Math. Systems Theory 13, 187–207 (1980)

Mathematical Systems Theory

Relative Complexity of Operations on Numeric and Bit-String Algebras

Nancy Lynch[1*†] and Edward K. Blum[2*‡]

[1]School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

[2]Department of Mathematics, University of Southern California, Los Angeles, California 90007

Abstract. Sets of primitive operations for algebras with numerical and bit string domains are classified according to their computational efficiency. The relative complexity of certain basic operations on such algebras is determined.

1. Introduction

Many different algebras with domain N (the non-negative integers) or $\{0,1\}^*$ can be shown to be equivalent in the sense that flowcharts over those algebras have the ability to compute exactly the partial recursive functions. However, it is intuitively clear that not all such algebras can be used to compute with equal "efficiency." It is the purpose of this paper to provide a classification for the relative complexity of different sets of operations over N and $\{0,1\}^*$.

The present work is a specialized outgrowth of work in [1], where relative complexity of arbitrary algebras is studied in a more general setting. The techniques and results of the present paper and of [1] are intended to suggest a more "modular" approach to complexity analysis than is commonly taken. The framework defined in [1] is used for studying *coding-independent* relative complexity of algebras, whereas in all of the problems of this paper, a fixed standard coding is used. Furthermore, (with one exception) the problems of this paper involve algebras over two specific domains, N and $\{0,1\}^*$. Therefore, the full generality of the framework in [1] is not required here. A preliminary report [2]

*This research was partially supported by NSF Grant DCR75-02373.
†This research was partially supported by NSF Grants MCS77-15628 and MCS78-01689.
‡This research was partially supported by NSF Grant MCS78-07461.

0025/5661/80/0013-0187$04.20

includes outlines of the results of the present paper as well as those of [1]. Henceforth, unless otherwise stated, all algebras are over N and $\{0,1\}^*$.

Section II contains notation and basic definitions. Many types of program scheme classes could be used as bases for relative complexity classification of algebras. For definiteness, we emphasize the classification yielded by finite flowcharts. In order to prove results about finite flowcharts, however, we consider two other scheme classes, the linear recursive schemes and the effective schemes.

In Section III, a classification is established for algebras, based on their efficient computing power. An algebra is defined to be "adequate" if, in a standard coding of its domain, its flowcharts allow functions to be computed at least as efficiently as do Turing machines (to within a polynomial). Several common algebras are classified as adequate or inadequate. Extension of the concept of "adequacy" to algebras with domains other than N and $\{0,1\}^*$ is also discussed briefly.

In Section IV, classification finer than that provided by general polynomials is considered. Upper and lower bounds are obtained for the flowchart complexity of various functions on particular numeric and bit string algebras. The problems selected for consideration are representative of a large class of possible questions. The present results are unified by the general methods used for their proofs. Namely, in each case, our best upper bound arises from a (complexity-increasing) compilation of an interpreted linear recursive scheme, using a technique of Chandra [3], while our best lower bound applies to effective schemes as well as to flow charts.

Section V contains suggestions for further work.

2. Notation and Definitions

An *algebra* $\mathcal{Q} = \langle \mathrm{Dom}_{\mathcal{Q}}; \mathrm{Fun}_{\mathcal{Q}}; \mathrm{Rel}_{\mathcal{Q}} \rangle$ is a set $\mathrm{Dom}_{\mathcal{Q}}$ (the *domain* of $\mathcal{Q}$) together with a finite family $\mathrm{Fun}_{\mathcal{Q}}$ of partial functions (more often called "operations" in the algebraic literature) and a finite family $\mathrm{Rel}_{\mathcal{Q}}$ of partial relations on that set. Distinguished constants are 0-ary functions.

(The inclusion of relations is a departure from the usual definitions given in, say, [11, 12]. However, it agrees with the notion of *algebraic system* in [13].)

For an algebra $\mathcal{Q}$, a flowchart F over $\mathcal{Q}$ is constructed in the usual way as a directed graph having a finite number of boxes (i.e. nodes) of the types:

Start:	$\boxed{\text{Start}}$
Input:	$\boxed{x_i := \text{input}_j}$
Copy:	$\boxed{x_i := x_j}$
Function:	$\boxed{x_i := f(x_{j_1}, \ldots, x_{j_k})}$
Relation:	$r(x_{i_1}, \ldots, x_{i_k})$

Function Output: $\boxed{\text{Output:} = x_i}$

Relation Output: $\boxed{\text{Output:} = \text{TRUE}}$

$\boxed{\text{Output:} = \text{FALSE}}$

where $f \in \mathrm{Fun}_{\mathcal{A}}$ and $r \in \mathrm{Rel}_{\mathcal{A}}$ and the x's are variables. (Strictly speaking, f and r are function and relation symbols for the members of $\mathrm{Fun}_{\mathcal{A}}$ and $\mathrm{Rel}_{\mathcal{A}}$. We shall rely on the reader to make this distinction.) Output boxes have no successors, relation boxes have two successors, and all others have one successor. A flowchart is either a *function flowchart*, in which case all output boxes are function output boxes, or a *relation flowchart*, in which case all output boxes are relation output boxes. There is exactly one start box.

For an algebra $\mathcal{A}$, a *linear recursive scheme* R over $\mathcal{A}$ is defined as in [3]. R consists of a finite collection of Algol-like procedure definitions, in which each procedure can call at most one other procedure. The Algol-like language has a fairly general instruction set, with conditions, typed variables (including Boolean variables with a fixed interpretation), vectors of parameters for procedures, but no looping constructs other than recursion. The basic function and relation symbols used are those in $\mathrm{Fun}_{\mathcal{A}}$ and $\mathrm{Rel}_{\mathcal{A}}$. Chandra's interpreted schemes compute elements whereas ours are intended to define functions and relations. Thus, in contrast to his definition, our notion of interpretation leaves inputs to a program uninterpreted. As before, we consider *function schemes* and *relation schemes*.

For an algebra $\mathcal{A}$, an *effective scheme* E over $\mathcal{A}$ is defined as in [4]. An effective scheme is composed of boxes of the same types as used for flowcharts. Rather than requiring that the number of boxes be finite, however, we require only that the formal scheme itself be generable in a recursively enumerable way. That is, there is a Turing machine able to construct a straightforward coding of the scheme, e.g. as a countable binary tree with a possibly infinite number of variables [4]. Again, we consider *function schemes* and *relation schemes*.

Semantics of all schemes are taken to be evident or as given in the references.

If a function f is undefined at inputs $x_1,\ldots,x_k$, we say that $f(x_1,\ldots,x_k) = \infty$. We use the conventions that $\infty \leq \infty$ and that $n < \infty$ for all n in N.

For any flowchart or effective scheme S, L_S denotes the natural path length function (the time complexity measure we will use); that is, for any inputs $x_1,\ldots,x_k$, $L_S(x_1,\ldots,x_k)$ denotes the number of boxes along the computation path in S for inputs $x_1,\ldots,x_k$. If R is a linear recursive scheme, L_R denotes the recursion depth function [3]. Any flowchart F can be "unfolded" into an equivalent effective scheme E, with $L_E = L_F$. Similarly, any linear recursive scheme R can be translated naturally into an equivalent effective scheme E, with $L_E \leq cL_R$ for some constant c depending on R but independent of the interpretation and input. (This follows because linear recursive schemes are loop-free.)

We require a "standard coding" mapping the domain N onto $\{0,1\}^*$. Define a total function 2adic: $N \to \{0,1\}^*$ by $\mathrm{2adic}(n) =$ the binary representation of

$n+1$, with the leading 1 removed. 2adic is a one-to-one correspondence between N and $\{0,1\}^*$.

We use m,n as number variables, and x,y as string variables. x,y are also used to denote elements of arbitrary domains. F is used for flowcharts, R for linear recursive schemes and E for effective schemes.

3. Adequate Algebras

A simple classification of the (flowchart) computing power of several algebras with domain N and $\{0,1\}^*$ appears in [1]. Namely, an algebra $\mathcal{Q}$ with domain N or $\{0,1\}^*$ is said to *have recursive power* provided each partial computable function (on N or $\{0,1\}^*$ as appropriate) is computable by flowchart over $\mathcal{Q}$. It is not difficult to show the following.

Theorem 3.1. *The following algebras have recursive power*:

(a) $\langle N; 0, \text{suc}; =\rangle$ (*where* $\text{suc}(n)=n+1$),

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

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

(d) $\langle N; 0, 1, +, \dot{-}; =\rangle$ (*where* $m \dot{-} n = 0$ *if* $m \leq n$, $m-n$ *otherwise*),

(e) $\langle N; 0, 1, +, ||; =\rangle$ (*where* $|n| = |2adic(n)|$, *the length of the 2adic representation of n, or equivalently*, $|n| = \lfloor \log(n+1) \rfloor$. *Note that this notation persists throughout the paper.*)

(f) $\langle\{0,1\}^*; \lambda, 0, 1, \text{tail}, \text{concat}; =\rangle$ (*where* λ *is the empty string*, $\text{tail}(x)=\lambda$ *if* $x=\lambda$, *all but the first symbol of* x, *otherwise, and* $\text{concat}(x,y)=xy$),

(g) $\langle\{0,1\}^*; \lambda, 0\text{suc}, 1\text{suc}; \text{prefix}\rangle$ (*where* $0\text{suc}(x)=x0$, $1\text{suc}(x)=x1$, *and* $\text{prefix}(x,y)$ *is true iff* x *is a prefix of* y),

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

(i) $\langle\{0,1\}^*; \lambda, 0, 1, \text{concat}; =\rangle$, *and*

(j) $\langle\{0,1\}^*; \lambda, 0, 1, \text{head}, \text{tail}, 0\text{suc}, 1\text{suc}, \text{reverse}; =\lambda, =0, =1\rangle$ (*where* $\text{head}(x)=\lambda$ *if* $x=\lambda$, *the first symbol of* x *otherwise, and where the predicates are tests for equality with short strings*).

Proof. By flowchart programming. For cases which pose some difficulties ((h) and (i)), proofs appear in [1]. □

It seems apparent that not all algebras with recursive power have the same "efficiency." Intuitively, we would not expect $\langle N; 0, \text{suc}; \leq\rangle$ to be an "adequate" algebra for a programming language because the successor operation constrains generation of new values to occur too slowly. By experience with LISP, however, we might expect that $\langle\{0,1\}^*; \lambda, 0, 1, \text{head}, \text{tail}, \text{concat}; =\rangle$ would be "adequate." A criterion for "adequacy" of an algebra with domain N or $\{0,1\}^*$ might be that a function computable in polynomial time by a Turing machine be computable in a polynomial number of steps by a flowchart over the system. We make this criterion precise and then classify several algebras as to their adequacy. (The results of this section are insensitive to polynomial variation, and within such variation, Turing machines are equivalent to any reasonable model of computation [5]. Finer classification than polynomial is not used in this section because of the fact that reducibility techniques are used, which require closure of bounding functions under composition.)

Definition. An algebra $\mathcal{A}$ with domain N or $\{0,1\}^*$ is *adequate* if for every polynomial-time computable function or relation f on N (or on $\{0,1\}^*$, as appropriate), there exist polynomial p and flowchart F over $\mathcal{A}$ satisfying the following conditions.

(a) F computes f,

(b) $L_F(x_1,\ldots,x_k) \le p\big(\max_{1\le i\le k} |x_i|\big)$, and

(c) if y is any value produced during the computation of F on inputs $x_1,\ldots,x_k$, then $|y| \le p\big(\max_{1\le i\le k} |x_i|\big)$.

The definition includes restrictions on both time and space; both are needed later, for example, for Lemma 3.5(b). Although the definition refers only to polynomial-time functions, Corollary 3.4 states that adequacy of $\mathcal{A}$ implies that computable functions of *any* complexity $t(x)$ are computable by flowcharts over $\mathcal{A}$ with complexity at most a polynomial in $t(x)$.

We consider first the classification of an algebra having operations as much like Turing machines as possible. Let $\mathfrak{B}$ denote the algebra $\langle\{0,1\}^*; \lambda, 0, 1, \text{head}, \text{tail}, 0\text{suc}, 1\text{suc}, \text{reverse}; =\lambda, =0, =1\rangle$ of Theorem 3.1(j). We prove that $\mathfrak{B}$ is adequate. In fact, $\mathfrak{B}$ can be used to carry out a step-by-step simulation, with at most a constant factor increase in complexity, of a multihead multitape Turing machine. Consider a Turing machine having a finite number of two-way read-write tapes, with a finite number of heads per tape. The first k tapes are input tapes. Initially, each input tape contains some string in $\{0,1\}^*$ with all heads for those tapes on the leftmost square of the input string. Tapes are otherwise blank, and all heads on each tape initially coincide. The last tape is the output tape. If and when the machine halts, it does so with all output heads on the leftmost square of the output string, and the output tape blank except for the output string. Any finite number of tape symbols is allowed, and the machine can detect coincidence of heads on the same tape.

Lemma 3.2. *Assume $f:(\{0,1\}^*)^k \to \{0,1\}^*$ is a partial function computed by a Turing machine as above, and $t:(\{0,1\}^*)^k \to N$ is a partial function such that the machine halts within $t(x_1,\ldots,x_k)$ steps on inputs $x_1,\ldots,x_k$. Then there exist flowchart F over $\mathfrak{B}$ computing f with*

$$L_F(x_1,\ldots,x_k) \le c\max(|x_1|,\ldots,|x_k|, t(x_1,\ldots,x_k)),$$

where c is a nonzero constant.

Proof. By constructions of [6, 7], it suffices to restrict attention to multitape machines with one head per tape. We simulate the machine by the well-known technique of replacing a tape by two pushdown stacks. The stack operations can be simulated in $\mathfrak{B}$ as follows:

Push $\alpha \in \{0,1\}$ onto stack x: $x := \alpha\,\text{suc}(x)$,

Test top of stack x against α: $\text{head}(\text{reverse}(x)) = \alpha$,

Pop stack x : $x := \text{reverse}(\text{tail}(\text{reverse}(x)))$.

Since two stacks can simulate a tape with no time loss, these equations indicate that $\mathfrak{B}$ can simulate a Turing machine in linear time □

Theorem 3.3. *$\mathfrak{B}$ is adequate.*

Proof. Immediate from Lemma 3.2 and the fact that the functions of $\mathfrak{B}$ cannot increase lengths of intermediate results very rapidly. □

Corollary 3.4. *Assume $\mathfrak{A}$ with domain* $\{0,1\}^*$ (resp. N) *is adequate, and let τ denote the identity function* (resp. 2adic). *Assume $f:(\{0,1\}^*)^k \to \{0,1\}^*$ is a partial function computed by a Turing machine as above, and $t:(\{0,1\}^*)^k \to N$ is a partial function such that the machine, on input $x_1,\ldots,x_k$, halts within $t(x_1,\ldots,x_k)$ steps. Then there exists flowchart F over $\mathfrak{A}$ and polynomial p satisfying the following.*

(a) *F computes (the composite function) $\tau^{-1} \circ f \circ \tau$,*

(b) $L_F(x_1,\ldots,x_k) \le p\left(\max\limits_{1\le i\le k} |x_i|,\ t(\tau(x_1),\ldots,\tau(x_k))\right)$,

(c) *if y is any value produced during the computation of F on inputs $x_1,\ldots,x_k$, then* $|y| \le p\left(\max\limits_{1\le i\le k} |x_i|,\ t(\tau(x_1),\ldots,\tau(x_k))\right)$.

Proof. If $\mathrm{Dom}_{\mathfrak{A}} = \{0,1\}^*$, then F can be obtained as follows. A flowchart over $\mathfrak{B}$ is obtained as in the proof of Lemma 3.2, and the (polynomial computable) basic operations of $\mathfrak{B}$ are replaced by their flowcharts over $\mathfrak{A}$ guaranteed by the adequacy of $\mathfrak{A}$. The fact that the basic functions of $\mathfrak{B}$ cannot increase lengths of intermediate results very rapidly implies the required bound. If $\mathrm{Dom}_{\mathfrak{A}} = N$, the construction needs to be only slightly modified to handle the isomorphism. □

Next, we turn to adequacy classification for other algebras. Rather than reason directly about Turing machine computation, we use the adequacy of $\mathfrak{B}$ and a "reducibility" to infer the adequacy of other algebras.

Definition. Let $\mathfrak{A}$ and $\mathfrak{A}'$ be algebras with domain N or $\{0,1\}^*$, and with all basic operations total. Let $\tau:\mathrm{Dom}_{\mathfrak{A}'} \to \mathrm{Dom}_{\mathfrak{A}}$ denote the identity function, 2adic or $\mathrm{2adic}^{-1}$ as appropriate. We write $\mathfrak{A} \le^{\mathrm{poly}} \mathfrak{A}'$ provided the following are satisfied.

(a) For each f in $\mathrm{Fun}_{\mathfrak{A}}$ there exist a flowchart F over $\mathfrak{A}'$ computing $\tau^{-1} \circ f \circ \tau$ and a polynomial p such that

(a1) $L_F(x_1,\ldots,x_k) \le p(\max\{|x_i| : 1 \le i \le k\},\ |f(\tau(x_1),\ldots,\tau(x_k))|)$, and

(a2) if y is any value produced during the computation of F on inputs $x_1,\ldots,x_k$, then $|y| \le p(\max\{|x_i| : 1 \le i \le k\}, |f(\tau(x_1,\ldots,\tau(x_k))|)$.

(b) For each r in $\mathrm{Rel}_{\mathfrak{A}}$ there exist a flowchart F over $\mathfrak{A}'$ computing $r \circ \tau$ and a polynomial p such that

(b1) $L_F(x_1,\ldots,x_k) \le p(\max\{|x_i| : 1 \le i \le k\})$, and

(b2) if y is any value produced during the computation of F on inputs $x_1,\ldots,x_k$, then $|y| \le p(\max\{|x_i| : 1 \le i \le k\})$.

(An alternative to this definition would treat functions in the same way as predicates, omitting the term $|f(\tau(x_1),\ldots,\tau(x_k))|$ from (a1) and (a2)). Such a definition would suffice for the results to be proved here. However, the key Lemma 3.5 is true for the generalized definition, and slightly more direct proofs are sometimes possible using the generalization.)

Lemma 3.5. (a) *$\le^{\mathrm{poly}}$ is transitive.*

(b) *If $\mathfrak{A} \le^{\mathrm{poly}} \mathfrak{A}'$ and $\mathfrak{A}$ is adequate, then $\mathfrak{A}'$ is adequate.*

Proof. By flowchart substitution. □

Theorem 3.6. *The following are adequate.*

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

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

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

(d) $\langle\{0,1\}^*;\lambda,0,1,\text{tail},\text{concat};=\rangle$, *and*

(e) $\langle\{0,1\}^*;\lambda,0\text{suc},1\text{suc};\text{prefix}\rangle$.

Note. The reader might find it interesting at this point to compare this result with Theorem 3.7. Between them, the two theorems decide the adequacy of all algebras mentioned in Theorem 3.1.

Proof. Let the five algebras of the theorem be denoted by $\mathcal{A}_a-\mathcal{A}_e$. We use Theorem 3.3 and Lemma 3.5(b), to show (a)–(e) by a chain of reductions. Easily, $\mathcal{B}\leq^{\text{poly}}\langle\{0,1\}^*;\lambda,0,1,\text{head},\text{tail},\text{concat};=\rangle$, and *head* can trivially be reprogrammed in terms of the other primitives, thus showing $\mathcal{A}_d$ to be adequate. Then it is not difficult to show that $\mathcal{A}_d\leq^{\text{poly}}\mathcal{A}_e$, showing $\mathcal{A}_e$ to be adequate.

Next, we show $\mathcal{A}_e\leq^{\text{poly}}\mathcal{A}_a$. Since $2\text{adic}^{-1}(0\text{suc}(x))=2(2\text{adic}^{-1}(x))+1$, the only difficulty is in showing $\langle\{0,1\}^*;;\text{prefix}\rangle\leq^{\text{poly}}\mathcal{A}_a$. We break up this task by showing $\langle\{0,1\}^*;;\text{prefix}\rangle\leq^{\text{poly}}\langle\{0,1\}^*;\lambda,\text{head},\text{tail};=\rangle\leq^{\text{poly}}\mathcal{A}_a$. (Note that the interpolated algebra does not have recursive power.) The first reducibility is easily implemented as follows: to test if x is a prefix of y, compare successively longer prefixes of x with those of y. It remains to show the second reducibility.

First, we show that $\langle\{0,1\}^*;\text{head};\rangle\leq^{\text{poly}}\mathcal{A}_a$. Note that

$$2\text{adic}^{-1}(\text{head}(2\text{adic}(n)))=\begin{cases}0 \text{ if } n=0,\\ 1 \text{ if } 2^{k+1}<n+2\leq 2^{k+1}+2^k & \text{for some } k\geq 0,\\ 2 \text{ if } 2^{k+1}+2^k<n+2\leq 2^{k+2} & \text{for some } k\geq 0.\end{cases}$$

Then it is straightforward to see that $\langle\{0,1\}^*;\text{head};\rangle\leq^{\text{poly}}\langle N;0,1,+,\exp;\leq,=\rangle$, where $exp(n)=2^n$. (Note that the exponential function is used, but only on "small" arguments, thus satisfying the requirement on the size of intermediate values.) Then it is easy to show $\langle N;0,1,+,\exp;\leq,=\rangle\leq^{\text{poly}}\mathcal{A}_a$, implementing *exp* with a flowchart that does repeated doublings.

(This part of the proof is an example of how $\leq^{\text{poly}}$ can be used with intermediate systems having non-polynomial-time primitives, such as *exp*, to get results about systems with polynomial-time primitives. The key is the fact that the non-polynomial primitives are only used on small arguments.)

Next, we show that $\langle\{0,1\}^*;\text{tail};\rangle\leq^{\text{poly}}\mathcal{A}_a$. Note that

$$2\text{adic}^{-1}(\text{tail}(2\text{adic}(n)))=\begin{cases}0 \text{ if } n=0,\\ n-2^k \text{ if } 2^{k+1}<n+2\leq 2^{k+1}+2^k \text{ for some } k\geq 0,\\ n-2^{k+1} \text{ if } 2^{k+1}+2^k<n+2\leq 2^{k+2} \text{ for some } k\geq 0.\end{cases}$$

Thus, $\langle\{0,1\}^*;\text{tail};\rangle\leq^{\text{poly}}\langle N;0,1,+,\dot{-},\exp;\leq,=\rangle$. It remains only to show $\langle N;\dot{-};\rangle\leq^{\text{poly}}\mathcal{A}_a$. The reduction is carried out as follows. The difference $m\dot{-}n$ is accumulated by summing its powers of 2 in decreasing order; each power of 2 is obtained by successive doublings and comparisons.

Thus, $\mathcal{Q}_a$ is adequate. $\mathcal{Q}_b$'s adequacy follows easily from that of $\mathcal{Q}_a$, and $\mathcal{Q}_c$'s adequacy is left to the reader. (A finer version of (c) is shown in Theorem 4.3(e)). □

We turn next to proofs of inadequacy. Of particular interest are (a) and (b) below. Together they combine to give an adequate system, but each separately is not adequate.

Theorem 3.7. *The following are not adequate.*
(a) $\langle N; 0, 1, +; = \rangle$,
(b) $\langle N; 0, \text{suc}; \leq \rangle$,
(c) $\langle \{0,1\}^*; \lambda, 0\text{suc}, 1\text{suc}; = \rangle$, *and*
(d) $\langle \{0,1\}^*; \lambda, 0, 1, \text{head}, \text{concat}; = \rangle$.

Proof. (a) We show that $\leq$ cannot be computed over $\langle N; 0, 1, +; = \rangle$ with polynomial path length, even by an effective scheme. Assume that it can, and E is an effective scheme computing $\leq$, with $L_E(m,m') \leq p(\max(|m|,|m'|))$, p a monotone polynomial. Fix $n \in N$ with $p(|2n|) < n$ (recalling that $|m| = \lfloor \log(m+1) \rfloor$), and consider $A = \{(m,m') : n+1 \leq m \leq 2n \text{ and } 0 \leq m' \leq n\}$, $B = \{(m,m') : (m',m) \in A\}$. We show that some member of A and some member of B must follow the same path in E so that E cannot compute $\leq$.

Every input pair (m,m') causes a path in E to be followed, of length $\leq p(\max(|m|,|m'|))$, and ending with either output *TRUE* or *FALSE* according to whether $m \leq m'$ or $m > m'$. Each branch point in E results from an equality test which can be expressed in the form $am + bm' + c = a'm + b'm' + c'$, for some $a,b,c,a',b',c' \in N$. (The expression for each branch point can be constructed by ignoring the information obtained from tests along the path, and simply looking at uses of assignment and $+$.) Prune E by omitting all such tests (and subsequent "no" subtrees) in which $a = a'$, $b = b'$, and $c = c'$. Remaining is a tree T for which, at every branch point, all inputs (m,m') causing the "yes" branch to be taken lie on one straight line in 2-space.

Now consider the path in T which takes the "no" branch at each choice point. There are $n^2 + n$ pairs in A, at most $n+1$ of which lie on any given straight line. Thus, at most $n+1$ of the pairs in A can follow any particular "yes" branch. But each pair in A must follow a path with length $\leq p(|2n|) < n$, hence with fewer than n branchpoints. Thus some pair in A must follow the "no" branch at every choice point, and this path must terminate. A symmetric argument shows that some pair in B must follow the same path, a contradiction.

(b) Consider any polynomial computable $f: N \to N$. If $\langle N; 0, \text{suc}; \leq \rangle$ is adequate, then there exists an effective scheme E over this system computing f and a polynomial p with $L_E(m) \leq p(|m|)$. Fix n with $p(|n|) < n$. E on input n must halt in $p(|n|)$ steps, and so suc cannot span from 0 to n during this computation. Then if any $m > n$ is used in place of n, E will follow the same path as before, since $\leq$ will be unable to distinguish m from n. But consider how the output of E 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 $f(m) = c$ for all $m \geq n$, or else $f(m) = m + c$ for all $m \geq n$.

(c) Since $\langle \{0,1\}^*; \lambda, 0\mathrm{suc}, 1\mathrm{suc}; = \rangle \leq^{\mathrm{poly}} \langle N; 0, 1, +; = \rangle$, Lemma 3.5(b) suffices.

(d) We show that *tail* cannot be computed over $\langle \{0,1\}^*; \lambda, 0, 1, \mathrm{head}, \mathrm{concat}; = \rangle$ with polynomial path length, even by an effective scheme. Assume that it can, and E is such an effective scheme, with $L_E(x) \leq p(|x|)$, p a polynomial. Fix $n \in N - \{0\}$ with $p(n) + 1 < 2^{n-1}$, and consider $A = \{x \in \{0,1\}^* : |x| = n$ and $\mathrm{head}(x) = 0\}$. We will show that two distinct members of A must follow the same path in E.

Each branch point in E results from an equality test on two formal expressions, each built up from λ's, 0's, 1's and x's using *head* and *concat*. Restrict consideration to inputs $x \in A$. Then we can simplify the expressions using simple reduction rules so that each expression is a concatenation of 0's, 1's and x's.

We wish to show that each equation is satisfied either by *no* $x \in A$, *all* $x \in A$ or *exactly one* $x \in A$. By replacing the formal variable x by $x_1 \ldots x_n$ in an equation, we obtain a new equation $a_1 \ldots a_k = b_1 \ldots b_k$, where each $a_i, b_i \in \{0,1\} \cup \{x_j : 1 \leq j \leq n\}$. We allow each x_j to range over $\{0,1\}$, and show by induction on k (with n fixed) that the equation has either *no* solutions, 2^n solutions or *exactly one* solution. If $k = 0$, the equation has 2^n solutions. Let k be at least 1, and assume the equation has at least one solution. There are four cases:

(a) $a_1 = b_1 = 0$ *or* $a_1 = b_1 = 1$

Then $(x_1, \ldots, x_n)$ is a solution to the given equation if and only if it is a solution to the equation $a_2 \ldots a_k = b_2 \ldots b_k$. The conclusion follows by the inductive hypothesis.

(b) $a_i = b_i = x_i$, $1 \leq i \leq n$

Then $(x_1, \ldots, x_n)$ is a solution to the given equation if and only if it is a solution to the equation $a_{n+1} \ldots a_k = b_{n+1} \ldots b_k$. The conclusion follows by the inductive hypothesis.

(c) $a_i = x_i$ *and* $b_i \in \{0,1\}$, $1 \leq i \leq n$ (*or else* $b_i = x_i$ *and* $a_i \in \{0,1\}$, $1 \leq i \leq n$)

Then there is exactly one solution to the given equation.

(d) $a_i = x_i$, $1 \leq i \leq n$, $b_i \in \{0,1\}$, $1 \leq i \leq m < n$, *and* $b_{m+i} = x_i$, $1 \leq i \leq n$

Then $x_1 \ldots x_n$ is the length n prefix of the periodic string $b_1 \ldots b_m b_1 \ldots b_m \ldots$.

Thus, the induction holds and the equations have the desired solutions. Now prune E by omitting all tests (and subsequent "no" subtrees) for which all $x \in A$ satisfy the reduced question. Remaining is a tree T for which, at each branch point, at most one $x \in A$ causes the "yes" branch to be taken. Consider the path in T which takes the "no" branch at each choice point. There are 2^{n-1} strings in A, at most one of which can cause any particular "yes" branch to be followed. But each string in A must have a computation path with length $\leq p(n)$. Since $p(n) + 1 < 2^{n-1}$, at least two strings, x and y, in A must follow the "no" branch at each choice point, and this branch must terminate in an output statement.

Now consider how the output of T on input x or y is constructed. Reductions as above show that the output on input x is the result of (possible empty) concatenations of 0's, 1's and x's, while the output on input y is the result of the same concatenations with y replacing x. But since $|\mathrm{tail}(x)| < |x|$, x

cannot occur in these concatenations, nor can y. Thus, the output of E is identical for both inputs, which means tail(x)=tail(y), a contradiction. $\square$

Remarks. (a) The properties used to prove Theorem 3.7 are not very restrictive. In particular, since the result is proved for effective schemes, we have not used finiteness of the flowchart; also, in (a), (c) and (d), we have not used the rate (relative to the path length) at which different expressions can be built up from the given input values and functions. It would be anticipated that both of these restrictions would be important in some circumstances. For example, it seems that either or both might be useful for verifying the following:

Conjecture. $\langle N; 0, 1, +, \times; = \rangle$ *is not adequate.*

This conjecture seems to express a fundamental property of the expressiveness of polynomials.

(b) The proof of Theorem 3.7(b) shows that $\langle N; 0, \text{suc}; \leq \rangle$ fails to be adequate in a very strong way—there are *no* nontrivial functions of one variable which can be computed efficiently by this system. More generally, it can be shown that if $f: N^n \to N$ is computable in polynomial path length over $\langle N; 0, \text{suc}; \leq \rangle$, then f can in fact be computed in constant path length; this eliminates most interesting functions.

(c) There is a sense in which, for example, $\leq$ and $+$ do not help each other's computation over $\langle N; 0, \text{suc}; = \rangle$. Namely, $\leq$ can be computed over $\langle N; 0, \text{suc}; = \rangle$ by a flowchart F with $L_F(m,n) \leq c \times 2^{\max(|m|,|n|)}$ for some constant c. However, Theorem 3.7(a) can be sharpened to state that any effective scheme E for $\leq$ over $\langle N; 0, 1, +; = \rangle$ has $L_E(m,n) > d \times 2^{\max(|m|,|n|)}$ for some constant d and infinitely many pairs (m,n). Similarly, $+$ can be computed over $\langle N; 0, \text{suc}; = \rangle$ by a flowchart F with $L_F(m,n) \leq c \times 2^{\min(|m|,|n|)}$ for some constant c, whereas any effective scheme E for $+$ over $\langle N; 0, \text{suc}; \leq \rangle$ has $L_E(m,n) > d \times 2^{\min(|m|,|n|)}$ for some constant d and all (m,n).

As mentioned in the introduction, the results of this paper are an outgrowth of a general development of relative complexity of programming systems. Much work remains to be done in extending the ideas of this section to arbitrary algebras. One goal of the general theory is the development of criteria for comparing sets of basic operations over the same domain, to see which of two such sets is "more efficient" for programming. Another goal is the development of criteria for determining whether flowcharts over an algebra provide a realistic measure of "actual computing time."

There are several difficulties which arise when one attempts to generalize in the most obvious way the ideas in this section. First, there appears to be no natural general notion of "polynomial computability" over an arbitrary domain. Such a definition would most naturally rest on a particular coding of the domain into a basic domain such as N or $\{0,1\}^*$, but for general algebras there is not necessarily a single natural coding. Different codings might make different functions polynomial computable. Second, when polynomials and other closed-form functions are used to summarize complexity information, they must be based on some "size parameter" n. It is not obvious what should be used as a size measure for analysis involving arbitrary algebras. The general questions are not pursued here, rather being deferred to [8] and further papers. Instead, in the remainder of this section, we discuss extension of the present ideas to one other domain closely related to N and $\{0,1\}^*$, namely the set Z of integers.

The domain Z has at least two candidates for standard codings in $\{0,1\}^*$. First, define $\tau:\{0,1\}^*\to Z$ by

$$\tau(0x) = -2\text{adic}^{-1}(x)$$
$$\tau(1x) = +2\text{adic}^{-1}(x).$$

$\tau(0)=\tau(1)=0$, but τ is otherwise one-to-one. Second, define a "pairing function" coding τ' by

$$\tau'(x_1 1\ldots 1x_k 00 y_1 1\ldots 1y_\ell) = 2\text{adic}^{-1}(x_1\ldots x_k) - 2\text{adic}^{-1}(y_1\ldots y_\ell),$$

where $x_i, y_i \in \{0,1\}^*$ for all i. A total function or relation f on Z can be defined to be "τ-polynomial-computable" or "τ'-polynomial-computable" if an appropriate corresponding total function or relation g on $\{0,1\}^*$ is polynomial computable. By "corresponding" we mean, for instance, that the following diagram commutes.

$$\begin{tikzcd} Z \arrow[d, "f"'] & \{0,1\}^* \arrow[l, "\tau"'] \arrow[d, "g"] \\ Z & \{0,1\}^* \arrow[l, "\tau"'] \end{tikzcd}$$

It is straightforward to show that the τ-polynomial computable and τ'-polynomial computable operations are identical. Thus, it may be claimed that there is a natural notion of *polynomial computability* for Z.

In keeping with the previous definition of adequacy, we wish to define adequacy for algebras $\mathcal{A}$ with $\text{Dom}_{\mathcal{A}}=Z$ so that the polynomial computable functions on Z are all computable with polynomial path length flowcharts over $\mathcal{A}$. The problem with this idea is that it is not obvious what the parameter should be on which to base the polynomial path length. Since we rely on the particular codings τ and τ' for the definition of polynomial computability, we will similarly use the codings to obtain a parameter. Somewhat arbitrarily, define $|\tau(x)|=|x|$, so that $|n|$ is approximately the log of the absolute value of n for $n\in Z$.

Definition. An algebra $\mathcal{A}$ with domain Z is *adequate* if for every polynomial computable function or relation f on Z there exist polynomial p and flowchart F over $\mathcal{A}$ satisfying the following.

(a) F computes f,

(b) $L_F(n_1,\ldots,n_k)\le p(\max\{|n_i|:1\le i\le k\})$, and

(c) if m is any value produced during the computation of F on inputs $n_1,\ldots,n_k$, then $|m|\le p(\max\{|n_i|:1\le i\le k\})$.

Similar to Theorems 3.6 and 3.7 we can show:

Theorem 3.8. *The following are adequate*:

(a) $\langle Z;0,1,+,-;\le\rangle$

(b) $\langle Z;0,1,+,-;\text{pos}\rangle$ (where $\text{pos}(n)$ is true iff n is positive).

Proof. (a) Let f be a polynomial computable operation on Z, g a τ-corresponding polynomial computable total operation on $\{0,1\}^*$, and h the 2adic-equivalent (to g) total operation on N. (That is, the following diagram commutes.

$$\begin{tikzcd} Z \arrow[d, "f"'] & \{0,1\}^* \arrow[l, "\tau"'] \arrow[d, "g"] & N \arrow[l, "\text{2adic}"'] \arrow[d, "h"] \\ Z & \{0,1\}^* \arrow[l, "\tau"'] & N) \arrow[l, "\text{2adic}"'] \end{tikzcd}$$

Since h is a polynomial computable operation on N, there is a flowchart F over $\mathfrak{N} = \langle N; 0, 1, +; \leq \rangle$ computing h, with polynomial bounds on L_F and the size of intermediate values. (This is by the adequacy of $\mathfrak{N}$.) By identifying N with Z^+, F can be regarded as a flowchart over $\mathfrak{B} = \langle Z; 0, 1, +; \leq \rangle$.

Now all that is required is a polynomial computable way of translating from $n \in Z$ to an element of $\text{2adic}^{-1}(\tau^{-1}(n))$ and a polynomial computable way of translating from $n \in Z^+ - \{0\}$ to $\tau(\text{2adic}(n))$, both via flowcharts over $\mathfrak{Z}$. For then F can be composed with the translation flowcharts to produce a suitable efficient flowchart for f over $\mathfrak{Z}$.

In order to translate from $n \in Z$ to an element of $\text{2adic}^{-1}(\tau^{-1}(n))$, a flowchart first determines n's sign and its absolute value, abs(n). If n is non-negative (resp. negative), there is a polynomial computable function k (resp. ℓ) from abs(n) to an element of $\text{2adic}^{-1}(\tau^{-1}(n))$. Since k and ℓ can be regarded as functions over N, they have efficient flowcharts over $\mathfrak{N}$ and hence over $\mathfrak{Z}$.

Similarly, in order to translate from $n \in Z^+ - \{0\}$ to $\tau(\text{2adic}(n))$, a flowchart simulates the polynomial computable mappings from n to the sign of $\tau(\text{2adic}(n))$ and to abs($\tau(\text{2adic}(n))$). Since these can be regarded as operations over N, they have efficient flowcharts over $\mathfrak{Z}$ as above. Then the sign and absolute value are combined using operations of $\mathfrak{Z}$.

(b) This follows from (a), a trivial flowchart program of $\leq$, and an easy version of Lemma 3.5 for algebras with domain $\mathfrak{Z}$. □

Theorem 3.9. *The following are not adequate*:

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

(b) $\langle Z; 0, \text{suc}, \text{pred}; \leq \rangle$ (where $\text{pred}(x) = x - 1$),

(c) $\langle Z; 0, 1, +, -; =, |_k \rangle$ for any fixed k (where $|_k$ means divisibility by k).

Proof. Similar to Theorem 3.7. □

Referring back to the conjecture following Theorem 3.7, we similarly conjecture that $\langle Z; 0, 1, +, -, \times; = \rangle$ is not adequate. We can show that the two problems are related in one direction:

Theorem 3.10. *If* $\langle N; 0, 1, +, \times; = \rangle$ *is adequate, then* $\langle Z; 0, 1, +, -, \times; = \rangle$ *is adequate.*

Proof. Assume the hypothesis. By Theorem 3.8(b), it suffices to construct a flowchart F for *pos* over $\langle Z; 0, 1, +, -, \times; = \rangle$ such that L_F and the sizes of intermediate values are polynomial bounded. The construction is similar to that for Theorem 3.8(a).

Define $f:Z\to Z$ by $f(n)=2n^2+n$. ($f(n)$ is a coding of n as a non-negative integer.) Clearly, f has a polynomial bounded flowchart over $\langle Z;0,1,+,-,\times;=\rangle$. Identifying Z^+ with N, we define

$$g:N\to N \text{ by } g(n)=\begin{cases}0 \text{ if } n=f(m) \text{ for some } m>0,\\ 1 \text{ otherwise.}\end{cases}$$

g is a polynomial computable function on N, so (by the hypothesis) it has a polynomial-bounded flowchart F' over $\langle N;0,1,+,\times;=\rangle$. But F' can also be regarded as a flowchart over $\langle Z;0,1,+,\times;=\rangle$. Applying F' to $f(n)$ and testing the answer for equality with 0 completes the construction. □

Questions. Is $\langle Z;0,1,+,-,\times;=\rangle$ adequate?
Does the converse of Theorem 3.10 hold?
Is $\langle Z;0,1,+,-;=,|\rangle$ adequate, where $|$ is the divisibility relation?

4. Finer Classification for Particular Operations

The results of the preceding section allow polynomial variation, but for more useful analysis, a finer classification is appropriate. There are a virtually unlimited number of questions to be answered about relative flowchart complexity of specific operation sets. In this section, we give a sampling of upper and lower bounds on the complexity of operations over adequate algebras.

The given results all use similar proof techniques. In particular, the upper bounds arise from uniform translation into flowcharts of linear recursive schemes, using the following version of a result of [3].

Theorem 4.1. (Chandra). *Let R be a linear recursive scheme, ε any positive real. Then there exist flowchart F and constant c such that for any algebra $\mathfrak{A}$,*
(a) *F computes the same function or relation over $\mathfrak{A}$ as does R and*
(b) $L_F(x_1,\ldots,x_k)\le c(L_R(x_1,\ldots,x_k))^{1+\varepsilon}+c$.

Lemma 4.2. *There exist linear recursive scheme R and constant c such that:*
(a) *R computes $f(n)=n \bmod 2$ over $\langle N;0,1,+;\le\rangle$, and $L_R(n)\le c\log n$,*
(b) *R computes $\dot{-}$ over $\langle N;0,1,+;\le\rangle$ and $L_R(m,n)\le c\log(m\dot{-}n)$,*
(c) *R computes $\times$ over $\langle N;0,1,+;\le\rangle$ and $L_R(m,n)\le c\min(\log m,\log n)$,*
(d) *R computes $f(m,n)=m^n$ over $\langle N;0,1,+,\times;\le\rangle$ and $L_R(m,n)\le c\log n$,*
(e) *R computes $\le$ over $\langle N;0,1,+,||;=\rangle$, and $L_R(m,n)\le c\log m$,*
(f) *R computes reverse over $\langle\{0,1\}^*;\lambda,0\text{suc},1\text{suc};\text{prefix}\rangle$ and $L_R(x)\le c|x|$.*

Proof. Informal explanation is provided, followed by the detailed programs with assertions sufficient for their verification.

(a) The procedure determines recursively whether the current power of 2 (given by n_1) enters into the binary expansion of n_0. It also approximates n_0 by truncating its binary representation after the n_1 position.

```
Parity($n_0$)                                data: $n_1, n_2$
/*Given $n_0 \in N$, Parity($n_0$) returns $n_0$ mod 2.*/
```

```
START
$\langle n_1, n_2\rangle := \text{Approx}(n_0, 1)$;
RETURN $(n_2)$
```

```
Approx($n_0, n_1$)                          data: $n_2, n_3, n_4$
/*Given $n_0 \in N$, $n_1 \in N - \{0\}$, Approx($n_0, n_1$) returns two values:
    (a) The largest $m \le n_0$ such that $m$ is a multiple of $n_1$, and
    (b) for $m = a \times n_1$ as in (a), 0 or 1 if the parity of $a$ is even or odd,
respectively.*/

START
if $n_1 \le n_0$
then begin
    $\langle n_2, n_3\rangle := \text{Approx}(n_0, n_1 + n_1)$;
    $n_4 := n_2 + n_1$;
    if $n_4 \le n_0$ then RETURN $(n_4, 1)$ else RETURN $(n_2, 0)$;
    end
else RETURN (0, 0)
```

(b) This is similar to (a). The procedure recursively approximates $n_0 \dot{-} n_1$ by truncating its binary representation after the position holding the current power of 2, n_2.

```
Minus($n_0, n_1$)                           data: $n_2$
/*Given $n_0, n_1 \in N$, Minus($n_0, n_1$) computes $n_0 \dot{-} n_1$.*/

START
if $n_0 \le n_1$
then RETURN (0)
else begin
    $n_2 := \text{Approx}(n_0, n_1, 1)$;
    RETURN $(n_2)$
    end
```

```
Approx($n_0, n_1, n_2$)                     data: $n_3, n_4$
/*Given $n_0 > n_1 \in N$ and $n_2 \ge 1$, Approx($n_0, n_1, n_2$) returns the largest $m \le n_0$
$- n_1$ such that $m$ is a multiple of $n_2$.*/

START
if $n_1 + n_2 \le n_0$
then begin
    $n_3 := \text{Approx}(n_0, n_1, n_2 + n_2)$;
    $n_4 := n_3 + n_2$;
    if $n_1 + n_4 \le n_0$ then RETURN $(n_4)$ else RETURN $(n_3)$
    end
else RETURN (0)
```

(c) By possibly interchanging inputs, we insure that $n_0 \leq n_1$. The procedure recursively approximates n_0 by truncating its binary representation after the position holding the current power of 2, n_2. It also returns the corresponding approximation to the product $n_0 \times n_1$.

```
Mult($n_0,n_1$)                         data: $n_2$
/*Given $n_0,n_1 \in N$, Mult($n_0,n_1$) returns their product.*/

START
if $n_0 \leq n_1$ then $n_2 := $ Multl($n_0,n_1$) else $n_2 := $ Multl($n_1,n_0$);
RETURN ($n_2$)

Multl($n_0,n_1$)                        data: $n_2,n_3$
/*Given $n_0,n_1 \in N$ with $n_0 \leq n_1$, Multl($n_0,n_1$) returns their product.*/

START
if $n_0 \leq 0$
then RETURN (0)
else begin
    $\langle n_2,n_3 \rangle := $ Approx($n_0,n_1,1,n_1$);
    RETURN ($n_3$)
    end

Approx($n_0,n_1,n_2,n_3$)               data: $n_4,n_5,n_6$
/*Given $n_0 \leq n_1$, with $n_0 \neq 0$, $n_2 \geq 1$ and $n_3 = n_2 \times n_1$, Approx($n_0,n_1,n_2,n_3$)
returns two values.
    (a) the largest $m \leq n_0$ such that $m$ is a multiple of $n_2$, and
    (b) for $m$ as in (a), $m \times n_1$.*/

START
if $n_2 \leq n_0$
then begin
    $\langle n_4,n_5 \rangle := $ Approx($n_0,n_1,n_2+n_2,n_3+n_3$);
    $n_6 := n_4 + n_2$;
    if $n_6 \leq n_0$ then RETURN ($n_6,n_5+n_3$) else RETURN ($n_4,n_5$)
    end
else RETURN (0,0)
```

(d) The program is similar to that in (c) and is left to the reader.

(e) If $n_0 \neq n_1$ and $|n_0| \neq |n_1|$, the inequality test $n_0 \leq n_1$ is reduced to the same inequality test on the much smaller values $|n_0|$ and $|n_1|$. If $n_0 \neq n_1$ and $|n_0| = |n_1| \neq 1$, the procedure asks recursively whether n_0 and n_1 can be separated in length by addition of the same multiple of n_2 (a given power of 2) to each. If so, then the direction of the inequality between n_0 and n_1 is also determined. If not, then two numbers are determined with the same relative values as n_0 and n_1, but such that adding n_2 to either number would increase its length.

```
Lteq($n_0, n_1$)                          Boolean: $b_0$
```

/*Given $n_0, n_1 \in N$, Lteq returns *true* or *false* depending upon whether $n_0 \le n_1$ or not.*/

```
START
if $n_0 = n_1$
then RETURN true
else begin
    if $|n_0| = |n_1|$ then $b_0$: = Compare($n_0, n_1$) else $b_0$: = Lteq($|n_0|, |n_1|$);
    RETURN ($b_0$)
    end
```

```
Compare ($n_0, n_1$)          Data: $n_2, n_3$          Boolean: $b_0, b_1$
```

/*Given n_0, n_1 with $n_0 \ne n_1$, but $|n_0| = |n_1|$, Compare(n_0, n_1) returns *true* or *false* depending upon whether $n_0 \le n_1$, or not.*/

```
START
if $|n_0| = 1$
then if $n_0 = 1$ then RETURN (true) else RETURN (false)
else begin
    $\langle n_2, n_3, b_0, b_1 \rangle$: = Approx($n_0, n_1, 1$);
    RETURN ($b_1$)
    end
```

```
Approx($n_0, n_1, n_2$)          Data: $n_3, n_4$          Boolean: $b_0 b_1$
```

/*Given $n_0, n_1, n_2 \in N$, $n_0 \ne n_1$, $|n_0| = |n_1| \ne 1$, n_2 a power of two, $|n_2| \le |n_0|$, Approx(n_0, n_1, n_2) returns four values:
If $|n_0 + cn_2| = |n_1 + cn_2|$ for all $c \in N$, then $\langle m_0, m_1, \mathit{false}, \mathit{false} \rangle$ are returned, where m_0 and m_1 are the unique values with
$|m_0| = |n_0| = |n_1|$, $m_0 - n_0 = m_1 - n_1 = cn_2$ for some c, and $|m_0 + n_2| = |m_1 + n_2| = 1 + |m_0| = 1 + |m_1|$.
If $|n_0 + cn_2| \ne |n_1 + cn_2|$ for some $c \in N$, then $\langle 0, 0, \mathit{true}, a_0 \rangle$ are returned, where $a_0 = \mathit{true}$ if $n_0 < n_1$, *false* otherwise.*/

```
START
if $|n_0| = |n_2|$
then RETURN ($n_0, n_1$, false, false)
else begin
    $\langle n_3, n_4, b_0, b_1 \rangle$: = Approx($n_0, n_1, n_2 + n_2$);
    if $b_0$ = true
    then RETURN (0, 0, true, $b_1$)
    else case
        if $|n_3 + n_2| = |n_3|$ and $|n_4 + n_2| = |n_4|$
        then RETURN ($n_3 + n_2, n_4 + n_2$, false, false)
        if $|n_3 + n_2| \ne |n_3|$ and $|n_4 + n_2| \ne |n_4|$
        then RETURN ($n_3, n_4$, false, false)
        if $|n_3 + n_2| = |n_3|$ and $|n_4 + n_2| \ne |n_4|$
        then RETURN (0, 0, true, true)
        if $|n_3+n_2| \neq |n_3|$ and $|n_4+n_2| = |n_4|$
        then RETURN $(0, 0, true, false)$
        endcase
    end
```

(f) The construction is straightforward and is left to the reader. (The procedure has inputs x_0 and x_1 a prefix of x_0. It determines recursively the reverse of the corresponding suffix of x_0.) □

Theorem 4.3. *For every positive real ε, there exist flowchart F and constant c such that*:

(a) *F computes $f(n) = n \bmod 2$ over $\langle N; 0, 1, +; \leq \rangle$ and $L_F(n) \leq c(\log n)^{1+\varepsilon}$*,

(b) *F computes $\dot{-}$ over $\langle N; 0, 1, +; \leq \rangle$ and $L_F(m,n) \leq c(\log(m \dot{-} n))^{1+\varepsilon}$*,

(c) *F computes × over $\langle N; 0, 1, +; \leq \rangle$ and $L_F(m,n) \leq c(\min(\log m, \log n))^{1+\varepsilon}$*,

(d) *F computes $f(m,n) = m^n$ over $\langle N; 0, 1, +, \times; \leq \rangle$ and $L_F(m,n) \leq c(\log n)^{1+\varepsilon}$*,

(e) *F computes ≤ over $\langle N; 0, 1, +, ||; = \rangle$, and $L_F(m,n) \leq c(\log m)^{1+\varepsilon}$*,

(f) *F computes reverse over* $\langle \{0,1\}^*; \lambda, 0\mathrm{suc}, 1\mathrm{suc}; \mathrm{prefix} \rangle$ and $L_F(x) \leq c|x|^{1+\varepsilon}$

Proof. By Theorem 4.1 and Lemma 4.2. □

It is possible use the translation from (loop-free) linear recursive schemes into flowcharts to obtain results about programs involving loops. Translation is performed on a top-down or bottom-up module of a scheme rather than on the full scheme. As an illustration, we give a lemma and a theorem applying the lemma to two problems, to a bottom-up module in each case. For an example of such a translation on a top-down module, the reader is referred to Lemma 4.4 of [1].

For $n_0, n_1 \in N$, define *power*(n_0, n_1) = if $n_0 > n_1$, then $2^{\lfloor \log_2(n_0 - n_1) \rfloor}$ else 0. (That is, power(n_0, n_1) is the largest power of 2 not greater than $n_0 - n_1$.)

Lemma 4.4. *There exist linear recursive scheme R and constant c such that R computes power over $\langle N; 0, 1, +, \times; \leq \rangle$ and $L_R(n_0, n_1) \leq c \log\log(n_0 \dot{-} n_1)$.*

Proof. The procedure is given inputs n_0, n_1 and n_2 (a number of the form 2^{2^a}). It recursively approximates the largest power of 2 not greater than $n_0 - n_1$, by truncating the binary representation of the *exponent* after the 2^a position. Note that multiplication is used to effect addition of exponents.

```
Power($n_0, n_1$)                        Data: $n_2$

START
if $n_0 \leq n_1$
then RETURN (0)
else begin
    $n_2$: = Approx($n_0, n_1$, 2);
    RETURN ($n_2$)
    end
```

```
Approx($n_0,n_1,n_2$)                    Data: $n_3$
/*Given $n_0 \geq n_1+1$, $n_2=2^{2^a}$ for some $a \geq 0$, Approx($n_0,n_1,n_2$) returns the
largest $m \leq n_0-n_1$ such that $m=2^{b\times 2^a}$ for some $b \in N$.*/

START
if $n_2+n_1 \leq n_0$
then begin
    $n_3$: = Approx($n_0,n_1,n_2 \times n_2$);
    if $n_3 \times n_2+n_1 \leq n_0$ then RETURN ($n_3 \times n_2$) else RETURN ($n_3$)
    end
else RETURN (1)
```
□

Theorem 4.5. *For every positive real ε, there exist flowchart F and constant c such that*

(a) *F computes* $f(n)=n \bmod 2$ *over* $\langle N;0,1,+,\times;\leq\rangle$ *and* $L_F(n) \leq c\log n(\log\log n)^{1+\varepsilon}$,

(b) *F computes* $\dot{-}$ *over* $\langle N;0,1,+,\times;\leq\rangle$ *and* $L_F(m,n) \leq c\log(m \dot{-} n)(\log\log(m \dot{-} n))^{1+\varepsilon}$.

Proof. (a) By Lemma 4.4 and Theorem 4.1, we obtain flowchart F' for *power* over $\langle N;0,1,+,\times;\leq\rangle$ with $L_{F'}(m,n) \leq c(\log\log(m \dot{-} n))^{1+\varepsilon}$ for some constant c. A flowchart F'' for $f(n)=n \bmod 2$ over $\langle N;0,1,+,\text{power};\leq\rangle$ is easily constructed with $L_{F''}(n) \leq c\log n$ for some constant c. Moreover, for all intermediate values m produced during the computation of F'' on n we have $m \leq n$. Substitution of F' for *power* in F'' yields the required bound.

(b) $\dot{-}$ may be computed over $\langle N;0,1,+,\text{power};\leq\rangle$ by flowchart F'' with $L_{F''}(m,n) \leq c\log(m \dot{-} n)$ for some c, and with all pairs of arguments to *power* having their difference at most $m \dot{-} n$. Substitution of F' in F'' yields the needed result. □

Remark. (a) represents a small improvement over the claimed upper bound on parity in [9].

Finally, we ask whether the bounds in Theorems 4.3 and 4.5 are optimal. In all cases, we are able to prove lower bounds which are reasonably close to the given upper bounds, but which do not include the ε in the exponent. The lower bounds are proved for effective schemes, and therefore apply to flowcharts as well.

Lemma 4.6. *There exists a positive constant c such that*

(a) *if effective scheme E computes* $f(n)=n \bmod 2$ *over* $\langle N;0,1,+;\leq\rangle$ *then* $L_E(n) \geq c\log n$ *for all n,*

(b) *if effective scheme E computes* $\dot{-}$ *over* $\langle N;0,1,+;\leq\rangle$ *then* $L_E(m,n) \geq c\log(m \dot{-} n)$ *for all n and (for each n) for all but finitely many m,*

(c) *if effective scheme E computes* $\times$ *over* $\langle N;0,1,+;\leq\rangle$, *then* $L_E(m,n) \geq c(\min(\log m,\log n))$ *for all m and n,*

(d) *if effective scheme E computes* $f(m,n)=m^n$ *over* $\langle N;0,1,+;\leq\rangle$, *then* $L_E(m,n) \geq c\log n$ *for all* $m \geq 2$ *and all n,*

(e) *if effective scheme E computes $\leq$ over $\langle N;0,1,+,||;=\rangle$, then $L_E(m,n) \geq c\log m$ for infinitely many m and n,*

(f) *if effective scheme E computes* reverse *over $\langle\{0,1\}^*;\lambda,0\text{suc},1\text{suc};\text{prefix}\rangle$, then $L_E(x)\geq c|x|$ for all x,*

(g) *if effective scheme E computes $f(n)=n \bmod 2$ over $\langle N;0,1,+,\times;\leq\rangle$, then $L_E(n)\geq c\log n$ for infinitely many n,*

(h) *if effective scheme E computes $\dot{-}$ over $\langle N;0,1,+,\times;\leq\rangle$, then $L_E(m,n) \geq c\log(m\dot{-}n)$ for infinitely many pairs (m,n).*

Proof. (a) Consider any (sufficiently large) n with $L_E(n)<\frac{1}{2}\log n$. Each branch point on the path for n results from an inequality test which may be expressed in the form $an+b\leq cn+d$ for some $a,b,c,d\in N$. Further, the numerical values of a, b, c and d are all at most approximately $\sqrt{n}$, because of the bound on $L_E(n)$. If $a=c$, the answer to the test is independent of n. If $a\neq c$, the answer depends only on the relative sizes of a and c, by the restriction on size of the coefficients and the fact that n is large. Thus, any $m\geq n$ will follow the same path in the scheme and will yield the same answer, a contradiction.

(b) Fix any n and assume that m is very large relative to n, and $L_E(m,n) <\frac{1}{2}\log(m\dot{-}n)$. Each branch point on the path for m and n arises from an inequality test of the form $am+bn+c\leq dm+en+f$, where a,b,c,d,e,f have numerical values $\leq\sqrt{m-n}$. If $a=d$, the answer to the test is independent of m. If $a\neq d$, the answer is determined by which of a and d is larger (since m is large relative to n). Thus, any (p,n) for $p\geq m$ will follow the same path in the scheme, and thus generate the same expression as output. But then the output for (m,n) is of the form $am+bn+c$, while the output for (p,n) is $ap+bn+c$. But $m-n=am+bn+c$ and $p-n=ap+bn+c$ together imply $m=p$.

(c) That number of steps is required just to generate the answer.

(d) As for (c).

(e) Consider any sufficiently large k, and $n=2^k-1$ (so $\text{2adic}(n)=\underbrace{0\ldots0}_{k}$)

Let $A=\{(m,m'):n<m\leq n+|n|, n+|n|<m'\leq n+2|n|\}$ and let $B=\{(m,m'):(m',m)\in A\}$. We show that if path length is too small, E cannot distinguish some pairs in A from some pairs in B. The argument is similar in outline to that used for Theorem 3.7(a).

Tests are equations between expressions involving m, m', 0, 1, + and $||$. We assume that $L_E(m,m')\leq e(\log m)$ for a sufficiently small positive constant e, and show that for any $p\leq L_E(m,m')$, an expression appearing p steps from the beginning on any path can be simplified to be of the form $am+bm'+c|n|+d$, where $a,b,c,d\leq 2^p$.

Proof of this simplification is by induction. The basis is easy. If an expression appears p steps from the beginning and is formed by + applied to two expressions each appearing within $p-1$ steps of the beginning then the result is clear. If the expression is formed by applying $||$ to a previous expression, some more work is needed.

Consider the new expression $|am+bm'+c|n|+d|$, where $a,b,c,d\leq 2^{p-1}$ and $n<m,m'\leq n+2|n|$. It is easy to see that $|(a+b)(n+1)|\leq|am+bm'+c|n|+d|\leq |(a+b)n+5\times 2^{p-1}|n|+2^{p-1}|$. But it can be shown that $|(a+b)(n+1)|=|(a+b)n +5\times 2^{p-1}|n|+2^{p-1}|=\lfloor\log(a+b)\rfloor+|n|$. Thus, the new expression can be rewritten as $\lfloor\log(a+b)\rfloor+|n|$. Since the first term is a constant, this expression is in the required form.

Thus, each test can be rewritten in the form $am+bm'+c|n|+d=0$, or equivalently $am+bm'+\lfloor c\log m\rfloor+d=0$. If $b\neq 0$, each m determines at most one m' which can satisfy this equation. If $b=0$, then at most one m in the given range can be a solution. Thus, at most $|n|$ solutions $(m,m')\in A$ exist for each test. By the bounds on path length and on m, there is a pair $(m,m')\in A$ which must answer "no" at each branch point; a similar argument holds for B, a contradiction.

(f) We leave this one for the reader.

(g) The proof of Theorem 1 in [9] suffices. In outline, an upper bound on the number of steps executed yields corresponding bounds on both the number of distinct polynomials which appear in relation boxes and on their degrees. These bounds in turn yield an upper bound on the number of solutions. But a parity program requires "separation" of consecutive pairs of integers, thus requiring many solutions. Careful comparison of these two bounds yields the inequality.

(h) An argument similar to that used for Theorem 1 of [9] suffices, where we consider input pairs of the form $(n,1)$. □

5. Suggestions for Further Research

There are, of course, many other operations over N and $\{0,1\}^*$ whose complexity might be determined with respect to various adequate algebras. In addition, similar questions can be asked for operations over other domains arising in computing and mathematics. General methods for defining polynomial computability and selecting parameters, as discussed near the end of Section III, should be developed in order that such problems might be treated consistently.

In none of the present work has any distinction been made between relative complexity of operations obtained via flowcharts, linear recursive schemes or effective schemes; the latter two scheme classes have been used only to obtain results about flowcharts. It might be interesting to compare the efficiency of the different scheme classes over various natural algebras. For instance, in Section IV, the upper and lower bounds differ by (at least) the ε in the exponent. This ε seems to represent a real difference between flowcharts and linear recursive schemes. It would be interesting to prove a lower bound, say based on one of the natural problems in Section IV, which is sensitive to the ε. For instance, one might be able to prove the following.

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

We emphasize that we seek *natural* examples of a complexity difference between scheme classes. It is not difficult to construct (probably uninteresting) free algebras for which a complexity difference between flowcharts and linear recursive schemes is provable, but it is unclear which insights such constructions yield. The specific operation sets chosen seem to play a crucial role in determining the comparative efficiency of scheme classes, but a thorough examination of this role remains to be done.

Acknowledgments

The authors would like to thank Arnold Rosenberg for his careful reading of the original manuscript and many helpful suggestions for its revision. This paper and the two others [1, 10] which evolved from the lengthy manuscript are much improved by his contributions.

References

1. N. A. Lynch and E. K. Blum, Relative Complexity of Algebras, (to be published).
2. N. A. Lynch and E. K. Blum, Efficient Reducibility Between Programming Systems, *Proceedings of Ninth Annual Symposium on Theory of Computation*, ACM, Boulder, 228–238 (1977).
3. A. Chandra, Efficient Compilation of Linear Recursive Programs, *14th Annual Symposium on Switching and Automata Theory*, October 15–17, 16–25 (1973).
4. D. Kfoury, Comparing Algebraic Structures up to Algorithmic Equivalence in Automata, *Languages and Programming*, Ed. M. Nivat, North-Holland/ Elsevier, 253–263 (1973). See also Translatability of Schemes over Restricted Interpretations, *J. Comp. and Syst. Sciences* 8, 387–408 (1974).
5. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1976.
6. P. Fischer, A. Meyer, and A. Rosenberg, Real-time Simulation of Multihead Tape Units, *JACM* 19, 590–607, (1972).
7. B. Leong and J. Seiferas, New Real-time Simulations of Multilevel Tape Units. *Ninth Annual Symposium on Theory of Computation*, ACM, Boulder, 239–247 (1977).
8. N. A. Lynch, Straight-Line Program Length as a Parameter for Complexity Measures, *Theoretical Computer Science*, (to appear). Also see *Proceedings of Tenth Annual ACM Symposium on Theory of Computing*, San Diego, 150–161, (1978).
9. L. J. Stockmeyer, Arithmetic Versus Boolean Operations in Idealized Register Machines, *IBM RC 5954*, (A 25 837).
10. N. A. Lynch and E. K. Blum, A Difference in Expressive Power Between Flowcharts and Recursion Schemes, *Mathematics Systems Theory* 12, 205–211 (1979).
11. G. Grätzer, *Universal Algebra*, Van Nostrand, Princeton, N.J., 1968.
12. P. M. Cohn, *Universal Algebra*, Harper and Row, N.Y., 1965.
13. A. I. Malcev, *Algebraic Systems*, Springer-Verlag, N.Y., 1973.

Received February 9, 1978, in revised form October 23, 1978 and June 27, 1979 and in final form August 15, 1979.