# APPLICATIONS OF COMPUTABILITY THEORY OVER ABSTRACT DATA TYPES

**J V TUCKER**

*Centre for Theoretical Computer Science*
*University of Leeds*
*Leeds LS2 9JT*
*England**

**To J W de Bakker.**

## PROLOGUE

*Prouder cities rise through the haze of time,*
*Yet, unenvious, all men have found is here.*
*Here is the loitering marvel*
*Feeding artists with all they know.*
Vernon Watkins

Amsterdam is one of the important cities of the world for computer science. It is the most important city for my own interests in the mathematical theory of programming and programming languages. It is a place where many people struggle with research of lasting value. It is always a pleasure for me to be welcomed here, and especially to celebrate Jaco de Bakker's jubilee at the CWI. Jaco is one of the architects of this city's outstanding reputation in computer science, through his personal research, encouragement of the subject and colleagues, and his judgement and belief in high standards. One cannot do better than be a worthy example to others.

I will lecture on applications of computability theory over abstract data types. I will show how this subject has significant applications in widely different areas of computer science. This is due to the fundamental nature of the concept of an abstract data type, as modelled by class of abstract algebras. It was in Amsterdam that I began to realise the beauty, and utility, of the combination of computability and algebra in computer science.

As a mathematics student at Warwick and Bristol, I was interested in computability and its applications, principally to algebra, influenced by J P Cleave and J C Shepherdson. But as a research fellow at Oslo (reading the work of E Engeler, Z Manna and others, for example) it became clear to me that to make progress with the subjects that interested me, I had to master the exciting mathematical concepts and results arising in theoretical computer science. In that period I was also encouraged by disussions with J A Bergstra, on his visits to our group of generalised recursion theorists in Oslo. He was one of the Logicians of Utrecht who had recently turned to theoretical computer science.

In January 1979, I arrived in Amsterdam excited at the opportunity of learning about the theory of programming languages, and of developing and applying my knowledge of mathematical logic in the context of computer science. The opportunity to learn was arranged by Jaco de Bakker. I wrote to him about my work, asking for assistance and advice about moving into computer science; we had not met. A mathematical logician interested in computer science was welcome in very few places in the world, but Jaco's department at the Mathematisch Centrum was one: I joined J I

* *From August 1989*: Department of Mathematics and Computer Science, University College of Swansea, Singleton Park, Swansea, SA2 8PP, Wales.

Zucker; K R Apt had recently left for Rotterdam; J A Bergstra and J W Klop joined later. And to complete the list of those who influenced me, I must mention the computer scientist R J R Back, who arrived with logical interests arising from his theory of program refinement.

I brought with me to Amsterdam one book: Fuchs' *Dutch art*. I soon acquired another: the typescript of Jaco de Bakker's *Mathematical theory of program correctness*. I studied this with help from Jeff Zucker, with whom I shared an office. Clearly all that was missing from computer science were the algebras...

## 1. INTRODUCTION

In the monograph Tucker and Zucker[88], the semantic and correctness theories of **while** programs, **while array** programs, and recursive programs established in de Bakker[80] are generalised to allow programs that compute not simply on the natural numbers but on any abstract data type. In addition, error or exceptional states are introduced into the semantics, which arise when a variable is called in a computation without first being initialised. These enhancements involved mathematical work on:

*many sorted algebras A and classes K of many sorted algebras;*
*error and exceptions in semantics and proof rules;*
*weak second order logic assertion languages;*
*theory of computable functions on A and K.*

But the basic semantical theory of programs we relied on is that in de Bakker[80]. The ideas and results involved in the two books have proved to be a significant influence on our subsequent research and teaching. We began planning our book in 1979 and owe a great debt to Jaco de Bakker for his unfailing support over those years of composition.

In this lecture I will sketch some of the subjects I am currently studying, using methods that are dependent on those in Tucker and Zucker [88], particularly on the computability theory for classes of many sorted algebras. The subjects are:

*synchronous concurrent algorithms and their application in hardware design*; and
*logic programming modules and abstract data types.*

These apparently disparate applications are part of a programme, based on abstract data type theory, that aims at the formulation and analysis of the many interesting notions of *computability*, *specifiability*, and *verifiability* that exist in differents areas of computer science.

In Section 2 some important concepts concerning abstract data types will be explained, including the technical role of classes of algebras.

Section 3 summarises the computability theory on abstract data types as it stands in Tucker and Zucker[88]; it includes the *Generalised Church-Turing Thesis* for the process of *deterministic computation* on abstract data types. This computability theory is founded upon recursive functions on algebras and classes of algebras. An important point is that it distinguishes between *primitive recursion* and *course-of-values recursion*, as these are no longer equivalent on absract algebras. The computability theory was developed to prove that important sets of program states, such as the *weakest preconditions* and *strongest post conditions* of programs and assertions, were expressible in a weak second order many sorted logical language. (At the heart of this exercise is a theorem that represents generalised recursively enumerable sets in the language.)

In Section 4 the idea of a synchronous concurrent algorithm (sca) is defined. This is a network of processors and channels that compute and communicate in parallel, synchronised by a global clock. Such algorithms compute on infinite streams of data and are characteristic of hardware. This type of computation is formalised using the course-of-values recursive functions over classes of stream algebras. The study of this and other models of these clocked algorithms, and their application, is a substantial task: it aims at a general mathematical theory of computation based on hardware. The special case of unit delays throughout the network corresponds with the use of primitive recursive functions and has been studied intensively in joint work with B C Thompson (Manchester). Significant contributions to the theory, case stuides and software have been made by: K Meinke (Manchester), N A Harman (Swansea); S M Eker and K Hobley (Leeds); and A R Martin (Infospec Computers Ltd).

In Section 5 the idea of a module appropriate for logic programming is examined. This leads to the need to distinguish between specification and computation, and to begin work on the scope and limits methods for the specification of relations on abstract data types, to complement the computation theory described in Section 3. This work with J I Zucker is a natural continuation of our studies of subjects we met in the preparation of our book.

I should also remark here that the study of some the semantic issues raised in Tucker and Zucker[88], whose roots lie in de Bakker[80], has also been continued in C Jervis [88].

## 2. ABSTRACT DATA TYPES

### 2.1 Modules and Algebras

In the theory of abstract data types, computation is characterised by a *module* whose semantics is a class K of many-sorted *algebraic structures* or *models*. A many-sorted algebraic structure or model A consists of a number of sets of data, and operations, and relations on the data. Such a structure, possibly satisfying some further properties, can be used to model semantically a *concrete implementation* of a module. A class K of structures, again possibly satisfying some further properties, can be used to model semantically the module abstractly as a *class of implementations*. For example, one standard definition of an abstract data type is that it is an isomorphism type i.e. a class of all isomorphic copies of an algebra: see Goguen, Thatcher, Wagner and Wright [78]. Among the properties of algebras that we will use to characterise meaningful or useful implementations are: *minimality, initiality, and computability* (see 2.3).

See Ehrig and Mahr [85] and Meinke and Tucker [89] for background material on algebra.

### 2.2 Modularisation

Consider the general idea that programming can be formulated as the creation of modules:

*Programming creates a program module P from a component module C in order to obtain a task module T.*

Consider a programming task in the case when the modules define implementation algebras. Suppose we want to compute a function F and a relation R on a set A using operations $f_i$ and relations $r_i$. Then the *task algebra* is (A: F,R) and the *component algebra* is (A: $f_i$ : $r_i$ ). The program may introduce a new data set B and functions $g_j$ and relations $s_j$ leading to *program algebra* (A, B: $f_i$, $g_j$, F: $r_i$, $s_j$, R). Of course this generalises to the case of many sets in the obvious way. To a user the finished product is the task algebra, and only the function F and the relation R are visible for the $f_i$, $g_j$ and $r_i$, $s_j$ are hidden. When, in the normal case, the modules define *classes* of implementation algebras we uniformise the above. These ideas are derived from those of R Burstall and J Goguen.

### 2.3 Classes of models

For the purposes of specification and verification, we expect K to be a subclass of Mod(T), the class of all models of some axiomatic theory T. Among the classes of interest are:

| | |
|---|---|
| K=Mod(T) | K={A ε Mod(T): A is finite} |
| K= {A ε Mod(T): A is computable} | K= {A ε Mod(T): A is semicomputable} |
| K={A ε Mod(T): A is initial} | K={A ε Mod(T): A is final} |

Each of these classes formalises an interesting aspect of the semantics of T. For example, consider initiality and computability. If model A is initial then it is (isomorphic to) a certain form of *standard implementation* by computer. To study the computation of F and R with respect to this K is to study the properties of a program module for F and R uniformly across all standard implementations.That a model A is computable means that it is *implementable in some way* by computer, according to the classical Church-Turing Thesis. To study the properties of F and R with respect to this K is to study the properties of a program module for F and R uniformly across all possible computer implementations.

### 2.5 Higher-order computation and logic

Many sorted algebra and logic may be employed as a unified framework for representing higher order computation and logic. For example, two forms of second order computation and logic over a structure A can be represented by applying first order computation and logic to the extended structures, $\underline{A}$ and $A^*$, being A with streams, and arrays adjoined (with appropriate evaluation operations), respectively.

**2.5.1 Augmentation of time cycles and streams** Let A be an algebra and add to the carriers of A a set T of natural numbers and, for each carrier $A_i$ of A, the set $[T \to A_i]$ of functions $T \to A_i$ or *streams* over $A_i$. To the operations of A we add the simple constant 0 and operation of successor t+1 on T, and

$eval_i : T \to A_i$ defined by $eval_i(t,a) = a(t)$.

Let this new algebra be $\underline{A}$.

**2.5.2 Augmentation of arrays** Let A be an algebra and add to the carriers of A a set of natural numbers T and the simple constant 0 and operation of successor t+1 on T. Next add an undefined symbol $u_i$ to each carrier $A_i$ of A, and extend the operations by strictness. This forms an algebra $A_u$ with carriers $A_{i,u} = A_i \cup \{u\}$. We model a *finite array* over $A_i$ by a pair $(\alpha, l)$ where

$\alpha: T \to A_{i,u}$ and $l \in T$ such that $\alpha(t) = u$ for all $t > l$.

Thus the pair is an infinite array, uninitialised almost everywhere. Let $A_i^*$ be the set of all such pairs. We add these sets to the algebra $A_u$ to create carrirers of the array algebra $A^*$. The new constants and operations of $A^*$ are: the everywhere undefined array; and functions that evaluate an array at a number address; update an array by an element at a number address; evaluate the length and update the length.

We will use both extensions of A in the next sections.The second augmentation is an enhancement of the array algebras in Tucker and Zucker [88] that we made in Tucker and Zucker [89]. There are many interesting extensions and potential applications.

## 3. COMPUTABILITY THEORY

### 3.1 Computable Functions on Abstract Data Types

In Tucker and Zucker[88] we have examined some classes of functions over an adt that are generated from its basic operations by means of

*sequential composition;*
*parallel composition;*
*simultaneous primitive recursion;*
*simultaneous course of values (cov) recursion;*
*search operators.*

These function building operations are defined by straight-forward generalisations of the classical concepts on the natural numbers to concepts over a class K of many-sorted algebras whose domains include the natural numbers; such structures are called *standard algebras*. An important class COVIND(A) of functions on A is that of the *course-of-values (cov) inductively definable functions*, formed by combining sequential and parallel composition, course of values recursion, and least number search. If course-of-values recursion is replaced by primitive recursion in this definition then the class IND(A) of functions obtained is the class of inductively definable functions. In either case, the functions are defined by a *parallel deterministic* model of computation. The simultaneous recursions, which are responsible for this parallelism, are also required when computing on many-sorted structures. The basic definitions and theory are taken from Tucker and Zucker [88] in which it is argued in detail that whilst effective calculability is ill-defined as an informal idea when generalised to an abstract setting, the ideas of deterministic computation and operational semantics are meaningful and equivalent and, furthermore, the following is true:

**Generalisation of the Church-Turing Thesis** *Consider a deterministic programming language over an abstract data type D. The set of functions and relations on a structure A, representing an implementation of the abstract data type D, that can be programmed in the language,* *is contained in the set of cov inductively definable functions and relations on* A. *The class of functions and relations over a class* K *of structures, representing a class of implementations of the abstract data type D, that can be programmed in the language, uniformly over all implementations of* K, *is contained in the class of cov inductively definable functions and relations over* K.

Much new work on translating recursions is necessary: see Simmons[88].

### 3.2 Computation on A*

For many purposes it is convenient to present the recursive functions on A using primitive recursion and the least number search operator on A*. This is possible because of the following fact

**Lemma** *Let f be a function on* A. *Then* $f \in$ COVIND(A) *if and only if* $f \in$ IND(A*).

## 4. SYNCHRONOUS CONCURRENT ALGORITHMS AND PARALLEL DETERMINISTIC COMPUTATION

A *synchronous concurrent algorithm* (sca) is an algorithm based on a network of modules and channels, computing and communicating data in parallel, and synchronised by a global clock. Synchronous algorithms process infinite streams of input data and return infinite streams of output data. Examples of scas include: *clocked hardware; systolic algorithms; neural nets; cellular automata;* and *coupled map lattice dynamical systems.*

### 4.1 A general functional model of synchronous concurrent computation.

To represent an algorithm, we first collect the sets $A_i$ of data involved, and the functions $f_i$ specifying the basic modules, to form a many sorted algebra A. To this algebra we adjoin a clock $T=\{0,1,...\}$ and the set $[T\rightarrow A_i]$ of *streams*, together with simple operations, to form a stream algebra $\underline{A}$ as in 2.5.1. This stream algebra defines the level of computational abstraction over which the sca is built.

An sca implements a specification that is a mapping of the form

$$F:[T \rightarrow A^n] \rightarrow [T\rightarrow A^m]$$

called a *stream transformer.*

The network and algorithm is then represented by means of the following method.

Suppose the algorithm consists of k modules and, for simplicity, that each module has several input channels, but only one output channel. Suppose that each model is connected to either other modules or the input streams.

Let us also suppose that each module produces an output at each clock cycle.

To each module $m_i$ we associate a total function $V_i:T \times [T \rightarrow A^n] \times A^k \rightarrow A$ which defines the value $V_i(t,a,x)$ that is output from $m_i$ at time t, if the algorithm is processing stream $a=(a_1, ...,a_k)$ from initial state $x=(x_1, ...,x_k)$. The behaviour of the algorithm is represented by the parallel composition of functions $V_1, ..., V_k$.

More precisely, suppose that each module $m_i$ has p(i) input channels, 1 output channel, and is specified by a function $f_i : A^{p(i)} \rightarrow A$. Suppose that the module $m_i$ is connected to the modules $m_{\beta(i,1)},...,m_{\beta(i,p(i))}$ or to input streams $a_{\beta(i,1)},...,a_{\beta(i,p(i))}$.

We will assume that there is a delay along the channels that is specified by functions

$$\delta_{i,j} : T \times [T \rightarrow A^n] \times A^k \rightarrow T$$

which are subject to the condition that

$$\delta_{i,j}((t,a,x),a,x) < t$$

for all t,a,x. The maps $V_i$ for i=1,...,k are defined by the following:

For any i,

$$V_i(0,a,x) = x_i.$$

For i an input module,

$$V_i\,(t,a,x) = f\,(a_{\beta(i,1)}(\delta_{i,1}(t,a,x),a,x),\ldots,\ a_{\beta(i,p(i))}(\delta_{i,p(i)}(t,a,x),a,x))).$$

For i another module,

$$V_i\,(t,a,x) = f\,(V_{\beta(i,1)}(\delta_{i,1}(t,a,x),a,x),\ldots,\ V_{\beta(i,p(i))}(\delta_{i,p(i)}(t,a,x),a,x))).$$

This represents a *course-of-values recursion* on the stream algebra $\underline{A}$.

**4.2 A constant delay model** There are several simple conditions we may impose on the $\delta_{i,j}$ that directly reflect operational properties of the module, channels or network. For instance, we may assume that a fixed constant delay $d_{i,j}$ is assigned to each channel so that

$$\delta_{i,j}\,((t,a,x),a,x)\ = \min(t - d_{i,j},\ 0)$$

and the algorithm is given by these equations for the maps $V_i$ for $i=1,\ldots,k$:

$$V_i\,(t,a,x) = x_i \quad \text{for } t < d_{i,j} + 1$$

$$V_i\,(t,a,x) = f\,(a_{\beta(i,1)}(t - d_{i,j},a,x),\ldots,\ a_{\beta(i,p(i))}(t - d_{i,j},a,x)))$$

$$V_i\,(t,a,x) = f\,(V_{\beta(i,1)}(t - d_{i,j},a,x),\ldots,\ V_{\beta(i,p(i))}(t - d_{i,j},a,x)))$$

**4.3 A unit delay model** If we take $d_{i,j} = 1$ then $\delta_{i,j}\,((t,a,x),a,x)\ = t - 1$ and we can rewrite the equations for the maps $V_i$ for $i=1,\ldots k$ :

$$V_i\,(0,a,x) = x_i$$

$$V_i\,(t+1,a,x) = f(a_{\beta(i,1)}\ (t,a,x),\ldots,\ a_{\beta(i,p(i))}\ (t,a,x)))$$

$$V_i\,(t+1,a,x) = f(V_{\beta(i,1)}\ (t,a,x),\ldots,\ V_{\beta(i,p(i))}\ (t,a,x)))$$

This is a *simultaneous primitive recursion over* $\underline{A}$.

Let us note that we have considered computation over a single algebra A and its stream algebra $\underline{A}$. *In practice* the above discussion invariably applies to a class of algebras. For example, often in the case of systolic algorithms, we design for the class of all initial (=standard) models of an axiomatisation of the integers or characters; or for some subclass of the class of all commutative rings.

**4.4 Applications to hardware of the unit delay model** Much of my research arisen from the aim to create a unified and comprehensive theory of hardware designs based on the concept of scaa and the methods and tools of algebra. To achieve this, B C Thompson and I have concentrated on the simple unit delay case which is already general enough to treat a huge number of interesting examples. The emphasis has been on case studies that evaluate practically applicable formal methods and software tools, and are useful in teaching. The programme can be divided into the following categories:

**4.4.1 Models** In addition to the functional model based on simultaneous recursive functions on $\underline{A}$, which is suited to work on specification and verification, we have considered other models in the unit delay case:

A von Neumann model based on concurrent assignments and function procedures on A; this is suited to work on programming, simulation and testing.

A directed graph model based on A; this is suited to work on architecture and layout.

Some models from each of these families have been formally defined by means of small languages, and, in particular, proved to be computationally equivalent. In formulating and classifying models of synchronous computation we are following the pattern of work associated with computability theory and which ends with a Church Turing Thesis to establish the scope and limits of parallel deterministic models of computation. But, we are also motivated by consideration of a multirepresentational environment in which it is possible to intercompile between representations depending on one's work in the design of the algorithm.

See: Thompson and Tucker [85,88,89], Thompson [87], Meinke and Tucker [88] and Meinke [88].

**4.4.2 Specification of scas and hardware.** A substantial study of the specification of scas and their role in the process of designing hardware is underway. An important contibution is a very simple mathematical theory of clocks, and retimings between clocks, based on the following notions:

A *clock* is an algebra T=({0,1,2,...}| 0, t+1). A *retiming* of clock T to clock T' is a function r: T→ T' such that (i) r(0)=0; (ii) r is monotonic; and (iii) r is surjective.

Some theoretical results have been obtained on nonlinear retimings, hierarchical design, and synchronising clocks but the the main interest remains the application of the theoretical concepts in detailed case studies of the design of correlators and convolvers; counters; uarts; computers (including RISCII and VIPER).

The emphasis in this area is on the rigorous analysis of methodological models and practical formal methods. Very general methodological frameworks, based on formally defined notions of specifications as stream transformers over many sorted algebras, and their consistent refinement, have been developed.

See: Harman and Tucker [87,88a, 88b].

**4.4.3 Derivation** The systematic and formal derivation of scas have been studied in connection with rasterisation algorithms: see Eker and Tucker [87,88,89]. However, derivation is an area that requires further work. There is a large literature on developing systolic algorithms of various kinds, but much of it is *ad hoc*, informal, and application specific. Nevertheless research by H T Kung, P Quinton, and the more formal work of C Lengauer provide us with a platform on which to build an analysis of the algorithm design process that complements our analysis of the specification design process of mentioned in 4.4.2. Studies in the transformation of scas have been initiated in connection with a theoretical analysis of compilation of functional to graph descriptions. Using equational specifications for data types and term rewriting techniques, optimising transformations for scas have been defined as preprocessors to simple, but verified, compilers: see Meinke [88].

**4.4.4 Verification of algorithms.** The functional model is beautifully suited to verification and a substantial study of the verification of scas, based on that model, is well underway. A large number of case studies of hardware, and systolic algorithms (for linear algebra and string processing) have been verified: see Thompson and Tucker [85,89]; Hobley, Thompson and Tucker [88]; and Thompson [87].

In the light of this it is tempting to create independent software tools for verification, customised to our theories and techniques. However we see that the mathematical concepts and methods are of use in *many* existing approaches to machine assisted formal verification, including those of K Hanna, N Daeche and M Gordon (*HOL*, based on Church's type theory); R Constable (*Nuprl*, based on Martin Lof's type theory); and J Goguen (*OBJ*, based on term rewriting). Thus work with a number of existing theorem provers would be more useful for demonstrating the usefulness of our tools.

Of course, the logical foundations of the mathematical techniques are based on the use of many sorted first order logic found in Tucker and Zucker [88].

The recursive functions are closely related to equational logic and algebraic specification techniques based on initial algebra semantics. (and these in turn are easily related to Horn clauses and logic programming techniques). This is the subject of much research with J A Bergstra about the power of algebraic specification techniques to define computable algebras of various kinds: see Bergstra and Tucker [79,80,83,87], for example. With a general theory of computability the relevance of some of those ideas and techniques used in the proofs is revealed more clearly; and they are found to have practical application! As part of our work for a definitive paper on the unit delay model, B C Thompson and I have been using a generalisation of one of the simplest lemmas in Bergstra and Tucker [80] which was published as Bergstra and Tucker [87].

Let program algebra $A_f$ be A augmented by all the subfunctions involved in the definition of f, obtained from its parse tree as a primitive recuirsive function in a certain straight forward way.

In the terminology of 2.2, A is the component algebra, $A_f$ is the program algebra and (A,f) is the task algebra.

Let $(\Sigma_f, E_f)$ be the signature and equations obtained by adding the corresponding names for these functions and equations obtained from the definition of the primitive recursive function f.

**Theorem** *Let* $(\Sigma, E)$ *be an algebraic specification of the component algebra* A. *Let* f *be a primitive recursive function over* A. *Then the program algebra* $A_f \equiv I(\Sigma_f, E_f)$ *and hence* $(\Sigma_f, E_f)$ *is an algebraic specification for the task algebra* (A, f).

Using a detailed proof of this fact, the functional definition of f over $\Sigma$ can be mapped or compiled into an algebraic specification $(\Sigma_f, E_f)$.

We can now work on the application of the proof of this result: we take scas, represented by primitive recursive functions over stream algebras, and map them into algebraic specifications, in preparation for machine processing.

**Corollary** *Let* $(\Sigma, E)$ *be an algebraic specification of the stream algebra* $\underline{A}$. *Let* V*be primitive recursive over* $\underline{A}$, *representing an sca. Let program algebra* $\underline{A}_V$ *be* $\underline{A}$ *augmented by all subfunctions involved in the definition of* V. *Let* $(\Sigma_V, E_V)$ *be the signature and equations corresponding with the primitive recursive function* V.

*Then the program algebra* $\underline{A}_V \equiv I(\Sigma_V, E_V)$ *and hence* $(\Sigma_V, E_V)$ *is an algebraic specification for the sca algebra* (A, V).

(Some prototype programs have been constructed and applied to scas by B C Thompson.) This material will be included in a definitive paper on the unit delay model: Thompson and Tucker [89].

I may add that this machinery provides a huge number of algebraic specifictaions that are *not* related to the stack: clocked hardware; systolic arrays; cellular automata; neural nets, for instance!

**4.4.5 Software tools.** A detailed design of a programming language and programming environment based on the von Neumann model (ii) has been undertaken, and a prototype system has been constructed. This system includes: the language, which is called CARESS (for Concurrent Assignment REpresentation of Synchronous Systems); a C compiler; a preprocessor; and an interactive tracing/debugging tool. The prototype is robust and convenient enough to have been used in undergraduate teaching. A multilingual shell for animating, editing and debugging specifications of scas has been built. With this tool, it is possible to test specifications against their scas automatically. A test compiler from a functional notation for the recursive functions to Caress has been made.

### 4.5 Toward general theory of synchronous concurrent computation

The *Generalised Church-Turing Thesis* applies to delimit the class of computable functions over *any* algebra or class of algebras: the class is identified by COVIND(A) or IND(A*). Thus we have a tool to speed us toward the goal of establishing the scope and limits of synchronous computation in hardware by devising an appropriate generalisation of the Church-Turing Thesis. This synchronous computation is further identifed with the notion of *parallel deterministic computation over abstract data types with streams*. The class of stream transformations $F:[T \to A^n] \to [T \to A^m]$ is identified by their cartesian forms $F:[T \to A^n] \times T \to A^m$ in COVIND($\underline{A}$) or IND($\underline{A}$*).

However our interest in scas, and deep involvement with their applications, requires a comprehensive and indepently justifiable theoretical foundation. Thus full generalisations of the different models are needed to allow for more complex processing elements and timing characteristics; and these models must be compared with one-another and classified by constructing compilers. A central problem is to understand *partiality* in terms of scas, which affects all aspects of the theory.

It is possible to model asynchronous nondeterministic computation in terms of synchronous deterministic computation in several ways. This was done for nondeterministic data flow by D Park, for example. The study of the nondeterminism and asynchrony as abstractions of determinism and synchrony is an important foundational task that has applications in practical modelling of hardware.

Extensive research on the functional model and its connections with equational specifications of modules, and with logic programming techniques, is necessary in order to support work on verification and software tools.

## 5. LOGIC PROGRAMMING AND SPECIFICATION

**5.1 Specification and computation in logic programming** A *logic programming language* is a language for specification *and* computation in which the means of computation is deduction in a logical system. More precisely, a program is a module that uses axiomatic theories expressed in formally defined logical languages, such as many-sorted first order logic, to define classes of implementation algebras (recall 2.1). This introduces the proof theory and as the basis for the semantics of computation; and, in particular, the model theory of logical systems as the basis for the semantics of specification. Unfortunately, the subject of proof dominates research on logic programming, mainly by work on practical deduction for implementations, and the subject of the model-theoretic semantics of specification has hardly been examined.

Clearly, to create modules we need theoretical accounts of constructing sets, defining relations, and evaluating functions. Thus research involves a range of programming paradigms and their integration: functional, relational, logic and, in its use of modules, object oriented. A relevant discussion is contained in Goguen and Meseguer [87].

**5.2 Relational, functional and logic paradigms**
The relational paradigm is for specification and the functional paradigm is for computation. The connection made in logic programming is of the following kind:

Let A be a set and R a subset of $A^{n+m}$. A *selection function* for R is a map $f:A^n \rightarrow A^m$ such that $\forall x[R(x,y) \Rightarrow R(x,f(x))]$.

The result of a logic program is the definition of a relation R and the computation of some family of selection functions for R. These constitute the task module of 2.2. A more appropriate general formulation is as follows:

*Let T be a theory and let P be a logic program with goal relation R. Then we want to interpret P in a class K of models of T in order to specify relation R and compute selection functions f uniformly over some class K.*

Thus we want to design P to be valid over a class K of algebras, and to compute one or more f such that:

$$K \models \forall x[R(x,y) \Rightarrow R(x,f(x))].$$

Note that this central problem of specifying relations and functions is a motivating problem of classical model theory: quantifier elimination. And that the problem of computing selection functions by logical deduction is a motivating problem of proof theory, and of the programs as proofs paradigm.

**5.3 Scope and limits of specification**
We have begun to study the corresponding characterisation of specification by means of relations. In Tucker and Zucker [89], the following basic question is asked and answered: *Does Horn clause computability on K, with its nondeterminism and potential for parallelism, specify all and only the cov inductively definable functions and relations on* K? The answer requires the basic step of formalising Horn clause computability over *any* structure or class of structures. It has been shown that Horn clause definability is fundamentally stronger than cov inductively definablility in general. It corresponds with an extension we have called *projective cov inductive definablility.*

In the computability theory of 3.1, we define a *semicomputable* set or relation R on A or K to be a set that is the domain of a partial computable function on A or K. It can be proved that *R is* *computable if and only if R and its complement -R is semicomputable.*

A set or relation R is *projective semicomputable* if it is a projection of a semicomputable set: there is a computable function $f:A^n \times A^m \to A$ such that for all x, $R(x) \Leftrightarrow (\exists y \in A^m)[f(x,y)\downarrow]$.

Turning to course-of-values inductive definability, with Lemma in 3.2 in mind, we find:

**Theorem** *A relation R is definable by Horn clauses, uniformly over all A* in K* if, and only if, R is projective cov inductively semicomputable over K*.*

Not every set that is projective cov inductively semicomputable over A* is cov inductively semicomputable over A*. However, in the case of classes of *minimal structures*, Horn clause computability and cov semicomputability are equivalent.

These and other results begin to clarify the sense in which Horn clauses constitute a *specification language*, for it is possible to define Horn clause "programs" over certain models that cannot be executed deterministically. When axiomatic specifications of abstract data types are are employed in formal reasoning about programs it is not always possible to avoid such structures.

### 5.4 Other characterisations

Horn clause definability is equivalent to several other characterisations of nondeterministic models for specification including *while-array with initialisation; while-array with random assignments;* and older notions such as *search computability* in the sense of Y N Moschovakis, for example. J I Zucker and I are working on a generalisation of the Church-Turing Thesis for specification to complement that for computation in 3.1. This is a more complex task: for many more details see Tucker and Zucker [89].

### 5.5 Applications

This work is relevant to the development of the concept of a *logic programming module* that generalises the abstract data type module. There is a close connection between logic programming modules and algebraic specification modules: see Goguen and Meseguer [84], Tucker and Zucker [89] and Derrick, Fairtlough and Meinke[89]. This idea of a module is, of course, many sorted. Many sorted logic programming has been studied in depth by Walther[87] and Cohn[87], motivated by theorem prover efficiencies made possible by typing. See Derrick and Tucker [88] for a general discussions of these issues.

### Acknowledgements

I would like to thank K Hobley, H Simmons, B C Thompson and S S Wainer for valuable conversations in the course of preparing this lecture.

## 6. REFERENCES

J W de Bakker, *Mathematical theory of program correctness*, Prentice Hall, 1980.

J.A. Bergstra and J V Tucker A characterisation of computable data types by means of a finite equational specification method, in J.W. de Bakker and J. van Leeuwen (eds.) *Automata, Languages and Programming, Seventh Colloquium, Noordwijkerhout, 1980,* Springer Lecture Notes in Computer Science 81, Springer-Verlag, Berlin, 1980, pp. 76-90.

J.A. Bergstra and J V Tucker Algebraic specifications of computable and semicomputable data structures, Research Report IW 121, Mathematisch Centrum, 1980.

J.A. Bergstra and J V Tucker Algebraic specifications of computable and semicomputable data types, *Theoretical Computer Science*, 50 (1987) 137-181.

J.A. Bergstra and J V Tucker, Initial and final algebra semantics for data type specifications: two characterisation theorems, *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing*, 12 (1983) 366-387.

A G Cohn, A more expressive formulation of many sorted logic, *J. Automated Reasoning* 3 (1987) 113-200.

J Derrick, M Fairtlough and K Meinke, *Horn clause model theory and its applications in logic programming*, CTCS Report 1989.

J. Derrick and J V Tucker, Logic programming and abstract data types, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), 217-219.

H Ehrig and B Mahr, *Fundamentals of algebraic specifications 1 - Equations and initial semantics*, Springer Verlag, 1985.

S M Eker and J V Tucker, Specification, derivation and verification of concurrent line drawing algorithms and architectures, pp 449-516 in *Theoretical foundations of computer graphics and CAD*, ed R A Earnshaw, Springer Verlag (1988).

S.M. Eker and J.V. Tucker, Specification and verification of synchronous concurrent algorithms : a case study of the Pixel Planes architecture. Centre for Theoretical Computer Science, University of Leeds, Report 12.88, 1988.
In P.M. Dew, R.A. Earnshaw and T.R. Heywood (eds), *Parallel processing for computer vision and display*, Addison Wesley, to appear.

J A Goguen, J W Thatcher, E G Wagner, and J B Wright, An initial algebra approach to the specification, correctness and implementation of abstract data types, pp 80-149 in *Current trends in programming methodology: IV Data structuring*, ed R T Yeh, Prentice Hall, 1978.

J A Goguen and J Meseguer, Equality, types, modules (and why not?) generics for logic programming, *J Logic Programming*, 2 (1984) 179-210.

J A Goguen and J Meseguer, Unifying functional, object-oriented and relational programming with logical semantics, Report SRI-CSL-87-7, SRI International, 1987.

N.A. Harman and J V Tucker, Clocks, retimings, and the formal specification of a UART, In G. Milne (ed) *The fusion of hardware design and verification* (Proceedings of IFIP Working Group 10.2 Working Conference), North-Holland, pp 375-396.

N.A. Harman and J.V. Tucker, The formal specification of a digital correlator I : User specification process.
Centre for Theoretical Computer Science, University of Leeds, Report 9.87, 1987.
In K. McEvoy and J.V. Tucker, *Theoretical aspects of VLSI design*, Cambridge University Press, to appear.

N.A. Harman and J V Tucker, Formal specifications and the design of verifiable computers, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), pp 500-503.

K.M. Hobley, B.C. Thompson, and J V Tucker Specification and verification of synchronous concurrent algorithms: a case study of aconvolution algorithm, In G. Milne (ed.) *The fusion of hardware design and verification* (Proceedings of IFIP Working Group 10.2 Working Conference) North-Holland, pp 347-374.

J W Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1984.

A I Mal'cev, *Algebraic systems*, Springer-Verlag, 1973.

A I Mal'cev, *The metamathematics of algebraic systems: Collected Papers 1936-1967*, North-Holland, 1971.

A.R. Martin and J.V. Tucker, The concurrent assignment representation of synchronous systems In J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds), *PARLE : Parallel Architectures and Languages Europe, Vol II Parallel languages*, Springer Lecture Notes in Computer Science 259, Springer-Verlag, 1987, 369-386.A revised and expanded edition is appears in *Parallel Computing* 9 (1988/89) 227-256.

K Meinke, A graph theoretic model of synchronous concurrent algorithms, PhD Thesis, School of Computer Studies, University of Leeds, Leeds, 1988.

K. Meinke and J.V. Tucker, Specification and representation of synchronous concurrent algorithms.
Centre for Theoretical Computer Science, University of Leeds, Report 22.88, 1988.
In F H Vogt (ed) *Concurrency* '88, Springer Lecture Notes in Computer Science, Springer-Verlag, 163-180.

K Meinke and J V Tucker, *Universal algebra* in S Abramsky, D Gabbay, T Maibaum (eds) *Handbook of logic in computer science*, OUP, to appear.

H Simmons, The realm of primitive recursion, *Archive Math Logic* 27 (1988) 117-188.

B.C. Thompson, A mathematical theory of synchronous concurrent algorithms.
PhD Thesis, School of Computer Studies, University of Leeds, 1987.

B.C. Thompson and J.V. Tucker, Theoretical considerations in algorithm design, In R.A. Earnshaw (ed), *Fundamental algorithms for computer graphics*, Springer-Verlag, 1985, 855-878.

B.C. Thompson and J V Tucker A parallel deterministic language and its application to synchronous concurrent algorithms, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), pp 228-231.

B C Thompson and J V Tucker, Synchronous concurrent algorithms, in preparation, 1989.

J V Tucker and J I Zucker, *Program correctness over abstract data types, with error state semantics*, North Holland, Amsterdam, 1988.

J V Tucker and J I Zucker, Horn programs and semicomputable relations on abstract structures, Proceedings of the *Automata, Languages and Programming 1989, Sixteenth Colloquium, Stresa,* Springer Lecture Notes in Computer Science, Springer-Verlag, 1989.

C Walther, *A many sorted calculus based on resolution and paramodulation*, Pitman, 1987.