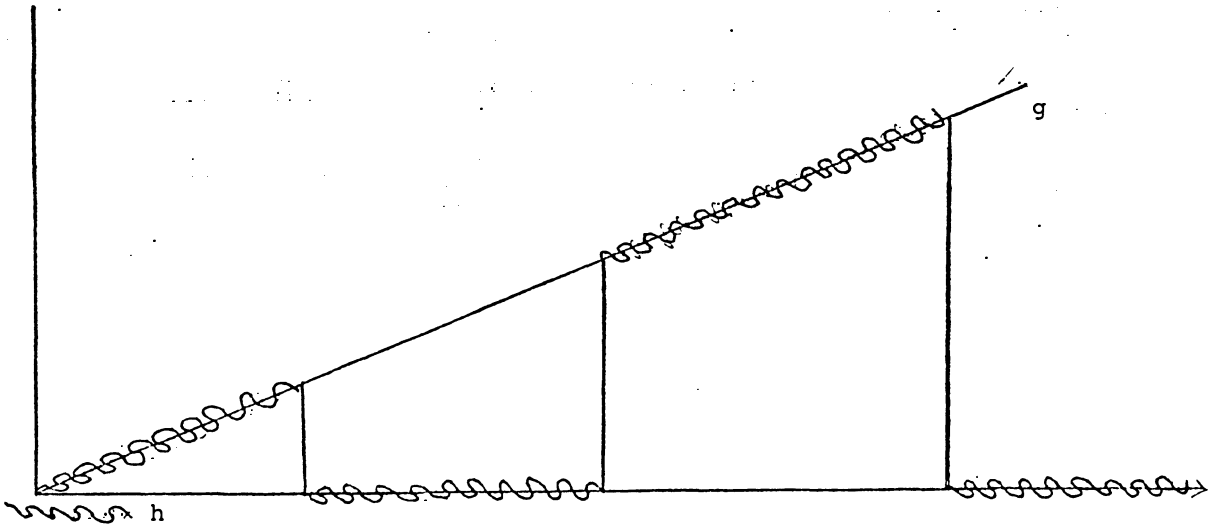
Proof: We have to construct a function h such that

(1) $h \leq_{pol} g$

(2) $g \not\leq_{pol} h$, i.e. $g \neq Op_i[h]$ for all i

(3) $h \not\leq_{pol} 0$, i.e. $h \neq Op_i[0]$ for all i

h is constructed in pieces as indicated in the figure.



During the construction of h we can be in either one of two modes (2) or (3) and we try to cancel some index i. Assume we are in mode (2) and we want to cancel index i; i can be cancelled if we have ensured that $g \neq Op_i[h]$. At any point of the construction only a finite initial segment of h is defined. If we would extend h to be the constant zero almost everywhere then certainly $g \neq Op_i[h]$; hence $g(x) \neq Op_i[h](x)$ for some x. This shows that if we define $h(x) = 0$ for sufficiently many arguments then $g \neq Op_i[h]$. We must also be able to detect that $g \neq Op_i[h]$. Since $g \neq Op_i[h]$ implies $g(x) \neq Op_i[h](x)$ for some x we will be able to detect this fact in $|y|$ steps for sufficiently long y. At this argument we will cancel i and change the mode to (3).

We describe now a program for h. Let $\{x_i\}_{i=1}^{\infty}$ be an enumeration of Σ^*.

To compute h(x):

(A): let $n = |x|$;
execute n steps of the following program
"compute $g(x_1)$, $g(x_2)$,... and build up a table G of the values";
execute n steps of the following program
"compute $h(x_1), h(x_2)$,... and build up a table H of the values"
let G_0 and H_0 be the tables after n steps.
Using the tables G_0 and H_0 execute n steps of the following program
"mode ← (2), index ← 1;
Loop: i ← 1
while $G(x_i) = Op_{index}[H](x_i)$ do i ← i + 1;
mode ← (3);
i ← 1;
while $H(x_i) = Op_{index}[0](x_i)$ do i ← i + 1
mode ← (2);
goto Loop"
let $mode_0$ be the value of mode after n steps.

(B) if $mode_0 = (2)$ then output ← 0
else output ← g(x)

The execution time of this operator is bounded by $c_1 + c_2|x|$ for some constants c_1, c_2; hence $h \leq_{pol} g$. To show $0 <_{pol} h <_{pol} g$ it is sufficient to show that mode and index do not reach stable values. Assume otherwise, say mode reaches the stable value (2). Then $h(x) = 0$ for almost all x; from $g = Op_{index}[h]$ and h is a function of finite support we infer $g \leq_{pol} 0$. But $0 <_{pol} g$ by hypothesis. If the stable value of mode is (3) then $h(x) = g(x)$ for almost all x. $h = Op_{index}[0]$ implies $g \leq_{pol} 0$.

The use of recursion in part A of the program for h can be justified by an application of the recursion theorem. This is done in section VI.

The next theorem can be proved using essentially the same technique.

Thm 3.2: Let f,g be such that $f <_{pol} g$. Every countable partial ordering can be embedded into the set $\{h;\ f <_{pol} h <_{pol} g\}$

Corollary 3.3: Every countable partial ordering can be embedded into the set of functions of subexponential complexity.

The proofs in [10] can be adapted to prove:

Thm 3.4: There are functions f,g with $0 <_{pol} f$ and $0 <_{pol} g$ such that $h \leq_{pol} f$ and $h \leq_{pol} g$ imply $h \leq_{pol} 0$.

Thm 3.5: The polynomial classes do not form a lattice under set inclusion.

IV. Honest Polynomial Classes:

In this section we study the relation between the reducibility 'polynomial computable in' and computational complexity. First we show that the reducibility orders functions according to their complexity. Then we define honest polynomial class, i.e. a class which is generated by a running time, and study their relation to complexity classes. Honest classes possess a modified Ritchie-Cobham property; a polynomial class is honest iff it is a complexity class. In [12] similar results are shown for the elementary honest classes.

Def 4.1: Let M be any Turing machine; say M has n input tapes. $T: (\Sigma^*)^n \to \{a_1\}^\infty$ (= the set of finite or infinite strings of a_1's) is the running time of M (is a time bound for M) if for all $\vec{x} \in (\Sigma^*)^n$ M halts in exactly (less than) $|T(x)|$ steps on input $\vec{x}$.

For us a running time (time bound) maps strings to strings. Usually ([8]) a time bound is a mapping from integers to integers, such that the length of no computation on an input of a certain length exceeds the time bound applied to that length. The standard definition is too coarse for our purposes; most of the following theorems are false for the standard definition.

Lemma 4.1: Let M be any TM. Then there exists a TM P which computes the running time T_M of M such that

$$T_P = T_M$$

Proof: P looks almost like M, except that M's output tape is a worktape for P. P simulates M; at every step it prints a_1 on its output tape and advances the head by one square.

Lemma 4.1 states that running times are very honest. They can be computed in a time bound, which is equal to their size. The next lemma states an important simulation property.

Lemma 4.2: For every Turing machine M there exists a polynomial time bounded TM S_M which on input $\vec{x}$, t simulates M on $\vec{x}$ for exactly $|t|$ steps.

Proof: obvious.

Corollary 4.1: Let f be a computable function and let the TM M compute f. Then $f \in \mathcal{P}ol(T_M)$.

Proof: By lemma 4.2 there is a simulator $S_M \in \mathcal{P}ol$ for M. Also by lemma 4.2 $M(x) = S_M(x, T_M(x))$. Hence $f \in \mathcal{P}ol(T_M)$.

The next theorem states that $\leq_{pol}$ orders the computable functions according to their complexity.

Thm 4.2: Let $f \in \mathcal{P}ol(g)$, let M compute g and let T_M be the running time of M. Then there is a TM P computing f such that

$$T_P \in \mathcal{P}ol(T_M)$$

Proof: Let R be an oracle machine in $\mathcal{P}ol(\ )$ which maps g into f, let $G \in \mathcal{E}_2^{\Sigma}(\ )$ be a bounding expression for R. P is the result of replacing the oracle calls in R by the TM M for g; symbolically

$$P = R + M$$

The following inequality holds for the running times.

$$|T_P(\vec{x})| \leq |T_{R[g]}(x)| + \sum_{\substack{g(\vec{y}) \text{ is} \\ \text{called during} \\ \text{the computation } R[g](\vec{x})}} |T_M(\vec{y})| + \text{overhead}$$

$$\leq |G[g](\vec{x})| + \sum_{g(\vec{y}) \text{ is called}} |T_M(y)| + \text{overhead} \quad .$$

From Corollary 4.1 we know that $g \in \mathcal{P}ol(T_M)$. Hence $G[g] \in \mathcal{P}ol(T_M)$. The overhead is bounded by a polynomial in the length of the input. It remains to show that

$$(*) \qquad \sum_{g(\vec{y}) \text{ is called}} |T_M(\vec{y})| \leq |\mathcal{P}ol(T_M)|$$

We proceed in two stages. First we construct an operator R' from R. R' simulates R and whenever R calls the oracle R' prints an encoded version of the input of the oracle call on its output tape (e.g. one can use the encoding: $a_i \to a_2 a_i$; $a_1 a_1$ serves as a delimiter). Apparently $|T_{R'}[g](\vec{x})| \leq 2\,|T_R[g](\vec{x})|^2$. Another oracle machine reads this string, divides it into its components and applies the oracle for T_M to them. Its running time is bounded by the length of the input. The combined operator establishes (*).

If we view the operators in $\mathcal{P}ol(\ )$ as possible ways of extracting information then $\mathcal{P}ol(f)$ is the information content of f [13]. Using this intuitive concept Thm 4.2 reads as follows: If the information content of f is smaller than the information content of g then the complexity of f is smaller than the complexity of g. The converse is not true in general. This is due to the fact that information content is a "two-dimensional" concept; it depends on the size of the function values and on their interdependence.

Def 4.2: a) $\mathcal{P}ol(f)$ is a honest polynomial class if $\mathcal{P}ol(f) = \mathcal{P}ol(T_M)$ where T_M is the running time of some TM M.

b) f is honest if $\mathcal{P}ol(f)$ is honest.

The next theorem shows that definition 4.2 agrees with the conventional definition of honesty: a function is honest if its complexity does not differ too much from the function itself.

Thm 4.3: f is honest iff there is a TM M which computes f such that

$$T_M \in \mathcal{P}ol(f).$$

Proof: <= : Assume M computes f and $T_M \in \mathcal{P}ol(f)$. By corollary 4.1 $f \in \mathcal{P}ol(T_M)$, hence $\mathcal{P}ol(T_M) = \mathcal{P}ol(f)$.

=> : Let $\mathcal{P}ol(f) = \mathcal{P}ol(T_P)$ for some TM P. By lemma 4.1 there is a TM Q computing T_P such that $T_Q = T_P$. By theorem 4.2 there is a TM M which computes f such that

$$T_M \in \mathcal{P}ol(T_Q) = \mathcal{P}ol(T_P) = \mathcal{P}ol(f).$$

Def 4.3: Let $t: \Sigma^* \to \mathbb{N}$. The complexity class determined by t is defined as:

$$C_t = \{f;\ \text{there is a TM M computing f such that } |T_M(x)| \leq t(x) \text{ a.e.}\}$$

Corollary 4.4: g is honest iff $\mathcal{P}ol(g) = \bigcup_{h \in \mathcal{P}ol(g)} C_{|h|}$

Proof:

=> : Assume g is honest. Hence there is a TM M, which computes g such that $T_M \in \mathcal{P}ol(g)$.

$\supseteq$: Assume $f \in C_{|h|}$ and $h \in \mathcal{P}ol(g)$. By lemma 4.2 therefore $f \in \mathcal{P}ol(g)$.

$\subseteq$: Assume $f \in \mathcal{P}ol(g)$. By theorem 4.2 there is a TM P computing f such that $T_P \in \mathcal{P}ol(T_M) = \mathcal{P}ol(g)$. Hence $f \in C_{|T_P|} \subseteq \bigcup_{h \in \mathcal{P}ol(g)} C_{|h|}$.

<= : obvious.

Machtey [14] has shown that honest polynomial classes cannot satisfy the Ritchie-Cobham-property. However they satisfy the modified Richie-Cobham property of corollary 4.4. The next theorem establishes the connection between honest classes and complexity classes.

Thm 4.5: g is honest iff $\mathcal{P}ol(g)$ is a complexity class.

Proof:

=>: Let g be honest; then $\mathcal{P}ol(g) = \bigcup_{h \in \mathcal{P}ol(g)} C_{|h|}$ by corollary 4.4. Let $\{h_i\}_{i=1}^{\infty}$ be an enumeration of $\mathcal{P}ol(g)$. We define

$$\hat{h}_i(x) = h_1(x)\ldots h_i(x)$$

Then $\hat{h}_i \in \mathcal{P}ol(g)$ and $|\hat{h}_i| \leq |\hat{h}_j|$ for $i < j$. The union theorem [7] implies that

$$\mathcal{P}ol(g) = \bigcup_{h \in \mathcal{P}ol(g)} C_h = \bigcup_{i \in \mathbb{N}} C_{|\hat{h}_i|} = C_t$$

for some $t: \Sigma^* \to \mathbb{N}$.

<=: Assume $\mathcal{P}ol(g) = C_t$ for $t: \Sigma^* \to \mathbb{N}$. Since $g \in \mathcal{P}ol(g)$ there is a TM M such that

$$|T_M(x)| \leq t(x) \text{ for almost all } x.$$

By lemma 4.1 there is a TM P, which computes T_M such that $T_P = T_M$. Hence $|T_P(x)| \leq t(x)$ for almost all x and hence $T_M \in C_t = \mathcal{P}ol(g)$. By theorem 4.3. g is therefore honest.

Thm 4.6: Let f,g be honest functions with $f <_{pol} g$. Then every countable partial ordering can be embedded into the functions $\{h;\ f <_{pol} h <_{pol} g$ and h honest$\}$

Proof: Adapt the proof of theorem 3.2.

Open problem: Are the honest polynomial classes a lattice under set inclusion?

V. Abstract Subrecursive Reducibilities

Axt [1] and Machtey [12] study the properties of the primitive recursive reducibility. Machtey [12] notes that several properties of the primitive recursive reducibility are true for other reducibilities including elementary, doubly recursive, triply recursive, ..., and multiple recursive reducibility. Ladner [10] observes that the results have an even wider range. He investigates reducibilities which are induced by resource bounded oracle machine computations. All results of [10] hold true provided the class of resource bounds (time or tape bounds) satisfies reasonable closure properties. Only sets are allowed as oracles. We extend his results in two directions: we remove the restriction to reducibilities, which are definable in terms of Turing machine space or time, and we remove the restriction to 0-1 valued oracles.

Subrecursive reducibilities are always induced by a class of general recursive operators [19]; i.e. operators which are defined on all total functions and take total functions into total functions. E.g. in section II we defined the reducibility 'polynomial computable in' in terms of the class of polynomial time computable operators. An abstract subrecursive reducibility is defined by a class of general recursive operators, which satisfies reasonable closure properties. The closure properties are in the spirit of Strong's definition of Basic Recursive Function Theories [20].

Def 5.1: Let $\mathcal{R}$ be the set of computable functions, let $\mathcal{R}^*$ be the set of 0-1 valued computable functions and let $\{\phi_i[\ ]\}$ be an acceptable indexing of the recursive operators.

Let $\mathcal{C}$ be either $\mathcal{R}$ or $\mathcal{R}^*$. $s: \mathbb{N} \to \mathbb{N}$ defines an abstract subrecursive reducibility over $\mathcal{C}$ if

(1) $Op_i[\ ] = \phi_{s(i)}[\ ]$ is a general recursive operator

(2) there is a constant c such that $Op_c[\ ] = APPLY[\ ]$, i.e. $Op_c[f](x) = f(x)$

(3) there is a computable function const such that

$$Op_{const(i)}[f](x) = i$$

(4) there is a computable function comp such that

$$Op_{comp(i,j)}[f](x) = Op_i[\lambda y\ Op_j[f](y)](x)$$

(5) there is a computable function cond such that

$$Op_{cond(i,u,k,\ell)}[f](x) = \underline{\text{if}}\ Op_i[f](x) = Op_j[f](x)\ \underline{\text{then}}\ Op_k[f](x)\ \underline{\text{else}}\ Op_\ell[f](x)$$

(6) a) for every i, g and finite function f there is a j such that

$$Op_j[g](x) = \begin{cases} f(x) & \text{if } x \in \text{dom } f \\ Op_i[g](x) & \text{otherwise} \end{cases}$$

b) for every i, g, f such that g = f almost everywhere there is a j such that

$$Op_j[g] = Op_i[f]$$

(7) there is a computable function sim such that

$$Op_{sim(i)}[f](x) = \phi_i[f](\phi_{n(i)}[f](x))$$

and $\phi_{n(i)}[f]$ is a non-decreasing function for every i and f. Furthermore if $\phi_i[f](y)\downarrow$ for all $y \leq x$ then $x \in$ range $\phi_{n(i)}[f]$.

In (1) - (7) f and g are to be taken from $\mathscr{C}$.

Remark: An abstract subrecursive reducibility is given by a list of total operators (1), which contains the operator APPLY[ ] (2), the constants (3), which is closed under composition (4), definition by cases (5) and finite variants (6) and which has a rudimentary simulation feature (7).

As we stated the definition, computations are done over the integers. Equally well we could have stated the definition, such that computations are executed over Σ^*. All reducibilities which are mentioned in the literature satisfy definition 5.1.

Thm 5.1: Each of the following is an abstract subrecursive reducibility:

a) the class of primitive recursive, doubly recursive,... operators (for $\mathscr{R}$)
b) the class of operators in $\mathscr{E}_n(\ )$ [6] for $n \geq 1$ (for $\mathscr{R}$)
c) the class of polynomial operators (for $\mathscr{R}$)
d) $\mathscr{T}$- time reducibility for time class $\mathscr{T}$(for $\mathscr{R}^*$) [10]
e) $\mathscr{S}$- space reducibility for space class $\mathscr{S}$(for $\mathscr{R}^*$) [10]
f) linear space, polynomial time reducibility
g) log-space reducibility [9]

Def 5.2: Let $f,g \in \mathscr{C}$. $f \leq_s g$ iff $f = Op_i[g]$ for some i.

Thm 5.2: $\leq_s$ is transitive and reflexive.

Proof: Let $f \leq_s g$ and $g \leq_s h$, say $f = Op_i[g]$ and $g = Op_j[h]$. Then $f = Op_i[\lambda y\ Op_j[h]y] = Op_{comp(i,j)}[h]$ by property (4). Hence $f \leq_s h$.

From APPLY[f] = f and APPLY[ ] = $Op_c[\ ]$ for some c we infer $f \leq_s f$.

Next we study the density properties of abstract reducibilities. Theorem 5.3 is the abstract analog to theorem 3.1.

Thm 5.3: Let g be such that $0 <_s g$. (0 denotes the constant function with value 0 everywhere). Then there is a h such that $0 <_s h <_s g$.

Proof: We have to construct h such that

(1) $h \leq_s g$

(2) $h \not\leq_s 0$, i.e. $h \neq Op_i[0]$ for all i

(3) $g \not\leq_s h$, i.e. $g \neq Op_i[h]$ for all i

h is constructed in stages as in the proof of theorem 3.1; for an informal discussion of the proof technique see theorem 3.1. In the proof of theorem 3.1 we used recursion; we justify this use now by an application of the recursion theorem.

We define a computable function e such that range $e \subseteq$ range s. Let i_0 be the fixed point of e; i.e.

$$\phi_{e(i_0)}[\ ] = \phi_{i_0}[\ ]$$

$h = \phi_{i_0}[g]$ is the desired function.

Let $\{x_i\}_{i=0}^{\infty}$ be an enumeration of the integers (domain of computation). We will first define a general recursive operator $\phi_{e_1(i)}[\ ]$; i is a free variable.

(A) $\phi_{e_1(i)}[g](x)$:

execute the following program for x steps
"compute $g(x_1), g(x_2), \ldots$ and build up a table G of the values"
execute the following program for x steps
"compute $\phi_i[g](x_1), \phi_i[g](x_2), \ldots$ and build up a table H of the values"
execute the following program for x steps
(using the tables G and H)
"initialize mode ← 2; j ← 1;
<u>while</u> $j < \infty$ <u>do</u>
<u>begin</u> k ← 1
    ¢ we check if $h \neq Op_j[0]$ ¢
    <u>while</u> $H(x_k) = Op_j[0](x_k)$
    <u>do</u> k ← k + 1;
    mode ← 3;
    k ← 1;
    ¢ we check if $g \neq Op_j[h]$ ¢
    <u>while</u> $G(x_k) = Op_j[h](x_k)$
    <u>do</u> k ← k + 1;
    mode ← 2
<u>end</u>"
the output of $\phi_{e_1(i)}[g](x)$ is the current value of mode

$\phi_{e_1(i)}[\ ]$ is a general recursive operator. By the simulation property (7) there is a non-decreasing unbounded (even range $B_{e_1(i)}[g] = \mathbb{N}$) function $B_{e_1(i)}[g]$ such that for all x

$$Op_{sim(e_1(i))}[g](x) = \phi_{e_1(i)}[g](B_{e_1(i)}[g](x))$$

We define now

(B): $\phi_{e(i)}[g](x) =$ <u>if</u> $Op_{sim(e_1(i))}[g](x) = 2$
<u>then</u> $g(x)$
<u>else</u> 0

$$= \underline{\text{if}}\ Op_{sim(e_1(i))}[g](x) = Op_{const(2)}[g](x)$$

$$\underline{\text{then}}\ Op_c[g](x)$$

$$\underline{\text{else}}\ Op_{const(0)}[g](x)$$

$$= Op_{cond(sim(e_1(i)),const(2),c,const(0))}[g](x)$$

$$= \phi_{s(cond(sim(e_1(i)),const(2),c,const(0)))}[g](x)$$

by properties (2), (3) and (5) of abstract reducibilities. Let i_0 be the fixed point of e, i.e.

$$\phi_{e(i_0)}[\ ] = \phi_{i_0}[\ ].$$

By definition of e there is a i_0' such that

$$Op_{i_0'}[\ ] = \phi_{e(i_0)}[\ ] = \phi_{i_0}[\ ].$$

Let $h = Op_{i_0'}[g]$. It remains to show that requirements (2) and (3) are satisfied. First note that range $B_{e_1(i_0)}[g] = N$. Hence larger and larger tables are generated in part (A) of the algorithm. Therefore it is sufficient to show that j becomes arbitrarily large. Assume otherwise; say j reaches a stable value j_0. Then mode also reaches a stable value. If this stable value is equal to 2 then we define $h = g$ almost everywhere; also $h = Op_{j_0}[0]$. By property (6a) of abstract reducibilities therefore $g \leq_s 0$, which contradicts the hypothesis of the theorem. If the stable value of mode is equal to 3 then we define $h = 0$ almost everywhere; also $g = Op_{j_0}[h]$. By property (6b) of abstract reducibilities therefore $g \leq_s 0$, which contradicts the hypothesis of the theorem. g.e.d.

For the remainder of the section we also need property (8).

(8) there are constants encode, decode 1 and decode 2 such that

$$Op_{decode1}[\lambda y\ Op_{encode}[f,g](y)](x) = f(x)$$

$$Op_{decode2}[\lambda y\ Op_{encode}[f,g](y)](x) = g(x)$$

Proofs for the following theorems are given in the full paper.

Thm 5.4: $\leq_s$ is an upper semi-lattice.

Thm 5.5: Let f,g be computable functions such that $f <_s g$. Then there is a computable function h such that $f <_s h <_s g$.

Thm 5.6: Let f,g be computable functions such that $f <_s h <_s g$. Every countable partial ordering can be embedded into the set $\{h;\ f <_s h <_s g\}$.

Thm 5.7: There are computable functions f,g such that $0 <_s f$, $0 <_s g$ and $h <_s f$ and $h \leq_s g$ implies $h \leq_s 0$.

Acknowledgement:

I wish to thank Professor R.L. Constable for his valuable discussions with me on the topics of this paper.

References:

[1] Axt, P., "On a Subrecursive Hierarchy and Primitive Recursive Degrees", TAMS, 92(1959), 85-105.

[2] Cobham, A., "The Instrinsic Computational Difficulty of Functions", Logic, Methodology and Philosophie of Science, Proc. 1964 Conference, North Holland, Amsterdam, 1965.

[3] Constable, R.L., "Type Two Computational Complexity", 5th ACM Symposium on Theory of Computing, 1973.

[4] Constable, R.L. and A.B. Borodin, "Subrecursive Programming Languages Part I: Efficiency and Program Structure", JACM, 19,3, July 1972.

[5] Cook, S.A., "The Complexity of Theorem Proving Procedures", 3rd ACM Symposium on Theory of Computing, 1971.

[6] Grzegorczyk, A., "Some Classes of Recursive Functions", Rozprawy Matematyczne, Warsaw 1953.

[7] Hartmanis, J. and J. Hopcroft, "An Overview of the Theory of Computational Complexity", JACM, 18,3, July 1971.

[8] Hopcroft, J. and J. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, 1969.

[9] Jones, N.D., "Recursibility among Combinatorial Problems in log space", Princeton Conference, 1973.

[10] Ladner, R.E., "Subrecursive Reducibilities", TR 73-03-13, Computer Science Group, University of Washington.

[11] Lynch, N., "Relativization of the Theory of Computational Complexity", Project MAC, TR-99.

[12] Machtey, M., "Augmented Loop Languages and Classes of Computable Functions", JCSS, 6,6, Dec. 1972.

[13] Machtey, M., "On a Notion of Helping", 14th SWAT Symposium, 1973.

[14] Machtey, M., "Honesty Techniques and Polynomial Degrees", Project MAC Conference on Concrete Complexity, 1973.

[15] Mehlhorn, K., "The 'Almost All' Theory of Subrecursive Degrees is Decidable", TR 73-170, Computer Science Department, Cornell University.

[16] Meyer, A.R. and D.M. Ritchie, "A Classification of the Recursive Functions", Zeitschr. fur math. Logik, Vol. 18, 1972.

[17] Mostowski, "Ueber gewisse universelle Relationen, Ann. Soc. Polon. Math., 17 (1938).

[18] Ritchie, R.W., "Classes of Predictably Computable Functions", Pacific Journal of Math., 1965.

[19] Rogers, Jr., H., Theory of Recursive Functions and Effective Computability, New York, 1967.

[20] Strong, H.R., "Algebraically Generalized Recursive Function Theory", IBM J. Res. Develop..

[21] Weihrauch, K., Doctoral Dissertation, Bonn.