1978

Applications of Classical Recursion Theory to Computer Science

Carl H. Smith

Report Number:
79-313

Applications of Classical Recursion Theory to Computer Science

Carl H. Smith

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
CSD Technical Report 313

ABSTRACT

Three applications of intensional recursion theory to computer science are presented. Proof techniques are used, in some cases, as a criterion of applicability. The first example distinguishes various forms of recursion as programming tools. A highly recursive proof of the famous speed up theorem from complexity theory is presented. The final application concerns the synthesis of programs given examples of their intended behavior.

Supported by NSF grant MCS 7903912. This report is based on a lecture delivered at Logic Colloquium '79, Leeds England, and will appear as part of the proceedings of that conference.

Applications of Classical Recursion Theory to Computer Science

Carl H. Smith

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

1. Introduction

In introducing his definitive work on recursion theory, Rogers [1967, pp. 10] remarks:

> "... our emphasis will be extensional , in that we shall be more concerned with objects named (functions) than with objects serving as names (algorithms)."

Computer scientists are interested in algorithms; their existence, expression, relative efficiency, comprehensibility, accuracy, and structure. Many computer scientists are more interested in algorithms themselves, rather than what is computed by them. In the sequel will emphasize and exploit the intensional aspects of recursion theory. Our intention is to substantiate the claim that the formalisms and techniques of recursive function theory can be applied to obtain results of interest in computer science. In what follows we will present results which yield insights into the nature of recursive programming techniques, complexity of programs, and the inductive inference of programs given examples of their intended input-output behavior. The results presented below are from the field of intensional

recursion theory in that they make assertions concerning algorithms and their proofs use the recursion theoretic techniques of diagonalization and recursion. Furthermore, in many instances, these techniques are applied intensionally in that they are used to specify an algorithm which manipulates other algorithms syntactically without necessarily any knowledge of the function specified by the manipulated algorithms. We include proofs only to illustrate the intensionality of their techniques.

Next we introduce the basic concepts and notation of recursive function theory from the perspective of a computer scientist. We start with the selection of a programming language with which to express algorithms. Hence, program is synonymous with algorithm. The only constraints we will place on our selection are that the language chosen be "powerful" enough to express any algorithm and that its syntax be unambiguous so that we may (algorithmically) determine of any string of symbols whether or not that string constitutes a well formed program. Any programming language satisfying the above constraints will suffice. Examples of such include PASCAL, ALGOL, FORTRAN and the language of Turing Machines. For convenience we will use the language of the UNIX operating system [Ritchie and Thompson 1978] as our programming language. Suppose that a lexicographically ordered list of strings of symbols of the programming language has been given. Let pr_0 denote the first syntactically well formed program in the list, and pr_1, the second, and pr_2 the third,... . Every partial recursive function is computed by at least one of the programs pr_i. ϕ_{pr_i} will denote the function of one integral argument

computed by program pr_i. In the sequel we will freely exploit the correspondence between the natural numbers, N, and our list of syntactically well formed programs by abbreviating "pr_i" as "i". The list ϕ_0, ϕ_1, ϕ_2, ... is called a programming system [Machtey and Young 1978]. Integers serve not only as names for functions but also as the domain of the data on which the functions operate. The basic entity of manipulation in the UNIX operating system is a file. Files may contain either programs or data. Hence, files in UNIX are analogous to integers in a programming system. We will occasionally exploit an implicit one-to-one correspondence between natural numbers and file names when discussing UNIX as a programming system in the technical sense. For example, when considering the application of some ϕ_i to more than one argument we will implicitly use the operation of file concatenation to combine the argument list into a single file. The file system of UNIX is derived from that of an older system, CTSS [Corbato et. al. 1965] see [Wilkes 1972]. We will have occasion to exploit features unique to the language of UNIX in the discussions below.

The programming system developed in the previous paragraph was generated algorithmically. Explicit in the above is an algorithm which given any natural number i, returns a complete description of the i^{th} well formed program. Such an algorithm is easily transformed into one which, on inputs i and x, generates program i and then simulates program i's behavior on input x, i.e. computes $\phi_i(x)$. Explicit in the language of UNIX is a shell command which takes two files as input, interpreted as a program

file and a data file, and simulates the program on the given data [Bourne 1978]. Hence, an enumeration or universal machine theorem holds in the programming system $\langle\phi_i\rangle$. More formally, there is a program u such that for any i and x, $\phi_u(i,x)=\phi_i(x)$. Note that program u interprets its datum i as a program.

The ability to effectively compose programs is another interesting and natural property of the programming system $\langle\phi_i\rangle$. Clearly there is an algorithm which given inputs i and j outputs a program, called c(i,j), which on any input x, just simulates program j on input x, and then simulates program i on the result, i.e. computes $\phi_i \circ \phi_j$. In the programming language used as an example here one, rather elegantly, pipes the output of program j into the input of program i. Storing the commands to pipe the output of j into i constitutes the creation of the program referred to above as c(i,j). A program to compute the function c merely takes its inputs i and j and creates a file containing "j pipe i". Formally, there is a function c such that for all i, j and x, $\phi_{c(i,j)}(x)=\phi_i(\phi_j(x))$. Note that any program which computes the function c above, interprets its two inputs as programs and then manipulates those programs as data to produce a third program.

Since the programming system $\langle\phi_i\rangle$ satisfies the enumeration and composition theorems it is <u>acceptable</u> [Machtey and Young 1978]. Furthermore, each acceptable programming system satisfies the s-m-n theorem [Hamlet 1974, 8.3]. Hence, the programming system $\langle\phi_i\rangle$ is, more traditionally, an indexing, a Godel number-

ing, or an acceptable numbering of all and only the partial recursive functions [Rogers 1958].

The programming system $\langle \phi_i \rangle$ was developed in a very natural manner. As previously noted, any such natural programming system has certain properties. In fact, even the most haphazardly designed computing systems actually in use today share these properties with the most elegant of the existing and imagined computing systems. Two features inherent in any acceptable programming system are the capability to interpret a datum as a program and the capability to manipulate a program as data. The ambiguity of the representation of program and data is a fundamental feature of nearly all modern day computing systems and also is the essence of the stored program concept [Burks, Goldstine and von Neumann 1963]. The ability to manipulate a program as data and to interpret a datum as a program in our programming system is not merely a ramification of associating programs with natural numbers. Programs are associated one-to-one with natural numbers in Friedberg's [1958] programming system which satisfies neither the s-m-n nor the composition theorems. In the next section we continue the discussion of programming techniques which are applicable in any acceptable programming system.

2. Recursion

Recursive programming techniques have become popular. Wirth [1976] notes that "Recursion ... is an important and powerful concept in programming." The word which titles this section has been used to name several theorems that hold in any acceptable

programming system and which embody and generalize the above mentioned programming technique. Rogers [1967, pp. viii] maintains that these theorems constitute "a fundamental tool in the theory." In this section we discuss the following two such theorems and their relationships to each other and to acceptable programming systems.

Kleene Recursion Theorem. [1938] For any program i there exists a program e, which can be found effectively from i, such that for any x, $\phi_e(x)=\phi_i(e,x)$.

Fixed Point Theorem. [Rogers 1967, 11.2] For any recursive function f there exists a program e, which can be found effectively from a program for f, such that $\phi_e=\phi_{f(e)}$.

Both of the above recursion theorems hold in any acceptable programming system. The intuitive content of these theorems is that we may write programs using a copy of the completed program as an implicit parameter in any effective calculation we choose. In practice, recursive programs in procedural programming languages almost always invoke themselves with arguments smaller than those of the original call. By the recursion theorems we may, in principle, write recursive programs which invoke themselves on arguments larger than those of the original call; which invoke effective distortions of themselves on a variety of arguments; which measure their own complexity; and which perform a myriad of transformations on their own description. In practice, the use of recursion, as in the recursion theorems, amounts to

deciding on the name of the file which will contain the code for the recursive program to be written. Then, when writing the program, the chosen file name may be used in conjunction with shell and edit commands to implement any use of recursion alluded to above.

The two recursion theorems stated above are equivalent in any acceptable programming system. However, the proof of the fixed point form of the recursion theorem requires the invocation of the universal function. Kleene's proof uses only the composition and s^1_1 functions: let v be a program to compute $\lambda xy[\phi_i(s^1_1(x,x),y)]$, then the desired e is given by $s^1_1(v,v)$. In the sequel we state and interpret several results with extensional content and intensional proofs which serve to further distinguish the two stated recursion theorems.

A programming system satisfies the padding theorem if given any program it is possible to effectively enumerate infinitely many distinct programs, each of which computes the same function as the given program. In our programming system the padding theorem follows from the composition theorem: pad any program by composing it with a program for the identity function. Riccardi [1980] has shown that any programming system which satisfies the enumeration, padding, and Kleene recursion theorems is acceptable. His proof exploits the one-to-oneness of the algorithm which produces the appropriate self referential program. A programming system which satisfies the enumeration, padding, the fixed point theorems but which is not acceptable is constructed

in Machtey, Winklmann and Young [1978]. The fixed point function exhibited in their proof is also one-to-one. Royer (private communication) has constructed an unacceptable programming system which satisfies the enumeration, padding, and non one-to-one Kleene recursion theorems. Hence, the Kleene recursion theorem , with an enumeration theorem, is distinctly more potent than the fixed point theorem in the absence of effective composition, i.e. without the capability to manipulate programs as data.

The fixed point recursion is more powerful than the Kleene recursion in the presence of a composition theorem and the absence of an enumeration theorem. Any programming system satisfying the composition theorem also satisfies the Kleene recursion theorem. The following theorem, obtained in collaboration with Machtey (private communication), is a sharpening, in the effectiveness of the exhibited composition function, of a result of Riccardi [1980] which generalizes, from sub recursive to acceptable programming systems, a result noticed by Alton [1976] and Kozen [1978].

Theorem. There is a programming system satisfying the effective composition theorem but not the effective fixed point theorem. Furthermore, a program for the composition function can be effectively located in the programming system.

Proof: Suppose p is a program in the acceptable programming system $\langle \phi_i \rangle$ for the function $\lambda x,y[4(x,y)+2]$. Let $\langle \psi_i \rangle$ be the r.e. sequence of functions given by:

i) $\pi_{4i} = \phi_i$;

ii) π_{4i+1} = Either $\lambda x[0]$, $\lambda x[1]$, or $\lambda x[2]$, whichever differs from both π_{4i} and π_{4i+2};

iii) $\pi_{4i+2} = \pi_j \circ \pi_k$, where i is the pairing of j and k;

iv) π_{4i+3} = Either $\lambda x[0]$, $\lambda x[1]$, or $\lambda x[2]$, whichever differs from both π_{4i+2} and π_{4i+4}.

By i) $\langle \pi_i \rangle$ is a programming system. Furthermore, π_{4p} is its composition function. The constant p can easily be found, by our judicious choice of initial programming system, from some simple pairing and coding programs. The function $\lambda x[x+1]$ has no fixed point (by ii) and iv)). Furthermore, using p, a few other simple programs, and an algorithm of [Machtey, Winklmann and Young 1978], it is also possible to effectively locate a program for s^1_1 in the programming system $\langle \pi_i \rangle$.

[]

We interpret the above theorem as indicating the strength of the fixed point recursion theorem over the Kleene recursion theorem in computational realms witnessing the ability to manipulate programs as data but not the ability to interpret data as a program.

We conclude this section with a brief discussion of multiple recursion theorems. Smullyan [1961] proved a double analogue of the fixed point theorem for r.e. relations. Riccardi [1980] proved that the double analogue of the Kleene recursion theorem for partial recursive functions with the enumeration theorem

implies the s-m-n theorem. Hence, acceptable programming systems can also be characterized as programming systems with enumeration and double Kleene recursion theorems. Case [1974] proved an infinitary analogue of the Kleene recursion theorem which we state below and use in subsequent sections.

Theorem [Case 1974]. Suppose Θ is an effective operator [Rogers 1967, 9.8]. Then one can effectively find a recursive one-to-one monotone increasing function p such that for all i and x, $\phi_{p(i)}(x)=\Theta(p)(i,x)$.

Intuitively, the operator recursion theorem allows one to construct a sequence of programs, p(0), p(1), ..., each of which can use its own index within p and descriptions of other programs in the range of p as implicit additional parameters.

3. Complexity

In this section we present two recursion theorem arguments establishing facts concerning the complexity of computations. Abstract complexity theory is a well studied area, for an introduction see Hartmanis and Hopcroft [1971], or Brainerd and Landweber [1974], or Machtey and Young [1978]. In the sequel we need only an acceptable programming system and the following definition due to Blum [1967].

A list of functions $\langle\Phi_i\rangle$ is a complexity measure for the acceptable programming system $\langle\phi_i\rangle$ iff

a) for any program i, domain (Φ_i) = domain (ϕ_i) and

b) the set $\{(i, x, y) \mid \Phi_i(x) = y\}$ is recursive.

Intuitively $\Phi_i(x)$ may be thought of as the running time (or the amount of any computational resource used in the execution) of program i on input x. The first result, which generalizes a result of [Blum 1967], asserts that there are arbitrarily expensive algorithms to compute any function.

Theorem [McCreight 1969]. For any program i and any recursive function h there is a program e which computes ϕ_i and for all x, $\Phi_e(x) > h(x)$.

Proof: By implicit use of the recursion theorem there exists (effectively in i and a program for h) a program e such that

$$\phi_e(x) = \begin{cases} \phi_i(x), & \text{if } \Phi_e(x) > h(x); \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

For any x, if $\Phi_e(x) \leq h(x)$, then $\phi_e(x)$ is undefined, a contradiction. Hence, $\phi_e = \phi_i$, and Φ_e bounds h.

[]

The use of recursion above is intensional. Program e uses self referencing only to compare its complexity with the given function h. Next we present a version of the speed-up theorem using an application of a double analogue of the operator recursion theorem. The proof we present was suggested by Case [19??] and bears a very strong resemblance to a proof of the same theorem due to Young [1973]. The use of recursion below is similarly intensional. The application of recursion is more

complicated since the self reference involves programs in an r.e. sequence referring to other programs in the same sequence. The proof below, as noted by Case, can be extended to proof the operator speed-up theorem of Meyer and Fischer [1972].

Theorem [Blum 1967]. For any recursive function h, which is monotone nondecreasing in the second argument, there exists a recursive function f such that for any program i, if $\phi_i = f$ then there exists (uniformly and effectively in i) a program j such that $\phi_j = f$ and $h(x,\Phi_j(x)) \leq \Phi_i(x)$, for almost all x.

Proof: Suppose that h is a recursive function of two arguments, monotone nondecreasing in the second one. By implicit use of a double analogue of the operator recursion theorem we construct below two r.e. sequences of programs $p_{(0,0)}$, $p_{(0,1)}$, $p_{(1,0)}$, ... and t(0), t(1), t(2), ... The master program, t(0), computes the function which we will later name f. For i>0, program t(i) computes a finite variant of $\phi_{t(0)}$. For each i and every y program p(i,y) computes a patched version of $\phi_{t(i)}$. Program t(i) is faster than program t(j), for j<i, because t(i) checks fewer programs for cancellation at each stage of the construction below. For a suitably chosen y, the patching program p(i,y) compensates for t(i)'s lack of consideration and computes $\phi_{t(0)}$. In the construction below, D_i denotes the i^{th} set of ordered pairs in some fixed but unspecified canonical listing of all such sets of ordered pairs. By convention, D_0 is the empty set. The max of the null set is, by convention, 0.

$$\phi_{p(i,y)}(x) = \begin{cases} D_y, & \text{if } x \text{ is in domain } D_y; \\ \phi_{t(i+1)}(x), & \text{otherwise.} \end{cases}$$

Stage x in the construction of $\phi_{t(n)}$.

If all previous stages converged set,

$$\phi_{t(n)}(x) = \begin{cases} 1+\max\{\phi_i(x) \mid n \leq i < x \text{ and } i \text{ was not cancelled at a previous stage and } \Phi_i(x) \leq \max\{h(x,\Phi_{p(i,y)}(x)) \mid y<x\}\}; \\ \text{divergent otherwise.} \end{cases}$$

Then, if the computation of $\phi_{t(n)}(x)$ converged, cancel all the previously uncancelled i for which $\phi_i(x)$ was included in the max above.

End Stage x.

We proceed to establish the totality of the functions computed by the programs in the range of t. Immediately from the above construction we have that $\phi_{t(n)}(x)=1$ whenever $n \geq x$. Note that for any i, $\phi_{p(i,0)}$ total implies $\phi_{p(i,y)}$ total for any y. Hence, for any $n<x$, in order to have $\phi_{t(n)}(x)$ defined it suffices to have $\Phi_{p(x-1,0)}(x)$, $\Phi_{p(x-2,0)}(x)$, ... , and $\Phi_{p(n,0)}(x)$ all defined and $\phi_{t(n)}(0)$, $\phi_{t(n)}(1)$, ... , and $\phi_{t(n)}(x-1)$ all defined. Hence, it suffices to have $\phi_{t(x)}(x)$, $\phi_{t(x-1)}(x)$, ... , $\phi_{t(n+1)}(x)$; $\phi_{t(n)}(0)$, $\phi_{t(n)}(1)$, ... , and $\phi_{t(n)}(x-1)$ all defined.

Proceeding inductively, assume that $\lambda n[\phi_{t(n)}(x')]$ is total for all $x'<x$. We need to show that $\phi_{t(0)}(x)$, ..., $\phi_{t(x-1)}(x)$ are all defined. By the argument of the previous paragraph, $\phi_{t(x)}(x)$ being defined is sufficient for $\phi_{t(x-1)}(x)$ to be defined. For $\phi_{t(x-2)}(x)$ to be defined it suffices to have both $\phi_{t(x)}$ and $\phi_{t(x-1)}$ defined on argument x. Hence, $\phi_{t(x-2)}(x)$ is defined. By similar arguments, $\phi_{t(x-3)}$, ..., $\phi_{t(0)}$ are all convergent on argument x. Hence, $\lambda[n[\phi_{t(n)}(x)]$ is total, completing the proof that all the programs in the range of t compute total functions.

Let $f=\phi_{t(0)}$. Suppose $\phi_i=f$. Observe that $\phi_{t(i+1)}(x)=t(x)$, for all x greater than or equal to the least s such that for all $j<i+1$, if j is ever cancelled by program t(0) then j is cancelled by t(0) before stage s. Choose y such that $D_y=\{(z,\phi_{t(0)}(z)) \mid z<s\}$. Then $\phi_{p(i,y)}=f$. If $\Phi_i(x)<h(x,\Phi_{p(i,y)}(x))$ for any $x>i$ then $\phi_{t(0)}$ and ϕ_i would differ.

[]

Note that recursion is used by programs t(i) in the above proof to interrogate the complexity of programs simulating the programs enumerated by t. The interrogation of complexity is an intensional use of recursion. Using recursion for the purposes of simulation is an extensional application. Despite the extensional techniques used in the above proof, the above result is unquestionably from the field of intensional recursion theory as it asserts that there are functions for which there is no best program.

4. Inductive Inference

In this section we survey some results concerning the inductive inference of programs given examples of their intended input-output behavior. The recursion theoretic approach to the problem of inductive inference, as formalized by L. and M. Blum [1975], constitutes a continuation of three distinct lines of research. Many of the fundamental definitions and concepts are taken from Gold's work [1967] in grammatical inference [Biermann and Feldman 1972]. Inductive inference can be viewed as a problem of synthesising Turing machines. Barzdin initiated an investigation of the synthesis of automata [Trakhtenbrot and Barzdin 1973]. Philosophers of science, most notably Carnap [Schilpp 1963], have investigated the process by which an empirical scientist examines some experimental data and conjectures an hypothesis intended to explain the data and to predict the outcome of future experiments. Philosophical implications of the recent recursion theoretic work on inductive inference, including a mechanistic repudiation of the principle expounded by Popper [1968] that every scientific explanation ought to be refutable, can be found in [Case and Smith 1978], the source of much of the material presented below.

An inductive inference machine (abbreviated: IIM) is an algorithmic device with no a priori bounds on how much time or memory resource it shall use, which takes as its input the graph of a function from N into N an ordered pair at a time (in any order), and which from time to time, as it's receiving its input,

outputs computer programs.

We will introduce several notions of what it means for an IIM to succeed at eventually finding an explanation for a function. The first is essentially from [Gold 1967], but see also [Blum and Blum 1975]. We say M EX identifies a recursive function f (written: f is in EX(M)) iff M, when fed the graph of f in any order, outputs over time but finitely many computer programs the last of which computes (or explains) f. No restriction is made that we should be able to algorithmically determine when (if ever) M on f has output its last computer program.

An IIM M is said to be order independent iff for any function f, the corresponding sequence of programs output by M, is independent of the order in which f is input. Clearly, any IIM M can be effectively transformed into an IIM M' which preprocesses any recursive function f and feeds it to M in the order (0,f(0)), (1,f(1)), (2,f(2)), ... An order independence result that covers the case of partial functions appears in [Blum and Blum 1975]. In what follows we shall suppose without loss of generality that all IIMs are order independent.

We define the class of sets of functions EX = { S |(there exists an IIM M)[S is included in EX(M)]}. EX is the collection of all sets S of recursive functions such that some IIM EX identifies every function in S. For example, Gold [1967] showed that {f|f is primitive recursive} is in EX. As noted in [Blum and Blum 1975], Gold's proof can be easily modified to show that any recursively enumerable class [Rogers 1967] of recursive functions

is EX Identifiable.

We now motivate our first generalization of Gold's result. Newton [1792] in introducing his own work said, "...I now demonstrate the frame of the System of the World." He apparently believed that he had converged on (unique?) explanations for a small finite class of phenomena of such general and astronomical scope that explanations for all other phenomena that actually occur in the real world could be obtained (perhaps with great difficulty) by a suitable fleshing out of these framework explanations. Newton himself was aware of a difficulty with the explanation of light; namely, that light was required to have an ostensibly contradictory dual, wave and particle nature. We know from hindsight that Newtonian physics and its classical extensions have some serious flaws or anomalies. Here is an example from optics. Classically the molecules of a medium through which light passes can be modeled as simple harmonic oscillators which then delay re-emitting light of some frequencies more than others. This provides a pretty good quantitative model of dispersion, the breaking up into colors of white light sent through a prism. Unfortunately, the classical model of dispersion does not correctly predict the outcome of dispersion experiments involving x-rays. X-ray dispersion is, therefore, referred to as anomalous dispersion. The quantum mechanical explanation of dispersion also covers the x-ray case, but the point is that physicists will sometimes actually employ an explanation which has an anomaly in it, an explanation which fails to correctly predict the outcome of one experiment but which is correction all other experiments.

We say M EX^1 identifies a recursive function f (written: f is in $EX^1(M)$) iff M, when fed the graph of f in any order, outputs a last computer program which computes f except perhaps at one anomalous input. For recursive functions f and g, "f is a singleton variant of g" is written $f=^1g$. We analogously define the class EX^1 = { $\underline{S}$ |(there exists an IIM M)[$\underline{S}$ is included in $EX^1(M)$}.

Putnam [1963] showed that there is no general purpose robot scientist in the sense that a naturally restricted subclass of EX does not contain all the recursive functions. Gold [1967] showed that no single inductive inference machine can EX identify every recursive function. $\{f|\phi_{f(0)}=^1f\}$ is in the class EX^1 but not the class EX indicating that if the goal set of mechanized scientists is relaxed to allow a possible single anomaly in explanations, then, in general, they can identify strictly larger classes of recursive functions than those that are error intolerant.

There are two possible kinds of single anomalies in an explanatory program. The first kind occurs when the program on some one input actually gives an output which is incorrect. This kind of single anomaly eventually can be found out, refuted, and patched. The second kind occurs when the program on some one input fails to give any output at all; the explanation is incomplete. This latter kind of anomaly, in general, cannot be algorithmically found out [Rogers 1967]; the explanation is not, in general, (algorithmically) refutable. If we define $EX^{=1}$ identification just as we defined EX^1 identification but we replace "ex-

cept perhaps at one anomalous input" by "except at exactly one anomalous input" , we have that $EX^{=1} = EX$. This is because exactly one anomaly (of either kind) can be patched in the limit: Patch in the correct output for input 0 until (if ever it is discovered that the output was already correct on input 0, then patch in the correct output for input 1 until ... . Eventually the patch will come to rest on the single anomaly which needed patching. It follows that the strength of EX^1 identification must come from two sources: Possibly incomplete explanations and our inability to test algorithmically for incompleteness. The proof that $EX=EX^1$ reflects this last observation. L. and M. Blum [1975] proved that the union of two EX identifiable sets of functions is not necessarily EX identifiable. Their result can be obtained as a simple corollary of the EX and EX^1 separation result.

The remainder of this section will cover several more general separation results. For any natural number n, define EX^n identification and the class EX^n analogously with EX^1 identification and the class EX^1. The notation $f=^n g$ means that the recursive function f is an n-variant recursive f the function g, e.g. $\{x|f(x)\neq g(x)\}$ has at most n members. Then, for any natural number n $\{f|\phi_{f(0)}=^{n+1}f\}$ is a member of EX^{n+1} but not EX^n. Hence, the more tolerant a learning procedure is of errors (anomalies) in it's output the better the chances, in general, of success.

Allowing a finite but unbounded number of anomalies in a final explanation constitutes an inference criterion more general

than any discussed above. We say M EX^* identifies a recursive function f (written: f is in $EX^*(M)$) iff M, when fed any enumeration of the graph of f, outputs but finitely many programs, the last of which computes f except perhaps on finitely many anomalous inputs. The class EX^* is defined in the usual way. For functions f and g, "f is a finite variant of g" will be written $f=^*g$. EX^* identification coincides with almost everywhere identification introduced in [Blum and Blum 1975] and subidentification in [Minicozzi 1976]. A sharpening of a result mentioned in [Blum and Blum 1975] is that $\{f \mid \phi_{f(0)} =^* f\}$ is a member of the class EX^* but is not a member of the union over all natural numbers n of the classes EX^n.

Hence, the EX classes form a hierarchy of more and more general inference criteria. Notice that for any n the set $\{f \mid \phi_{f(0)} =^n f\}$ can be EX^n identified by an IIM which, when fed the graph of f, outputs f(0) as its only conjecture, i.e. can be EX^n identified by an IIM which outputs a single conjecture and never later changes its mind. This last observation leeds to the following definitions. a, b, c and d will denote members of {natural numbers} $\cup$ {*}. For any a and b we say M EX^a_b identifies f (written: f is in $EX^a_b(M)$) iff M EX^a identifies f after no more than b mind changes (no restriction when b=*). The class $EX^a_b = \{ S \mid (\text{there exists an } M)[S \text{ is included in } EX^a_b(M)]\}$. Observe that for any a EX^a_* identification coincides with EX^a identification above. By convention, any natural number is <*. Then $EX^a_b \subseteq EX^c_d$ iff $a \leq c$ and $b \leq d$. Hence, learning procedures can not, in general, infer more accurate solutions by making more attempts.

Next we introduce a notion of inference without convergence to a fixed explanation. We say an IIM M BC identifies a recursive function f (written: f is in BC(M)) iff M, when fed the graph of f (in any order) outputs over time an infinite sequence of computer programs all but finitely many of which compute f. The class $BC=\{ \underline{S} \mid (\text{there is an IIM M})[\underline{S} \text{ is included in } BC(M)]$. The class BC is defined in the usual manner. Barzdin [1974] acting on an observation of Feldman [1972] independently defined a notion, referred to as GN^{OO} in the Russian literature, which coincides with our BC. John Steel (private communication) first observed that EX^* is included in BC. That the inclusion is proper is a result from [Case and Smith 1978] obtained in collaboration with Leo Harrington (private communication). Barzdin [1974] proved that EX is properly contained in BC. His proof actually shows that EX^* is properly contained in BC. However, Barzdin's proof makes no use of recursion theorems.

Theorem Let $\underline{S}$ be $\{f \mid$ for all but finitely many x, $\phi_{f(x)}=f\}$. Then $\underline{S}$ is an element of BC but not EX^*.

Hence, if we allow our learning procedures to accumulate larger and larger size explanations, then they can, in general, infer larger classes of phenomenon.

Proof: Clearly $\underline{S}$ is a member of BC. Let M be given. We suppose without loss of generality that for all σ, $M(\sigma)$ is defined. It remains to exhibit a function f which is a member of $(\underline{S} - EX^*(M))$. By implicit use of the operator recursion theorem [Case 1974], we obtain a repetition free r.e. sequence of pro-

grams p(0), p(1), p(2), ... such that one of these programs computes such an f. We proceed to give an informal effective construction of the $\phi_{p(i)}$'s in successive stages $s \geq 0$. p(1) is just a program for $\phi_{p(0)}$ which differs from p(0). $\phi^s_{p(i)}$ denotes the finite initial segment of $\phi_{p(i)}$ defined before stage s. For each i, $\phi^0_{p(i)}$ is empty. $q^s = M(\phi^s_{p(0)})$.

Begin stage s. Simultaneously execute the following three substages until (if ever) either suitable x and σ are found in substage (i) or a mind change is found in substage (ii).

(i) Dovetail a search for x and σ such that $\phi^s_{p(0)} \subset \sigma$, range $(\sigma - \phi^s_{p(0)}) = \{p(0), p(1)\}$, x is a member of domain $(\sigma - \phi^s_{p(0)})$, and program q^s converges on x to a value $\neq \sigma(x)$.

(ii) See if there is a τ such that $\phi^s_{p(0)} \subset \tau \subseteq$ what has been put into $\phi_{p(s+2)}$ so far and $M(\tau) \neq q^s$. (Before stage s, $\phi^s_{p(s+2)}$ is made $= \phi^s_{p(0)}$.)

(iii) Make $\phi_{p(s+2)}$ have value p(s+2) at more and more successive arguments not yet in its domain.

Condition(1). x and σ are found in substage (i) before a mind change is found in substage (ii). Set $\phi^{s+1}_{p(0)} = \sigma$, do not extend $\phi_{p(s+2)}$ any further, and set $\phi^{s+1}_{p(s+3)} = \phi^{s+1}_{p(0)}$.

Condition(2). A mind change is found in substage (ii) before or at the same time as suitable x and σ are found in substage (i). Set $\phi^{s+1}_{p(0)}$ = what has been put into $\phi_{p(s+2)}$ so far, make program p(s+2) from this point on simulate program p(0) on

all inputs not yet in its domain so that $\phi_{p(s+2)}$ will = $\phi_{p(0)}$, and set $\phi^{s+1}_{p(s+3)} = \phi^{s+1}_{p(0)}$.

End stage s.

Case (1). Some stage s never terminates. Then by substage (iii), $\phi_{p(s+2)}$ is a (total) recursive function and for all but finitely many x, $\phi_{p(s+2)}(x) = p(s+2)$. Set $f = \phi_{p(s+2)}$. Clearly f is a member of $\underline{S}$. Program q^s is M's last output on input f; furthermore, q^s never converges on any x not in domain $(\phi_{p(0)}{}^s)$ since, if it did, it could not converge to both p(0) and p(1) and so substage (i) would find suitable x and σ. It follows that ϕ_{q^s} is a finite function and hence not a finite variant of f. Therefore, f is a member of $(\underline{S} - EX^*(M))$.

Case (2). Not Case (1). Then $\phi_{p(0)}$ is a (total) recursive function and everything in its range is a program for $\phi_{p(0)}$: By Condition (1) p(0)'s and p(1)'s are introduced into its range and these compute $\phi_{p(0)}$; by Condition (2) p(s+2)'s are introduced into its range, but then p(s+2) also computes $\phi_{p(0)}$. Set $f = \phi_{p(0)}$. Clearly, then, f is a member of $\underline{S}$. Suppose M on f outputs a last program q. Then Condition (1) holds at all but finitely many stages s. Hence, infinitely often, f is defined to differ from ϕ_q. Therefore, f is a member of $(\underline{S} - EX^*(M))$.

[]

Of particular interest in the above theorem is the highly intensional use of recursion in the construction of the $\phi_{p(i)}$'s.

The range of p was used as the range of the $\phi_{p(i)}$'s. Previously determined initial segments of $\phi_{p(0)}$ are fed into an IIM and the resulting program is simulated. None of the programs in the range of p is simulated directly. In the proof of the speed up theorem in section 3, the calculation of $\phi_{t(i)}(x)$ involved the simulation of $\phi_{t(i+1)}$, ..., $\phi_{t(x)}$, all on argument x.

5. Summary

We presented three example applications of recursion theory to computer science. The first application dealt with the role of recursion in the infrastructure of programming techniques. Briefly mentioned was the use of recursion in abstract complexity theory. The last application was from the relatively new area of inductive inference.

Acknowledgements

We gratefully acknowledge many conversations with J. Case, M. Machtey, and G. Riccardi. Computer time was provided by the Department of Computer Sciences, Purdue University.

References

Alton, D., "Natural complexity measures, subrecursive languages, and speed-ups," Computer Science Department Technical Report 76-05, University of Iowa, 1976.

Barzdin, J., "Two theorems on the limiting synthesis of functions," in Barzdin (ed.) Theory of Algorithms and Programs, 1 ,Latvian State University, Riga, U.S.S.R, 1974, pp 82-84.

Biermann, A. and Feldman, J., "A survey of results in grammatical inference," in Watanabe (ed.) Frontiers of Pattern Recognition , Academic Press, 1972.

Blum, L. and Blum, M., "Toward a mathematical theory of inductive inference," Information and Control, 28 , 1975, pp 125-155.

Blum, M., "A machine-independent theory of the complexity of recursive functions," Journal of the ACM, 14 , 1967, pp 322-336.

Brainerd, W. and Landweber, L., Theory of Computation , John Wiley & Sons, New York, 1974.

Bourne, S., "The UNIX shell," The Bell System Technical Journal, 57 , 1978, pp 1971-1190.

Burks, A., Goldstine, H., and von Neumann, J., "Preliminary discussion of the logical design of an electronic computing instrument," in Taub (ed.) Collected Works of John von Neumann, 5 , Macmillan Company, New York, 1963, pp 34-79.

Case, J., "Periodicity in generations of automata," Mathematical Systems Theory, 8 , 1974, pp 15-32.

Case, J., "Operator speed-up for universal machines," to appear in IEEE Transactions on Computers.

Case, J. and Smith, C., "Anomaly hierarchies of mechanized inductive inference," Proceedings of the 10th ACM Symposium on the Theory of Computing , San Diego, California, 1978, pp 314-319.

Corbato, F., et.al., "The compatible time-sharing: a programmer's guide, second edition, M.I.T. Press, Cambridge, Mass., 1965.

Feldman, J., "Some decidability results on grammatical inference and complexity," Information and Control, 20 , 1972, pp 244-262.

Friedberg, R., "Three theorems on enumeration," Journal of Symbolic Logic 23 , 1953, pp 309-316.

Gold, E. M., "Language identification in the limit," Information and Control, 10 , 1967, pp 447-474.

Hamlet, R., Introduction to Computation Theory , Intext Educational Publishers, New York, 1974.

Hartmanis, J. and Hopcroft, J., "An overview of the theory of computational complexity," Journal of the ACM, 18 , 1971, pp 444-475.

Kleene, S., "On notation for ordinal numbers," Journal of Symbolic Logic, 3 , 1938, pp 150-155.

Kozen, D., "Indexing of subrecursive classes," to appear in Theoretical Computer Science.

Machtey, M., Winklmann, K. and Young, P., "Simple Godel numberings, isomorphisms, and programming properties," SIAM Journal of Computing, 7 , 1978, pp 39-59.

Machtey, M. and Young, P., An Introduction to the General Theory of Algorithms , Elsevier North-Holland, Inc., New York, 1978.

McCreight, E., "Classes of computable functions defined by bounds on computation", PhD dissertation, Carnegie-Mellon University, Pittsburgh, Penn., July 1969.

Meyer, A. and Fischer, P., "Computational speed-up by effective operators," Journal of Symbolic Logic, 37 , 1972, pp 48-68.

Minicozzi, E., "Some natural properties of strong-identification in inductive inference," Theoretical Computer Science, 2 , 1976, pp 345-360.

Popper, K., The Logic of Scientific Discovery , (second edition), Harper Torch Books, New York, 1968.

Putnam, H., "Probability and conformation," The Voice of America, Forum on the Philosophy of Science, 10 , U.S. Information Agency 1963; also appears in Putnam, H., Mathematics, Matter and Method , Vol. 1, Cambridge University Press, 1975.

Riccardi, G., The Independence of Control Structures in Abstract Programming Systems, Ph.D. Dissertation, SUNY at Buffalo, Buffalo New York, 1980.

Ritchie, D. and Thompson, K., "The UNIX time-sharing system," The Bell System Technical Journal, 57 , 1978, pp 1905-1930.

Rogers, H., "Godel numbers of partial recursive functions," Journal of Symbolic Logic, 23 , 1958, pp 331-341.

Rogers, H., Theory of Recursive Functions and Effective Computability , McGraw Hill, New York, 1967.

Schilpp, P., Library of Living Philosophers, 11 , "The philosophy of Rudolf Carnap," Open Court Publishing Co., LaSalle, Illinois, 1963.

Smullyan, R., Theory of formal systems, Annals of Mathematical Studies , no. 47, Princeton, New Jersey, 1961.

Trakhtenbrot, B. and Barzdin, J., Finite Automata: Behavior and Synthesis, North Holland/American Elsevier, New York, 1973.
Wilkes, M., Time-Sharing Computer Systems , second edition, American Elsevier, New York, 1972.

Wirth, N., Algorithms + Data Structures = Programs , Prentice Hall, Englewood Cliffs, New Jersey, 1976.

Young, P., "Easy constructions in complexity theory," Proceedings of the AMS, 37 , 1973, pp 555-563.