Efficient Reducibility Between Programming Systems: Preliminary Report*

Nancy A. Lynch
Florida International University
Miami, Florida 33199

Edward K. Blum
University of Southern California
Los Angeles, California 90007

I. Introduction

Much of the research on semantic theories has concentrated on qualitative properties such as definability (of such programming concepts as recursive procedures), equivalence (of different language constructs), and verifiability (of the correctness, or consistency, of one expression relative to another). Current qualitative theories are in a tentative state and much remains to be done. However, there is also a quantitative side to semantics. Indeed, many of the questions which any semantic theory must answer are at once qualitative and quantitative. We would like to draw upon complexity-theoretic techniques to answer such questions. However, first it is necessary to establish a mathematical framework within which the analysis of semantic complexity can be carried out. This framework should accommodate the software concepts which underlie existing sophisticated languages like ALGOL 68 and simpler languages like BASIC. Recent research [1-9] suggests a primarily algebraic framework. Algebra lends itself to a precise formulation of certain "static"(or non-effective) programming concepts such as "values," "data structures" in an extensible language [2,9], and"operators," whereas "dynamic" (or effective) concepts such as "control" must be based on the theory of computation, although even here algebraic notions may prove useful [1]. However, it appears that classical abstract algebraic notions such as the homomorphism concept, arising out of studies of the properties of similar algebraic structures, are inadequate to the task of computer science.

We are currently working on the development of new algebraic constructs to provide a mathematical framework for both qualitative and quantitative analysis of semantic problems. Some preliminary ideas in this direction appear in [15] and in Appendix A. The emphasis in the present paper is on obtaining some insight into the kinds of results that can be obtained. Specifically, we restrict attention to questions involving relative complexity of flowchart programming systems. We expect that isolation of the relevant algebraic properties sufficient to imply our results and their natural generalizations will be possible. We also hope that the present results have intrinsic interest in suggesting a more "relative" or "modular" approach to complexity analysis:

*This work was supported by the National Science Foundation through grant DCR75-02373.

Analysis of complexity of algorithms has generally been done in an "absolute" way, by counting the total "time" required by the algorithm when performed on a RAM with a specified operation set, or perhaps on a Turing machine. One difficulty with this approach is that it tends to de-emphasize similarities between computational problems. It has been noticed [10] that the underlying algebra is not really absolute; for different problems or at different times for the same problem, we might wish to measure complexity of an arithmetic function in terms of basic arithmetic operations on N, in terms of bit vector operations, or in terms of basic bit operations. Thus, in a sense, time complexity is most naturally thought of as a relative concept rather than as an absolute one.

Relative complexity is, of course, not a new idea; one form in which it has been studied is represented by [11,12,13], for example. This work uses Turing machines with oracles as a model for computation. But for very low-level complexity theory, the peculiarities of Turing machines sometimes becomes intermingled with the properties of the oracle set in determining relative complexity. We take the viewpoint that both are important; the basic operations of the Turing machine itself are considered to be no different from oracle functions and predicates, and both are here thought of as primitives of an algebra.

There are really two kinds of modularity to be treated. The first is the definition of a new operation from previously defined operations on a previously defined data type. (This is a very general description of the subject matter of "algebraic complexity" [14].) The second is the "implementation" of an entirely new data type, together with some new operations, relative to a previously defined data type. (For example, given bit vectors and some standard set of operations, how should we "implement" the rational numbers with an appropriate set of operations?) We consider both cases.

One motivation for considering implementation of an entire algebra rather than of one function at a time arises from the previously mentioned work on data structures. A second and very important motivation arises from coding considerations. Consider the situation in which we have a programming system based on bit vectors, with some natural set of operations, and wish to determine the "complexity of primeness for members of N." There is no a priori reason we could not assume a coding of N into bit vectors which includes

primeness in a quickly accessible way. Difficulties of this kind are generally resolved by specifying a particular coding. But meaningful results should be obtainable without resorting to such a drastic, specific solution. We regard "primeness" as existing not in a vacuum but along with other operations we want to perform on N (such as + or $\leq$). Restrictions on the complexity upper bound for the other operations on N serve to restrict the coding in such a way that coding-independent results can be obtained for the new operation.

In the remaining sections, we give our definitions and technical results. Results deal with relative complexity of certain basic operations on particular algebras, with the comparative expressive power of different program structures, and with the comparative efficiency of different relative codings of the same pair of algebras. We also try to capture what comprises an adequate (i.e. able to compute as efficiently as possible) set of basic operations over different algebras, and prove adequacy and inadequacy of some particular sets of operations. The main technical results of interest are those in Section V, Theorem 3, and possibly Theorem 1.

There are many open questions remaining to be considered; also, extensions of our definitions will be needed to express more complicated data structure and algorithm implementations. We have chosen here to examine the simplest possible definitions and explore their power.

II. Notation and Definitions

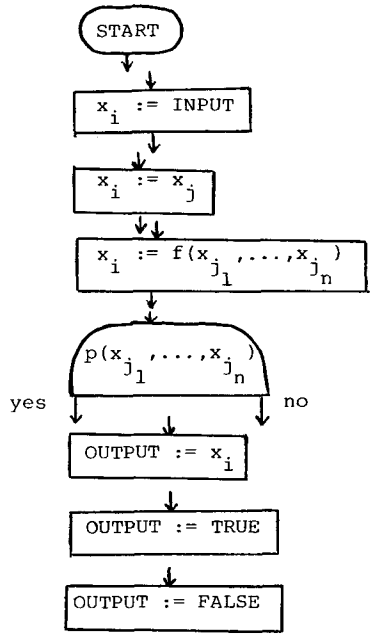
An algebra $S = \langle A; f_1,\ldots,f_k; p_1,\ldots,p_\ell \rangle$ is a set together with a collection of partial functions and partial predicates. Constants are 0-ary functions.

Definition: Let $S = \langle A; f_1,\ldots,f_k; p_1,\ldots,p_\ell \rangle$ and $S' = \langle A'; f_1',\ldots,f_k'; p_1',\ldots,p_\ell' \rangle$ be two algebras. Let $\tau: A' \to A$ be a possibly partial, onto, function. A partial function F_i on A' is a simulator of f_i if for all $a_1,\ldots,a_n$, if $f_i(\tau(a_1),\ldots,\tau(a_n))$ is defined, then $f_i(\tau(a_1)\ldots,\tau(a_n)) = \tau(F_i(a_1,\ldots,a_n))$. A partial predicate P_i on A' is a simulator of p_i if for all $a_1,\ldots,a_n$, if $p_i(\tau(a_1),\ldots,\tau(a_n))$ is defined, then $p_i(\tau(a_1),\ldots,\tau(a_n)) = P_i(a_1,\ldots,a_n)$.

Note that several representations in A' are permitted for each element of A. Predicates and functions are treated differently because of their different uses in programs. τ is required to be onto in order that every element of A have a representation, but is permitted to be partial since not every element of A need represent an element of A.

Of course, F_i and P_i so far have no relationship to the base operations of S'. In Appendix A, some purely algebraic relationships are suggested, but here we will restrict to flowchart programming systems.

Our flowcharts are composed of boxes of the following kinds:



Here, f and p represent function and predicate symbols of S respectively. We assume that each flowchart is consistent in its output type; it either outputs only values in A, or only values in {TRUE,FALSE}. Flowcharts are regarded as computing partial functions or predicates on A in the natural way, and we will take the notational liberty of identifying flowcharts with their functions or predicates.

Definition: Let $F_1,\ldots,F_k$ be flowcharts over S' which compute partial functions. Let $P_1,\ldots,P_\ell$ be flowcharts over S' which compute partial predicates. Then we say

$S \overset{\leq}{\underset{\tau}{}} (F_1,\ldots,F_k;P_1,\ldots,P_\ell)^{S'}$ [S is flowchart reducible to S' via $\tau,F_1,\ldots,F_k,P_1,\ldots,P_\ell$) provided each F_i is a simulator of the corresponding f_i, and each P_i is a simulator of the corresponding p_i. We write $S \overset{\leq}{\underset{\tau}{}} S'$ if $S \overset{\leq}{\underset{\tau}{}} (F_1,\ldots,F_k;P_1,\ldots,P_\ell)^{S'}$ for some $F_1,\ldots,F_k,P_1,\ldots,P_\ell$.

Of course, the same kind of definition could be made using other restrictions on the simulators. (See Appendix A.) For flowcharts (and many other natural classes), proper handling of the partialness will give the expected results regarding substitution, transitivity, congruence relations and quotient algebras.

Sometimes, (e.g. in Section V), we will wish to imagine a coding mapping as going from left to right instead of from right to left. For any algebra S, let $\bar{F}(S)$ (the free version of S) have as its domain all well-formed terms over the function symbols of S, (including constants) and its functions defined in the usual way for free algebras. Let $e(x)$, for $x \in$ domain $\bar{F}(S)$, be the "value" of x when evaluated in S. (e(x) may be undefined.) Let each predicate p on $\bar{F}(S)$ be defined by $p(x_1,\ldots,x_n) = p(e(x_1),\ldots,e(x_n))$ (so that $p(x_1,\ldots,x_n)$ may be undefined). Now, if the

situation is as in the preceding definitions, a natural (partial) mapping σ: domain $(\bar{F}(S)) \to A'$ may be constructed so that $e(x) = \tau(\sigma(x))$ for all x. σ is constructed inductively on the structure of terms in $\bar{F}(S)$, using $F_1,\ldots,F_k$. What this means is that our original definition, although it allows arbitrarily many representations in S' for each element of S, still distinguishes certain representations, the choice depending on the computation path used to generate the element.

To every element x of domain $\bar{F}(S)$ we associate a size $s(x)$, which is the minimum number of gates (each gate labelled by a function of $\bar{F}(S)$) in a no-input arbitrary fan-out circuit computing x. (This definition is used in Section V only.) For any flowchart F, L_F will denote the natural path length function and will be the complexity measure we will consider; i.e. $L_F(x)$ is the number of operations involved in the computation of F(x).

III. Expressiveness of Systems

The definitions in Section II provide a general framework for studying relative computability and relative complexity of systems. Our primary interest here is in complexity issues, and Sections IV-VI are devoted to consideration of different kinds of questions about relative complexity. Before moving to these questions, however, it is worthwhile to mention that there are some expressiveness issues to consider. For flowcharts alone, we have found no surprises. There are, however, several general questions that arise when we compare different program structures. For example, how dependent are our definitions on our use of flowcharts?

These questions are related to the comparative expressiveness questions studied in schematology (16,17,18,19). Paterson and Hewitt, for example, show that there is a recursion scheme not strongly equivalent to any flowchart scheme. The current framework suggests considering a similar question for interpreted schemes. (After all, when recursive programs are compiled, the underlying algebra is known.) Of course, if the algebra has exactly the power to compute the partial recursive functions ($\langle N;0,succ;=\rangle$ or $\langle \{0,1\}^*;\lambda,0succ;1succ;=\rangle$, for instance), then there is no difference in expressiveness between flowcharts and general r.e. tree schemes. In fact, the same is true for any algebra S with 0-ary and 1-ary functions only (but arbitrary predicates) and with $\langle N;0,succ;=\rangle \overset{\leq}{\tau} S$ for some τ. We ask whether there are algebras over which these classes differ:

Theorem 1: There exists an algebra $S = \langle \{0,1\}^*,\lambda,0succ;1succ;p\rangle$ (where p is a unary total recursive predicate) and a partial unary predicate q such that q is computed by a recursive program over S but q is not computed by any finite flowchart over S.

Proof: The techniques originate in [16] but become more complicated because we are no longer permitted to modify interpretations as they do. We must patch together a single diagonalizing interpretation. An outline of the proof appears as Appendix B. [X]

The moral is that our restriction to flowcharts is of some significance, even for expressiveness.

IV. Adequate Algebras With Domain N or $\{0,1\}^*$

There are many algebras with domain N or $\{0,1\}^*$ with (flowchart) expressive power exactly the partial recursive functions. Intuitively, however, not all of them are equally efficient. We give definitions for adequacy (in the sense of having efficiency comparable to Turing machines) of a set of operations over either of these domains and classify several commonly-used sets of operations as to their adequacy. In proofs of the classification results, we emphasize hierarchical techniques which fit our reducibility definitions.

For $x \in N$, define 2adic (x) to be the usual 2adic coding of x into 0's and 1's; 2adic is a bijection of N onto $\{0,1\}^*$. If $x \in \{0,1\}^*$, $|x|$ refers to the length of x. If $x \in N$, $|x|$ refers to $|2adic\ (x)|$.

Definition: An algebra S with domain N(or $\{0,1\}^*$) is adequate if for every polynomial computable function or predicate f on N(or $\{0,1\}^*$), there exist a polynomial p and flowchart F with:

(1) $\langle N;f\rangle \overset{\leq}{\lambda x\,[x]_{(F)}} S$
(or $\langle \{0,1\}^*;f\rangle \overset{\leq}{\lambda x\,[x]_{(F)}} S$),

(2) $L_F(x_1,\ldots,x_n) \leq p(\max |x_j|)$ for all inputs and

(3) the length of any value produced during the computation of F on inputs $x_1,\ldots,x_n$ is at most $p(\max |x_j|)$.

That is, tractable functions have tractable programs over S. The definition includes restrictions both on time and space; both will be needed to make the Lemma of this Section true. Although the definition only refers to tractable functions, it will follow that functions of any complexity on a Turing machine can be done over S with similar complexity.

Consider, for example, $\mathcal{B} = \langle \{0,1\}^*;\lambda,0succ,1succ,car,cdr,reverse;=\lambda,=0,=1\rangle$, where

$0succ\,(x) = x0$, $1succ(x) = x1$,

$car\,(x) = \lambda$ {if $x = \lambda$ the first symbol of x, otherwise, and

$cdr\,(x) = \lambda$ {if $x = \lambda$, all but the first symbol of x, otherwise. ($\mathcal{B}$ consists of a reasonable set of "unit-cost" string operations.) It is straightforward to show, by machine simulation, that $\mathcal{B}$ is adequate. In fact, $\mathcal{B}$ is able to simulate multihead multitape Turing machines in linear time. To show that other algebras are adequate, we would like to use the adequacy of $\mathcal{B}$ and our reducibility; to do so, we must introduce a complexity bound into the reducibility definition:

Definition: Let $S = \langle A;f_1,\ldots,f_k;p_1,\ldots,p_\ell\rangle$ and S' be algebras with domain N or $\{0,1\}^*$. Assume $S \overset{\leq}{\tau(F_1,\ldots,F_k;P_1,\ldots,P_\ell)} S'$

Assume there is some polynomial p for which:

(1) each F_i satisfies $L_{F_i}(x_1,\ldots,x_n) \leq p(\max |x_j| + |f_i(\tau(x_1),\ldots,\tau(x_n))|)$ for all inputs,

(2) the length of any value produced during the computation of F_i on $x_1,\ldots,x_n$ is at most

$p(\max |x_j| + |f_i(\tau(x_1),\ldots,\tau(x_n))|)$,

(3) each P_i satisfies $L_{P_i}(x_1,\ldots,x_n) \leq p(\max |x_j|)$ for all inputs, and

(4) the length of any value produced during the computation of P_i on $x_1,\ldots,x_n$ is at most $p(\max |x_j|)$.

Then we say $S \underline{\leq}_{\tau(F_1,\ldots,F_k;P_1,\ldots,P_\ell)}^{\text{poly}} S'$. We write $S \underline{\leq}_{\tau}^{\text{poly}} S'$ if $S \underline{\leq}_{\tau(F_1,\ldots,F_k;P_1,\ldots,P_\ell)}^{\text{poly}} S'$ for some $F_1,\ldots,F_k,P_1,\ldots,P_\ell$.

The "honesty" condition in (1) and (2) is used rather than a strict polynomial condition as in (3) and (4) primarily because even with this generalized definition, the following lemma is still true:

Lemma: Let S and S' be algebras over N or $\{0,1\}^*$, the identity $2a_{d_{1c}}{}^{-1}$ as appropriate. If $S \underline{\leq}_{\tau}^{\text{poly}} S'$ and S is adequate, then S' is adequate.

Proof: By substitution of flowcharts. ☒

It can now be shown that other common algebraic systems over N and $\{0,1\}^*$ are adequate:

Theorem 2: The following are adequate:

(a) $\langle \{0,1\}^*;\lambda,0,1,\text{cdr},\text{concatenation};= \rangle$

(b) $\langle \{0,1\}^*;\lambda,0\text{succ},1\text{succ};\text{prefix} \rangle$,

(where prefix $(x,y) = \begin{cases} \underline{\text{true}} \text{ if } x \text{ is a prefix of } y, \\ \underline{\text{false}} \text{ otherwise.} \end{cases}$)

(c) $\langle N;0,1,+;\underline{\leq} \rangle$

(d) $\langle N;0,1,+,\dot{-};= \rangle$

(e) $\langle N;0,1,+,\lambda x[|x|];= \rangle$,

Proof: Using the Lemma and the adequacy of $\mathcal{B}$. It is easy to show $\mathcal{B}_{\lambda x[x]} \underline{\leq}^{\text{poly}} \langle \{0,1\}^*;\lambda,0,1,\text{car},\text{cdr},\text{concatenation};= \rangle$, and car may be trivially reprogrammed in terms of the other primitives. The adequacy of (b) is then deduced from that of (a), of (c) from (b), and of (d) and (e) from (c). Obtaining (c) from (b) involves successive doubling and comparison to compute "prefix"; the other constructions are straightforward. ☒

It is also possible to prove that certain other systems, each with the same computing power as those in Theorem 2, are not adequate. Of particular interest are (a) and (b) below; together they combine to give an adequate system, but each separately is not adequate. In a sense which can be made precise, $\leq$ and + comprise a very low-level example of primitives that do not "help" each other (in the presence of $\langle N;0,\text{succ};= \rangle$).

Theorem 3: The following are not adequate:

(a) $\langle N;0,1,+;= \rangle$,

(b) $\langle N;0,\text{succ};\underline{\leq} \rangle$,

(c) $\langle \{0,1\}^*;\lambda,0\text{succ},1\text{succ};= \rangle$

(d) $\langle \{0,1\}^*;\lambda,0,1,\text{car},\text{concatenation};= \rangle$

Proof: (a) We show that $\leq$ cannot be computed over $N;0,1,+;=$ with polynomial path length. Assume that it can, and F is a flowchart computing $\leq$, with path length on inputs x,y at most $p(|x|,|y|)$, p a monotone polynomial. Choose $n \in N$ with $p(|2n|,|2n|) < n$, and consider

$A = \{(x,y) \mid n+1 \leq x \leq 2n \text{ and } 0 \leq y \leq n\}$ and

$B = \{(x,y) \mid (y,x) \in A\}$. We show that some member of A and some member of B must follow the same path in F.

Unwind F into an (infinite) tree T. Every input pair (x,y) causes a path in T. to be followed, of length $\leq p(|x|,|y|)$ and ending with either

OUTPUT := TRUE or OUTPUT := FALSE according

to whether $x \leq y$ or $x > y$. Each branch point in T results from an equals test which may be expressed in the form

$ax+by+c = a'x+b'y+c', \quad a,b,c,a',b',c' \in N.$

(The expression for each branch point may be constructed by ignoring the information obtained from tests along the path, and simply looking at uses of assignment and +.) Prune T by omitting all tests (and subsequent "no" subtrees) having a=a', b=b' and c=c'. Remaining is a tree T' for which, at every branch point, all inputs (x,y) causing the "yes" branch to be taken lie on one straight line. A counting argument now shows that some point in A (resp. B) must follow the "no" branch at every choice point, and this path must terminate.

(b) Consider F, a flowchart over $N;0,\text{succ};\leq$ which computes unary function f, and which has path length at most $p(|x|)$ on input x, for some polynomial p. Choose $n \in N$ with $p(|n|) < n$. Consider the behavior of F on input n and on input $x > n$. F must follow the same path in both cases, because succ cannot span from 0 to n in $p(|n|)$ steps, and $\leq$ is unable to distinguish n from x. But then consider how the output of F on input n was constructed. The output arose from a variable initialized either at 0 or n and increased by 1 a fixed number of times. Thus, for some $c \in N$ we have $f(x) = c$ for $x \geq n$, or else $f(x) = x+c$ for $x \geq n$.

(c) $\langle \{0,1\}^*;\lambda,\text{succ}0,\text{succ}1;= \rangle \underline{\leq}_{2a_{d_{1c}}}^{\text{poly}} \langle N;0,1,+;= \rangle$. Use the Lemma.

(d) We show that cdr cannot be computed over $\langle \{0,1\}^*;\lambda,0,1,\text{car},\text{concatenation};= \rangle$ with polynomial path length. Assume that it can, and F is such a flowchart, with path length on input x at most $p(|x|)$, p a polynomial. Choose $n \in N - \{0\}$ with $p(n) + 1 < 2^{n-1}$, and consider $A = \{x \in \{0,1\}^* \mid |x| = n \text{ and } \text{car}(x) = 0\}$. We will first show that two distinct members of A must follow the same path in F:

Unwind F into a tree T. Each branch point results from an equals test on two expressions, each built up from λ's, 0's, 1's and x's using car and concatenation. Restrict consideration to inputs $x \in A$; then we may simplify the expressions using simple reduction rules so that each expression is a (possibly empty) concatenation of 0's, 1's, and x's. But since all $x \in A$ are of the same length, each equation is satisfied by either no $x \in A$, all $x \in A$ or exactly one $x \in A$. Prune T by omitting all tests (and subsequent "no" subtrees) for which all $x \in A$ satisfy the reduced equation.

Remaining is a tree T' for which, at each branch point, at most one $x \in A$ causes the "yes" branch to be taken. A counting argument shows there are two values $x_1, x_2 \in A$ following the "no" branch at each point.

Consider the output from T' on inputs x_1 and x_2. Reductions as above show that the output on input x_1 is the value of a (possibly empty) concatenation of 0's, 1's and x_1's, while the output on x_2 is the value of the same expression with x_2 replacing x_1. But since $|cdr(x_1)| < |x_1|, x_1$ can cannot occur in the expression. Thus, the output is identical for both inputs.

X

We remark that Theorem 3 may be strengthened, with similar proofs, to be coding-independent.

One very interesting question we have not been able to resolve is:

Question: Is $\langle N;0,1,+,\times,=\rangle$ adequate?

This seems to be a fundamental one about the expressiveness of polynomials.

V. Optimal Codings of Arithmetic Systems into $\mathcal{B}$

We compare different possible codings of basic algebras into $\mathcal{B}$ of the preceding sections. We show that certain natural codings are nearly optimal, in the sense that nothing above a certain minimal level of complexity can be computed faster in any other coding. Thus, we have limits on the improvement possible through changes in coding. Ideas for proofs are fairly simple and much more general than presented here.

Consider coding $\langle N;0,1,+;\leq\rangle$ into $\mathcal{B}$. In the 2adic coding, + and $\leq$ can be done by flowcharts with path length linear in the log of the inputs in N. In comparing another coding to this one, it is reasonable to impose similar complexity restrictions on + and $\leq$ in the new coding. We obtain:

Theorem 4: Assume $\langle N;+;p\rangle \overset{\leq}{\tau}_{(F_+;P)} \mathcal{B}$, where p is a partial predicate. Assume there exists C such that $\tau(y)=x$ and $\tau(y')=x'$ imply $L_{F_+}(y,y') \leq C(|x|+|x'|)$. Further assume that $\tau(y)=x$ implies $L_P(y) \leq t(x)$, where t is a partial function.

Then there is a flowchart G and a constant C such that $\langle N;;p\rangle \overset{\leq}{2adic}^{-1}_{(G)} \mathcal{B}$, and $2adic^{-1}(y)=x$ implies $L_G(y) \leq C(|x|^2+t(x))$.

Proof: The 2adic representation of $x \in N$ allows us to quickly determine an efficient way to build up x using +. This allows for fast translation from the 2adic to a τ representation. More precisely, we show:

Lemma: Assume $\langle N;+;\rangle \overset{\leq}{\tau}_{(F_+)} \mathcal{B}$, and F_+ is as in Theorem 4. Then there is a flowchart G and a constant C such that $G(y)=y'$ implies $2adic^{-1}(y)=\tau(y')$, and $2adic(x)=y$ implies $L_G(y) \leq C|x|^2$.

Proof of Lemma: G first uses the bits of y to obtain a "parse" of $2adic^{-1}(y)$, i.e. a sequence of + operations, starting with 0 and 1, that generates $2adic^{-1}(y)$. The natural parse consists of a sequence of about $|y|$ operations, each involving either doubling, or doubling and adding 1. A fixed element of $\tau^{-1}(0)$ and one of $\tau^{-1}(1)$ are built into G. G then applies F_+ in the way described by the parse, using the fixed elements where needed. By the consistency of τ with F_+, an element of $\tau^{-1}(2adic^{-1}(y))$ is eventually obtained.

Each of the $|y|$ operations involves a bounded number of applications of F_+ to inputs which are τ-representations of integers with length at most $|y|$. Since $|2adic^{-1}(y)|=|y|$, the bound follows.

The Theorem is now an immediate consequence of the Lemma and the additional hypothesis. X

Stronger hypotheses are needed to obtain a similar result for functions as well as predicates:

Theorem 5: Assume $\langle N;+,f;\rangle \overset{\leq}{\tau}_{(F_+,F;P_<)} \mathcal{B}$, where f is a partial function. Assume F_+ is as in Theorem 4. Assume that $\tau(y)=x$ implies $L_F(y) \leq t(x)$, where t is a partial function. Further assume there exists C such that $\tau(y)=x$ and $\tau(y')=x'$ implies $L_{P_<}(y,y') \leq C(|x|+|x'|)$.

Then for any ε, there exist a flowchart G and a constant C such that $\langle N;f;\rangle \overset{\leq}{2adic}^{-1}_{(G)} \mathcal{B}$, and $2adic^{-1}(y)=x$ implies $L_G(y) \leq C(|x|^2+t(x)+|f(x)|^{2+\varepsilon})$.

Proof: 2adic (f(x)) is constructed bit-by-bit from a τ-representation of f(x). More precisely, we show:

Lemma: Assume $\langle N;+;\leq\rangle \overset{\leq}{\tau}_{(F_+;P_<)} \mathcal{B}$, with F_+ as in Theorem 4 and $P_<$ as in Theorem 5.

Then for any ε, there exist a flowchart G and a constant C with $G(y)=2adic(\tau(y))$ and $L_G(y) \leq C|\tau(y)|^{2+\varepsilon}$ for all y.

Proof of Lemma: Let $C_1 \in \tau^{-1}(1); C_1$ has a flowchart over $\mathcal{B}$. We first design a flowchart G_1 over $\mathcal{B}_1 = \langle \{0,1\}^*;\lambda,0succ,1succ,car,cdr,reverse,C_1,F_+;=,=\lambda,=0,=1;P_<\rangle$ which computes (the function computed by) G, and then obtain the needed flowchart G by replacement of the symbols C_1, F_+ and $P_<$ by their flowcharts.

The best G_1 we know arises from a compilation using techniques of Chandra [20], of a "loop-free linear recursive program" over $\mathcal{B}_1$. The relevant theorem appears at the beginning of Section VI. The recursive program we use is: (Notation is as in [20].)

Translate (x_0): data x_1, x_2

|* Given $x_0 \in \{0,1\}^*$, Translate (x_0) outputs $2adic(\tau(x_0))$ if $x_0 \in$ domain τ. Its behavior is otherwise unspecified. *|

START;

$\langle x_1, x_2\rangle$ Approx$(F_+(x_0, C_1), C_1)$;

RETURN (x_1);

Approx(x_0,x_1): data x_2, x_3

|* Given x_0 with $\tau(x_0)$ defined and ≥ 1, x_1 with $\tau(x_1)$ a power of 2, and $\tau(x_1) \leq \tau(x_0)$, Approx(x_0,x_1) returns two values: (1) the string obtained by deleting the leading 1 from the binary representation of $\lfloor \tau(x_0) \div \tau(x_1) \rfloor \times \tau(x_1)$ and (2) some value in $\tau^{-1}(\lfloor \tau(x_0) \div \tau(x_1) \rfloor \times \tau(x_1))$.*|

START;

if $P_<(F_+(x_1,x_1),x_0)$

then begin

$\langle x_2, x_3\rangle \leftarrow$ Approx$(x_0, F_+(x_1,x_1))$

if $P_{\leq}(F_+(x_3,x_1),x_0)$

then RETURN (succ1(x_2),$F_+(x_3,x_1)$);

else RETURN (succ0(x_2), x_3);

end;

else RETURN (λ, x_1);

The given assertions suffice to verify correctness of the program. On input y, the recursion depth is approximately $|\tau(y)|$. Also, if an argument z is generated during computation on input y, then $|\tau(z)|$ is at most approximately $|\tau(y)|$.

Using Chandra's theorem, we obtain G_1 computing G, with y ε domain (τ) implying $L_{G_1}(y) \leq C|\tau(y)|^{1+\varepsilon}$, for some constant C. Moreover, each argument z (with possibly finitely many exceptions) to which F_+ and $P_<$ are applied when G_1 is run on input y have $|\tau(z)|$ at most approximately $|\tau(y)|$.

Now obtain flowchart G by replacing in G_1, $F_+, P_<$ and C_1 by their flowcharts. The complexity bound follows from the hypotheses on F_+ and $P_{\leq}$.

The Theorem follows immediately. X

Theorems 4 and 5 limit the improved efficiency that could be obtained over the standard coding, at least for computation of "small" functions. The only possible improvement is a very local one arising from possible concise representations of large numbers (such an improvement is possible, for example, in floating-point codings.) In particular, polynomial size-bounded "polynomial-computable" functions and predicates in any coding satisfying the hypotheses of Theorems 4 and 5 are also "polynomial-computable" in the standard coding.

The reader may object to the complexity restrictions used in Theorem 4 and 5 - namely, all the codings have a uniform bound on the running time of flowcharts on all representations of an element. It is plausible that some efficient mappings might use infinitely many representations of some elements; in that case, a uniform bound is not the appropriate requirement. To prove optimality results for the standard coding when compared to codings allowing infinitely many representations for elements, we require a different way to compare complexity of codings. We will allow complexity to be calculated not only in terms of the natural number itself (as before) but also in terms of the way that number is generated in $\langle N;0,1,+;\leq\rangle$: Intuitively, the size of an expression evaluating to a number is a measure of the "size" of the number, and so it is probably reasonable to use that size as a parameter upon which to base measures of running time. Here we use F S), σ and s as defined in Section II.

Let $\mathcal{S}=\langle N;0,1,+;\leq\rangle$ and let s be the size measure on $F(\mathcal{S})$. In the standard coding, we may obtain flowcharts for + and ≤, σ': domain $(F(\mathcal{S}))\rightarrow\{0,1\}^*$ as in Section II and a constant C such that $L_{F_+}(\sigma'(x),\sigma'(x')) \leq C((s(x)+s(x'))$ and $L_{P_{\leq}}(\sigma'(x),\sigma'(x')) \leq C(s(x)+s(x'))$ for all x and x'.

So we impose similar restrictions on other codings in order to compare them. Allowing polynomial fudge-factors for simplicity, we obtain:

Theorem 6: Assume $\langle N;0,1,+;p\rangle \overset{\leq}{\tau} (F_0,F_1,F_+;P)\mathcal{B}$, and σ is as in Section II. Assume for some polynomial q, $L_{F_+}(\sigma(x),\sigma(x')) \leq q(s(x)+s(x'))$ and $L_p(\sigma(x)) \leq q(s(x))$ for all x, x'.

Then there exist a flowchart G and a polynomial q such that $\langle N;;p\rangle \overset{\leq}{2adic}^{-1}{}_{(G)}\mathcal{B}$, and $L_G(\sigma'(x)) \leq q(s(x))$ for all x.

Proof: By techniques similar to those used for Theorem 4. X

That is, even allowing codings with infinitely many representations for each element, the standard coding cannot be significantly improved upon for the computation of predicates. As before, a result for functions is obtainable under an additional hypothesis on $P_{\leq}$:

Theorem 7: Assume $\langle N;0,1,+,f;\leq\rangle \overset{\leq}{\tau}(F_0,F_1,F_+,F;P_{\leq})\mathcal{B}$, σ is as in Section II and F_+ is as in Theorem 6. Assume for some polynomial q, that $L_F(\sigma(x)) \leq q(s(x))$ for all x, that $L_{P_{\leq}}(\sigma(x),\sigma(x')) \leq q(s(x)+s(x'))$ for all x,x', and that $(\forall x)(\exists y)[F(\sigma(x)) = \sigma(y)$ and $s(y)\leq q(s(x))]$.

(This latter assumption represents a size restriction on the function computed by F.)

Then there exist flowchart G and polynomial q with $\langle N;f;\rangle \overset{\leq}{2adic}^{-1}{}_{(G)}\mathcal{B}$, and $L_G(\sigma'(x)) \leq q(s(x))$ for all x.

Similar optimality results are obtained for certain standard codings of Z and the positive rationals in $\mathcal{B}$. These results lead to some preliminary definitions for "adequacy " of operation sets over Z and Q_+ and to classification of some common operation sets as to their adequacy. Further details will have to be deferred to a longer paper.

VI. Low-Level Relative Complexity of Basic Operations on N and {0,1}*

Adequacy, as studied in Section IV, allows polynomial variation and is certainly not as fine a concept as we would like to consider. In this section, we reconsider commonly-used basic operations on N and {0,1}* to obtain a finer complexity classification. Results in this sections, since each deals with only a single algebra, use little of the expressive power of the present framework. The new framework simply provides a unification and an encouragement for hierarchical constructions. These results are only some of a virtually unlimited class of similar results which presumably could be obtained.

The primary aim here is to obtain upper and lower bounds on flowchart complexity; however, the best flowchart upper bounds we have obtained for the operations studied in this section have arisen through loop-free linear recursive programs compiled into flowcharts by Chandra's construction [20]. We use Chandra's formalism for linear recursive programs, except that we do not specify that inputs be distinguished constants of the algebra; our (interpreted) programs compute functions and predicates rather than single values. As he does, we use D_F (the recursion depth of program F) as the function giving a measure of complexity. We use as a lemma:

Theorem (Chandra): Assume F is a linear recursive program over S, and $\langle\{0,1\};0,1;\rangle \overset{<}{\underset{T}{\geq}} S$. Assume ε is a positive real. Then there exists F', a flowchart over S computing the same function or predicate as F, and $C \in N$ such that $L_{F'}(x_1,\ldots,x_n) \leq C(D_F(x_1,\ldots,x_n))^{1+\varepsilon}$ for all inputs. Moreover, except for a fixed finite set of possible exceptions, if a base operation of S is applied to an element of S during the execution of F' on a given input, then the same base operation was applied to the same element during the execution of F on the same input.

Theorem 8: For every positive real ε, there exist exists $C \in N$ and a flowchart F, such that (for all inputs):

(a) F computes parity over $\langle N;0,1,+;\leq\rangle$ and $L_F(x) \leq C(\log x)^{1+\varepsilon}$,

(b) F computes $\dot{-}$ over $\langle N;0,1,+;\leq\rangle$ and $L_F(x_1,x_2) \leq C(\log(x_1 \dot{-} x_2))^{1+\varepsilon}$

(c) F computes $\times$ over $\langle N;0,1,+;\leq\rangle$ and $L_F(x_1,x_2) \leq C(\min(\log x_1, \log x_2))^{1+\varepsilon}$,

(d) F computes $\lambda x_1,x_2\,[x_1^{x_2}]$ over $\langle N;0,1,+,\times;\leq\rangle$ and $L_F(x_1,x_2) \leq C(\log x_2)^{1+\varepsilon}$,

(e) F computes parity over $\langle N;0,1,+,\times;\leq\rangle$, and $L_F(x) \leq C(\log x)(\log\log x)^{1+\varepsilon}$,

(f) F computes $\dot{-}$ over $\langle N;0,1,+,\times;\leq\rangle$, and $L_F(x_1,x_2) \leq C(\log(x_1 \dot{-} x_2))(\log\log(x_1 \dot{-} x_2))^{1+\varepsilon}$

(g) F computes $\leq$ over $\langle N;0,1,+,||;=\rangle$, and $L_F(x_1,x_2) \leq C(\log x_1)^{1+\varepsilon}$,

(h) F computes reverse, over $\langle\{0,1\}^*;\lambda,0succ,1succ;prefix\rangle$ and $L_F(x) \leq C|x|^{1+\varepsilon}$.

Proof: We use Chandra's theorem; sometimes linear recursive programs are used to compute the needed function, and other times they are used to compute "modules" of the function. For instance, we show:

(c) Mult (x_0,x_1): data x_2

```
|* Given x0,x1 ∈ N, Mult(x0,x1) returns their
product.*|
START;
if  x0 ≤ x1
then  x2 ← Multl(x0,x1);
else  x2 ← Multl(x1,x0);
RETURN (x2);
```

Multl(x_0,x_1): data x_2,x_3

```
|* Given x0,x1 ∈ N with x0 ≤ x1, Multl(x0,x1)
returns their product. *|
START;
if  x0 ≤ 0
then RETURN (0);
else begin
     <x2,x3> ← Approx (x0,x1,1,x1);
     RETURN (x3);
     end;
```

Approx (x_0,x_1,x_2,x_3): data x_4,x_5

```
|* Given x0 ≤ x1 with x0 ≠ 0, x2 ≥ 1 and x3=x2·x1,
   Approx (x0,x1,x2,x3) returns two values:
  (1) the largest y ≤ x0 such that y is a multiple
      of x2, and
  (2) for y as in (1), y·x1.* |
START;
if  x2 ≤ x0
then begin
     <x4,x5> ← Approx (x0,x1,x2+x2,x3+x3);
     x6 ← x4+x2;
     if  x6 ≤ x0
     then  RETURN (x6,x5+x3);
     else  RETURN (x4,x5);
     end;
else RETURN (0,0);
```

(e) Power (x_0,x_1): data: x_2

```
|* If x0 > x1, Power(x0,x1) returns the largest
y ≤ x0-x1 such that y is a power of 2.  0 is re-
turned otherwise.  *|
START;
if  x0 ≤ x1
then RETURN (0);
```

```
else if  x0 ≤ x1+1
     then  RETURN(1);
     else begin
          x2 ← Approx (x0,x1,2);
          RETURN (x2);
          end;

Approx (x0,x1,x2):        data  x3

|* Given x0 ≥ x1+2, x2 = 2^(2^a)  for some a ≥ 0,
   Approx (x0,x1,x2) returns the largest y ≤ x0-x1
   such that y = 2^(b.2^a)  for some b ε N. *|

START;
if  x2+x1 ≤ x0
then begin
     x3 ← Approx (x0,x1,x2.x2);
     if  x3.x2 + x1 ≤ x0
     then  RETURN (x3.x2);
     else  RETURN (x3);
     end;
else RETURN (1);
```

For any ε, we may obtain a flowchart F' for Power over $\langle N;0,1,+,x;\leq\rangle$ with $L_{F'}(x_0,x_1) \leq C(\log\log(x_0 \dot{-} x_1))^{1+\varepsilon}$, for some constant C.

Now a simple flowchart F" may be constructed computing parity over $\langle N;0,1,+,\text{power};\leq\rangle$, with $L_{F"}(x) \leq C \log x$ for some C. Moreover, for all values y produced during the computation of F" on input x, we have $y \leq x$. Substitution of F' in F" yields the desired bound.

The other constructions are similar; a few programming tricks are needed, for example for (g). ☒

It is not difficult to obtain corresponding lower bounds, by techniques such as those used to prove Theorem 3 and by counting techniques of Stockmeyer [21]. In no case, however, is the obtained lower bound sensitive to the "ε". We have as yet been unable to prove an interesting distinction in complexity between linear recursive programs and flowcharts. We conjecture, for example:

Conjecture:. There does not exist a flowchart F and a constant such that F computes reverse over $\langle\{0,1\}^*;\lambda,0succ,1succ;prefix\rangle$ and $L_F(x) \leq C|x|$.

Appendix A: Algebraic Concepts

The complexity characterizations in this paper, while stated for flowchart simulators, do not require the full power of flowcharts for their proofs. We are working on the isolation of the relevant algebraic properties sufficient to derive lower bounds and realistic upper bounds for relative complexity of interesting problems. Some definitions which appear fruitful are:

Definition: Let S and S' be algebras, with notation as before. A mapping ϕ: A→A' is a genomorphism if ϕ has a partial left-inverse τ such that for all f_i and all $b_1,\ldots,b_n$,

$$f_i(b_1,\ldots,b_n) \in \tau([\phi(b_1),\ldots,\phi(b_n)]).$$

(The brackets refer to the algebraic closure.)

We note that the notation in the above definition differs from that in [15].

Definition: Let S be an algebra. A function F on A is closed if $F(a_1,\ldots,a_n) \in [a_1,\ldots,a_n]$ for all $a_1,\ldots,a_n$ in its domain.

Definition: Let S,S' be algebras, τ:A'→A a possibly partial, onto, function. τ is closed simulative if every basic function of S has a closed simulator over S'.

We note elementary implications between the two definitions:

Lemma: If τ is closed simulative, then every right inverse of τ is a genomorphism. If ϕ is a genomorphism whose image is a subalgebra, then there exists a left inverse of ϕ which is closed simulative.

Note that closed simulative maps may be used in the natural way to define a transitive reducibility between algebras. These weak definitions deal with basis functions only, and capture the concept of accessibility of values within algebras. Although expressiveness considerations may or may not be interesting for these definitions, natural definitions of complexity can be made, and some nontrivial low-level complexity results can be made to follow from them.

In order to introduce meaningful restrictions on predicate simulators, we define "tree predicates" (and "tree functions") over an algebra. Intuitively, these are predicates and functions computable by general tree schemes (possibly infinite, loop-free flowcharts) over the algebra. A precise algebraic characterization of such predicates and functions is not difficult.

Definition: Let S,S' be algebras, τ:A'→A a possibly partial, onto function. τ is tree simulative if every basic function and predicate of S has a tree simulator over S'.

Again, a transitive reducibility between algebras results. Also again, complexity considerations seem more interesting than expressiveness considerations, and several nontrivial complexity results (of the "decision tree" variety) can be made to follow from these definitions.

Since the "flowchart simulative" reducibility used earlier in this paper is (properly) stronger than the closed simulative and tree simulative reducibilities presented here, it is sensible to discuss the attribution of parts of inherent flowchart complexity of problems to closure and to tree considerations.

Appendix B: Proof Outline for Theorem 1

We diagonalize over unary predicate flowcharts $F_1,\ldots,F_n,\ldots$, each of which uses only operation symbols from S. We assume without loss of generality that F_n has at most n boxes (and n registers). Borrowing from Paterson and Hewitt, we define:

$q_p(x)$= if p(x)

then TRUE

else $q_p(x0) \wedge q_p(x1)$.

Note q_p is always either TRUE or undefined. We construct p so the q_p satisfies the needed conditions on q in the statement of the theorem; at stage n, we insure that F_n does not correctly compute q_p. We also construct three auxiliary functions s: N → {0,1}*, f: N→N and g: N→N such that

s(0) = λ

for all n, s(n) is a prefix of s(n+1),

length (s(n)) = f(n),

all strings put into p before stage n have lengths less than f(n),

(1) only extensions of s(n) get put into p at stage n,

and g(n) is "very large" relative to n and f(n).

Stage n: Let p_n be the set of values put into p before stage n, and assume s(n), f(n) and g(n) are known.

Let $x = s(n)0^{g(n)}$. Let $p^1 = p_n \cup \{xw \mid |w| = g(n)\}$.

There are two possibilities:

Case 1: F_n on input x with predicate symbol interpreted as p^1 halts with either TRUE or FALSE as output.

In this case, we claim that at most $(2^{f(n)}+2g(n)+1)^n.n$ steps were executed.

(For any predicate p, define equivalence relation R_p by wR_pw^1 iff $(\forall v)$ $p(wv)$ iff $p(w^1v)$. Then there are at most $(2^{f(n)}+2g(n)+1)$ equivalence classes in R_{p^1}. If there are two points in the computation where all registers have R_{p^1}-equivalent contents and control is at the same flowchart box, the computation is in a loop.) Therefore, since g(n) is large, we can find w,v with

$$|w| = |v| = g(n),$$

(2) xw was not generated in any register during the computation of F_n on x with p^1,

(3) no extension of s(n)v was generated in any register during the computation of F_n on x with p^1.

Let $p_{n+1} = p^1 \cup \{s(n)vy \mid |y| = g(n) \text{ and } y \neq w\}$.

Then we claim:

F_n on x and p_{n+1} behaves exactly like F_n on x and p^1, as far as register contents, predicate answers and eventual output are concerned (by (3)). Also, F_n on s(n)v and p_{n+1} behaves exactly like F_n on x and p^1, as far as predicate answers and eventual output are concerned (by (2) and (3)).

Let f(n+1) > f(n)+2g(n), and also be greater than the lengths of all strings generated during the computation of F_n on s(n)v, with p_{n+1}. Let s(n+1) be any string of length f(n+1) which is not an extension of s(n) vw.

(By (1), F_n on s(n)v and p behaves exactly like F_n on x and p^1, as far as predicate answers and output are concerned. Thus, F_n on s(n)v and p halts with either ouptut TRUE or FALSE. But q_p (s(n)v) is undefined, diagonalizing over F_n.)

Case 2: F_n on x with p^1 does not halt.

Let $p_{n+1} = p^1$. Let f(n+1)=1+f(n)+2g(n).

We claim there exists s(n+1) such that

s(n) is a prefix of s(n+1),

$|s(n+1)| = f(n+1)$,

and no extension of s(n+1) is generated during the non-halting computation of F_n on x with p^1.

(Since there are at most $(2^{f(n)} + 2g(n) + 1)$ equivalence classes in R_{p^1}, two of the first $(2^{f(n)} + 2g(n) + 1)^n.n+1$ steps must take us to the same box in F_n with all corresponding register contents R_{p^1} - equivalent. Thereafter, the computation is in a loop, repeating a sequence of flowchart boxes and R_{p^1} - equivalence classes with period $\pi \leq (2^{f(n)} + 2g(n) + 1)^n.n$.

We require some notation for register contents. Assume exactly n registers, for simplicity, and consider the following trace:

	. . .	. . .		. . .
	$C_{1,\pi-1}^{(0)}$	$C_{2,\pi-1}^{(0)}$	. . .	$C_{n,\pi-1}^{(0)}$
*	$C_{1,\pi}^{(0)}$	$C_{2,\pi}^{(0)}$	. . .	$C_{n,\pi}^{(0)}$
	$C_{1,1}^{(1)}$	$C_{2,1}^{(1)}$	. . .	$C_{n,1}^{(1)}$
	$C_{1,2}^{(1)}$	$C_{2,2}^{(1)}$	. . .	$C_{n,2}^{(1)}$
	. . .	. . .		. . .
**	$C_{1,\pi}^{(1)}$	$C_{2,\pi}^{(1)}$	. . .	$C_{n,\pi}^{(1)}$
	$C_{1,1}^{(2)}$	$C_{2,1}^{(2)}$	. . .	$C_{n,1}^{(2)}$
	. . .	. . .		. . .

* and ** indicate the lines which give the register contents at the two configurations which first put

us into R_{pl}- equivalence classes, as above.

$C_{i,j}^{(k)}$ denotes the contents of register i after k entrances into the loop and j steps into that interation. The (0)- superscript values are obtained from the values preceding the first entrance into the loop. Note the following dependency:

For every i,j, there exists $W_{i,j} \in \{0,1\}^*$ and $h(i,j) \in 1,\ldots,n$ with either

(4) $(\forall k \geq 1)\ C_{i,j}^{(k)} = W_{i,j}$

or (5) $(\forall k \geq 1)\ C_{i,j}^{(k)} = C_{h(i,j),\pi}^{(k-1)} W_{i,j}$.

Now for a given (i,j), there are three possibilities:

Case a: (4) holds. Then there are at most 2 strings $C_{i,j}^{(k)}$.

Case b: (5) holds, and there exists $m \geq 1$ with the pair

$$\underbrace{h(\ldots h(h(h}_{m}(i,j),\pi),\pi),\pi\ldots),\pi$$

satisfying (4). In this case, we can choose $m \leq n$, since there are only n registers. Then there are at most n + 2 values $C_{i,j}^{(k)}$.

Case c: (5) holds, and

$$\underbrace{h(\ldots h(h(h}_{a}(i,j),\pi),\pi),\pi\ldots) = \underbrace{h(\ldots h(h(h}_{b}(i,j),\pi),\pi,\pi\ldots)$$

for some $1 \leq a < b \leq n+1$. Then there are at most n strings $u_1,\ldots,u_n$, and strings v and y with all $C_{i,j}^{(k)}, k \geq n$ in $\{u_\ell\, y^*v \mid 1 \leq \ell \leq n\}$.

In summary, when F_n runs on x and p^1, the following values are produced: at most

$$n(2^{f(n)} + 2g(n) + 1)^n.n\ (2 + (n+2) + n)$$

"sporadic" values exist plus at most

$$n(2^{f(n)} + 2g(n) + 1)^n.n.n \text{ choices of } u,$$

y,v having the remaining values in uy^*v.

But $(\forall u,y,v,\ell)$ uy^*v includes extensions of at most $\ell+1$ distinct strings of length ℓ. Then since g(n) is so large relative to n and f(n), it follows that some extension s(n+1) of s(n), $|s(n+1)| = f(n+1) = 1 + f(n) + 2g(n)$ has none of its extensions in the set of values produced.

Choosing s(n+1) in this way insures that F_n on x and p will diverge, thus diagonalizing over F_n.) ☒

REFERENCES

1. C. C. Elgot, Monadic Computation and Iterative Algebraic Theories, IBM Report RC 4564, October 1973.

2. B. H. Liskov and S. N. Zilles, Specification Techniques for Data Abstractions, Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 7-19.

3. F. L. Morris, Correctness of Translations of Programming Languages-An Algebraic Approach Stanford U. Report CS-72-303, Aug. 72.

4. R. M. Burstall, An Algebraic Description of Programs with Assertions, Verification and Simulation, in Proc. ACM Conference on Proving Assertions About Programs SIGPLAN Notices 7,1 ACM 72.

5. G. Birkhoff, The Role of Algebra in Computing, in Computers in Algebra and Number Theory, Vol iv SIAM-AMS Proc. A.M.S. 1971.

6. J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright, A Junction Between Computer Science and Category Theory: I Basic Concepts and Examples, Part 1, IBM Report RC-4526 (Sept. 73); Part 2, IBM Report RC-5908 (March 1976).

7. R. M. Burstall and J. W. Thatcher, The Algebraic Theory of Recursive Program Schemes, Symposium on Category Theory Applied to Computation and Control, Lecture Notes in Computer Science 25(1975), 126-131.

8. J. B. Wright, J. A. Goguen, J. W. Thatcher and E. G. Wagner, Rational Algebraic Theories and Fixed-Point Solutions Proc. 17th Annual Symposium on Foundations of Computer Science, (Oct. 76) 147-158.

9. J. V. Guttag, E. Horowitz and D. R. Musser, Abstract Data Types and Software Validation. Research Report 76-48. Information Sciences Institute, Aug. 1976.

10. A. V. Aho, J. E. Hopcroft and U. D. Ullman, The Design and Analysis of Computer Algorithms Addison-Wesley, 1974.

11. S. A. Cook, The Complexity of Theorem-Proving Procedures, Third Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.

12. R. Karp, Reducibility Among Combinatorial Problems, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds. Plenum Press (1972) pp. 85-104.

13. R. Ladner, N. A. Lynch and A. L. Selman, Comparison of Polynomial-Time Reducibilities, Sixth Annual ACM Symposium on Theory of Computing, 1974, pp. 110-121.

14. A. Borodin and I. Munro,

15. E. K. Blum and D. R. Estes, A Generalization of the Homomorphism Concept, Algebra

Universalis, To appear.

16. M. S. Paterson and C. E. Hewitt, Comparative Schematology. Record of Project MAC Conference on Concurrent Systems and Parallel Computation (1970). pp. 119-128.

17. D. C. Luckham, D. M. R. Park and M. S. Paterson, On Formalised Computer Programs, J.C.S.S., Vol. 4. No. 3, June 1970, 220-249.

18. Stephen J. Garland and David C. Luckham, Program Schemes, Recursion Schemes and Form Formal Languages, J.C.S.S., Vol. 7, No. 2 April 1973, 119-160.

19. Takumi Kasai, Translatability of Flowcharts into While Programs, J.C.S.S., Vol. 9, No. 2 October 1974, 177-195.

20. A. K. Chandra, Efficient Compilation of Linear Recursive Programs, Stanford Artificial Intelligence Project MEMO AIM-167, April 1972.

21. L. J. Stockmeyer, Arithmetic versus Boolean Operations in Idealized Register Machines, IBM RC 5954, (A 25 837).