Math. Systems Theory 14, 193–214 (1981)

Mathematical
Systems Theory

Relative Complexity of Algebras

Nancy A. Lynch[1,2] and Edward K. Blum [1]

School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia

Department of Mathematics, University of Southern California, Los Angeles, California

Abstract. A simple algebraic model is proposed for measuring the relative complexity of programming systems. The appropriateness of this model is illustrated by its use as a framework for the statement and proof of results dealing with coding-independent limitations on the relative complexity of basic algebras.

I. Introduction

Much of the recent 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 still 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 quantitative questions as what effect certain operations have on the lengths of computations and what effect different codings have. These are questions of "semantic complexity." To treat or even formulate such questions, it seems useful 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 sophisticated languages such as ALGOL 68 and simpler languages such as BASIC. Recent research [1–10] suggests that this framework be primarily algebraic. It appears, however, that

[1]This research was partially supported by NSF Grant DCR75-02373 and MCS78-07461.
[2]This research was partially supported by NSF Grant MCS77-15628.

0025/5661/81/0014-0193$04.40

classical algebraic notions such as the homomorphism concept, arising from studies of the properties of similar algebraic structures, are inadequate to the tasks of computer science. Rather, more general notions of representation or simulation seem to be required.

In this paper, we define a framework for both qualitative and quantitative analysis of semantic problems, and give examples of its use. Although various programming languages can be considered, we focus here (for definiteness) on a flowchart language. We concentrate on questions involving the relative computability and the relative complexity of programming systems. Although intended principally to illustrate the model, the results should have intrinsic interest and suggest directions for further research.

A basic premise of our work is that analysis of the complexity of algorithms should be performed using a more "relative" or "modular" approach than is usual, in accordance with recent research in semantics, program verification and programming methodology. Complexity analyses have generally been done in an "absolute" way, by selecting a standard model of computation (a RAM with a specified operation set, or perhaps a Turing machine) and determining the total time or space required by the algorithms when executed on this model. One difficulty with this approach is that it tends to deemphasize certain similarities between problems. It has been noticed [11] that the underlying algebra for discrete algorithms 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 the non-negative integers N, in terms of basic bit string operations, or in terms of basic bit operations. This situation suggests that a computation model based on a single standard algebra is not to be sought; rather, in many cases, the interesting ideas seem to be inherently relative ones. Formal models for measuring relative complexity are needed; they should have the property that relative complexity measures calculated thereby can be combined in a systematic way to produce an "absolute" complexity measure of an algorithm.

Relative complexity is, of course, not a new idea; one form in which it has been studied is represented by [12–14], for example. This work uses Turing machines with oracles as a model for computation. For questions about computability, or about time complexity at the level of functions growing at least as fast as general polynomials, this model appears to be adequate. But for very slow-growing time bounds, the peculiarities of Turing machines become intermingled with the properties of the oracle set and of the problem under consideration, in determination of relative complexity. We take the viewpoint that the basic operations of the Turing machine are no different from oracle sets; both are here thought of as basic operations of an algebra.

There are two different kinds of modularity we wish to consider. The first is the definition of a new operation from previously defined operations on a previously defined data type. For example, given bit strings with some suitable set of "unit-cost" operations, how should concatenation be "implemented"? The second is the "implementation" of an entirely new data type (including its operations) relative to a previously defined data type. For example, given bit strings and some standard set of operations, how should the rational numbers with an appropriate set of operations be "implemented"? "Implementation" of

an entire algebra at once rather than of one function at a time is suggested by the work on data structures in [2, 9, 10]; another important motivation arises from coding considerations. Consider a programming system based on bit strings, with some natural set of operations, within which we wish to determine the "complexity of primeness" for the natural numbers, N. There is no a priori reason we could not assume a coding of N into bit strings which included primeness in a trivially accessible way. Thus, the problem has not been clearly formulated. Difficulties of this kind are generally resolved by specification of a particular coding, but some meaningful results should be obtainable without such a specific solution. We regard "primeness" as existing not in a vacuum but along with and relative to other operations on N (such as $+$ and $\leqslant$). By imposing complexity upper bounds on the other operations, we restrict the allowed codings sufficiently so that the relative complexity of primeness makes coding-independent sense. We then find that there are coding-independent tradeoffs in complexity for different operations of an algebra.

The basic algebraic framework for this relativistic approach to complexity is given in Section II of this paper. We introduce the notion of "simulators" in one algebra $\mathfrak{A}'$ for the operations of another algebra $\mathfrak{A}$. To be useful in measuring complexity, the simulators must be expressed in terms of the basic operations of $\mathfrak{A}'$. Although any scheme class could be used to define the simulators, we focus mainly on flowchart schemes. Some basic results and examples are given.

Section III contains a comparison of relative computability of algebras under different codings and with different scheme classes. We establish sufficient conditions on algebras and codings for the different computability definitions to be equivalent.

In Section IV, we consider relative complexity of algebras under different codings but using flowchart schemes. We quantify the restrictions imposed on codings by requiring that they permit certain operations to be simulated quickly. Thus, we consider a particular pair of domains, N and bit-strings, and show that there are well-defined ways in which a standard coding of the natural numbers into bit strings is close to optimal. In intuitive terms, we show that every operation on N which can be simulated quickly in any alternative encoding that can quickly simulate $+$ and $\leqslant$ can be simulated quickly in the standard coding as well. Although this intuitive statement seems easy to understand, it is not clear how the complexity bounds should best be expressed. The statement is formalized in several ways. Definitional issues are important here, for example, in the determination of appropriate parameters (in N) upon which to base complexity measures.

Section V contains suggested directions for further research.

There are several papers by the authors related to the present one. [15] contains a study of a generalization of the homomorphism concept called the "genomorphism." This generalization appears to be sufficiently powerful to yield some interesting results about representation and simulation. The definitional issues mentioned in Section IV form the basis for a study of size parameters in general algebras in [16, 17]. The general definition for size measure suggested in this paper is used there to measure many different types of complexity, and applications to complexity-bounded group theory and to fundamental lower bounds on relative complexity of basic algebras are given. In [18]

we presented a result, related to those studied in Section III, generalizing the Paterson–Hewitt result on recursion schemes. In [19], we study the efficiency of specific sets of basic operations over N and $\{0, 1\}^*$. Since the results in [19] hypothesize specific codings (and deal with the first of the two kinds of modularity we treat), they do not require the generality of the formalism of the present paper. A preliminary version of the results of the present paper, together with those of [18, 19], appears in [20]. Finally, work remains to be done in extending the present model to handle complexity of data structure algebras as studied in [2, 9, 10]. One such result in [37] gives a tradeoff lower bound on the time required for insertion and searching in a data base allowing only comparisons on its data items.

Several recent research efforts of others which use ideas similar to those in this paper suggest that such representation and simulation ideas are fundamental. [10] contains definitions for simulation which are close to the genomorphism definitions. The data space model of [21] includes related and very general simulation definitions of a type which may prove useful in handling data structure algebras. Results in [22] and [34] on graph embedding and in [23] on security involve representation-independent complexity in frameworks similar to ours.

II. Notation, Definitions and Basic Results on Simulation

An *algebra* $\mathcal{A}=\langle \mathrm{Dom}_{\mathcal{A}}; \mathrm{Fun}_{\mathcal{A}}; \mathrm{Rel}_{\mathcal{A}} \rangle$ is a set $\mathrm{Dom}_{\mathcal{A}}$ (the *domain* of $\mathcal{A}$) together with a finite collection, $\mathrm{Fun}_{\mathcal{A}}$, of partial functions (more often called "operations" in the algebraic literature) and a finite collection, $\mathrm{Rel}_{\mathcal{A}}$, of partial relations on $\mathrm{Dom}_{\mathcal{A}}$. Constants are 0-ary functions. The members of $\mathrm{Fun}_{\mathcal{A}}$ and $\mathrm{Rel}_{\mathcal{A}}$ are called *basic* functions and relations of $\mathcal{A}$.

Let $\tau: A' \to A$ be a partial, surjective function, where A' and A are arbitrary sets. Let f be a partial function on A and f' a partial function on A'. Then f' is a *τ-simulator* of f provided that for all $x_1, \ldots, x_n$ in A', if $f(\tau(x_1), \ldots, \tau(x_n))$ is defined, then so is $\tau(f'(x_1, \ldots, x_n))$ and their values are equal. Similarly, let r be a partial relation on A and r' a partial relation on A'. Then r' is a *τ-simulator* of r provided that if $r(\tau(x_1), \ldots, \tau(x_n))$ is defined, then so is $r'(x_1, \ldots, x_n)$, and their truth values are equal.

Note that several "representations" in A' are permitted for each element of A, and that not every element of A' need "represent" an element of A. Also note that the definition is slightly different for functions and for relations. This difference is intended to reflect the different ways in which functions and relations are used in programs. We think of A as the "represented" (or implemented) algebra and A' as the "representing" (or implementing) algebra.

To express the relative computability and relative complexity of two arbitrary algebras $\mathcal{A}$ and $\mathcal{A}'$, we consider a translation mapping $\tau: \mathrm{Dom}_{\mathcal{A}'} \to \mathrm{Dom}_{\mathcal{A}}$, which must be onto $\mathrm{Dom}_{\mathcal{A}}$. We wish to examine the computability and complexity of τ-simulators of the basic functions (including constants) and relations of $\mathcal{A}$. To relate $\mathcal{A}$ to $\mathcal{A}'$, we must relate these simulators to the basic functions and relations of $\mathcal{A}'$. This we do by specifying that the simulators be computable by programs in some language over the basic functions and relations of $\mathcal{A}'$. In this

paper, we focus on a simple flowchart programming language, but two other languages are also used to help prove results about the flowchart language. We shall not take time to give the full syntax and semantics of these languages. We assume, as usual, that symbols have been chosen for the elements, functions and relations of A. In our metalanguage, we use the same notation for these symbols and the objects they denote, relying on the reader's good will to make the distinction from the context.

For an algebra $\mathfrak{A}$, a *flowchart* over $\mathfrak{A}$ is composed in the usual way from a finite number of "boxes" of the types:

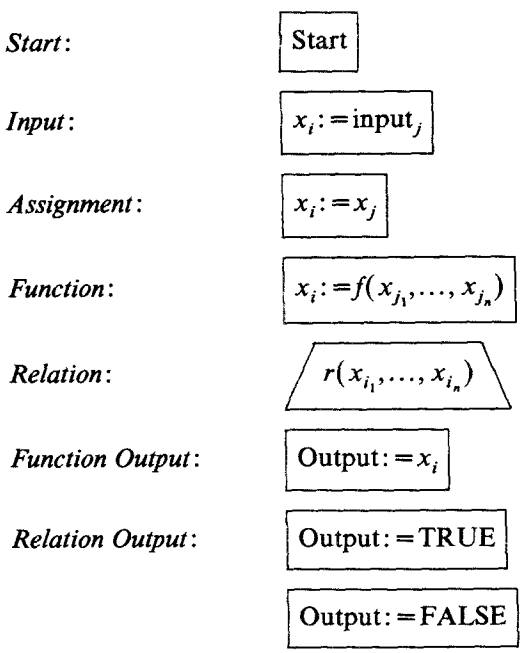

Fig. 1.

where $f \in \text{Fun}_{\mathfrak{A}}$, $r \in \text{Rel}_{\mathfrak{A}}$, the x's are variables and inputs are elements of $\text{Dom}_{\mathfrak{A}}$. Output boxes have no successors, relation boxes have two successors, and all others have one successor. There is exactly one start box. 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.

For an algebra $\mathfrak{A}$, a *linear recursive scheme* over $\mathfrak{A}$ is defined as in [24], with one modification. As in [24], a finite collection of procedure definitions is given, in which each procedure can call at most one other procedure. The language has a fairly general instruction set, with conditionals, typed variables (including Boolean variables with a fixed interpretation), vectors of parameters for procedures, but no looping other than by the use of recursion. The basic function and relation symbols used are those in $\text{Fun}_{\mathfrak{A}}$ and $\text{Rel}_{\mathfrak{A}}$. In contrast to Chandra's definition, however, 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* over $\mathcal{A}$ is defined as in [25]. Thus, an effective scheme is composed of boxes of the same types as used for flowcharts, but the number of boxes need not be finite; i.e. they can form an infinite binary tree. However, we require that the nodes of the tree, suitably coded, be generable in a recursively enumerable way; i.e. as the range of a total recursive function. Again, we consider *function schemes* and *relation schemes*.

A flowchart, linear recursive scheme or effective scheme P defines, in a natural way, a partial function fn_P or a partial relation rel_P. Semantics are assumed to be clear.

(Note that there are two different ways in which a syntactically correct interpreted flowchart, linear recursive scheme or effective scheme, acting on a particular vector of inputs, may fail to give an answer. The first is by entering a computational loop, and the second is by using one of the partial basic operations at a vector of arguments not in the domain of the operation. No distinction is made between the treatments of these two cases.)

Using these three kinds of "programs," we can present definitions for the relative computability of two algebras, $\mathcal{A}$ and $\mathcal{A}'$.

A mapping from one set of programs or functions to another is *arity-preserving* providing the image of each program or function has the same number of arguments as the program or function.

We write $\mathcal{A} \underset{\tau,\mathcal{P}}{\leqslant} \mathcal{A}'$ (resp. $\mathcal{A} \underset{\tau,\mathcal{P}}{\leqslant}^{\text{lin}} \mathcal{A}', \mathcal{A} \underset{\tau,\mathcal{P}}{\leqslant}^{\text{eff}} \mathcal{A}'$) if $\tau: \text{Dom}_{\mathcal{A}'} \to \text{Dom}_{\mathcal{A}}$ is a partial, surjective map, $\mathcal{P}$ is a total, arity-preserving mapping from $\text{Fun}_{\mathcal{A}} \cup \text{Rel}_{\mathcal{A}}$ to the set of flowcharts (resp. linear recursive schemes, effective schemes), $fn_{\mathcal{P}(f)}$ is a τ-simulator of f for each f in $\text{Fun}_{\mathcal{A}}$, and $\text{rel}_{\mathcal{P}(r)}$ is a τ-simulator of r for each r in $\text{Rel}_{\mathcal{A}}$. (In particular, for each constant c in $\text{Fun}_{\mathcal{A}}$, $\mathcal{P}$ selects a 0-input flowchart generating an element of $\tau^{-1}(c)$.) When $\mathcal{P}$ is irrelevant, we write $\mathcal{A} \underset{\tau}{\leqslant} \mathcal{A}'$ (resp. $\mathcal{A} \underset{\tau}{\leqslant}^{\text{lin}} \mathcal{A}', \mathcal{A} \underset{\tau}{\leqslant}^{\text{eff}} \mathcal{A}'$) for $(\exists \mathcal{P})\,[\mathcal{A} \underset{\tau,\mathcal{P}}{\leqslant} \mathcal{A}']$ (resp. $(\exists \mathcal{P})[\mathcal{A} \underset{\tau,\mathcal{P}}{\leqslant}^{\text{lin}} \mathcal{A}'], (\exists \mathcal{P})[\mathcal{A} \overset{\tau}{\underset{\tau,\mathcal{P}}{\leqslant}}^{\text{eff}} \mathcal{A}']$). We write $\mathcal{A} \underset{\tau}{\leqslant} \mathcal{A}'$ (resp. $\mathcal{A} \underset{\tau}{\leqslant}^{\text{lin}} \mathcal{A}', \mathcal{A} \underset{\tau}{\leqslant}^{\text{eff}} \mathcal{A}'$) for $(\exists \tau)[\mathcal{A} \underset{\tau}{\leqslant} \mathcal{A}']$ (resp. $(\exists \tau)[\mathcal{A} \underset{\tau}{\leqslant}^{\text{lin}} \mathcal{A}']$, $(\exists \tau)[\mathcal{A} \underset{\tau}{\leqslant}^{\text{eff}} \mathcal{A}']$). Clearly, $\mathcal{A} \underset{\tau}{\leqslant} \mathcal{A}'$ implies $\mathcal{A} \overset{\tau}{\underset{\tau}{\leqslant}}^{\text{eff}} \mathcal{A}'$, and $\mathcal{A} \underset{\tau}{\leqslant}^{\text{lin}} \mathcal{A}'$ implies $\mathcal{A} \overset{\tau}{\underset{\tau}{\leqslant}}^{\text{eff}} \mathcal{A}'$. A construction in [26] shows that $\mathcal{A} \overset{\tau}{\underset{\tau}{\leqslant}}^{\text{lin}} \mathcal{A}'$ implies $\mathcal{A} \overset{\tau}{\underset{\tau}{\leqslant}} \mathcal{A}'$

Theorem 2.1. *If* $\mathcal{A} \underset{\tau}{\leqslant} \mathcal{A}'$ *and* $\mathcal{A}' \underset{\tau'}{\leqslant} \mathcal{A}''$, *then* $\mathcal{A} \underset{\tau\circ\tau'}{\leqslant} \mathcal{A}''$. *If* $\mathcal{A} \underset{\tau}{\leqslant}^{\text{eff}} \mathcal{A}'$ *and* $\mathcal{A}' \underset{\tau'}{\leqslant}^{\text{eff}} \mathcal{A}''$, *then* $\mathcal{A} \underset{\tau\circ\tau'}{\leqslant}^{\text{eff}} \mathcal{A}''$.

Proof. By composition closure of flowcharts and effective schemes. □

One consequence of allowing multiple representations for each element of the represented algebra is that the quotient algebras are representable by their original algebras. We recall [35, 36] the notion of congruence, generalizing it somewhat. If $\mathcal{A}$ is an algebra, then an equivalence relation, R, on $\text{Dom}_{\mathcal{A}}$ is a *congruence on* $\mathcal{A}$ if

(a) for every $f \in \text{Fun}_{\mathcal{A}}$, if $x_i R y_i$, $1 \leqslant i \leqslant n$, then either $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n)$ are both undefined, or $f(x_1, \ldots, x_n) R f(y_1, \ldots, y_n)$, and

(b) for every $r \in \mathrm{Rel}_{\mathscr{Q}}$, if $x_i R y_i$, $1 \leqslant i \leqslant n$, then either $r(x_1, \ldots, x_n)$ and $r(y_1, \ldots, y_n)$ are both undefined or $r(x_1, \ldots, x_n) = r(y_1, \ldots, y_n)$.

Note the insistence that the function or relation values be either both defined or both undefined. In a congruence, equivalent elements are indistinguishable within the given algebra $\mathscr{Q}$. If R is a congruence on $\mathscr{Q}$, we define $\mathscr{Q}/R$ to be the quotient algebra in the usual way [35, 36]. The *natural* mapping $\tau: \mathscr{Q} \to \mathscr{Q}/R$ maps an element x onto its equivalence class $[x]$.

Theorem 2.2. $\mathscr{Q}/R \underset{\tau}{\leqslant} \mathscr{Q}$, *where* τ *is the natural mapping.*

Proof. Flowcharts consisting of a module computing a single basic operation suffice. □

Example 2.1. Let $\mathscr{Q}_b = \langle \{0, 1\}^*; +, \times; \equiv \rangle$ where the elements are regarded as binary representations of elements of N, possibly with leading zeros. $\equiv$ is defined to be true if the same natural number is represented by the two strings, possibly with different numbers of leading zeros. $+$ and $\times$ are defined arbitrarily (as regards leading zeros) as long as the members of N represented by the answers are the correct sums and products. Clearly, $\equiv$ is a congruence, so that $\mathscr{Q}_b/\equiv \underset{\tau}{\leqslant} \mathscr{Q}_b$ for the natural mapping τ. By transitivity, an "implementation" of $\mathscr{Q}_b$ relative to any other algebra may be thought of as leading directly to an "implementation" of $\mathscr{Q}_b/\equiv$, which is an algebra isomorphic to $\langle N; +, \times; = \rangle$.

If Fun is any set of function symbols including at least one 0-ary function, then *Exp(Fun)* denotes the set of all well-formed expressions over the symbols in Fun.

For an algebra $\mathscr{Q}$, $e \in \mathrm{Exp}(\mathrm{Fun}_{\mathscr{Q}})$, let *val(e)* be the "value" of e when evaluated in $\mathscr{Q}$ by applying the functions to the denoted elements that occur in e (val(e) may be undefined). We write well-formed expressions in infix operator form. Let $\mathsf{Free}(\mathscr{Q})$ be the free algebra of expressions having as its domain $\mathrm{Dom}_{\mathsf{Free}(\mathscr{Q})}$ the set of expressions e for which val(e) is defined. Its functions are defined in the usual way except that they are restrictions to $\mathrm{Dom}_{\mathsf{Free}(\mathscr{Q})}$. [For $f \in \mathrm{Fun}_{\mathscr{Q}}$ and $e_1, \ldots, e_n$ in $\mathrm{Dom}_{\mathsf{Free}(\mathscr{Q})}$, such that $f(e_1, \ldots, e_n)$ is defined, $f(e_1, \ldots, e_n)$ is the expression $f(e_1, \ldots, e_n)$.] Its relations are defined as follows. For $r \in \mathrm{Rel}_{\mathscr{Q}}$ and $e_1, \ldots, e_n$ in $\mathrm{Dom}_{\mathsf{Free}(\mathscr{Q})}$ we define $r(e_1, \ldots, e_n) = r(\mathrm{val}(e_1), \ldots, \mathrm{val}(e_n))$. Note that $r(e_1, \ldots, e_n)$ is defined if and only if $r(\mathrm{val}(e_1), \ldots, \mathrm{val}(e_n))$ is defined, and similarly for basic functions. Again, we have used the same metasymbols f, r, for functions and relations of $\mathscr{Q}$ and $\mathsf{Free}(\mathscr{Q})$ as for the symbols in the expressions.

An algebra $\mathscr{Q}$ is called *spanned* if every element of $\mathrm{Dom}_{\mathscr{Q}}$ can be generated by a finite number of applications of functions in $\mathrm{Fun}_{\mathscr{Q}}$ (to constants in $\mathrm{Fun}_{\mathscr{Q}}$). If $\mathscr{Q}$ is spanned, then it is clear that $\mathscr{Q}$ is a quotient algebra of $\mathsf{Free}(\mathscr{Q})$, with val as its natural mapping.

The "translation" map τ is from $\mathscr{Q}'$, the representing algebra, to $\mathscr{Q}$, the represented algebra. It is helpful also to consider a "simulation" map σ in the other direction. Since τ is permitted to be many-to-one, σ is not defined from $\mathrm{Dom}_{\mathscr{Q}}$ to $\mathrm{Dom}_{\mathscr{Q}'}$, but rather from $\mathrm{Dom}_{\mathsf{Free}(\mathscr{Q})}$ to $\mathrm{Dom}_{\mathscr{Q}'}$. Intuitively, for every generation of an element of $\mathrm{Dom}_{\mathscr{Q}}$, a naturally representing element of $\mathrm{Dom}_{\mathscr{Q}'}$ is selected.

If $\mathcal{A}$ is an algebra, Fun a set of function symbols, and $\mathcal{F}$ an arity-preserving total mapping from Fun to the set of partial functions on $\mathrm{Dom}_{\mathcal{A}}$, then a partial mapping $\sigma_{\mathcal{F}}: \mathrm{Exp(Fun)} \to \mathrm{Dom}_{\mathcal{A}}$ is defined inductively as follows. If c is a constant symbol, then $\sigma_{\mathcal{F}}(c) = \mathcal{F}(c)(\ )$. If $e_1, \ldots, e_n \in \mathrm{Exp(Fun)}$ then $\sigma_{\mathcal{F}}(f(e_1, \ldots, e_n)) = \mathcal{F}(f)(\sigma_{\mathcal{F}}(e_1), \ldots, \sigma_{\mathcal{F}}(e_n))$. If $\mathcal{P}$ is an arity-preserving total mapping from Fun to the set of partial functions on $\mathrm{Dom}_{\mathcal{A}}$, we write $\sigma_{\mathcal{P}}$ for $\sigma_{\mathcal{F}}$, where $\mathcal{F}(f) = fn_{\mathcal{P}(f)}$ for all $f \in \mathrm{Fun}$.

Theorem 2.3. *Let* $\mathcal{A} \underset{\tau, \mathcal{P}}{\leqslant} \mathcal{A}'$ (resp. $\mathcal{A} \underset{\tau, \mathcal{P}}{\overset{\text{lin}}{\leqslant}} \mathcal{A}'$, $\mathcal{A} \underset{\tau, \mathcal{P}}{\overset{\text{eff}}{\leqslant}} \mathcal{A}'$). *Then the following diagram commutes*:

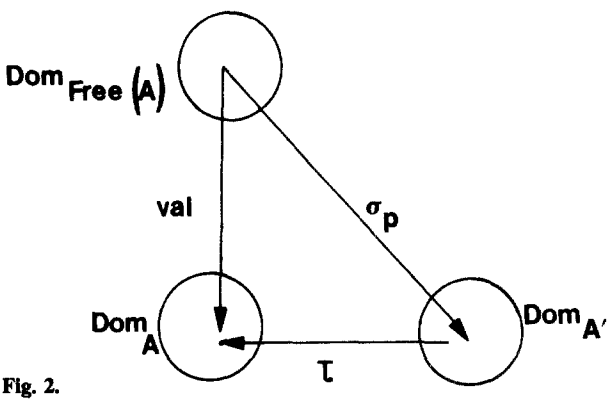

Fig. 2.

Proof. Left to the reader. (Note that $\mathcal{P}$ as defined assigns a particular element of $\mathrm{Dom}_{\mathcal{A}'}$ to each constant in $\mathrm{Fun}_{\mathcal{A}}$.) □

Thus, our reducibility definitions, by allowing arbitrarily many representations in $\mathrm{Dom}_{\mathcal{A}'}$ for an element of $\mathrm{Dom}_{\mathcal{A}}$, make it possible for a coding to distinguish the various computation paths used to generate the element.

III. Relative Computability

In this section we compare relative computability definitions involving different scheme classes and different codings. Theorem 3.1 shows that for a certain class of "sufficiently powerful" algebras, relative computability using effective schemes is no more general than relative computability using flowcharts. Later results show that relative computability under certain codings implies relative computability under certain other codings. In particular, Example 3.2 uses Theorem 3.2 to demonstrate the flowchart power of particular algebras with domain $\{0, 1\}^*$. Later, in Section IV, some of the ideas used to compare relative computability under different codings (in Theorem 3.2) are sharpened to allow comparison of relative complexity under different codings.

Let $\mathcal{A}$ and $\mathcal{A}'$ be algebras. Without loss of generality, we can take $\mathrm{Dom}_{\mathcal{A}}$ and $\mathrm{Dom}_{\mathcal{A}'}$ to be disjoint (by possibly renaming elements). Let $\underline{\mathcal{A} \cup \mathcal{A}'}$ denote the

algebra $\langle \mathrm{Dom}_{\mathfrak{A}} \cup \mathrm{Dom}_{\mathfrak{A}'}, \mathrm{Fun}_{\mathfrak{A}} \cup \mathrm{Fun}_{\mathfrak{A}'}, \mathrm{Rel}_{\mathfrak{A}} \cup \mathrm{Rel}_{\mathfrak{A}'} \rangle$. Let *succ* denote the function $\lambda x[x+1]$ on N.

Lemma 3.1. *Assume $\mathfrak{A}$ has only 0-ary and unary functions (and arbitrary relations). If f is a partial function computed by an effective scheme over $\mathfrak{A}$, then $\langle \mathrm{Dom}_{\mathfrak{A}}; f; \rangle \overset{\tau}{\leqslant} \mathfrak{A} \cup \langle N; 0, \mathrm{succ}; = \rangle$, where $\tau(x) = x$ if $x \in \mathrm{Dom}_{\mathfrak{A}}$ and $\tau(x)$ is undefined if $x \in N$.*

(The same result holds for relation r.)

Proof. An effective scheme over $\mathfrak{A}$ can be simulated by a flowchart scheme over $\mathfrak{A} \cup \langle N; 0, \mathrm{succ}; = \rangle$ by allowing $\langle N; 0, \mathrm{succ}; = \rangle$ to code the recursively enumerable control steps as in [28, §3]. (The availability of flowcharts over $\langle N; 0, \mathrm{succ}; = \rangle$ essentially allows the simulation of any finite number of counters.) Results of application of functions can be represented as numerically-coded formal expressions until a basic relation of $\mathfrak{A}$ is to be applied (or until an output is required). At such a time, the formal expressions involved must be evaluated by the simulating flowchart. The arity hypothesis insures that intermediate results of this evaluation can be stored in a finite number of flowchart locations [28]. □

Next, we show how an "auxiliary" algebra can sometimes be absorbed into another algebra by means of an arbitrary coding.

Lemma 3.2. *Assume $\mathfrak{A} \overset{\tau}{\leqslant} \mathfrak{A}' \cup \mathfrak{A}''$, where domain $(\tau) \subseteq \mathrm{Dom}_{\mathfrak{A}'}$. Assume $\mathfrak{A}'' \leqslant \mathfrak{A}'$. Then $\mathfrak{A} \overset{\tau}{\leqslant} \mathfrak{A}'$.*

Proof. Since there is essentially no intersection between $\mathfrak{A}'$ and $\mathfrak{A}''$, flowcharts over $\mathfrak{A}' \cup \mathfrak{A}''$ which have inputs from $\mathrm{Dom}_{\mathfrak{A}'}$ only can be simulated by flowcharts over $\mathfrak{A}'$, using any coding τ' of $\mathfrak{A}''$ in $\mathfrak{A}'$. □

Theorem 3.1. *Assume $\mathfrak{A}$ has only 0-ary and unary functions (and arbitrary predicates), and assume $\langle N; 0, \mathrm{succ}; = \rangle \leqslant \mathfrak{A}$. Then the effective schemes and the finite flowcharts compute the same classes of partial functions and predicates over $\mathfrak{A}$.*

Proof. Assume f is a partial function computed by an effective scheme over $\mathfrak{A}$. Then $\langle \mathrm{Dom}_{\mathfrak{A}}; f; \rangle \overset{\tau}{\leqslant} \mathfrak{A} \cup \langle N; 0, \mathrm{succ}; = \rangle$ by Lemma 3.1, where $\tau(x) = x$ if $x \in \mathrm{Dom}_{\mathfrak{A}}$, undefined if $x \in N$. By Lemma 3.2, $\langle \mathrm{Dom}_{\mathfrak{A}}; f; \rangle \overset{\iota}{\leqslant} \mathfrak{A}$, where ι is the identity function. That is, f is computed by a flowchart over $\mathfrak{A}$. The argument for relations is the same. □

In [18] an algebra $\mathfrak{A}$ is constructed with 0-ary and unary functions, over which there is a provable difference in computing power between effective schemes (in fact, recursive schemes) and flowchart schemes. Thus, the "sufficient power" condition $\langle N; 0, \mathrm{succ}; = \rangle \leqslant \mathfrak{A}$ is crucial.

Corollary 3.1. *Assume $\mathfrak{A} \leqslant^{\mathrm{eff}} \mathfrak{A}'$, $\mathfrak{A}'$ has 0-ary and unary functions (and arbitrary relations) and $\langle N; 0, \mathrm{succ}; = \rangle \overset{\tau}{\leqslant} \mathfrak{A}'$. Then $\mathfrak{A} \overset{\tau}{\leqslant} \mathfrak{A}'$.*

Proof. Immediate by Theorem 3.1. □

The construction of [18] is not strong enough to resolve the following:

Question 3.1. Do there exist algebras $\mathcal{A}$ and $\mathcal{A}'$ ($\mathcal{A}'$ with only 0-ary and unary functions and arbitrary relations) for which $\mathcal{A} \leqslant^{\text{eff}} \mathcal{A}'$ but for which it is false that $\mathcal{A} \leqslant \mathcal{A}'$?

Similar questions can be asked for other scheme classes.

An algebra $\mathcal{A}$ is *skeletal* if $\mathcal{A}$ is spanned, if $\text{Fun}_{\mathcal{A}}$ consists of unary total functions and constants, and $\text{Rel}_{\mathcal{A}}$ is $\{=\}$. Algebra $\mathcal{A}$ is a *skeleton* of algebra $\mathcal{A}'$ provided $\mathcal{A}$ is skeletal and $\text{Dom}_{\mathcal{A}} = \text{Dom}_{\mathcal{A}'}$. (Note that $\text{Fun}_{\mathcal{A}}$ is not required to be a subset of $\text{Fun}_{\mathcal{A}'}$.)

The next few results compare relative (flowchart) computability under different codings. The main result, Theorem 3.2, says that if a (sufficiently "powerful") algebra $\mathcal{A}'$ can simulate another algebra $\mathcal{A}$ in any coding τ, and also can simulate a skeleton for $\mathcal{A}$ both in τ and in another coding τ', then $\mathcal{A}'$ can simulate all of $\mathcal{A}$ in coding τ'. We first prove a lemma yielding a translation between two codings τ and τ'.

Lemma 3.3. *Assume algebras $\mathcal{A}, \mathcal{A}'$ and mappings τ, τ' satisfy the following:*

(a) *$\mathcal{A}$ is skeletal.*

(b) $\mathcal{A} \leqslant \mathcal{A}'$.

(c) $\mathcal{A} \overset{\tau}{\leqslant} \mathcal{A}'$.

(d) $\langle N; 0, \text{succ}; = \rangle \leqslant \mathcal{A}'$.

Then there is a unary partial function f on $\text{Dom}_{\mathcal{A}'}$, computable by a flowchart over $\mathcal{A}'$, such that $\tau'(y) = \tau(f(y))$ for all $y \in \text{domain}(\tau')$.

Proof. Let $\mathcal{P}, \mathcal{P}'$ be such that $\mathcal{A} \underset{\tau, \mathcal{P}}{\leqslant} \mathcal{A}'$ and $\mathcal{A} \underset{\tau', \mathcal{P}'}{\leqslant} \mathcal{A}'$.

We describe a flowchart F which, on input $y \in \text{domain}(\tau')$, outputs $\sigma_{\mathcal{P}}(x)$ for some x such that $\tau'(y) = \text{val}(x)$; this suffices by Theorem 2.3. F enumerates recursively the elements of $\text{Dom}_{\text{Free}(\mathcal{A})}$, by using $\langle N; 0, \text{succ}; = \rangle$ to code the enumerated expressions and to manage the necessary bookkeeping (by simulating several counters).

(1) For each such x in turn, F carries out the following two steps.

(1a) F computes $\sigma_{\mathcal{P}'}(x)$. (In order to do this, F follows the inductive definition of $\sigma_{\mathcal{P}'}$. Since the functions in $\text{Fun}_{\mathcal{A}}$ are all 0-ary or unary, F can keep the intermediate results of this computation in a finite number of registers. Again, necessary bookkeeping steps are handled using $\langle N; 0, \text{succ}; = \rangle$.)

(1b) F discovers whether $\tau'(y) = \text{val}(x)$. (In order to do this, F applies $\mathcal{P}'(=)$ to the given input y and $\sigma_{\mathcal{P}'}(x)$, thereby discovering whether $\tau'(y) = \tau'(\sigma_{\mathcal{P}'}(x))$ $(= \text{val}(x))$.) If not, F goes back to step (1a) to consider the next x in the enumeration. If so, then F goes on to step (2), retaining the current value of x.

(2) F computes and outputs $\sigma_{\mathcal{P}}(x)$. (The method is similar to that described in step (1a) for the computation of $\sigma_{\mathcal{P}'}$.)

Note that this construction requires that the operations of $\text{Fun}_{\mathcal{A}}$ be total. Also note that step (1) is guaranteed to terminate for $y \in \text{domain}(\tau')$, because $\mathcal{A}$ is spanned.

The situation described in this proof can be represented by the following diagram:

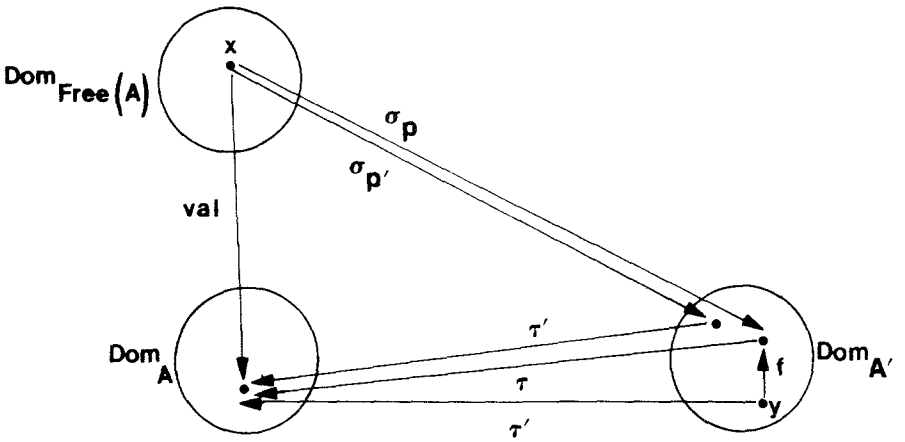

Fig. 3. □

Theorem 3.2. *Let algebras* $\mathcal{Q}, \mathcal{Q}', \mathcal{Q}''$ *and mappings* τ, τ' *satisfy the following*:

(a) $\mathcal{Q} \leqslant \mathcal{Q}'$,

(b) $\mathcal{Q}''^{\tau}$ *is a skeleton of* $\mathcal{Q}$,

(c) $\mathcal{Q}'' \leqslant \mathcal{Q}'$,

(d) $\mathcal{Q}'' \overset{\tau}{\underset{\tau'}{\leqslant}} \mathcal{Q}'$, *and*

(e) $\langle N; 0, \mathrm{succ}; = \rangle \leqslant \mathcal{Q}'$.

Then $\mathcal{Q} \underset{\tau'}{\leqslant} \mathcal{Q}'$.

Proof. Lemma 3.3 is applied twice, yielding two partial mappings f and g on $\mathrm{Dom}_{\mathcal{Q}'}$, each computable by a flowchart over $\mathcal{Q}'$, such that $\tau'(y) = \tau(f(y))$ if $y \in \mathrm{domain}(\tau')$, and $\tau(y) = \tau'(q(y))$ if $y \in \mathrm{domain}\ (\tau)$. For each function and predicate of $\mathcal{Q}$, a τ'-simulator is constructed by composing f and g with the given (in (c)) τ-simulators. For instance, if h is a unary function in $\mathrm{Fun}_{\mathcal{Q}}$ and h' its τ-simulator, then for y such that $h(\tau'(y))$ is defined, we have $h(\tau'(y)) = h(\tau(f(y))) = \tau(h'(f(y))) = \tau'(g(h'(f(y))))$. Hence, $g \circ h' \circ f$ is a τ'-simulator of h. Similarly, $r(\tau'(y))) = r(\tau(f(y))) = r'(f(y))$, so that $r' \circ f$ is a τ'-simulator of r. □

Example 3.1. Let $\mathcal{Q} = \langle N; 0, \mathrm{succ}; =, \in K \rangle$, where K is the halting set of some Gödel numbering [29], and let $\mathcal{Q}' = \langle N; 0, \mathrm{succ}; = \rangle$. Then it cannot be the case that $\mathcal{Q} \leqslant^{\mathrm{eff}} \mathcal{Q}'$. For if $\mathcal{Q} \leqslant^{\mathrm{eff}} \mathcal{Q}'$, then $\mathcal{Q} \leqslant \mathcal{Q}'$ by Theorem 3.1, and thus $\mathcal{Q} \underset{\iota}{\leqslant} \mathcal{Q}'$ by Theorem 3.2, where ι is the identity mapping. But this conclusion contradicts the undecidability of K. This example indicates that our simulation definitions are strong enough to preserve undecidability independently of the coding used.

Example 3.2. (Recursive power). We give a classification of the (flowchart) power of simple algebras with domain N or $\{0,1\}^*$. For examples involving the domains N and $\{0,1\}^*$, a "standard" coding is required. For this purpose, we

define a total function *2adic*: $N \to \{0,1\}^*$ by $2\text{adic}(x)$ = the binary representation of $x+1$, with the leading 1 removed. 2adic is a one-to-one correspondence between N and $\{0,1\}^*$. An algebra $\mathfrak{A}$ with $\text{Dom}_{\mathfrak{A}} = N$ or $\{0,1\}^*$ is said to *have recursive power* provided $\langle N; f; \rangle \overset{\tau}{\leqslant} \mathfrak{A}$ whenever f is partial recursive, and $\langle N; \ ; p \rangle \overset{\tau}{\leqslant} \mathfrak{A}$ whenever p is partial recursive, where τ is the identity or 2adic^{-1} as appropriate.

The algebra $\langle N; 0, \text{succ}; = \rangle$ can be shown to have recursive power by the construction of flowcharts for all partial recursive functions f, inductively on the definition of f by systems of recursion equations. Then simple flowchart programming and transitivity imply that many other algebras also have recursive power; for example, $\langle N; 0, \text{succ}, \leqslant \rangle$, $\langle N; 0, 1, +; = \rangle$, $\langle \{0,1\}^*; \lambda, 0, 1, \text{tail}, \text{concat}; = \rangle$ (where λ is the empty string; $\text{tail}(x) = \lambda$ if $x = \lambda$, all but the first symbol of x, otherwise; and $\text{concat}(x, y) = xy$), and $\langle \{0,1\}^*; \lambda, 0\text{succ}, 1\text{succ}; \text{prefix} \rangle$ (where $0\text{succ}(x) = x0$, $1\text{succ}(x) = x1$, and $\text{prefix}(x, y)$ is true iff x is a prefix of y) can be thus shown to have recursive power.

It appears somewhat less obvious that the algebras $\mathfrak{A} = \langle \{0,1\}^*; \lambda, 0\text{succ}, 1\text{succ}; = \rangle$ and $\mathfrak{A}' = \langle \{0,1\}^*; \lambda, 0, 1, \text{concat}; = \rangle$ also have recursive power. To see that they do, note that $\mathfrak{A}$ satisfies the hypotheses of Theorem 3.1 and that the effective schemes compute all partial recursive functions and predicates over $\mathfrak{A}$. (The operations of $\mathfrak{A}$ can be used to identify the input and generate the output, while the major work of the computation is done by the effective control.) The power of $\mathfrak{A}'$ follows from that of $\mathfrak{A}$ and transitivity.

Although the algebras given in this example all have the same flowchart computing power, it seems apparent that they are not all equally "efficient." Intuitively, it seems clear that some partial recursive functions are much "more quickly" computed over $\langle \{0,1\}^*; \lambda, 0\text{succ}, 1\text{succ}, \text{prefix} \rangle$ than over $\langle N; 0, \text{succ}; = \rangle$. A formal classification of the efficiency of the given algebras is studied in [19].

IV. Relative Complexity

Probably the most interesting questions to be considered in our framework involve determination of the coding-independent relative complexity of particular algebras. As an example, we consider in this section the flowchart complexity of $\mathfrak{N} = \langle N; 0, 1, +; \leqslant \rangle$ relative to $\mathfrak{B} = \langle \{0,1\}^*; \lambda, 0, 1, \text{head}, \text{tail}, 0\text{succ}, 1\text{succ}, \text{reverse}; = \lambda, = 0, = 1 \rangle$, (where the predicates of $\mathfrak{B}$ are tests for equality with short strings and where $\text{head}(x) = \lambda$ if $x = \lambda$, the first symbol of x, otherwise). The algebra $\mathfrak{B}$ can in some sense be considered to be a "unit-cost" algebra. However, the presence of the assignment operator in flowcharts makes the "unit-cost" intuition somewhat imperfect. Further discussion and use of $\mathfrak{B}$ appears in [19]. It should be clear that the ideas used in this section are generalizable to bounds other than those given and to algebras other than $\mathfrak{N}$ and $\mathfrak{B}$.

For any flowchart F, let L_F denote the natural path length function. This is the time complexity measure we shall use in what follows.

In the standard 2adic coding of $\mathfrak{N}$ into $\mathfrak{B}$, it is clear that $+$ and $\leqslant$ can be simulated by flowcharts with path lengths linear in the length of the coded inputs in N. (We define the *length* $|x|$ of $x \in N$, by $|x| = |2\text{adic}(x)|$, the length of

the 2adic bit string. Note that $|x| = \lfloor \log_2(x+1) \rfloor$, hence is really independent of 2adic.) It is conceivable that some other coding of N into $\{0,1\}^*$ (say, one that resembles floating-point coding) might allow the computation of some functions and relations to be done much more efficiently than the standard coding. The following results show that the only way that situation could occur is if either $+$ or $\leqslant$ became more complex in that other coding (a rather undesirable property). For simplicity in proving these results, we hypothesize maintenance of the linear complexity of $+$ and $\leqslant$. The results have straightforward modifications for larger complexity bounds on $+$ and $\leqslant$.

The following lemma uses ideas which are extensions of those in Lemma 3.3. It shows that a linear bound on $+$ alone in any coding is sufficient to yield a quadratic translation from the 2adic coding to the new coding of any number. It is important to note that the linear and quadratic bounds are functions of parameters derived entirely within the *coded* system $\mathfrak{N}$, without reference to the *coding* system $\mathfrak{B}$. This is necessary, of course, for meaningful comparison of the effects of codings on efficiency. In most of the results, we use the length $|x|$ as the parameter, but others derived within $\mathfrak{N}$ might also provide significant parameters on which to base complexity comparisons. One such parameter is considered later in this section, in Lemma 4.3 and Theorem 4.5.

Where no confusion is likely, we take the notational liberty of using the same symbol for a flowchart and for the function or predicate it computes. In the remainder of this section, we let k denote an arbitrary constant and p an arbitrary polynomial.

Lemma 4.1. *Assume* $\langle N; +; \rangle \underset{\tau, \mathcal{P}}{\leqslant} \mathfrak{B}$ *and* $L_{\mathcal{P}(+)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$ *for all* $x, y \in \text{domain}(\tau)$. *Then there exist a flowchart* F *over* $\mathfrak{B}$ *and a constant* c *such that* $\tau(F(x)) = 2\text{adic}^{-1}(x)$ *and* $L_F(x) \leqslant c(|2\text{adic}^{-1}(x)|^2 + 1)$ *for all* $x \in \{0,1\}^*$.

Proof. F uses the bits of x to determine a sequence of $+$ operations that would generate $2\text{adic}^{-1}(x)$, starting with 0 and 1. The sequence consists of about $|x|$ operations, each involving either doubling, or doubling and adding 1. F then uses a fixed element, b_0, of $\tau^{-1}(0)$ and a fixed element, b_1, of $\tau^{-1}(1)$ and applies $\mathcal{P}(+)$ in the way described by the above sentence, using b_0 and b_1 in place of 0 and 1. Since $\mathcal{P}(+)$ computes a τ-simulator of $+$, an element of $\tau^{-1}(2\text{adic}^{-1}(x))$ is thereby obtained.

Each of the approximately $|x|$ operations involves at most two applications of $\mathcal{P}(+)$ to elements which are τ-representations of integers n with $|n| \leqslant \max(|x|, 1)$. (Note that it is *the integers n themselves, not their τ-representations,* whose lengths are thus bounded.) Since $|2\text{adic}^{-1}(x)| = |x|$, the needed bound follows from the bound on $L_{\mathcal{P}(+)}$.

The situation can be depicted as follows:

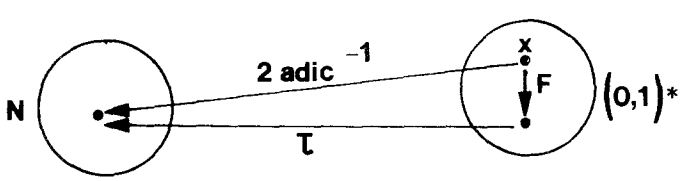

Fig. 4. □

The one-way translation above is sufficient to imply that any relation of at least quadratic complexity can be computed just about as efficiently in the standard coding as in any other coding (if linear complexity of + is to be preserved). For simplicity, we state the result for unary relations only.

Theorem 4.1. *Let t be a partial function. Assume $\langle N; +; r\rangle \overset{\tau,\mathcal{P}}{\leqslant} \mathfrak{B}$, where $L_{\mathcal{P}(+)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$ and $L_{\mathcal{P}(r)}(x) \leqslant t(\tau(x))$ for all $x, y \in$ domain(τ). Then there exist a flowchart G over $\mathfrak{B}$ and a constant c such that G computes a 2adic^{-1}-simulator of r and $L_G(x) \leqslant c(|2\text{adic}^{-1}(x)|^2 + 1) + t(2\text{adic}^{-1}(x))$.*

Proof. By Lemma 4.1 and composition of flowcharts F and $\mathcal{P}(r)$. □

If we wish to obtain a result similar to Theorem 4.1, for functions rather than relations, a bound for translating from the τ coding back to the standard coding is also required. The most efficient flowchart we know for this translation is obtained from a direct compilation of a linear recursive scheme into the flowchart language, using techniques of [24]. If P is a linear recursive scheme, let L_P denote the recursion depth function. (Since linear recursive schemes are loop-free, L_P is a good estimate of the running time.) We use the following version of a result of [24].

Theorem 4.2 (Chandra). *Let P be a linear recursive scheme, ε any positive real. Then there exist flowchart F and constant c such that for any interpretation algebra $\mathfrak{A}$,*

(a) *F computes the same function (or relation) over $\mathfrak{A}$ as does P,*

(b) *$L_F(x_1, \ldots, x_n) \leqslant c((L_P(x_1, \ldots, x_n))^{1+\varepsilon} + 1)$ for all inputs $x_1, \ldots, x_n$, and*

(c) *if a basic operation of $\mathfrak{A}$ is applied during the execution of F on given inputs, then the same basic operation is applied to the same arguments during the execution of P on the same inputs.*

Proof. See [24]. □

It may seem that conditions (a) and (b) capture the important complexity relationship maintained by the translation. Indeed, the results in [24] explicitly give only these two conditions. However, this suffices only if the basic operations of $\mathfrak{A}$ are thought of as atomic. If the intention is to substitute flowcharts for the basic operations of $\mathfrak{A}$, then some condition such as (c) is needed for a complexity comparison to remain invariant through the substitution. This point is illustrated below in the proof of Lemma 4.2. (See [30] for a study of scheme complexity in which a condition similar to (c) is central.)

The following lemma yields the translation needed for functions. Linear bounds are now imposed both on + and $\leqslant$.

Lemma 4.2. *Assume $\langle N; +; \leqslant\rangle \overset{\tau,\mathcal{P}}{\leqslant} \mathfrak{B}$, where $L_{\mathcal{P}(+)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$ and $L_{\mathcal{P}(\leqslant)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$ for all $x, y \in$ domain(τ). Let ε be a positive real. Then there exist a flowchart F' over $\mathfrak{B}$ and a constant c such that $2\text{adic}^{-1}(F'(x)) = \tau(x)$ and $L_{F'}(x) \leqslant c(|\tau(x)|^{2+\varepsilon} + 1)$ for all $x \in$ domain(τ).*

Proof. We augment $\mathfrak{B}$ to a new algebra $\mathfrak{B}'$, design a flowchart G over $\mathfrak{B}'$ for the needed translation, and then obtain F' by replacing the operation symbols of

the new algebra by flowcharts over $\mathfrak{B}$. The process is done in two stages because part of the construction involves a translation of a linear recursive scheme; the replacement flowcharts involve loops and linear recursive schemes are required to be loop-free.

Fix $b_1 \in \tau^{-1}(1)$, $f = fn_{\mathscr{P}(+)}$ and $r = \mathrm{rel}_{\mathscr{P}(<)}$. Let $\mathfrak{B}' = \langle \{0,1\}^*; \lambda, 0\mathrm{succ}, 1\mathrm{succ}, \mathrm{head}, \mathrm{tail}, \mathrm{reverse}, b_1, f; =\lambda, =0, =1, r\rangle$. Consider the following linear recursive scheme interpreted over $\mathfrak{B}'$. Notation is as in [24].

Translate(x_0): *data*: x_1, x_2
/*Given $x_0 \in \{0,1\}^*$, Translate(x_0) outputs 2adic$(\tau(x_0))$ if $x_0 \in \mathrm{domain}(\tau)$. Its behavior is otherwise unspecified.*/

START
$\langle x_1, x_2\rangle \leftarrow \mathrm{Approx}(f(x_0, b_1), b_1)$;
RETURN(x_1)

Approx(x_0, x_1): *data*: x_2, x_3
/*Given x_0 with $\tau(x_0)$ defined and $\geqslant 1$, x_1 with $\tau(x_1)$ a power of 2 and $\tau(x_1) \leqslant \tau(x_0)$, Approx$(x_0, x_1)$ returns two values:
(1) the string obtained by deleting the leading 1 from the binary representation of $\lfloor \tau(x_0) \div \tau(x_1) \rfloor$, and
(2) some value in $\tau^{-1}(\lfloor \tau(x_0) \div \tau(x_1) \rfloor \times \tau(x_1))$.
Its behavior is otherwise unconstrained.*/

START
if $r(f(x_1, x_1), x_0)$
then begin
$\langle x_2, x_3\rangle \leftarrow \mathrm{Approx}(x_0, f(x_1, x_1))$;
if $r(f(x_3, x_1), x_0)$
then RETURN $(1\mathrm{succ}(x_2), f(x_3, x_1))$
else RETURN $(0\mathrm{succ}(x_2), x_3)$;
end
else RETURN (λ, x_1)

By definition, 2adic$(\tau(x_0))$ is the binary representation of $\tau(x_0)+1$, with the leading 1 removed. The main program τ-simulates the addition of 1 to $\tau(x_0)$. The procedure determines recursively the binary representation of the quotient of $\tau(x_0)$ and the current power of 2 (given by $\tau(x_1)$), with the leading 1 removed. It also determines a τ-representation of the approximation to $\tau(x_0)$ obtained by truncating its binary representation after the $\tau(x_1)$ position.

Note that on any input $x \in \mathrm{domain}(\tau)$, the recursion depth is approximately $|\tau(x)|$, and f and r are applied only to elements y of domain(τ) with $|\tau(y)|$ at most approximately $|\tau(x)|$. By Theorem 4.2, we obtain flowchart $\mathcal{G}$ computing *Translate* over $\mathfrak{B}'$, with $L_{\mathcal{G}}(x) \leqslant c(|\tau(x)|^{1+\varepsilon}+1)$ whenever $x \in \mathrm{domain}(\tau)$, where c is a constant. Moreover, each argument y to which f and r are applied when G is run on input x has $|\tau(y)|$ at most approximately $|\tau(x)|$ (because of conclusion (c) of Theorem 4.2).

Now obtain F' over $\mathfrak{B}$ from G by replacing b_1, f and r by their flowcharts over $\mathfrak{B}$. The complexity bound follows from the hypotheses on $L_{\mathscr{P}(+)}$ and $L_{\mathscr{P}(<)}$. □

Now we obtain a result similar to Theorem 4.1 for functions rather than relations.

Theorem 4.3. *Let t be a partial function. Assume $\langle N; +, f; \leqslant \rangle \leqslant_{\tau,\mathcal{P}} \mathfrak{B}$, where $L_{\mathcal{P}(+)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$, $L_{\mathcal{P}(\leqslant)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$, and $L_{\mathcal{P}(f)}(x) \leqslant t(\tau(x))$ whenever $x, y \in$ domain(τ). Let ε be a positive real. Then there exist a flowchart G over $\mathfrak{B}$ and a constant c such that G computes a 2adic^{-1}-simulator of f and*

$$L_G(x) \leqslant c\big(|2\text{adic}^{-1}(x)|^2 + |f(2\text{adic}^{-1}(x))|^{2+\varepsilon} + 1\big) + t(2\text{adic}^{-1}(x)).$$

Proof. By Lemmas 4.1 and 4.2 and composition of flowcharts F (of Lemma 4.1), F' (of Lemma 4.2) and $\mathcal{P}(f)$. The situation can be depicted as follows:

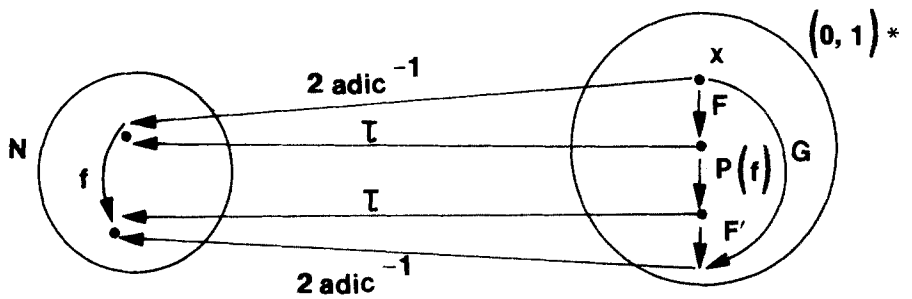

Fig. 5. □

Theorems 4.1 and 4.3 delimit in a particular way the improvement in efficiency that can be obtained by choosing an alternative to the standard coding of $\mathfrak{N}$ in $\mathfrak{B}$. Above the quadratic level (i.e. $t(x)=c|x|^2$), no order of magnitude improvement is possible for predicates and "small" functions ["small" meaning that $|f(x)| \leqslant t(x)$]. The only possible improvement is a "local" one arising from possible concise representations of large numbers. Such an improvement is possible, for example, in a "floating-point" coding, τ, such as the following.

Example 4.1. For a string $w=a_1 1 a_2 1 \ldots 1 a_k 00 b_1 1 b_2 \ldots 1 b_\ell$, where all $a_i, b_j \in \{0,1\}$, we define $\tau(w)=2\text{adic}^{-1}(a_1 \ldots a_k) \times 2^{2\text{adic}^{-1}(b_1 \ldots b_\ell)}$, that is, τ is a partial function from $\{0,1\}^*$ onto N such that $\tau(w)=n \times 2^m$, where $2\text{adic}(n)=a_1 \ldots a_k$ and $2\text{adic}(m)=b_1 \ldots b_\ell$. The existence of simulators of $+$ and $\leqslant$ with linear complexity bounds is easy to verify, if we recall that the relevant parameter is length measured in the system $\mathfrak{N}$. Thus, by Theorem 4.3, for small functions of at least quadratic complexity, the floating-point coding provides no improvement over the standard coding. But a large function like the exponential $=\lambda x[2^x]$ has a linear complexity flowchart in the τ-coding, whereas all flowcharts for the exponential function in the standard coding require exponential path length simply to generate the needed representation.

We summarize the preceding results by stating a result which says that any function or predicate which is "polynomial-computable" in any coding of $\mathfrak{N}$ in $\mathfrak{B}$ is also polynomial-computable in the standard coding.

Theorem 4.4. (a) *Assume* $\langle N; +; r\rangle \underset{\tau,\mathscr{P}}{\leqslant} \mathfrak{B}$, *where* $L_{\mathscr{P}(+)}(x, y) \leqslant p(|\tau(x)| + |\tau(y)|)$ *and* $L_{\mathscr{P}(r)}(x) \leqslant p(|\tau(x)|)$ *for all* $x, y \in domain(\tau)$. *Then there exists a flowchart F and a polynomial q such that F computes a* 2adic^{-1}*-simulator of r and* $L_F(x) \leqslant q(|2\text{adic}^{-1}(x)|)$.

(b) *Assume* $\langle N; +, f; \leqslant\rangle \underset{\tau,\mathscr{P}}{\leqslant} \mathfrak{B}$, *where* $L_{\mathscr{P}(+)}(x, y) \leqslant p(|\tau(x)| + |\tau(y)|)$, $L_{\mathscr{P}(<)}(x, y) \leqslant p(|\tau(x)| + |\tau(y)|)$ *and* $L_{\mathscr{P}(f)}(x) \leqslant p(|\tau(x)|)$ *for all* $x, y \in domain(\tau)$. *Assume further that* $|f(x)| \leqslant p(|x|)$. *Then there exist a flowchart F over* $\mathfrak{B}$ *and a polynomial q such that F computes a* 2adic^{-1}*-simulator of f and* $L_F(x) \leqslant q(|2\text{adic}^{-1}(x)|)$.

An objection can be raised to the form in which the linear bounds are expressed in the preceding results. For all the codings, a *uniform* bound is hypothesized on the running time for flowcharts on all representations of an element. For instance, in Lemma 4.1 we use the condition $L_{\mathscr{P}(+)}(x, y) \leqslant k(|\tau(x)| + |\tau(y)| + 1)$. This may be too restrictive a condition for a model which allows infinitely many representations for each element. Perhaps it should be expected that a flowchart should take more time on some representations of an element than on others. It might, for instance, be desirable to have a flowchart run fast on those representations generated in a small number of steps of a "user program," whereas it is possibly less important that it run fast on representations which take many steps for a user program to generate (since the program is using considerable time in any case). One way to do this is to express complexity in terms of a parameter other than size of the represented element. The parameter should depend on the way a representation is generated, yet still be defined within the represented algebra. We define here one such parameter applicable to all algebras. The "user program step" intuition will be formalized by application of this general parameter to the associated free algebra. We introduce a numerical measure of this parameter as follows.

Let $\mathfrak{A}$ be any algebra and $A, B \subseteq \text{Dom}_{\mathfrak{A}}$. We define *size*$_{\mathfrak{A}}$*(A : B) the size, in* $\mathfrak{A}$*, of A relative to B)* as follows:

(a) $\text{size}_{\mathfrak{A}}(A : B) = 0$ iff $A \subseteq B$.

(b) $\text{size}_{\mathfrak{A}}(A : B) = k + 1$ iff both (b1) and (b2) hold.

(b1) There exist $C \subseteq \text{Dom}_{\mathfrak{A}}$, $f \in \text{Fun}_{\mathfrak{A}}$, $x_1, \ldots, x_n \in C$, such that $A \subseteq C \cup \{f(x_1, \ldots, x_n)\}$ and $\text{size}_{\mathfrak{A}}(C : B) = k$.

(b2) $\text{size}_{\mathfrak{A}}(A : B) \not\leqslant k$.

(c) $\text{size}_{\mathfrak{A}}(A : B)$ is otherwise undefined (and is said to be equal to ∞).

By convention, $n < \infty$ for all $n \in N$, and $\infty \leqslant \infty$. From (b), we see that $\text{size}_{\mathfrak{A}}(A : B)$ is the number of $\mathfrak{A}$ operations required by a straight-line program to generate the elements in A given the elements in B. We write $\underline{size_{\mathfrak{A}}(x : B)}$ for $\text{size}_{\mathfrak{A}}(\{x\} : B)$, $\underline{size_{\mathfrak{A}}(A)}$ for $\text{size}_{\mathfrak{A}}(A : \varnothing)$, and $\underline{size_{\mathfrak{A}}(x)}$ for $\text{size}_{\mathfrak{A}}(\{x\} : \varnothing)$. In this general notation, if $x \in \{0, 1\}^*$, then $\text{size}_{\langle\{0,1\}^*; \lambda,\ 0\text{succ}, 1\text{succ}; \rangle}(x) = |x|$, and if $x \in N$, then $\text{size}_{\langle N; 0, 1, +; \rangle}(x)$ is the length of a minimal addition chain generating x (See [31], p. 402).

Example 4.2. Consider $\text{size}_{\mathfrak{N}}(9)$. It is easy to see that $\text{size}_{\mathfrak{N}}(1) = 1$, $\text{size}_{\mathfrak{N}}(2) = 2$, $\text{size}_{\mathfrak{N}}(4) = 3$, $\text{size}_{\mathfrak{N}}(8) = 4$, and thus $\text{size}_{\mathfrak{N}}(9) = 5$. Although one can compute 9 in $\mathfrak{N}$ by adding 1 nine times, clause (b2) shows that $\text{size}_{\mathfrak{N}}(9) \neq 9$.

We now obtain comparisons of codings using $\text{size}_{\text{Free}(\mathcal{N})}(x)$ as a parameter. Intuitively, this allows complexity to be calculated not only in terms of the magnitude of a number, but also in terms of *the way that number is generated* (by $+$). Evidence that this definition provides a useful general parameter for arbitrary algebras appears in [16, 17].

Remark. It is easy to see that $\mathcal{N} \leqslant_{2\text{adic}-1,\mathcal{P}} \mathcal{B}$ for some $\mathcal{P}$ such that $L_{\mathcal{P}(+)}(\sigma_{\mathcal{P}}(x), \sigma_{\mathcal{P}}(y)) \leqslant c(\text{size}_{\text{Free}(\mathcal{N})}(x) + \text{size}_{\text{Free}(\mathcal{N})}(y))$ and $L_{\mathcal{P}(\leqslant)}(\sigma_{\mathcal{P}}(x), \sigma_{\mathcal{P}}(y)) \leqslant c(\text{size}_{\text{Free}(\mathcal{N})}(x) + \text{size}_{\text{Free}(\mathcal{N})}(y))$, where c is a constant. Thus, there is a complexity bound for the computation of $+$ and $\leqslant$ in the standard 2adic coding which is linear in the new size parameter. So it is reasonable to impose corresponding restrictions on other codings in order to compare them. Analogous to Lemma 4.1, we obtain Lemma 4.3. It states again that quadratic translation can be accomplished from the standard coding to any other coding in which $+$ and $\leqslant$ are linearly simulatable. For the remainder of the paper, $\mathcal{P}$ is fixed as described earlier in this paragraph.

Lemma 4.3. *Assume* $\langle N; 0, 1, +; \rangle \leqslant_{\tau,\mathcal{Q}} \mathcal{B}$, *and* $L_{\mathcal{Q}(+)}(\sigma_{\mathcal{Q}}(x), \sigma_{\mathcal{Q}}(y)) \leqslant k(\text{size}_{\text{Free}(\mathcal{N})}(x) + \text{size}_{\text{Free}(\mathcal{N})}(y))$ *for all* $x, y \in \text{Dom}_{\text{Free}(\mathcal{N})}$. *Then there exist a flowchart* F *and a constant* c *such that*

$$(\forall x)(\exists y)\big[\, F(\sigma_{\mathcal{P}}(x)) = \sigma_{\mathcal{Q}}(y),\ \sigma_{\mathcal{P}}(x) = \sigma_{\mathcal{P}}(y),\ \text{size}_{\text{Free}(\mathcal{N})}(y) \leqslant c\ \text{size}_{\text{Free}(\mathcal{N})}(x)$$

and $L_F(\sigma_{\mathcal{P}}(x)) \leqslant c(\text{size}_{\text{Free}(\mathcal{N})}(x))^2$].

Proof. The situation can be depicted as follows:

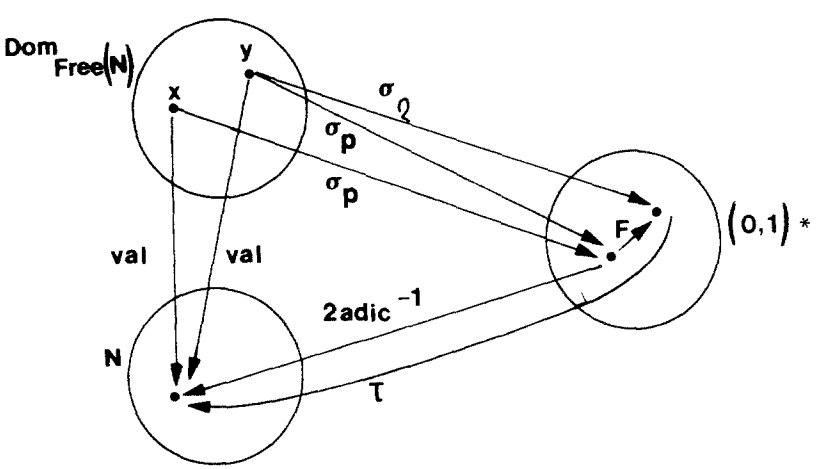

Fig. 6.

(Thus, $\text{val}(x) = \text{val}(y)$, $\text{val} = 2\text{adic}^{-1} \circ \sigma_{\mathcal{P}} = \tau \circ \sigma_{\mathcal{Q}}$, and $2\text{adic}^{-1} = \tau \circ F$.)

Since $\sigma_{\mathfrak{P}}$ is many-one, one of the $\sigma_{\mathfrak{P}}$-preimages of $\sigma_{\mathfrak{P}}(x)$, y, must be selected. The construction is similar to that for Lemma 4.1. F uses the bits of $\sigma_{\mathfrak{P}}(x)$ to determine a sequence of $+$ operations that would generate val(x), starting with 0 or 1. The sequence consists of about $|\sigma_{\mathfrak{P}}(x)|$ operations, each involving either doubling, or doubling and adding 1. By its inductive definition, $\sigma_{\mathfrak{P}}(x)$ has at most about $\text{size}_{\text{Free}(\mathfrak{A})}(x)$ bits, so that the operation sequence obtained has at most about $\text{size}_{\text{Free}(\mathfrak{A})}(x)$ operations. Let y be the infix expression describing this operation sequence. The size bound on y is clearly true.

F then begins with $\mathfrak{Q}(0)$ and $\mathfrak{Q}(1)$, and repeatedly applies $\mathfrak{Q}(+)$ according to the given sequence. Each application is to bit-strings which are $\sigma_{\mathfrak{Q}}$-images of expressions, t, having $\text{size}_{\text{Free}(\mathfrak{A})}(t)$ at most about $\text{size}_{\text{Free}(\mathfrak{A})}(x)$. The hypothesis on $\mathfrak{Q}(+)$ yields the quadratic bound. □

Similarly, analogues of Theorem 4.1, Lemma 4.2 and Theorem 4.3 can be obtained, based on $\text{size}_{\text{Free}(\mathfrak{A})}$ as a parameter. Since some complicated details are thereby introduced, we simply give summary versions allowing polynomial variance.

Theorem 4.5. (a) *Assume* $\langle N; 0, 1, +; r\rangle \underset{\tau,\mathfrak{Q}}{\leqslant} \mathfrak{B}$, *where*

$$L_{\mathfrak{Q}(+)}(\sigma_{\mathfrak{Q}}(x), \sigma_{\mathfrak{Q}}(y)) \leqslant p\big(\text{size}_{\text{Free}(\mathfrak{A})}(x) + \text{size}_{\text{Free}(\mathfrak{A})}(y)\big)$$

and $L_{\mathfrak{Q}(r)}(\sigma_{\mathfrak{Q}}(x)) \leqslant p(\text{size}_{\text{Free}(\mathfrak{A})}(x))$ *for all* $x, y \in \text{Dom}_{\text{Free}(\mathfrak{A})}$. *Then there exist a flowchart* G *computing a* $2adic^{-1}$*-simulator of* r *and a polynomial* q, *such that* $L_G(\sigma_{\mathfrak{P}}(x)) \leqslant q(\text{size}_{\text{Free}(\mathfrak{A})}(x))$.

(b) *Assume* $\langle N; 0, 1, +, \leqslant\rangle \underset{\tau,\mathfrak{Q}}{\leqslant} \mathfrak{B}$, *where* $L_{\mathfrak{Q}(+)}(\sigma_{\mathfrak{Q}}(x), \sigma_{\mathfrak{Q}}(y)) \leqslant p(\text{size}_{\text{Free}(\mathfrak{A})}(x) + \text{size}_{\text{Free}(\mathfrak{A})}(y))$ *and* $L_{\mathfrak{Q}(<)}(\sigma_{\mathfrak{Q}}(x), \sigma_{\mathfrak{Q}}(y)) \leqslant p(\text{size}_{\text{Free}(\mathfrak{A})}(x) + \text{size}_{\text{Free}(\mathfrak{A})}(y))$ *for all* $x, y \in \text{Dom}_{\text{Free}(\mathfrak{A})}$. *Then there exist a flowchart* F' *and a polynomial* q *such that*

$$(\forall x)(\exists y)\big[F'(\sigma_{\mathfrak{Q}}(x)) = \sigma_{\mathfrak{P}}(y),\ \text{val}(x) = \text{val}(y),$$

and $L_{F'}(\sigma_{\mathfrak{Q}}(x)) \leqslant q(\text{size}_{\text{Free}(\mathfrak{A})}(x))]$.

(c) *Assume* $\langle N; 0, 1, +, f; \leqslant\rangle \underset{\tau,\mathfrak{Q}}{\leqslant} \mathfrak{B}$, *where* $L_{\mathfrak{Q}(+)}$ *and* $L_{\mathfrak{Q}(<)}$ *are as in* (b), *where* $L_{\mathfrak{Q}(f)}(\sigma_{\mathfrak{Q}}(x)) \leqslant p(\text{size}_{\text{Free}(\mathfrak{A})}(x))$ *and with the size restriction*

$$(\forall x)(\exists y)\big[\mathfrak{Q}(f)(\sigma_{\mathfrak{Q}}(x)) = \sigma_{\mathfrak{Q}}(y) \text{ and } \text{size}_{\text{Free}(\mathfrak{A})}(y) \leqslant p\big(\text{size}_{\text{Free}(\mathfrak{A})}(x)\big)\big].$$

Then there exist flowchart G *computing a* 2adic^{-1}*-simulator of* f *such that* $L_G(\sigma_{\mathfrak{P}}(x)) \leqslant q(\text{size}_{\text{Free}(\mathfrak{A})}(x))$, *where* q *is a polynomial.*

Proof. (a) As for Theorem 4.1, using Lemma 4.3. Note that the size bound on y obtained in Lemma 4.3 is required here.

(b) The situation can be depicted as follows:

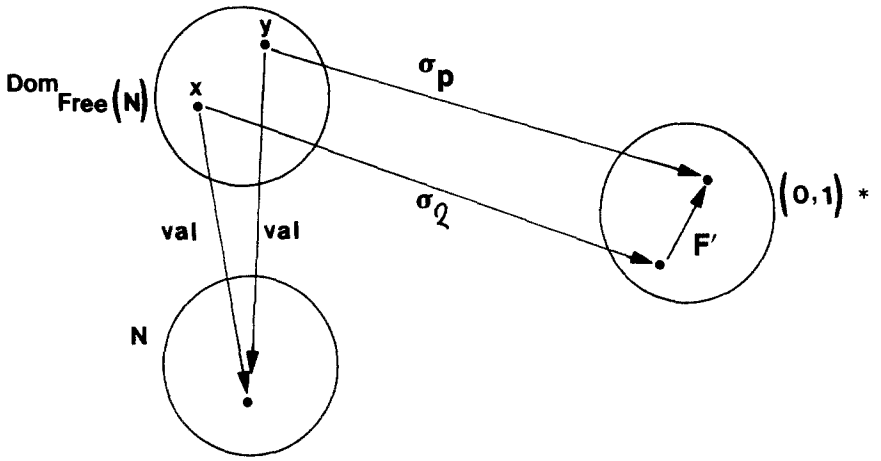

Fig. 7.

The proof is similar to that of Lemma 4.2, but several complications are introduced by the new parameter. Since polynomial variance is allowed, we simplify matters by performing a more direct translation than that for Lemma 4.2, simply determining the bits of the binary representation of $\mathrm{val}(x)+1$ in order, high to low order. The functions computed by $\mathfrak{Q}(0)$, $\mathfrak{Q}(1)$, $\mathfrak{Q}(+)$ and $\mathfrak{Q}(\leqslant)$ are used to translate from $\sigma_{\mathfrak{Q}}(x)$ to $\sigma_{\mathfrak{P}}(y)$, and then replaced by their hypothesized flowcharts. (A similar idea was used for Lemma 4.2.) The number of applications of $\mathfrak{Q}(+)$ and $\mathfrak{Q}(\leqslant)$ in this translation can be bounded by $q_1(\log(\mathrm{val}(x)))$ for some polynomial q_1. Furthermore, $\mathfrak{Q}(+)$ and $\mathfrak{Q}(\leqslant)$ are only applied to $\sigma_{\mathfrak{Q}}(x)$ and to elements of the form $\sigma_{\mathfrak{Q}}(w)$, where $\mathrm{size}_{\mathrm{Free}(\mathfrak{N})}(w) \leqslant q_2(\log(\mathrm{val}(x)))$ for some polynomial q. Then by the hypotheses on $\mathfrak{Q}(+)$ and $\mathfrak{Q}(\leqslant)$, the total time for the translation is bounded by $q_3(\log(\mathrm{val}(x)), \mathrm{size}_{\mathrm{Free}(\mathfrak{N})}(x))$ for some polynomial q_3. Since $\log(\mathrm{val}(x)) \leqslant q_4(\mathrm{size}_{\mathrm{Free}(\mathfrak{N})}(x))$ for some polynomial q_4, the bound follows.

(c) By Lemma 4.3 and (b), using composition of flowcharts. The size bound on y obtained in Lemma 4.3 and the new size restriction are both required here to control the growth of the parameter. □

We remark that (c) leaves open the possibility that some improvement may occur in an alternative coding if the size restriction assumed in (c) is violated.

V. Further Work

In Section III, some comparison was given of relative computability definitions for different scheme classes. Such comparisons should also be carried out for relative complexity definitions. Somewhat more specifically, Theorem 3.1 can be paraphrased by saying that if enough "power" is present in an algebra, then that algebra can "simulate" with a flowchart the result of any effective scheme. It is

intuitively plausible that if enough "efficient power" is present in an algebra, then that algebra can "efficiently simulate" with a flowchart the result of any effective scheme (or recursive scheme, for example). The power hypothesis was expressed in terms of a reducibility. The efficient power hypothesis should be similarly expressible in terms of a suitable efficient reducibility. (Remarks in Section IV and in [16, 17] suggest ways of defining efficient reducibilities.)

Section IV represents a small beginning for classification of coding-independent relative complexity of specific algebras. Many interesting technical questions remain to be formulated. Subsequent classification efforts might focus on basic numeric and bit string algebras, finitely generated groups and perhaps algebras arising in finite set theory. For such algebras, reasonable progress should be possible with only minor extensions of the definitions in this paper.

The model of this paper is not sufficiently general for the study of the implementation of data structure algebras [2, 9, 10]. Appropriate extension to a suitably general representation model with general size parameter is needed. The model should allow realistic treatment of coding-independent complexity of implementations of data structures. Some possible directions are suggested by [21, 32, 33].

Acknowledgments

The authors would like to thank Arnold Rosenberg and Ann Yasuhara for their careful readings of several versions of the manuscript and their many helpful suggestions for its revision. This paper is very much improved by their contributions.

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, 7–19, March 1975.
3. F. L. Morris, Correctness of Translations of Programming Languages—An Algebraic Approach, Stanford U. Report CS-72-303, August 1972.
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 (September 1973); 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, 126–131, (1975).
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, 147–158, (October 1976).
9. J. V. Guttag, E. Horowitz and D. R. Musser, Abstract Data Types and Software Validation. Research Report 76-48. Information Sciences Institute, August 1976.

10. J. A. Goguen, J. W. Thatcher and E. G. Wagner, An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, IBM Thomas J. Watson Research Center, manuscript.
11. A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
12. S. A. Cook, The Complexity of Theorem-Proving Procedures, Third Annual ACM Symposium on Theory of Computing, 151–158, 1971.
13. R. Karp, Reducibility among Combinatorial Problems, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, 85–104, (1972).
14. R. Ladner, N. A. Lynch and A. L. Selman, Comparison of Polynomial-Time Reducibilities, Sixth Annual ACM Symposium on Theory of Computing, 110–121, 1974.
15. E. K. Blum and D. R. Estes, A Generalization of the Homomorphism Concept, Algebra Universalis, July, 1977.
16. N. A. Lynch, Straight-Line Program Length as a Parameter for Complexity Measures. Tenth Annual ACM Symposium on Theory of Computing, 1978.
17. N. A. Lynch, Straight-Line Program as a Parameter for Complexity Analysis, to appear in Theoretical Computer Science.
18. N. A. Lynch and E. K. Blum, A Difference in Expressive Power Between Flowcharts and Recursion Schemes, Math. Syst. Theory, 205–211, 1979.
19. N. A. Lynch and E. K. Blum, Relative Complexity of Operation Sets for Numeric and Bit String Algebras, Math. Syst. Theory 13, 187–207 (1980).
20. N. A. Lynch and E. K. Blum, Efficient Reducibility Between Programming Systems, Proceedings of Ninth Annual Symposium on Theory of Computation, 1977.
21. A. Cremers and T. Hibbard, Formal Modelling of Virtual Machines. To appear in IEEE Transactions on Software Engineering.
22. R. Lipton, S. Eisentat, and R. DeMillo, Space and Time Hierarchies for Classes of Control Structures and Data Structures, JACM, Vol. 23, No. 4, October 1976.
23. R. Rivest, L. Adleman, and M. Dertouzos, On Data Banks and Privacy Homomorphisms, in Foundations of Secure Computation, Academic Press, Inc., 171–179, 1978.
24. A. Chandra, Efficient Compilation of Linear Recursive Programs, Stanford Artificial Intelligence Project MEMO AIM-167, April 1972.
25. D. Kfoury, Comparing Algebraic Structures up to Algorithmic Equivalence. In Automata, Languages and Programming, Ed. M. Nivat, North-Holland/Elsevier, 253–263, 1973.
26. M. S. Paterson and C. E. Hewitt, Comparative Schematology, Record of Project MAC Conference on Concurrent Systems and Parallel Computation 119–128, (1970).
27. J. Guttag, The Specification and Application to Programming of Abstract Data Types, University of Toronto, Computer Systems Research Group, Technical Report CSRG-59, September 1975.
28. H. R. Strong and S. A. Walker, Characterization of Flowchartable Recursions in Fourth Annual ACM Symposium on Theory of Computing, May 1972.
29. H. Rogers, Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.
30. K. Weihrauch, On the Computational Complexity of Program Schemata, Cornell University Department of Computer Science, TR 74-196, February 1974.
31. D. Knuth, The Art of Computer Programming, 2, Fundamental Algorithms, Addison-Wesley, 1968.
32. A. Schönhage, Real-Time Simulation of Multi-Dimensional Turing Machines by Storage, Modification Machines, Project MAC Technical Memorandum 7, MIT (1973).
33. R. E. Tarjan, Reference Machines Require Non-Linear Time to Maintain Disjoint Sets, in 9th Annual ACM Symposium on Theory of Computing, May 1977.
34. A. L. Rosenberg, Data Encodings and Their Costs, Acta Inform. 9, 273–292, (1978).
35. G. Grätzer, Universal Algebra, Van Nostrand, 1968.
36. P. Cohn, Universal Algebra, Harper and Row, 1965.
37. A. Borodin, L. J. Guibas, N. A. Lynch and A. C. Yao, Efficient Searching via Partial Ordering, submitted for publication.

Received February 9, 1978; received in final form May 30, 1980