Acta Informatica 17, 451–476 (1982)

Acta Informatica

Complexity Theory and the Operational Structure of Algebraic Programming Systems

P.R.J. Asveld* and J.V. Tucker**

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Summary. An algebraic programming system is a language built from a fixed algebraic data abstraction and a selection of deterministic, and non-deterministic, assignment and control constructs. First, we give a detailed analysis of the operational structure of an algebraic data type, one which is designed to classify programming systems in terms of the complexity of their implementations. Secondly, we test our operational description by comparing the computations in deterministic and non-deterministic programming systems under certain space and time restrictions.

0. Introduction

Algorithms are written in a *definite*, possibly high level, programming formalism $\mathscr{L}$ and are designed to compute functions on data structures belonging to a *definite*, possibly complicated, collection of data types. Here we consider the semantical problems involved in assessing the complexity of such computations in the context of small scale programming systems whose data types have been designed using the algebraic specification methods first worked out by S. Zilles [23, 31, 32], J.V. Guttag [14, 15] and, in particular, the ADJ Group [1, 2].

Typically, we shall have in mind a programming system $\mathscr{PS}$ possessing a selection of deterministic, and non-deterministic, control and assignment constructs and whose data types are characterised in a formal specification naming a set of primitive operators Σ, on different kinds of data, which satisfy a set of axioms E. We shall devise an operational view of the semantics of such programming systems for the purpose of analysing the complexities intrinsic to the computations they support and, especially, in order to make a useful

* *Present address:* Department of Mathematics and Computer Science, Delft University of Technology, Julianalaan 132, 2628 BL Delft, the Netherlands

** *Present address:* Department of Computer Studies, The University of Leeds, Leeds, LS29JT England

0001-5903/82/0017/0451/$05.20

classification of programming systems based upon the space and time resources involved in their implementation.

Of course, the central problem is how to delimit the complexity of implementing the data types underlying a system $\mathscr{PS}$. If one supposes $\mathscr{PS}$ to be implemented in some general purpose program language $\mathscr{L}$ - say, by implementing the data types of $\mathscr{PS}$ as functional procedures in which the type specifications have the status of comments - then one quickly sees the investigation sink into incidental features of the definition of $\mathscr{L}$. This is even true of those languages which support data abstraction such as CLU, see Liskov [21]. It is precisely here that the algebraic ideas about data abstraction play their elegant, and essential, rôles. Syntactically, the data types of $\mathscr{PS}$ are described by an algebraic specification (Σ, E); semantically, the data types of $\mathscr{PS}$ are grouped together and modelled by a many-sorted algebra A, unique up to isomorphism. Our concepts for the complexity of an implementation of the data types of $\mathscr{PS}$ are derived from general characteristics of *how* the semantics of (Σ, E) defines A with the result that these concepts turn out to be intrinsic invariants of data types. We set ourselves the goal of framing definitions of *polynomial time*, and *space*, *implementable data types* which are as uncontentious and useful as the concepts of finite and computable data types. That this goal can be achieved is not accidental. It is merely a reflection of one of the manifold advantages of using autonomous *specification languages* to deal with data types; in this case, semantically *concrete* implementations begin to assume certain normal forms and these reflect the *abstract* structure of the data types.

After this operational view of a programming system's data types is explained, in the first three sections, we *test* our analytical machinery by using it to compare programming systems $\mathscr{PS}$ and $\mathscr{PS}'$ sharing the same data types, but having different memory structures and deterministic and non-deterministic control constructs. For example, for general systems based upon what we call *polynomial space enumerable data types*, a class of data types containing all the polynomial space implementable types, we show that the computational abilities within polynomial space restrictions are equivalent; a result which rests on an extreme generalisation of Savitch's theorem [27] on the equivalence of deterministic and non-deterministic polynomial space bounded Turing machine computations. For $\mathscr{PS}$ and $\mathscr{PS}'$ built from *polynomial time enumerable data types* we can only confirm the existence of efficient simulations of non-deterministic control constructs by deterministic constructs when certain conditions are placed on the memory structures available in the non-deterministic system. The $P=NP$ problem for these polynomial time enumerable programming systems with unrestricted memory is shown to be reducible to the $P=NP$ problem for Turing machines.

Henceforth, it is assumed that the reader is acquainted with the ideas and technical work in the theory of algebraic data types, at least with the basic paper ADJ [2] (but the more the reader knows about specification languages, such as AFFIRM [12, 25] or CLEAR [10], the better). Only the rudiments of complexity theory are required and these can be found in the book [19].

1. Algebraic Data Types: Semantics, Specification, Implementation

Syntactically, the programming systems in which we are interested are those modelled by a pair

$$\mathscr{PS} = [(\Sigma, E), \mathscr{L}(\Sigma)]$$

consisting of algebraic specification (Σ, E) for the data types of the language, and a set of program schemes $\mathscr{L}(\Sigma)$, based upon the operator names contained in the signature Σ, which formalise the programming constructs available for the encoding of algorithms in the language. Semantically, we may model such a programming system, denotationally, by a pair

$$[A, \mathscr{L}(A)]$$

wherein A is a (single-sorted) algebraic structure of signature Σ defined by the specification (Σ, E) uniquely up to isomorphism, and $\mathscr{L}(A)$ is the set of all partial functions on A computable by the schemes of $\mathscr{L}(\Sigma)$ interpreted over A according to the rules of some "standard account" of $\mathscr{L}(\Sigma)$ computations. The only requirement on A which is worth mentioning is that it is a structure finitely generated by elements named in its signature.

Let us straightaway observe that the extent to which such a programming system $\mathscr{PS}$ represents a high-level language is determined solely by the denotational meaning A of its data types and that this is achieved by using algebraic isomorphism as the sharpest notion of semantical equivalence for data types. For example, in an algebraic manipulation language A might contain a ring of elementary analytic functions over the complex numbers, faithfully represented at a lower level by an elaborate symbolic implementation.

Now, how the schemes of $\mathscr{L}(\Sigma)$ compute in A is commonly described in terms of the combinatorial activities of a virtual machine whose states are defined using A as the value set for program variables. Perhaps the reader had such a semantics for $\mathscr{L}(\Sigma)$ in mind when we spoke of $\mathscr{L}(A)$ a moment ago. The point is that an operational view of $\mathscr{L}(\Sigma)$ relative to the structure A is not a problem: *in devising an operational view of the programming system $\mathscr{PS}$, the problem lies in settling on an operational structure of the data abstraction A.* This problem we will explore in this and the following two sections.

Our point of departure, and technical motivation, lies in the initial algebra semantics for data types created by the ADJ Group [2]. An axiomatic specification (Σ, E) for a data type distinguishes the class $ALG(\Sigma, E)$ of all structures of signature Σ satisfying the properties in E, and in order to fix a unique meaning for (Σ, E) one must assign an algebra $\mathscr{M}(\Sigma, E) \in ALG(\Sigma, E)$, unique up to isomorphism. This done, one can then say a given data type semantics A is correctly defined by a specification (Σ, E) if $\mathscr{M}(\Sigma, E) \cong A$. When (Σ, E) is an algebraic specification, $\mathscr{M}(\Sigma, E)$ can be defined to be the initial algebra $I(\Sigma, E)$ of $ALG(\Sigma, E)$, necessarily unique up to isomorphism. This is a natural step to take because it corresponds to the decision that two terms t and t' over the operator signature Σ are made *semantically identical* if, and only if, t and t' can be *proved* equal from the axioms in E; in the obvious notation,

$$\mathscr{M}(\Sigma, E) = I(\Sigma, E) \models t = t' \quad \text{if, and only if,} \quad E \vdash t = t'. \tag{1}$$

In its turn, this initial algebra $I(\Sigma, E)$ can be uniquely defined as a factor algebra of the syntactic algebra $T(\Sigma)$ of all terms over Σ because $T(\Sigma)$ is initial in the category of all Σ-algebras. Let $I(\Sigma, E) \cong T(\Sigma, E) = T(\Sigma)/\equiv_E$ where $\equiv_E$ is the unique congruence corresponding to the provability clause of (1). Now if A is a data type semantics there is a unique epimorphism $v_A: T(\Sigma) \to A$. Therefore, we can always uniquely write $A \cong T(\Sigma, A) = T(\Sigma)/\equiv_A$ where $\equiv_A$ is the congruence induced on $T(\Sigma)$ by v_A. If (Σ, E) specifies A then $\equiv_E$ and $\equiv_A$ coincide.

Many of the perplexing conceptual and technical problems to do with data types find exact expressions through this handful of algebraic ideas, and can be perspiciously studied by the meticulous dissection of programming problems [2–6] or by highly theoretical work aimed at establishing general facts [8, 9]; and this seems to be true of the problem of finding an operational structure for a data type A from which both particular and general questions about computations over A may be answered.

Now there are two operational parameters for A which are obvious and fundamental: a chosen data representation and a chosen mechanism for evaluating basic operations which together make an *implementation* of the type. To treat these parameters, and their complexity, in a general and uniform way, we focus attention on *transversals* for $\equiv_A$.

A *transversal* for $\equiv_A$ is a set of terms $\Omega \subseteq T(\Sigma)$ such that for each $t \in T(\Sigma)$ there is some $t' \in \Omega$ for which $t \equiv_A t'$ and if $t, t' \in \Omega$ and $t \neq t'$ then $t \not\equiv_A t'$. For any given resource characteristic of syntax, a transversal is meant to fix the complexity of data representation and operations in some implementation of $T(\Sigma, A)$.

The idea of a transversal originates in an algebraic implementation technique for data types defined by algebraic specifications using initial algebra semantics. Given a specification (Σ, E), the semantic and proof theoretical equivalence (1) determines an operational meaning for (Σ, E) in the shape of a *deductive term rewriting system* $\to_E$ on $T(\Sigma)$ defined by E. A transversal for $\equiv_E$ then represents a *complete set of normal forms* for the reduction rules making up $\to_E$. Thus, for an account of the complexity of an implementation of A *defined by a specification* (Σ, E) it seems reasonable to analyse the complexities involved in operating the reduction relation $\to_E$. Actually, the semantics of algebraic specifications in the AFFIRM specification and verification language is defined, in this operational way, as a rewrite system: see Musser [25]. (For an introduction and survey of research into equational replacement systems, including results on their complexity, see Huet and Oppen [20].) We will *not* need to bring in as a new parameter the specification of a data type A in order to deal with its implementation. In what follows *we make no hypotheses about the concept of implementation save that whatever its mathematical model may be it will produce some distinguished transversal for* $\equiv_A$. *And that the complexity properties of the type according to such a model will be faithfully represented in properties of that transversal.* With this understanding, we make statements about implementations of data types and their complexity, and think of the *semantical* complexity of the type through properties of the class of all implementations, all transversals.

How transversals for $T(\Sigma, A)$ can be made to characterise the operational structure of a data type A will be explained by using them to support a classification of the intrinsic complexity of a data type semantics. *Just as one can presently speak of finite or computable data types* [8, 9] *one wants to be able to speak of polynomial time or space implementable data types* because these latter concepts would determine a resource based classification of the algebraic programming systems, for example. Framing reliable definitions for such notions is a delicate matter and we shall divide the task between two sections. First, we consider the complexity of data representation and then, in Sect. 3, the complexity of the primitive operators of a type. All the results we subsequently prove about the complexity of computations on abstract data types are meant to test analytic value and reliability of our operational description of data types.

2. Normed Data Types

Measuring the complexity of computations on a data type A rests primarily on an assessment of the complexity of data from A which we invest in the concept of a *norm* on A, being a function $N_A: A \to \omega$ specially tailored to the algebra of A. Secondly, it rests on the charges made for applying the operations of A which we formalise as *charge functions* associated to the norm. If α is a program which computes on A then we might say α runs in *polynomially many steps* over A with respect to norm N_A if there is a polynomial $p_\alpha: \omega^n \to \omega$ such that for each input $a_1, \ldots, a_n \in A$, the number of *steps* involved in computing $\alpha(a_1, \ldots, a_n)$ is bounded by $p_\alpha(N_A(a_1), \ldots, N_A(a_n))$. But to speak realistically about *time* in this way we must take into account the dictates of some charge function.

The starting point for norming a data type A is some decision on charging the syntax involved in its specification represented in a norm $N: T(\Sigma) \to \omega$. From an implementation of $T(\Sigma, A)$, inducing a transversal Ω, there arises a canonical implementation norm $N_\Omega: T(\Sigma, A) \to \omega$ which in turn induces the final norm N_A on A. So in this way, $N_A(a)$ will represent the charge made on $a \in A$ as this is *determined* by an implementation of $T(\Sigma, A)$ and *measured* by N. Along similar lines the charge functions on A are created.

First, we will develop a bit of theory about norms; despite their simplicity and generality, these definitions will support quite complicated conceptual and technical discussions later on and should be mastered here and now.

A *norm* on an algebra A is a map $N: A \to \omega$ and is intended to structure the data in A by giving A a prewellorder,

$$a \leqq b \quad \text{if, and only if,} \quad N(a) \leqq N(b)$$

(cf. Fig. 2.1). Given such a norm N, to each k-ary operation σ of A is associated a *resource charge function* with respect to N, $C_\sigma: A^k \to \omega$, defined from a numerical function $r_\sigma: \omega^k \to \omega$, so that

$$C_\sigma(a_1, \ldots, a_k) = r_\sigma(N(a_1), \ldots, N(a_k))$$

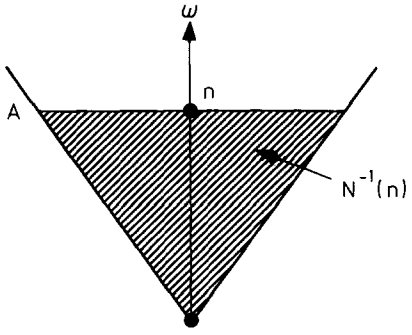


Fig. 2.1

is the cost of applying σ to $a_1, \ldots, a_k \in A$ in terms of norm N. The resource may be time or space, for example.

Being interested in asymptotic behaviour in infinite, but finitely generated, algebras prompts us to make these definitions. Let $N^{-1}(n) = \{a \in A : N(a) \leqq n\}$.

A norm $N: A \to \omega$ is *trivial* if there exists some n for which $A \subseteq N^{-1}(n)$.

A norm $N: A \to \omega$ is *finite* if for each n, $N^{-1}(n)$ is a finite set.

Notice that finite norms on infinite algebras are never trivial. We will often abbreviate $N(a)$ by $|a|_N$ or simply $|a|$.

Examples

Strings 2.1. The semigroup of words over an alphabet X is normed by string length: if $w = x_1 \ldots x_k \in X^+$ then $|w| = k$. The charge function with respect to either time or space for concatenation is defined $C(w_1, w_2) = |w_1| + |w_2|$. And $|\cdot|$ is a finite norm iff X is finite.

Arithmetic 2.2. The semiring of natural numbers is finitely normed by $|n| = n$ and by $|n| = 1 + \lceil \log_2(n) \rceil$ where the second norm measures complexity in terms of the binary representation of natural numbers. For example, for this second norm the obvious charge functions for space satisfy $C_+(n, m) \leqq 1 + \max\{|n|, |m|\}$ and $C_\times(n, m) \leqq |n| + |m|$.

Polynomials 2.3. Let R be a ring and $R[X] = R[X_1, \ldots, X_n]$ a polynomial ring over R. Then the degree function $deg: R[X] \to \omega$ is a norm and it is a finite norm iff R is finite. As a norm it is biased towards multiplication for its charge function for space satisfies $C_\times(p, q) \leqq deg(p) + deg(q)$ while addition satisfies $C_+(p, q) \leqq \max\{deg(p), deg(q)\}$ where $p, q \in R[X]$.

Polynomials Again 2.4. In a programming system for algebraic manipulation, polynomial degree is a rather pointless measure of the complexity of data in its computations because the number and size of the coefficients defining a polynomial are ignored. Assume the ring R is already defined and normed by $N: R \to \omega$. An obvious way of representing $R[X] = R[X_1, \ldots, X_n]$ is to use

arrays of elements of R so that if $p \in R[X]$ has degree d then the length of the array representing p is

$$\sum_{l=0}^{d} \binom{n+l-1}{n-1}.$$

A sensible norm $\hat{N}: R[X] \to \omega$ would be to take $\hat{N}(p)$ as the sum of the norms of the coefficients appearing in p. Clearly, if N is a finite norm then $\hat{N}$ is too.

Notice that if $N(r)=1$ for all $r \in R$ then with this trivial norm on R we have

$$\hat{N}(p) = \sum_{l=0}^{d} \binom{n+l-1}{n-1}$$

Only when $n=1$ do we "recover" polynomial degree. □

For the moment, we concentrate on norms and say nothing of their charge functions, postponing that subject to the next section and, in particular, Sect. 5.

Although a norm on an algebra A is meant to express, locally, the complexity of data in A it also expresses something, globally, about the complexity of construction of algebras:

Let A be an algebra and $N: A \to \omega$ a finite norm. The *growth function* of A with respect to N is the map $g_N: \omega \to \omega$ defined by $g_N(n) = \text{card}[N^{-1}(n)]$.

The algebra A is said to be of *polynomial growth* with respect to norm $N: A \to \omega$ if there is a polynomial $p: \omega \to \omega$ such that for all $n \in \omega$, $g_N(n) \leqq p(n)$. And A is of *exponential growth* with respect to norm $N: A \to \omega$ if there is an exponential function $e: \omega \to \omega$ such that for all $n \in \omega$, $g_N(n) \leqq e(n)$.

Example 2.5. Consider the norm $\hat{N}: R[X_1, \ldots, X_n] \to \omega$ derived from the given norm $N: R \to \omega$ as defined in Example 2.4. Assume N is a finite norm with growth function $g: \omega \to \omega$ and let $\hat{g}: \omega \to \omega$ denote the growth function of $\hat{N}$. Then a formula for $\hat{g}$ is

$$\hat{g}(k) = \sum_{l=1}^{k} \sum_{z_1 + \ldots + z_k = l} \prod_{i=1}^{k} G(z_i)$$

wherein $G(z) = \text{card}\{r \in R: N(r) = z\}$. Clearly, $\hat{g}$ is *not* bounded by a polynomial even if g is. □

It is easy to construct finite norms, say on the natural numbers, with non-exponential growth.

In order to pin down the extent to which the complexities of data representation are semantic invariants of data types we at least need to establish a criterion for the equivalence of two norms. We make the following natural choice.

Let N and M be norms on A. Then N is *linearly reducible* to M (in symbols: $N \leqq M$) if there is $\lambda \in \omega$ such that for all $a \in A$, $N(a) \leqq \lambda M(a)$. And N is *linearly equivalent* to M (in symbols: $N \equiv M$) if $N \leqq M$ and $M \leqq N$. As usual, two functions $f, g: \omega \to \omega$ are linearly reducible $f \leqq g$ if there is some $\lambda \in \omega$ so that $f(n) \leqq g(\lambda n)$ for all $n \in \omega$; and they are linearly equivalent if $f \leqq g$ and $g \leqq f$.

Lemma 2.6. *Let N and M be finite norms on A. If $N \leqq M$ then $g_M \leqq g_N$ and if $N \equiv M$ then $g_N \equiv g_M$.*

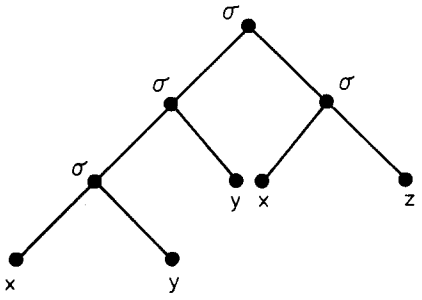


Fig. 2.2

Proof. Clearly, $N \leqq M$ implies $\{a \in A\colon M(a) \leqq n\} \subseteq \{a \in A\colon N(a) \leqq \lambda n\}$ for each $n \in \omega$. Thus $g_M(n) \leqq g_N(\lambda n)$. □

Now we can shape our measures of the complexity of the data belonging to a data type A derived from its syntactic implementations.

Norms on Syntax 2.7. Observe that the common definitions of "term height" are important norms on the syntax $T(\Sigma)$. Consider norms N_1 and N_2 which take the value 1 on the constants of Σ and are elsewhere defined inductively by

$$N_1(\sigma(s_1, \ldots, s_k)) = 1 + \max\{N_1(s_i)\colon 1 \leqq i \leqq k\},$$

$$N_2(\sigma(s_1, \ldots, s_k)) = 1 + \sum_{i=1}^{k} N_2(s_i).$$

When one parses a term one obtains a tree and $N_1(t)$ calculates (one plus) the height of this tree (the supremum of the heights of all paths in the tree) while $N_2(t)$ calculates the number of nodes in this tree. For example, if Σ contains the binary operation σ and constants x, y, z then for $t = \sigma(\sigma(\sigma(x, y), y), \sigma(x, z))$, whose tree is that in Fig. 2.2, we have $N_1(t) = 4$ and $N_2(t) = 9$.
Quite generally, for any term $t \in T(\Sigma)$, $N_2(t) \leqq M^{N_1(t)-1}$ where M is the maximum arity of the operation symbols in t. N_1 and N_2 are finite iff Σ is finite. Mathematically, the essential property of norms for syntax is this.

A norm $N\colon T(\Sigma) \to \omega$ is said to be *inductive* if for all $t = \sigma(s_1, \ldots, s_k) \in T(\Sigma)$,

$$N(t) \geqq 1 + \max\{N(s_i)\colon 1 \leqq i \leqq k\}$$

or, quite simply, $N(t) \geqq N_1(t)$. The connection between inductiveness and non-trivial syntactic norms is obvious, as indeed is the following fact:

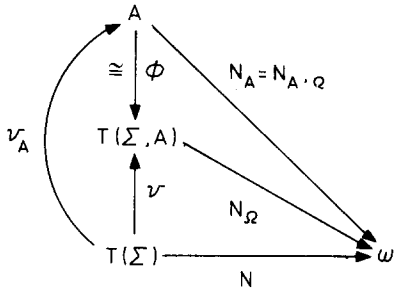
Lemma 2.8. *Let $N\colon T(\Sigma) \to \omega$ be an inductive norm. Then N_1 is linearly reducible to N. If g is the growth function of N and g_1 is the growth function of N_1 then $g_1 \geqq g$. In particular, the growth function g of N is bounded by an exponential function.* □

And so now we are able to explain our measures for data in A as they are determined by a measure of term complexity and a term model implementation of A.

Data Type Norms and Implementations 2.9. Let $N: T(\Sigma) \to \omega$ be any syntactic norm. Let A be a data type isomorphic with $T(\Sigma, A)$ by the unique map $\phi: A \to T(\Sigma, A)$. Let Ω be a transversal for an implementation of $T(\Sigma, A)$. An implementation now uniquely defines a norm $N: T(\Sigma, A) \to \omega$ by

$$N_\Omega([t]) = N(t_0) \quad \text{for that unique } t_0 \in \Omega \text{ such that } t \equiv_A t_0.$$

So let $N_A = N_{A,\Omega} = N_\Omega \circ \phi: A \to \omega$ be the *norm on A determined by implementation* Ω *from norm* $N: T(\Sigma) \to \omega$. The situation is illustrated in this commutative diagram,



Lemma 2.10. *Let* $N: T(\Sigma) \to \omega$ *be an inductive norm and let* $N_{A,\Omega}: A \to \omega$ *be a norm determined by implementation* Ω *from* N. *Then the growth function* $g = g_{A,\Omega}$ *of* $N_{A,\Omega}$ *is bounded by an exponential function.* □

Thus, it is fair to say that most naturally occurring norms for data types are finite norms of exponential growth.

Each norm $N_A = N_{A,\Omega}$ is uniquely determined on A by an implementation transversal Ω; moreover, it is uniquely determined by Ω up to algebraic isomorphism. To obtain norms on A which are full isomorphism invariants we have only distinguish special implementations:

A transversal Ω for $T(\Sigma, A)$ is said to be *minimal* or *optimal* with respect to norm $N: T(\Sigma) \to \omega$ if for each $t \in \Omega$ there does not exist a term $t' \in T(\Sigma)$ such that $t' \equiv_A t$ and $N(t') < N(t)$.

A norm N_A derived from such an optimal transversal Ω represents the most economical data representation available to any implementation of the type A as this is judged by the underlying syntactic norm N. Thus, in case $N = N_1$ we are taking a tree representation as compact as possible from the point of view of its height; in case $N = N_2$ we are taking a tree representation with a minimal number of nodes. Clearly, any two optimal transversals for a given syntactic norm define precisely the same norm on A, and this type of norm is a general isomorphism invariance.

For a fixed measure of syntactic complexity, how do the derived data representations differ between different choices of initial values for the data type? The norms on an algebra A determined by the standard norms N_1 and N_2 on $T(\Sigma)$ enjoy a rather special invariance property in this respect which we shall formulate in Lemma 2.11.

A function $h: \omega^k \to \omega$ is *semilinear* if it can be written in the form $h(x) = c + f(x)$ for $c \in \omega$ and with f satisfying this linearity condition: for all $x_1, \ldots, x_k \in \omega$ and all $\lambda \in \omega$, $f(\lambda x_1, \ldots, \lambda x_k) = \lambda f(x_1, \ldots, x_k)$.

A norm $N: T(\Sigma) \to \omega$ is *semilinear* if it is inductively defined by semilinear mappings: for each k-ary operation symbol $\sigma \in \Sigma$ there is a k argument semilinear map $c_\sigma + f_\sigma(x)$ such that if $t = \sigma(s_1, \ldots, s_k) \in T(\Sigma)$ then

$$N(t) = c_\sigma + f_\sigma(N(s_1), \ldots, N(s_k)).$$

Clearly, the norms N_1 and N_2 are semilinear.

Let A be an algebra of signature Σ and assume A is generated by two sets of generators $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_m\}$ not yet named in Σ. Let $T(\Sigma, X)$ be the algebra of polynomials over Σ in the symbols of X which we shall use to name the a_i and b_i. $T(\Sigma, X)$ is merely $T(\Sigma \cup X)$, of course.

Unicity Lemma 2.11. *Let N be a semilinear norm on $T(\Sigma, X)$. Let A be a Σ-algebra finitely generated by $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_m\}$. Then the corresponding norms on A induced by these generating sets and any pair of implementations are linearly equivalent and their growth functions are linearly equivalent.*

Proof. First we derive a lemma about semilinear norms on syntax.

Recall that a map $f: \omega^k \to \omega$ is *extensive* if for every $x_1, \ldots, x_k \in \omega$, $f(x_1, \ldots, x_k) \geqq x_i$, $1 \leqq i \leqq k$.

Composition Lemma 2.12. *Let $|\cdot|: T(\Sigma, X) \to \omega$ be a semilinear norm. Then for each $t_0(X_1, \ldots, X_n)$ and $t_i(X_1, \ldots, X_m) \in T(\Sigma, X)$, for $1 \leqq i \leqq n$, we have*

$$|t_0(t_1, \ldots, t_n)| \leqq |t_0| \cdot [q(|t_1|, \ldots, |t_n|) + c]$$

where $c \geqq \max\{c_\sigma : \sigma \in \Sigma\}$ and q is any extensive map.

Proof. This is done by induction on the structure of t_0. The basis is obvious so we consider only the induction step. Let $t_0 = \sigma(s_1, \ldots, s_k)$ and $t = (t_1, \ldots, t_n)$. Assume that lemma holds for all terms of lower complexity than t_0. The calculation runs as follows:

$$\begin{aligned}
|t_0(t)| &= |\sigma(s_1(t), \ldots, s_k(t))| \\
&= f_\sigma(|s_1(t)|, \ldots, |s_k(t)|) + c_\sigma \\
&\leqq f_\sigma(|s_1|[q(|t|) + c], \ldots, |s_k|[q(|t|) + c]) + c_\sigma \quad \text{by induction} \\
&\leqq [q(|t|) + c] \cdot f_\sigma(|s_1|, \ldots, |s_k|) + c_\sigma \quad \text{by semilinearity} \\
&\leqq [q(|t|) + c] \cdot (f_\sigma(|s_1|, \ldots, |s_k|) + c_\sigma) \\
&\leqq |t_0| \cdot [q(|t|) + c]. \quad \square
\end{aligned}$$

The proof of the Unicity Lemma 2.11 now proceeds as follows. We show $N_a \leqq N_b$. Let $w = t(a_1, \ldots, a_n) \in A$ and let $a_i = t_i(b) = t_i(b_1, \ldots, b_m)$, $1 \leqq i \leqq n$, wherein the terms chosen are from respective transversals Ω_a and Ω_b. Thus we have $|w|_a = N(t)$ and $|a_i|_b = N(t_i)$, $1 \leqq i \leqq n$. Now $|w|_b = |t(t_1(b), \ldots, t_n(b))|_b$ and by the Composition Lemma 2.12,

$$N(t(t_1(X), \ldots, t_n(X))) \leqq N(t)[q(N(t_1), \ldots, N(t_n)) + c]$$

where $X = (X_1, \ldots, X_m)$. Setting $l = [q(|t_1(b)|_b, \ldots, |t_n(b)|_b) + c]$ and substituting we get

$$|w|_b \leqq l \cdot |w|_a.$$

So for given generating sets $\{a_1, \ldots, a_n\}$, $\{b_1, \ldots, b_m\}$ there exists a constant l to linearly reduce $|\cdot|_b$ to $|\cdot|_a$. The converse reduction follows mutatis mutandis. □

We conclude this section on data representation with a sophisticated mathematical example.

Groups with Polynomial Growth 2.13. Consider a finitely generated group as a data type. Let $\Sigma = \{\cdot, ^{-1}, 1, X_1, \ldots, X_m\}$ be the signature of a group with m names for generators adjoined. The equational laws which define group structures E taken over Σ can be used to define $\mathscr{G}(m)$ the class of all m generator groups whose initial object $T(\Sigma, E)$ is $F(m)$, the free group of rank m.

In group theory, $T(\Sigma)$ is identified as the set of all finite strings over the alphabet $X = \{X_1, \ldots, X_m, X_1^{-1}, \ldots, X_m^{-1}\}$ and is implicitly normed by string length just as in Example 2.1. The *Normal Form Theorem* proved for $F(m)$ is a result which effectively assigns to each string of $T(\Sigma)$ $(= X^*)$ a word in *reduced normal form.* The set of all reduced normal forms is a transversal Ω for $T(\Sigma, E) \cong F(m)$ and indeed it is optimal with respect to the norm $N: T(\Sigma) \to \omega$. The induced norm $N_\Omega: T(\Sigma, E) \to \omega$ is what a group theorist means by *word length;* notice that $T(\Sigma, E)$ *does not* have polynomial growth with respect to N_Ω.

Repeating these constructions for $\mathscr{AG}(m)$, the class of all m generator abelian groups, leads to an initial object which *does* have polynomial growth.

Which groups in general have polynomial growth with respect to an optimal transversal's norm?

According to J. Wolf [30], if G has a nilpotent subgroup of finite index (meaning: a nilpotent subgroup $N < G$ such that G/N is finite) then G has this polynomial growth property. Thus, if G is nilpotent then G has polynomial growth; nilpotence is a generalisation of abelianness. Amazingly, if G is soluable – a generalisation of nilpotence! – and has polynomial growth then G has a nilpotent subgroup of finite index. This result is obtained by Wolf's paper together with J. Milnor [24].

From Tits' theorem, which classifies those finitely generated groups which are (isomorphic to) matrix groups, a similar *algebraic characterisation* of linear groups with polynomial growth is possible, see J. Tits [28].

It is presently an open problem of group theory as to whether or not a *finitely generated group has polynomial growth, if, and only if, it has a nilpotent subgroup of finite index.*

3. Polynomial Time and Space Implementable Data Types

What is left for us to do, to complete our operational classification of data types and, by extension, programming systems is to consider the charge functions determined by an implementation with respect to a norm on syntax. It is here that automata-based complexity theory enters in an essential way to analyse the complexity of transversals and the operations which must be performed on them in implementing a type $T(\Sigma, A)$. This is quite easy to do and naturally leads us to the important kinds of data type whose names are given to this section.

$T(\Sigma)$ is a subset of the set of all strings over the finite alphabet $\Sigma \cup \{(\ ,\ ),\ ,\}$ and as such is context-free considered as a formal language. The complexity of a transversal Ω we will identify with the complexity of Ω as a formal language within $T(\Sigma)$: it might be a regular language or an r.e. set, for example. (Indeed it need not be computable at all: if $\equiv_A$ is r.e. and Ω is r.e. then $\equiv_A$ is recursive and this means $T(\Sigma, A)$ has soluable word or term problem. Many data types with finite, equational specifications have $\equiv_A$ as r.e., but not recursive. See [8, 9] for further information on this point; obviously here we wish to stay well within the world of the computable data types.)

In modelling implementations of A it is, of course, essential to consider the complexities involved in manipulating data representations because it is these which determine the charge functions for the primitive operators of A. Let A be a data type and Ω a transversal for $T(\Sigma, A)$. Define the function $COMPOSE: \Sigma \times \Omega^{\leqq M} \to \Omega$, where M is the maximum arity of operations in Σ, by

$$COMPOSE(\sigma, s_1, \ldots, s_k) = t_0 \text{ for that unique } t_0 \in \Omega \text{ such that } t_0 \equiv_A \sigma(s_1, \ldots, s_k).$$

COMPOSE implements the operations of $T(\Sigma, A)$. If $N: T(\Sigma) \to \omega$ is a norm then the complexity of computing $COMPOSE_\sigma$ with respect to N, by means of some automaton, we define to be the charge function C_σ of $T(\Sigma, A)$ with respect to N_Ω.

We give the following basic definition assuming the reader is acquainted with the idea of (*deterministic*) polynomial time computation, see for example Hopcroft and Ullman [19].

Let $N: T(\Sigma) \to \omega$ be a norm and let A be a data type semantics. Then A is said to be a *polynomial time implementable data type with respect to* N if there exists a transversal Ω for $T(\Sigma, A)$ for which

1) The set Ω may be recognised in polynomial time within $T(\Sigma)$; and moreover the set Ω may be enumerated in polynomial time without repetitions in increasing order with respect to N. More precisely, given some norm on ω, there exists a polynomial time bounded Turing machine $e: \omega \to \Omega$ satisfying

(i) e is surjective;

(ii) if $m \neq n$, then $e(m) \neq e(n)$; and

(iii) if $m < n$, then $N(e(m)) \leqq N(e(n))$.

2) The function $COMPOSE: \Sigma \times \Omega^{\leqq M} \to \Omega$ is computable in polynomial time over $T(\Sigma)$.

3) The function $g_\Omega(n) = \text{card}[\{t \in \Omega: N(t) \leqq n\}]$ is bounded by a polynomial.

Condition (3) is equivalent to saying $T(\Sigma, A)$ has polynomial growth with respect to N_Ω, of course.

Of course, there are other functions which have a bearing on the notion of an implementation, and consequently on its efficiency, but which we have not mentioned. For example, we could define an inverse to *COMPOSE*, $DECOMPOSE: \Omega \to \Sigma \times \Omega^{\leqq M}$ by choosing one of the functions that satisfy

$$DECOMPOSE(t) = (\sigma, s_1, \ldots, s_k) \quad \text{where } s_1, \ldots, s_k \in \Omega \text{ and } t \equiv_A \sigma(s_1, \ldots, s_k)$$

and assume that the transversals considered in connection with norms $N: T(\Sigma) \to \omega$ have the property that if $COMPOSE(\sigma, s_1, \ldots, s_k) = t$ then

$N(t) > N(s_i)$ for $1 \leqq i \leqq k$ and if $DECOMPOSE(t) = (\sigma, s_1, \ldots, s_k)$ then $N(t) > N(s_i)$ for $1 \leqq i \leqq k$. But we prefer to let the definition stand on what we consider to be its three essential conditions.

Example 3.1. Abelian Semigroups

Consider a finitely generated abelian semigroup as a data type. Let $\Sigma = \{\cdot, X_1, \ldots, X_m\}$ be the signature of a semigroup with m names for generators adjoined. The equational laws which define abelian semigroups E taken over Σ can be used to define $\mathscr{AS}(m)$ the class of all m generator commutative semigroups whose initial object $T(\Sigma, E)$ is the free-abelian semigroup of rank m. The algebra $T(\Sigma, E)$ is invariably thought of as the set of all strings $\Omega = \{X_1^{\lambda_1} \ldots X_m^{\lambda_m} : \lambda_i \in \omega, 1 \leqq i \leqq m\}$ with a commutative concatenation:

$$(X_1^{\lambda_1} \ldots X_m^{\lambda_m}) \cdot (X_1^{\mu_1} \ldots X_m^{\mu_m}) = X_1^{\lambda_1 + \mu_1} \ldots X_m^{\lambda_m + \mu_m}.$$

Remembering that $T(\Sigma)$ is the free groupoid over Σ - no associativity, no commutativity - one realises this Ω *is* the obvious transversal for $T(\Sigma, E)$. Since abbreviations in complexity arguments are misleading, write $\cdot$ as the binary function symbol c and by $X_1^{\lambda_1} \ldots X_m^{\lambda_m}$ *mean*

$$c(X_1, c(X_1, \ldots \quad \ldots (c(X_m, X_m)) \ldots))$$
$$\lambda_1 + \ldots + \lambda_m - 1 \text{ times.}$$

Under the usual norm $N_2 : T(\Sigma) \to \omega$ it is easy to check that Ω is polynomial time computable, since it is a *context-free language*, and has polynomial growth as in condition (3), because $g_\Omega(n) \leqq n^m$. Explicit analysis with Turing machine will demonstrate that *DECOMPOSE* is *real-time computable* (see [26]) while *COMPOSE* is *quadratic time computable*. Thus $T(\Sigma, E)$ is a polynomial time implementable data type.

Revising these calculations for the class $\mathscr{S}(m)$ of all m generator semigroups yields its obvious transversal, that usually, and informally, written

$$\Omega = \{X_{\lambda_1}^{\mu_1} \ldots X_{\lambda_R}^{\mu_R} : 1 \leqq \lambda_i \leqq m;\ \mu_i, i \in \omega\},$$

to be a context-free language with real-time computable *DECOMPOSE*, *linear time* computable *COMPOSE*, but of exponential growth. Thus the initial object of $\mathscr{S}(m)$ is *not* such a data type.

The definition of a *polynomial space implementable data type* derives from that of polynomial time implementable data type mutato nomine; it is clearly a broader concept.

This concludes our discussion of data types in isolation.

4. Programming Constructs

The assignment and control constructs of our algebraic programming systems are modelled by various sets of deterministic and non-deterministic program

schemes $\mathscr{L} = \mathscr{L}(\Sigma)$ based upon operator signatures Σ. Assignments in $\mathscr{L}$ are of three kinds and are defined by

1) $X := Y$;
2) $X := c$ for a constant $c \in \Sigma$;
3) $X := \sigma(Y_1, \ldots, Y_k)$ for an operation symbol $\sigma \in \Sigma$.

The simplest programming system with which we deal is based upon a set of program schemes obtained by closing assignments (1), (2) and (3) under composition and the control structures **if** B **then** $*$ **else** $*$ **fi** and **while** B **do** $*$ **od**. Here B is any test of the form $Y_1 = Y_2$, $Y_1 \neq Y_2$ or $R(Y_1, \ldots, Y_k)$ where R is a basic relation from Σ. This set of schemes will be denoted by $\mathscr{F}$, the *language of well-structured flow chart programs*. This basic formalism $\mathscr{F}$ will be extended with constructs like:

arrays;
counters, i.e. special variables of auxiliary sort natural number which one can increase, decrease and test a counter variable for being zero or equal to another counter;
recursion.

Thus, we obtain the sets of schemes $\mathscr{F}\mathscr{A}$, $\mathscr{F}\mathscr{C}$ and $\mathscr{F}\mathscr{R}$ respectively. And, by combining these new facilities, we obtain languages such as $\mathscr{F}\mathscr{A}\mathscr{C}$. We intend to compute membership in sets $X \subseteq A^n$ so assume our programming languages contain the halting statements **accept** and **reject**.

The semantics of any set of program schemes $\mathscr{L}$ must be *operationally* defined over each data type A. We do this informally by assigning to any program α of $\mathscr{L}$ over A an A-register machine which we imagine to be able to process the instructions (1), (2) and (3) deterministically as they occur in the program α (cf. [29]). The operational semantics of those languages including arrays or counters is defined in the usual way, while each recursive procedure is considered to be an abbreviation of the (possibly infinite) program obtained by procedure body replacement. We must also assume the reader to be familiar with the comparative power of these programming constructs. See Greibach [13], Manna [23], Tucker [29] and the references there cited.

Introducing non-determinism into programming languages is possible in many different ways. Here we add a non-deterministic analogue of the control structure **if** B **then** $*$ **else** $*$ **fi**. This construct is **choose** $*$ **or** $*$ **ro**, and the operational semantics of this non-deterministic choice between two statements is informally defined as follows.

In a computation, when we encounter a **choose** S_1 **or** S_2 **ro** we follow both branches determined by S_1 and S_2 in parallel. Meeting an **accept** statement in some branch terminates the computation with an acceptance of the input. But meeting a **reject** statement signals only the end of the branch in which it occurs and not the end of the computation. Thus a is in the set X_α of all elements of A^n accepted by the (non-deterministic) program α if, and only if, a computation $\alpha(a)$ includes some **accept** statement. This non-determinism is that known as *breadth first with ignoring* in the studies of non-determinism of D. Harel and V.R. Pratt; see [17, 18] for details. Adding the **or**-statement to the definitions of sets of schemes, such as $\mathscr{F}$ and $\mathscr{F}\mathscr{A}\mathscr{C}$, makes their non-deterministic counter-parts $\mathscr{N}\mathscr{F}$ and $\mathscr{N}\mathscr{F}\mathscr{A}\mathscr{C}$, and so on.

Unless it is stated otherwise, $\mathscr{L}$ always denotes a set of non-deterministic program schemes. The computations of a program α in $\mathscr{L}$ over A are described in terms of states, transformations of states, and of computation trees.

A *state* of a computation of α on an input $(a_1, \ldots, a_n) \in A^n$ consists of an instruction from α (viz. the next instruction to be executed) together with a list of those variables occurring in α and their actual values in A used so far in the computation of α on $(a_1, \ldots, a_n)$.

The *initial state* of a computation $\alpha(a_1, \ldots, a_n)$ consists of the first instruction of α and the n input variables to which the inputs $a_1, \ldots, a_n$ are assigned. A state is *final* if its first entry is either an **accept** or a **reject** statement. A state s is transformed in the obvious way to a new state s' by executing the instruction in the first entry of s, and s' is called a *direct successor* of s. Note that a state s has two direct successors when the first entry in s is a **choose*or*ro** instruction. This "direct successor" relation gives rise to the *computation tree* of $\alpha(a_1, \ldots, a_n)$, the root of which is the initial state, and the sons of a node are its direct successors. A path from the root in this tree is finite if a final state occurs in this path (which will be the last state of that path). Whenever α happens to be deterministic, the computation tree is a chain. Obviously this chain is finite if, and only if, α terminates on $(a_1, \ldots, a_n)$.

5. Complexity Measures and Complexity Classes

We may now turn to complexity considerations involving time and space in our programming systems. Typically, we are computing over A with a program α in $\mathscr{L}$ and we want to measure the complexity of a computation $\alpha(a)$ for $a \in A^n$ as a function of the norm of the input $N(a)$. For example, in Sect. 2 we spoke of the *unit cost criterion for time:* the shortest distance from the root of the computation tree to a final accepting state. The *unit cost criterion for space* counts the number of data locations (corresponding to variables or counters) accessed in the computation $\alpha(a)$ as a function of $N(a)$. But neither of these are particularly refined and, indeed, that for space is trivial in programming systems which fix bounds to the number of variables appearing in computations by their programs, such as those using $\mathscr{F}$ of the previous section, but not $\mathscr{F}\mathscr{A}$. Thus we work with respect to the so-called *weighted cost criterion* and it is for precisely this reason we have carried the charge functions in Sects. 2 and 3, of course.

Under a weighted cost criterion each step is charged for the "work involved in that step". In the case of time we take the sum over all steps of a computation. The cost for a single step depends on the instruction applied and is represented in Table 5.1 for all programming constructs we use. (By $N(Y)$ we mean the norm for the element from A contained in the location named by Y.)

In the case of space the situation is slightly more involved. Again the cost of a single step depends on the instruction; the assumptions we make in Table 5.1 reflect the idea of bit-wise information transport in implementations. For assignment statements we have to store the result which requires additional space (locations corresponding to variables X_i will be charged for $N(X_i)$ units). So the space consumed in a single step consists of the space

Table 5.1. Charge functions

Instruction	Charge (Time)	Charge (Space)
$X:=Y$	$N(Y)$	$\log N(Y)$
$X:=c$	$N(c)$, $c\in\Sigma$ is a constant	$\log N(c)$
$X:=\sigma(Y_1, \ldots, Y_k)$, $\sigma\in\Sigma$	$t_\sigma(N(Y_1), \ldots, N(Y_k))$	$s_\sigma(N(Y_1), \ldots, N(Y_k))$
accept	1	0
reject	1	0
if B **then** $*$ **else** $*$ **fi**	$1+t_B$	s_B
while B **do** $*$ **od**	$1+t_B$	s_B
choose $*$ **or** $*$ **ro**	1	0
test:		
$B\equiv Y_1=Y_2$	$t_B=N(Y_1)+N(Y_2)$	$s_B=\log(N(Y_1)+N(Y_2))$
$B\equiv Y_1\neq Y_2$	$t_B=N(Y_1)+N(Y_2)$	$s_B=\log(N(Y_1)+N(Y_2))$
$B\equiv R(Y_1, \ldots, Y_k)$, $R\in\Sigma$	$t_R(N(Y_1), \ldots, N(Y_k))$	$s_R(N(Y_1), \ldots, N(Y_k))$

Counters: Assuming a given norm on ω they are treated similarly.

Recursion: We consider the corresponding (possibly infinite) program obtained by procedure body replacement, and we charge each **call** and **return** instruction for 1 with respect to time and for 0 with respect to space

required to perform the instruction (this space is reusable!) and the sum of the norms of the elements stored in all locations. Finally, we take the maximum over all states in the computation tree up to the first **accept** statement.

Since we are unable to reduce in general a resource bound by a constant factor (i.e. "linear speed up", "linear space compression") as in Turing machine based complexity theory, we define complexity classes in terms of $\mathcal{O}(f(n))$ for some resource bound $f(n)$ rather than in terms of $f(n)$ itself.

Let $f:\omega\to\omega$ be a monotonic function. Let A be a data type normed by $N:A\to\omega$ and let $\mathscr{L}$ be a set of program schemes over A. The class of all subsets of A^m, for all m, which are accepted by deterministic $\mathscr{L}$ programs within time $\mathcal{O}(f(n))$ with respect to N we denote $DTIME^A_{\mathscr{L}}(f(n))$. Similarly, we let $DSPACE^A_{\mathscr{L}}(f(n))$ designate the class of all subsets of A^m, for all m, which are accepted by deterministic $\mathscr{L}$ programs within space $\mathcal{O}(f(n))$ with respect to N. The full classes of sets recognized by $\mathscr{L}$, allowing its non-deterministic features, but still under resources bounded by $f(n)$, we denote $NTIME^A_{\mathscr{L}}(f(n))$ and $NSPACE^A_{\mathscr{L}}(f(n))$, respectively. As usual we define

$$P(A,\mathscr{L})=\bigcup_{k\geq 1} DTIME^A_{\mathscr{L}}(n^k), \qquad NP(A,\mathscr{L})=\bigcup_{k\geq 1} NTIME^A_{\mathscr{L}}(n^k),$$

$$PSPACE(A,\mathscr{L})=\bigcup_{k\geq 1} DSPACE^A_{\mathscr{L}}(n^k), \qquad NPSPACE(A,\mathscr{L})=\bigcup_{k\geq 1} NSPACE^A_{\mathscr{L}}(n^k),$$

for each $\mathscr{L}$ over A.

6. Time and Space Bounded Computations

In this section, we try to test the reliability of our definitions for polynomial time and space implementable data types by comparing the complexity classes

determined by the high-level programming systems these data types support. What sort of theorems ought to be expected? It must be remembered that our data types are *not* polynomial time, or space, implementable in any *generalised* complexity-theoretic sense. Rather, the data types are complicated general structures which can be constructed and operated in polynomial time, or space, in the *ordinary* sense. Thus, whatever results about complexity classes are obtained they must be consistent with the basic facts of life for automaton based complexity theory.

We organise the comparison theorems for the complexity classes by proving them from conditions on a general programming system which are weaker than implementability whenever this is possible. It should be emphasised that establishing other results, known in the Turing-machine based theory, might well require stronger hypotheses on the data types, but the comparisons between determinism and non-determinism do not.

To begin with, let $[A, \mathscr{L}]$ be a programming system wherein $\mathscr{L}$ is some set of deterministic or non-deterministic program schemes and A is a data type with some given finite norm $N: A \to \omega$. We denote the growth function of A with respect to N by $g: \omega \to \omega$. And henceforth $f: \omega \to \omega$ is always a monotonic function satisfying $f(n) \geqq n$ for each $n \in \omega$.

The first condition we must enforce throughout the section is one which concerns the complexity measures on a data type; it requires the charge functions to behave "properly":

Assumption 6.1. For each k-ary operation or relation symbol ξ in the signature Σ of $A (k \geqq 0)$, the corresponding time charge function $t_\xi: \omega^k \to \omega$ and space charge function $s_\xi: \omega^k \to \omega$ satisfy: there exists a natural number $\gamma_\xi \geqq 2$ such that

$$s_\xi(x_1, \ldots, x_k) \leqq t_\xi(x_1, \ldots, x_k) \leqq \gamma_\xi^{s_\xi(x_1, \ldots, x_k)}$$

for all arguments $(x_1, \ldots, x_k) \in \omega^k$. □

The following fact is now immediate from this assumption and Table 5.1.

Observation 6.2. For each instruction i, the corresponding time charge function t_i and space charge function s_i satisfy: there is a $\gamma_i \geqq 2$ such that

$$s_i(x_1, \ldots, x_k) \leqq t_i(x_1, \ldots, x_k) \leqq \gamma_i^{s_i(x_1, \ldots, x_k)}$$

for all arguments $(x_1, \ldots, x_k) \in \omega^k$ when t_i and s_i are k-ary. □

From Observation 6.2 and the fact that one cannot "visit" more space than there is time available we obtain our first expected comparison:

Proposition 6.3. *For each programming system* $[A, \mathscr{L}]$,

$$DTIME_{\mathscr{L}}^A(f(n)) \subseteq DSPACE_{\mathscr{L}}^A(f(n))$$

and

$$NTIME_{\mathscr{L}}^A(f(n)) \subseteq NSPACE_{\mathscr{L}}^A(f(n)). \quad \square$$

Clearly, for any programming system we know that

$$P(A, \mathscr{L}) \subseteq PSPACE(A, \mathscr{L}) \tag{1}$$

and

$$P(A, \mathscr{L}) \subseteq NP(A, \mathscr{L}) \subseteq NPSPACE(A, \mathscr{L}). \tag{2}$$

Our next task is to show that under certain assumptions on the data type and schemes of a programming system it is indeed the case that

$$NPSPACE(A, \mathscr{L}) \subseteq PSPACE(A, \mathscr{L})$$

(Corollary 6.5); thus, for such a system

$$P(A, \mathscr{L}) \subseteq NP(A, \mathscr{L}) \subseteq PSPACE(A, \mathscr{L}) = NPSPACE(A, \mathscr{L})$$

which is a situation familiar in Turing machine complexity. This we prove from a generalisation of Savitch's theorem to programming systems with so-called *$f(n)$-space enumerable data types.*

Let A be a data type with norm $N: A \to \omega$. Assume the basic operators of A are augmented by a constant $FIRST$ and a unary operator $NEXT$ which together enumerate A by satisfying these axioms: (i) $A = \{NEXT^n(FIRST): n \in \omega\}$; (ii) $NEXT$ is injective; and (iii) $N(a) \leqq N(NEXT(a))$ for each $a \in A$.

Now A is called *$f(n)$-space enumerable* if the charge functions C_σ^s with respect to space for all its operators, including $NEXT$, satisfy

$$C_\sigma^s(x_1, \ldots, x_k) \leqq f(\max\{N(x_i): 1 \leqq i \leqq n\}).$$

A is called *polynomial space enumerable* if A is $f(n)$-space enumerable for some polynomial function f.

Clearly, *any polynomial space implementable data type is polynomial space enumerable;* although this latter concept is quite weak it can carry an efficient deterministic simulation of non-deterministic computations:

Theorem 6.4. *Let A be a data type which is $f(n)$-space enumerable with respect to norm N, and let $\mathscr{L}$ be a set of program schemes over A which allow counters and recursion. If $X \in NSPACE_{\mathscr{L}}^A(f(n))$, then there exists a constant c depending on X such that $X \in DSPACE_{\mathscr{L}}^A(f^2(n) \cdot \log g(cf(n)))$, where g is the growth function of A. In particular $X \in DSPACE_{\mathscr{L}}^A(f^3(n))$ whenever A has exponential growth with respect to N.*

Proof. Let α be an $f(n)$-space bounded non-deterministic $\mathscr{L}$-program over A which accepts X. We may assume that before α enters an accepting state it first erases deterministically all the registers used during the computation. This modification gives rise to a finite number of accepting states.

The maximal number of different states encountered during a computation of α on an input of norm n is roughly bounded by $|\alpha| \cdot (g(c \cdot f(n)))^{f(n)}$ for some constant c depending on α and hence on X, where $|\alpha|$ is the number of instructions in α, and g is the growth function of A.

We will show that a modification of Savitch's original argument [27] as described in [7, 19] enables us to simulate α deterministically within space $\mathcal{O}(f^2(n) \cdot \log(g(cf(n))))$.

For each accepting state C_f we determine whether it can be achieved from the initial state C_0. This is done by the recursive procedure $TEST(C_1, C_2, i)$ as given in [7, p. 370]. In this procedure there are two space comsuming state-

ments, viz., the test whether either $C_1 = C_2$ or C_2 is a direct successor of C_1, and the **for**-loop that enumerates all possible intermediate states.

Now checking the equality $C_1 = C_2$ can be performed in space $f(n)$. And in determining whether C_2 is a direct successor of C_1 we need no more space than α already consumed since exactly the same (space) charge functions are involved.

Enumerating all states (in the **for**-loop) that occupy no more space than $f(n)$ can easily be programmed. Using the fact that A is $f(n)$-space enumerable, it follows that this enumeration requires at most $\mathcal{O}(f(n))$-space.

It is now a routine matter to verify that the space bound on the deterministic simulation of α is $f^2(n) \cdot \log g(cf(n))$. Clearly, $\log g(cf(n))$ is of order at most $f(n)$ if g is bounded by an exponential function. □

Corollary 6.5. *Let A be a polynomial space enumerable data type of exponential growth and let $\mathscr{L}$ be a set of program schemes which allow counters and recursion. Then in the programming system $[A, \mathscr{L}]$,*

$$NPSPACE(A, \mathscr{L}) = PSPACE(A, \mathscr{L}). \quad \square$$

Actually, for polynomial space enumerable data types with exponential growth we can improve on Theorem 6.4.

Theorem 6.6. *Let A be an $f(n)$-space enumerable data type of exponential growth and let $\mathscr{L}$ be a set of program schemes which allow counters and recursion. Then in the programming system $[A, \mathscr{L}]$,*

$$NSPACE^A_{\mathscr{L}}(f(n)) \subseteq DSPACE^A_{\mathscr{L}}(f^2(n)). \quad \square$$

Proof. In essence the argument is the same as in establishing the previous theorem except that now we are able to obtain a tighter estimate on the number of states.

We will show that the number of different states does not exceed $|\alpha| \cdot 2^{f(n)-1} c^{f(n)}$ for some constant c. (The combinatorial background material used in proving this bound can be found in e.g. [16, Chap. 4].) This in turn implies a deterministic simulation in space $\mathcal{O}(f^2(n))$.

The number of different states equals the product of the size $|\alpha|$ of α, and the number $C(f(n); g)$ of different ways we can fill at most $f(n)$ registers such that the total amount of space does not exceed $f(n)$. For sake of simplicity we write K for $f(n)$. Moreover, we consider for a while the growth function g being a parameter of C (although for a given A provided with a norm, g is fixed). Then

$$C(K; g) = \sum_{k=1}^{K} \sum_{i_1 + \ldots + i_k = K} \prod_{j=1}^{k} g(i_j).$$

The innerproduct equals the number of ways we can fill k registers such that the total space does not exceed $i_1 + \ldots + i_k$. Then we bound $i_1 + \ldots + i_k$ by K, and finally we take all possible values of k.

Since the growth function $g(n)$ satisfies $b \leqq g(n) \leqq c^n$ for some constants $b \geqq 1$ and $c \geqq 2$, for all $n \geqq 1$, we now have

$$C(K; b) \leqq C(K; g(n)) \leqq C(K; c^n).$$

For the lower bound we obtain

$$C(K;b)=\sum_{k=1}^{K}\sum_{i_1+\ldots+i_k=K} b^k=\sum_{k=1}^{K}\binom{K-1}{k-1}b^k$$
$$=b\left(\sum_{l=0}^{K-1}\binom{K-1}{l}b^l\right)=b(b+1)^{K-1},$$

while for the upper bound we have similarly,

$$C(K;c^n)=\sum_{k=1}^{K}\sum_{i_1+\ldots+i_k=K}\exp_c\left(\sum_{j=1}^{k} i_j\right)$$
$$=\sum_{k=1}^{K}\sum_{i_1+\ldots+i_k=K} c^K=c^K\left(\sum_{k=1}^{K}\binom{K-1}{k-1}\right)=2^{K-1}c^K.$$

Summarising, we obtain with $b=1$,

$$2^{f(n)-1}\leqq C(f(n);g(n))\leqq 2^{f(n)-1}c^{f(n)}.$$

Thus $C(f(n);g(n))$ is bounded by $\frac{1}{2}(2c)^{f(n)}$, in which c depends on g, and therefore on A and its norm. □

From the proof of Theorem 6.6 it follows that even for "slowly" growing data types (i.e. $g(n)<c^n$ for some $c\geqq 2$) the number of configurations is still exponential in $f(n)$. So an improvement of the $f^2(n)$-space bound in the simulation requires - even for those slowly growing data types - essentially more powerful techniques than Savitch's divide-and-conquer argument.

We shall now begin to involve time in our discussion.

Theorem 6.7. *Let A be an $f(n)$-space enumerable data type and let $\mathscr{L}$ be a set of program schemes. If $X\in DSPACE^A_{\mathscr{L}}(f(n))$, then there exist constants $c\geqq 1$ and $d\geqq 2$ depending on X such that $X\in DTIME^A_{\mathscr{L}}(\exp_d[f(n)\log g(cf(n))])$, where g is the growth function of A.*

Proof. Let α be a deterministic $\mathscr{L}$-program which accepts X within space $f(n)$. A rough bound on the number of different states occurring in an accepting computation of α on an input of norm n is $|\alpha|\cdot[g(cf(n))]^{f(n)}$ for some $c\geqq 1$, which is of order $\exp_{d'}[f(n)\log g(cf(n))]$ for some suitable $d'\geqq 2$.

Going from one of these states to another by means of any instruction i in α takes time at most

$$\max_i t_i(\ldots)\leqq\max_i \gamma_i^{s_i(\ldots)}\leqq[\max_i \gamma_i]^{f(n)}$$

(cf. Observation 6.2 and the $f(n)$-space enumerability of A).

Let d be the maximum of d' and the γ_i's. Then after consuming an amount of time greater than $\exp_d[f(n)\log g(cf(n))]$, α will enter the same state twice, and therefore α will never halt. So $X\in DTIME^A_{\mathscr{L}}(\exp_d[f(n)\log g(cf(n))])$. □

Corollary 6.8. *Let A be an $f(n)$-space enumerable data type of exponential growth, and let $\mathscr{L}$ be a set of program schemes over A. Then in the programming system $[A,\mathscr{L}]$,*

$$DSPACE^A_{\mathscr{L}}(f(n))\subseteq\bigcup_{d\geqq 2} DTIME^A_{\mathscr{L}}(\exp_d[f(n)]).$$

Proof. Due to the assumption on the norm on A, we can – as in the proof of Theorem 6.6 – bound the number of different states by $|\alpha| \cdot 2^{f(n)-1} c^{f(n)}$ for some $c \geqq 2$. From this the statement easily follows in a way similar to the proof of the previous theorem. □

Using Proposition 6.3, Theorem 6.4 and Theorem 6.7 (in that order) it is possible to simulate non-deterministic time-bounded computations deterministically. However, for those programming languages that are in $\mathcal{NFAC}$ a direct simulation turns out to be more efficient.

We take the definitions of *$f(n)$-time enumerable* and *polynomial time enumerable* data types mutatis nomine from the corresponding ideas about space resources.

Theorem 6.9. *Let A be an $f(n)$-time enumerable data type, and let $\mathcal{L}$ be a set of program schemes which is included in $\mathcal{NFAC}$. Then for each $X \in NTIME^A_{\mathcal{L}}(f(n))$, there exist constants $c \geqq 1$ and $d \geqq 2$ depending on X, such that $X \in DTIME^A_{\mathcal{FAC}}(\exp_d[f(n) \log g(cf(n))])$, where g is the growth function of A.*

Proof. The argument consists of a straightforward simulation which for a given non-deterministic program α enumerates for each input $(a_1, \ldots, a_n) \in A^n$ all possible states in the computation tree of $\alpha(a_1, \ldots, a_n)$, and searches for the shortest accepting path in that tree. To do this, the deterministic simulating language must involve counters and arrays, because there is no a priori bound on the number of different states in a computation corresponding $\alpha(a_1, \ldots, a_n)$.

The deterministic simulating algorithm β determines for each state in the computation tree of $\alpha(a_1, \ldots, a_n)$ its successor states. Each successor state s is stored temporary, after which it is compared with all previously computed (and definitely stored) different states of $\alpha(a_1, \ldots, a_n)$. When s happens to be "new", it is also stored definitely. As soon as an accepting state is encountered, β halts and accepts the input.

For storing the different states of $\alpha(a_1, \ldots, a_n)$, β uses a doubly indexed array. The first index refers to a number provided by β in order to distinguish different states; the latter index corresponds to the variable (or array-entry) as it occurs in the original program α.

The number of different states is again bounded by $|\alpha| g(cf(n))^{f(n)}$. Computing and storing a successor state takes time at most $f(n)$. And determining whether this successor state is "new" requires no more time than $|\alpha| f^2(n) \cdot [g(cf(n))]^{f(n)}$. So the total time β needs for an input of norm n is $|\alpha| [g(cf(n))]^{f(n)}(f(n) + |\alpha| f^2(n)[g(cf(n))]^{f(n)})$. Using the facts that $f(n) \geqq 1$ and that g is monotonically non-decreasing, and by increasing c appropriately, it is easy to show that this is of order $\exp_d[f(n) \log g(cf(n))]$ for some $d \geqq 2$. □

In a way similar to Theorem 6.6 and Corollary 6.8 we obtain the following:

Corollary 6.10. *Let A be an $f(n)$-time enumerable data type with exponential growth and let $\mathcal{L}$ be a set of program schemes which is included in $\mathcal{NFAC}$. Then in the programming system $[A, \mathcal{L}]$,*

$$NTIME^A_{\mathcal{L}}(f(n)) \subseteq \bigcup_{d \geqq 2} DTIME^A_{\mathcal{FAC}}(\exp_d[f(n)]). \quad \square$$

For the program constructs represented by $\mathscr{F}$ and $\mathscr{FC}$ there is for each program α a fixed bound $M=M_\alpha$ on the number of variables and counters occurring in α. Consequently, the number of computation states for such programs is bounded by $|\alpha|[g(cf(n))]^M$. Therefore we have:

Corollary 6.11. *Let A be an $f(n)$-time enumerable data type with exponential growth and let $\mathscr{L}$ be a set of program schemes included in $\mathscr{NFC}$. Then in the programming system $[A, \mathscr{L}]$,*

$$NTIME^A_{\mathscr{L}}(f(n)) \subseteq \bigcup_{\substack{M \geqq 1 \\ c \geqq 1}} DTIME^A_{\mathscr{FAC}}([g(cf(n))]^M). \quad \square$$

Corollary 6.12. *Let A be a polynomial time enumerable data type and let $\mathscr{L}$ be a set of program schemes which is included in $\mathscr{NFC}$. If A has polynomial growth then*

$$NP(A, \mathscr{L}) \subseteq P(A, \mathscr{FAC}). \quad \square$$

We shall now prove that the full abstract $P=NP$ problem for programming systems, with polynomial time enumerable data types and allowing programs which do not have restrictions on the size of memory they may access, reduces to the $P=NP$ problem for Turing machine computation. The argument is a rather straightforward adaptation of the argument for Cook's Theorem [11] which says that the satisfiability problem for formulae of the Propositional Calculus is NP-complete.

Let ${}^n\mathscr{PF}$ denote the set of all propositional formulae of the Propositional Calculus in propositional variables $P_1, \ldots, P_n$ and let 0 and 1 denote true and false respectively. Then the *satisfaction relation* for ${}^n\mathscr{PF}$ is the predicate ${}^nsat \subseteq {}^n\mathscr{PF} \times \{0, 1\}^n$ defined by

$${}^nsat(F(P_1, \ldots, P_n), x_1, \ldots, x_n) \Leftrightarrow F(x_1, \ldots, x_n) = 0.$$

Whence the *satisfiability predicate* is defined by

$${}^nSAT(F(P_1, \ldots, P_n)) \Leftrightarrow \exists\, x = (x_1, \ldots, x_n) \in \{0, 1\}^n \cdot {}^nsat(F(P_1, \ldots, P_n), x).$$

Now ${}^nsat$ is a relation which is decidable in polynomial time with respect to *formula length* and uniformly so in n. Clearly, for fixed n, ${}^nSAT$ is polynomial time decidable with respect to formula length. It is the "exponential search" as n varies which gives rise to the NP-completeness of satisfiability: Cook's theorem says that $SAT = \bigcup_{n \in \omega} {}^nSAT$ is NP-complete on $\mathscr{PF} = \bigcup_{n \in \omega} {}^n\mathscr{PF}$.

Theorem 6.13. *Let A be a polynomial time enumerable data type and let $\mathscr{L}$ be a set of program schemes which is included in $\mathscr{NFAC}$. Assume A has polynomial growth. Then for each non-deterministic $\alpha \in \mathscr{L}$ which recognises the set $X_\alpha \subseteq A^m$ in time bounded by polynomial p_α there is a reduction function r_α which maps each input $a \in A^m$ to a propositional formula*

$$r_\alpha(a) = F_{\alpha, a}$$

in $f_\alpha(a)$ propositional variables such that

$$a \in X_\alpha \Leftrightarrow {}^{f_\alpha(a)}SAT(F_{\alpha, a})$$

and these maps r_α and f_α are polynomial time computable with respect to the norm on A and formula length. Moreover, the reductions are uniform in the program α and the polynomial bounding function for its computations over A, being polynomial time computable in program length and polynomial degree.

Proof. Let α be a non-deterministic $\mathscr{L}$-program over A which accepts X in time bounded by polynomial p. Let $N(a)=n$ and suppose that $a\in X$. Then a is accepted within $p(n)$ time and there exists a sequence of computation states $C=C_1, \ldots, C_q$ with C_1 an initial state, C_q a final state and for $1\leqq t\leqq q\leqq p(n)$, $C_t\vdash C_{t+1}$ meaning C_{t+1} is a direct successor of C_t. Clearly, each C_t involves no more than $p(n)$ data locations; and not more than $g(n+p(n))=k$ distinct elements of the type A may appear in the computation C.

The formula $F_{\alpha,a}$ is made along the same lines as in the Turing machine reduction except that our propositional variables are chosen as follows:

$D(i,j,t)$ represents "the i-th data location contains the j-th element of A at time t" where $1\leqq i\leqq p(n)$, $1\leqq j\leqq k$, $1\leqq t\leqq p(n)$.

$S(l,t)$ represents "the l-th instruction of α is to be processed at time t" where $1\leqq l\leqq|\alpha|$ and $1\leqq t\leqq p(n)$.

Thus we have $p(n)\cdot k\cdot p(n)+|\alpha|\cdot p(n)=p(n)\cdot(kp(n)+|\alpha|)$ propositional variables at our disposal from which we can make propositional formulae $\Phi_1, \ldots, \Phi_6$, corresponding to the 6 statements about C given below, such that $F_{\alpha,a}=\Phi_1\wedge\ldots\wedge\Phi_6$.

1) *Each C_t has one and only one element in each location.*
2) *Each C_t has one and only one instruction.*
3) *At most one location is altered in the passage from C_t to C_{t+1}.*
4) *The transition of C_t to C_{t+1} is legal according to the instruction of C_t.*
5) *C_1 is initial.*
6) *C_q is final and marks acceptance.*

Clearly, we then have

$$\begin{aligned} a\in X &\Leftrightarrow \text{there exist } C_1, \ldots, C_q \text{ satisfying statements (1)–(6).}\\ &\Leftrightarrow \exists\, x\in\{0,1\}^B\cdot {}^B sat(F_{\alpha,a}, x)\\ &\Leftrightarrow {}^B SAT(F_{\alpha,a}) \end{aligned}$$

where $B=p(n)\cdot(k\cdot p(n)+|\alpha|)$.

The construction of these formulae is straightforward because it follows Cook's proof so closely. For example, statements (1) and (2) are based upon the fact that the mutually exclusive disjunction $U(P_1, \ldots, P_r)$ of proposition variables which when written out is a formula of length $\mathcal{O}(r^2)$, see [7]. We consider (3) as an illustration.

Define $\phi(i,j,t)\equiv D(i,j,t+1)\leftrightarrow D(i,j,t)$ and notice the length of this formula is constant. Its interpretation is "*the j-th element of A is in location i at time $t+1$ iff it was there at time t*". The formula Φ_3 is defined by

$$\Phi_3\equiv \bigwedge_{t=1}^{p(n)} \bigvee_{i_0=1}^{p(n)} \bigwedge_{\substack{i=1\\ i\neq i_0}}^{p(n)} \bigwedge_{j=1}^{k} \varphi(i,j,t)$$

and the order of its length is given by

$$p(n)\cdot p(n)\cdot(p(n)-1)\cdot k.$$

On completing such formalisations, the reader will find that the length of the formula $F_{\alpha,a}$ is of order bounded by

$$p^3(n) \cdot [g(p(n))]^M$$

where M is the maximum arity of the operations of the data type, a parameter which creeps into case (4).

We have only to check the complexity of this construction as the uniformity R: $\mathscr{L}_0 \times A^n \to \mathscr{PF}$ where $\mathscr{L}_0$ is the set of pairs of $\mathscr{L}$-programs and their bounding functions. Most of the computation $R(\alpha, a) = F_{\alpha,a}$ is work for a Turing machine on $\mathscr{PF}$ such as in Cook's proof, but it depends on the number n, which is obtained by the enumeration function; the bound p, which is given by the data; and the number k, which is given by the growth function, an invariant determined by the data type A and its norm. The hypotheses of polynomial time enumerability clearly entail that all this information is available so that $R(\alpha, a)$ is polynomial time computable in $|\alpha|$ and $N(\alpha)$. □

Corollary 6.14. *Let A be a polynomial time enumerable data type with polynomial growth and let $\mathscr{L}$ be a set of program schemes which is included in $\mathscr{NFAC}$. Then all non-deterministic computations in the programming system $[A, \mathscr{L}]$ are reducible to the satisfiability problem for propositional formulae.* □

7. Concluding Remarks

From the point of view of the general theory of program semantics, we have tried to think seriously, and in a precise mathematical way, about the algebraic semantics of high-level computations: specifically by thinking operationally of the simple minded equations,

$$\begin{aligned} \text{Data Types} &= \text{Specifications} + \text{Implementation} \\ \text{Programs} &= \text{Assignments} + \text{Control Structures} \end{aligned}$$

and fusing them together by the equation

$$\text{Algorithms} = \text{Data Types} + \text{Programs}.$$

From this point of view the complexity theory is meant as a stiff test of the semantical theory.

From the point of view of complexity theory, we have simply tried to lift all the conceptual equipment for conducting analyses of computational resources into a general algebraic setting, but without loosing sight of the fact that it is only in computations on the hard ground of syntax that any realistic measure of complexity must set down its root. Certainly most of the arguments used in Sect. 6 are routine generalisations of known techniques *once one has the conceptual equipment at hand;* perhaps Theorem 6.6 and its corollaries may be claimed to be novel. And it may be of interest to realise that such familiar ideas as those used in proving Savitch's theorem are in no sense specific to Turing machines.

In any case, it seems to us that, whatever the shortcomings in our own work reported here, it is only through the organizing framework of the ADJ Group's initial algebra methodology that some mathematical unity between models of high and low level computations can be achieved.

References

1. ADJ [Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.]: Abstract data types as initial algebras and correctness of data representations. In: Proceedings ACM Conference on Computer Graphics, Pattern Recognition and Data Structure, ACM, New York (1975), pp. 89–93
2. ADJ [Goguen, J.A., Thatcher, J.W., Wagner, E.G.]: An initial algebra approach to the specification, correctness and implementation of abstract data types. In: Current Trends in Programming Methodology IV, Data Structuring, R.T. Yeh (ed.), Prentice-Hall, Englewood Cliffs, N.J. (1978), pp. 80–149
3. ADJ [Thatcher, J.W., Wagner, E.G., Wright, J.B.]: Data type specification: parameterization and the power of specification techniques. IBM Yorktown Heights Research Report RC 7757, Yorktown Heights (1979)
4. ADJ [Ehrig, H., Kreowski, H.-J., Thatcher, J.W., Wagner, E.G., Wright, J.B.]: Parameterized data types in algebraic specification languages. In: Automata, Languages and Programming, 7th Colloquium 1980, J.W. de Bakker and J. van Leeuwen (eds.). Lecture Notes in Computer Science **85**, Springer: Berlin Heidelberg New York (1980), pp. 157–168
5. ADJ [Thatcher, J.W., Wagner, E.G., Wright, J.B.]: More on advice on structuring compilers and proving them correct. IBM Yorktown Heights Research Report, RC 7588 (1979)
6. ADJ [Goguen, J.A.]: Abstract errors for abstract data types. In: IFIP 1977 Working Conference on Formal Description of Programming Concepts. North-Holland: Amsterdam (1977), pp. 21.1–21.32
7. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley: Reading, Mass. (1974)
8. Bergstra, J.A., Tucker, J.V.: A characterisation of computable data types by means of a finite, equational specification method. In: Automata, Languages and Programming, 7th Colloquium 1980, J.W. de Bakker and J. van Leeuwen (eds.). Lecture Notes in Computer Science **85**, Springer: Berlin Heidelberg New York (1980), pp. 76–90
9. Bergstra, J.A., Tucker, J.V.: Initial and final algebra semantics for data type specification: two characterisation theorems. Report IW 142/80, Department of Computer Science, Mathematical Centre, Amsterdam (1980)
10. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. Proc. 5th Internat. Conf. on Artificial Intelligence: Cambridge, Mass. (1977), pp. 1045–1058
11. Cook, S.A.: The complexity of theorem proving procedures. Proc. 3rd Ann. ACM Symp. Theory of Computing (1971), pp. 151–158
12. Gerhart, S., Wile, D.S.: Preliminary report on the delta experiment: specification and verification of a multiple-user file updating module. In: Proc. Specifications of Reliable Software Conf., Boston, Mass. (1979)
13. Greibach, S.A.: Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science **36**, Springer: Berlin Heidelberg New York (1975)
14. Guttag, J.V.: The specification and application to programming of abstract data types. Ph.D. Thesis, University of Toronto, Department of Computer Science: Toronto (1975)
15. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informat. **10**, 27–52 (1978)
16. Hall, M.: Combinatorial Theory. Blaisdell: Waltham, Mass. (1967)
17. Harel, D.: On the total correctness of non-deterministic programs. IBM Yorktown Heights Research Report, RC 7691 (1979)
18. Harel, D., Pratt, V.R.: Nondeterminism in logics of programs. MIT Laboratory for Computer Science Research Report MIT/LCS/TM-98 (1978)
19. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley: Reading, Mass. (1979)

20. Huet, G., Oppen, D.C.: Equations and rewrite rules: a survey. Report TR CSL-111, S.R.I. International: Menlo Park, Ca. (1980)
21. Liskov, B.: CLU reference manual. MIT Laboratory for Computer Science Research Report MIT/LCS/TM-225 (1979)
22. Liskov, B., Zilles, S.: Specification techniques for data abstractions. IEEE Transactions on Software Engineering **1**, 7–19 (1975)
23. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill: London (1974)
24. Milnor, J.: Growth of finitely generated solvable groups. J. Differential Geometry **2**, 447–449 (1968)
25. Musser, D.R.: Abstract data type specification in the AFFIRM system. IEEE Transactions on Software Engineering **6**, 24–32 (1980)
26. Rosenberg, A.L.: Real-time definable languages. J. Assoc. Comput. Mach. **14**, 645–662 (1967)
27. Savitch, W.J.: Relationships between non-deterministic and deterministic tape complexities. J. Comput. Systems Sci. **4**, 177–192 (1970)
28. Tits, J.: Free subgroups in linear groups. J. Algebra **20**, 250–270 (1972)
29. Tucker, J.V.: Computing in algebraic systems. In: Recursion theory, its generalisations and applications, F.R. Drake and S.S. Wainer (eds.). Cambridge University Press: Cambridge, pp. 215–235, 1980
30. Wolf, J.: Growth of finitely generated solvable groups and curvature of Riemannian manifolds. J. Differential Geometry **2**, 421–446 (1968)
31. Zilles, S.: Algebraic specification of data types. Project MAC Progress Report **11**, MIT: Cambridge, Mass. (1974)
32. Zilles, S.: An introduction to data algebras (working paper). IBM Research Laboratory: San José, Ca. (1975)

Received November 28, 1980 / May 28, 1982