[illegible]

7/25/68

# TOWARD AN ALGEBRA OF COMPUTATION

A THESIS

Presented to

The Faculty of the Division of Graduate

Studies and Research

by

Jesse H. Poore, Jr.

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy in the

School of Information and Computer Science

Georgia Institute of Technology

August, 1970

# TOWARD AN ALGEBRA OF COMPUTATION

Approved:

[illegible]

Chairman: Lucio Chiaraviglio

[illegible]

Visiting Member: Saul Gorn
Moore School of Electrical Engineering
University of Pennsylvania

[illegible]

Member: John M. Gwynn, Jr.

[illegible]

Member: Joseph J. Talavage

Date Approved by Chairman: October 6 1970

## ACKNOWLEDGMENTS

I would like to thank Professor V. Slamecka for creating the stimulating research community in which I wrote my thesis.

I would like to thank the members of my doctoral guidance committee, Professors Gough, Gwynn, Jensen, Talavage, and Valach, for their invaluable advice.

Professor Saul Gorn of the University of Pennsylvania discussed the proposed research with me in its early stages. His criticism of an early draft of the dissertation led to a substantial improvement of the paper. I wish to express my gratitude to Professor Gorn for his continued interest and assistance.

Professor Lucio Chiaraviglio must be credited with the perception and insight that led me from vague notions to the theory contained herein. I would like to thank Professor Chiaraviglio for his many contributions of technical expertise, for his professional and gentlemanly conduct of the doctoral program, and for his intellectual stimulation, guidance, encouragement, and companionship.

The work reported herein has been sponsored in part by the National Science Foundation Grant GN-655. This support is gratefully acknowledged.

TABLE OF CONTENTS

## LIST OF TABLES

# CHAPTER I

# INTRODUCTION

A program for the development of a mathematical science of computation was begun in 1963 by J. McCarthy with his publication of two papers [13,14] which expounded the basis and aspirations of such a theory. The second step in the program was made in 1967 by R. J. Orgass and F. B. Fitch [17,18]. Orgass and Fitch presented a formal system of combinatory logic and demonstrated its adequacy for representing all the entities to be studied by a theory of computation. They established the linguistic base. This paper suggests that the third step is the explication of the algebraic structures that ensue from the linguistic formalization. At the level of algebraic structures the relationship of computation to other areas of mathematics will become clear. Furthermore, it will become possible to study the models of interesting linguistic systems with known and powerful algebraic methods. We proceed to recount the first two developments and to motivate the third.

## A Basis for Computation

The basis proposed by McCarthy included a statement of the objects of study of the proposed science and a sample of the kinds of theorems about these objects that an appropriate theory would yield. Problems, procedures, data spaces, programs, programming languages and computers were the objects of study and they were related to each other in [13] as follows. A problem was defined by its solution acceptance criterion; a procedure together with a data space prescribed a method of meeting the acceptance test, hence solving the problem. Procedures and data spaces were to be defined in a precise manner after the fashion of the construction of primitive recursive functions so that complex procedures were to be constructed from elementary ones and similarly for data spaces. The relationship between them was that procedures are operations closed on data spaces. Programs are linguistic expressions of procedures and, of course, programming languages are the linguistic complexes which govern the construction of programs. Computers are finite automata which execute programs. These were the objects of study of computation according to the proposal.

We wish to prove theorems about the relationships that exist between categories of objects and among the objects of a given category. A specific problem can be solved by any one of several procedures, each of these may be realized in any one of several programming languages and each of these may be executed on a wide selection of computers. This wealth of possibilities leads us to seek a theory of computation with sufficient power to prove theorems of the following kinds:

(1) Problems of the type $T_1, T_2, \ldots, T_n$ can be solved only by procedures having properties $P_1, P_2, \ldots, P_m$.

(2) Programming language $PL_i$ is capable of expressing procedures having properties $P_1, P_2, \ldots, P_m$.

(3) Procedure $P_i$ translates programs written in $PL_i$ to equivalent programs in $PL_j$.

(4) Programming language $PL_i$ can express procedures that computer $C_j$ cannot compute.

(5) Computer $C_i$ can compute everything that computer $C_j$ can compute.

All of these statements are qualitative but of course we wish eventually to have a quantitative theory. Within a quantitative theory one could state various criteria of optimality and would be able to state precisely how one procedure compares with another; how one computer compares with another, etc. All of McCarthy's originally stated aspirations for the science of computation are included in the above examples.

Many papers were produced in the wake of McCarthy's clarification of the scope of computation, each of them attempting to deal with an isolated problem or sub-area. For example, the problem of proving the equivalence of two procedures expressed in a canonical notation has been dealt with, for several canonical notations. Programming languages were formulated in such a way that proofs about programs expressed in them could be devised. Mathematical procedures and formalisms from recursive function theory have been construed as programming languages and machines in the hope of obtaining guidance in the development of programming languages. Notable among these are the Markov algorithms, Post systems, Thue systems, and the lambda calculi. Studies in the syntax and semantics of programming languages are numerous. Indeed, there is one for every type of language now in use as well as several that were designed especially for the occasion. One may read a documented account of these efforts in the introduction to the dissertation of R. Orgass [15] and in a survey of this field compiled by J. G. Sanderson [22].

## A Formal System for Computation

Orgass recognized the need for a unifying treatment of the syntax and semantics of programming languages. In the course of achieving a theory in which both syntax and semantics could be formulated, Orgass chose a formulation that included both the classical theory of computation and the ability to model computing machines and allied utilities. The chosen system is combinatory logic based on the minimal logic of Fitch [3,4]. Thus the system is basic in the sense of Fitch. This means that every system of logic is definable within it. The original intent of this capability was recovery of the systems of logic developed from *Principia Mathematica* through the 1930's. The present, added significance of this capability is that the various objects and processes of computation can be obtained in the combinatory system. Within this system of combinatory logic, Orgass develops the following: (1) A representation of natural numbers, (2) a representation of each partial recursive function, (3) a representation of each n-ary predicate in the Kleene-Mostowski hierarchy, (4) a representation of each primitive recursive, recursive and recursively enumerable relation among formulas of the system, (5) a representation of ordered n-tuples, (6) representations of computer memories, instructions, central processors and computations, and (7) a representation for each programming language and, for several concepts of the meaning of a program, the relation of a program to its meaning. Hence, we now have an adequate and uniform formal description of the objects of interest in a science of computation.

In [16] Orgass demonstrates the manner in which his theory deals with problems of practical interest. Several notions of equivalence of computations, equivalence of programs and of the relation of a program to the function it causes a computer to compute are formulated. In each case the system of combinatory logic is shown to be negation complete. There is a formula in the system that is a theorem just in the case that the relation holds and its negation is a theorem just in the case that the relation fails to hold. The approach used is general. The results that are obtained hold for classes of computers and programming languages. Thus, this theory is of much greater interest than the parochial results obtained heretofore.

The work of Orgass provides us with a universal locus and logic for the problems of computation. The importance of this step forward in the establishment of a science of computation cannot be overestimated. We are now in the position of being able to identify a precise logical nature for the objects of computation and their interrelations. Nevertheless, there are some unavoidable limitations of any program of representation since we must still deal with the individual problems or class of problems in a piecemeal fashion. A problem or class of problems is given a combinatory *alter ego* and then we seek a solution to the problems in their combinatory setting. There may be a wide variety of combinatory settings for such problems and a wide variety of combinatory tools for their solution. A level of uniformity has been gained since settings and tools are all combinatory. Yet it may be possible to take a further unifying step. We here propose to take such a further step by characterizing the algebra of a portion of combinatory logic that seems to offer an adequate setting and furnish reasonable tools for the solution of a wide range of problems of computation.

## An Algebra for Computation

We call a theory of the character described by McCarthy a predictive theory, for obvious reasons. The theory developed in this paper is a suggestive theory in that it suggests a continuing development that shows a great deal of promise for leading to a fully predictive theory. Suggestive and predictive theories are also characterized by their qualitative and quantitative properties, respectively. Thus, we cannot actually solve any of the problems posed earlier; however, we can show them in a setting in which they will be solvable when the necessary measures are developed. It is an important aspect of the setting that it is conducive to a development of measures. Our approach to the development of this theory is as follows.

Every postulate system determines a class of structures. Each member of a structure is a model of the system. It is now a workable thesis that all the intuitive problems of computation can be expressed in a combinatory logic. If we could characterize an algebra of combinatory logic, then all the models of this logic would be at hand and we could use the resources of algebraic theory and model theory to provide us with those broad and general results that characterize an ideally mature science.

As a matter of fact, the algebraic structure of combinatory logic has not yet been explicated. However, for the purposes of the science of computation it may not be necessary to have the algebra of all of combinatory logic. If certain economies can be realized in the linguistic base and the portion of combinatory logic that is necessary and sufficient to the study of computation can be identified, then only the algebra of this portion must be determined. One may hope that such a restriction of the algebraic problem will expedite its solution.

This hope has been realized and is reported in [19,20]. Some of the details of this work will be brought out in the next section; at this time it is necessary only to say that a subset of combinatory logic has been proved necessary and sufficient for the study of hardware structures and that a similar subset has been proved necessary and sufficient for the study of software structures. The strategy in this dissertation is to determine the algebra appropriate to the distinguished subset of combinators. This algebra is developed in Chapter III and shown to be a transformation Boolean algebra with non-increasing, normed, additive, and idempotent operators. The contribution that this dissertation makes to information and computer science is the explication of the algebra of computation.

## Plan of Presentation

Chapter II is devoted to the presentation of the Boolean and combinatory basis on which this research rests. The broadest setting within which this research takes place is now termed the theory of models. The character of the theory of models is presented. Within the theory of models, the development of the theory of Boolean algebras with structure preserving operators is outstanding. Combinatory logic is fundamental to this research program but it does not appear explicitly in the algebraic development. Therefore, the presentation here is directed toward the representation of hardware and software structures which are part of the mathematical basis. The combinatory basis is completed with the sections on the representation of hardware and software facilities in combinatory logic.

The algebras of computation and computers are developed in Chapter III. The appropriate concepts in Boolean duality which lead to computation spaces are also developed there.

Chapter IV contains the concluding remarks. Some limitations to the algebraic approach have been found and these are discussed. We also discuss the advantages of the algebraic approach and make some suggestions as to how the limitations might be overcome. A number of possible further algebraic studies are suggested. The chapter is concluded with a brief summary of the motivation, the algebraic development, and the results.

# CHAPTER II

# BOOLEAN AND COMBINATORY BASIS

## Boolean Algebras with Operators

The theory of models (see Tarski [23]) is a part of the semantics of axiomatic systems. Each axiom system determines a set of sentences and every mathematical system in which every provable sentence of the axiom system holds is a model of the axiom system. Thus, a class of models is associated with each axiom system. An objective of the theory of models is to transfer linguistic problems to a mathematical setting so that mathematical methods can be employed in solving the problems. Formal properties of the axiom system become structural properties of the models and the mathematically determined properties of the models lead to formal properties of the linguistic systems.

Tarski [11,12] points out that several significant new algebras have been obtained in recent years via the generalization of models developed for specific formal systems. For example, Halmos [6,7] obtained polyadic algebras as the structures of the predicate calculus and Tarski [9,10] obtained cylindric algebras for the same system. Other examples are the development of closure algebras, projective algebras and relation algebras. All of these systems have the same algebraic structures; they are Boolean algebras with operators. The algebraic structures appropriate to computation are shown in the next chapter to also be Boolean algebras with operators.

Each of these algebras presents its developer with the mathematical obligation of establishing its identity through a representation theorem. In the work that follows we present two representation theorems. The first (Theorem 19) is the natural representation theorem in that it establishes a theory of models for the abstract algebras in question on the basis of the paradigm which motivated the algebras. The second (Theorem 26) is a representation that follows on Boolean considerations only. By combining the results of Jonsson and Tarski [11, 12] in a general study of Boolean algebras with hemimorphic operators, and of Halmos [8] in the general theory of Boolean duality, with the representation theory for Boolean algebras, we were able to obtain the topological dual of the algebras developed in the next chapter. This topological dual identifies the algebras in question in the larger mathematical setting.

Throughout this thesis the well-known developments in the theory of Boolean algebras are assumed.

## System of Combinatory Logic

The system of combinatory logic used in the representation of hardware and software structures is completely formalized and applicative and is presented in the format prescribed in Curry and Feys [2].

### Combinatory System H

A. Objects (ob)

(1) The primitive objects are constants S, K, and perhaps others.

(2) The primitive operation is application and is indicated by juxtaposition.

(3) If a and b are obs then (ab) is an ob. (Association is to the left unless explicitly indicated.)

B. Elementary Theorems

(1) The axioms are:

(i) $a = a$ for any ob a.

(ii) $Sxyz = xz(yz)$ where x, y, z are indeterminants with respect to H.

(iii) $Kxy = x$ (x,y, and z as above).

(2) The rules of inference, for obs a, b, and c, are:

(i) If $a = b$, then $b = a$.

(ii) If $a = b$ and $b = c$, then $a = c$.

(iii) If $a = b$, then $ac = bc$.

(iv) If $a = b$, then $ca = cb$.

The system that includes among its objects the indeterminants of H is called an object extension of H. The definition of such an extension must list the indeterminants with the primitive objects and extend the domain of applicability of the axioms and rules to the new set of atoms. Also, the principle of extensionality must be included in the rules of inference.

Principle of Extensionality. If x does not occur in a or b, then $ax = bx$ implies $a = b$ where x is an indeterminant with respect to H and a and b are obs over the extension.

Equality is used in the sequel in the sense of extensional equivalence for the system H.

Some applicative formal systems have the property of combinatorial completeness which asserts for each function definable intuitively via substitution of constants for indeterminants the existence of a formal object to which the intuitive function is congruent. In Curry and Feys it is proved that the system H has the property of combinatorial completeness. Thus, the intuitive functions are defined in terms of S and K. The following is a list of the usual combinators, which represent basic functions, which facilitate manipulation of combinators. Explicit definition is indicated '=df'.

1. I =df (SKK)
2. 0 =df (KI)
3. W =df (SS0)
4. B =df (S(KS)K)
5. C =df(S(BBS)(KK))
6. T =df(CI)
7. N =df (CI0)
8. V =df W(BC(BW(B(BB(C(BB(BWN))N)))))

For any combinator X the following recursive definition schemata hold.

9. $X^0$ =df I

   $X^{n+1}$ =df $(BXX^n)$

10. $X_{(0)}$ =df X

    $X_{(n+1)}$ =df $(BX_{(n)})$

11. $X_{[0]}$ =df I

    $X_{[n+1]}$ =df $(B(BX_{[n]})X)$

12. $X^{[0]} =_{df} I$

$X^{[n+1]} =_{df} \{BX(BX^{[n]})\}$

13. $L^{(n)} =_{df} C_{[n]}I$

14. $X^{(n)}_{\|0\|} =_{df} L^{(n)}$

$X^{(n)}_{\|k+1\|} =_{df} C^{[2k]}(B^{2}_{(2k)}X^{(n)}_{\|k\|})X$

15. $X^{(n)}_{|0|} =_{df} X^{(n)}_{\|n\|}$

$X^{(n)}_{|k+1|} =_{df} C^{[k]}X^{(n)}_{|k|}$

## Combinatory Definability of Hardware and Software

Poore, Baralt, and Chiaraviglio [19,20] have exhibited the subsets of combinatory logic that are appropriate to the study of hardware and software structures. In order to convey the significance of the Hardware Definability Theorem of [19] and the Software Definability Theorem of [20] as expeditiously as possible, a number of definitions and procedural details are suppressed in the present treatment.

Consider now several combinators that are defined intuitively. Since the underlying system of logic is combinatorially complete, all of the following combinators have formal definitions which may be found in the above cited papers. Using the notational abbreviations '$\bar{x}$' for '$x_1x_2\ldots x_n$' for '$(((x_1x_2)x_3)\ldots x_n)$' we have

$$L^{(n)}\bar{x}a = a\bar{x},$$

$$N^{(k,n)}L^{(k)}\overline{(L^{(n)}\bar{x})} = L^{(k)}\overline{(L^{(n)}\overline{Nx})},$$

$$W^{(k,n)}\left(L^{(k)}\overline{(L^{(n)}\bar{x})}\right)\left(L^{(k)}\overline{(L^{(n)}\bar{y})}\right) = L^{(k)}\overline{\left(L^{(n)}(\overline{Wxy})\right)}$$

Now let

$$G^{(k,n)} = \left\{L^{(k)}\overline{(L^{(n)}\bar{x})}\,\middle|\,x_i\epsilon\{0,I\}\right\}$$

and we can state Theorem 1.

Theorem 1. $(G^{(k,n)},W^{(k,n)},N^{(k,n)})$ is a Boolean algebra isomorphic to the Boolean algebra $\left((\{0,1\}^n)^k,+,'\right)$.

The following are explicit definitions within the combinatory system that describe the combinators central to the definability of hardware and software.

16. $D =df\ L^{(k)}\overline{(L^{(n_j)}\bar{x})}$

17. $M_j =df\ L^{(n_j)}\bar{x}$

18. $H_j =df\ C^{[2]}\left(T(L^{(n_j+1)}V^{(n_j)}_{|n_j|})\right)(W_{[n_j]}0)$

19. $Z_s =df\ L^{(k)}\overline{(H_jM_j)}$

20. $E =df\ N_{(2)}\left[C^{[2]}\left(L^{(2)}(L^{(k+1)}I^{(k)}_{|k|})\right)\right](W_{[k]}0)$

21. $Y_t =df\ L^{(m_t)}\overline{(EZ_s)}$

22. $Q_t =df\ T\left[S_{[m_t]}\left(K(W_{[m_t]}0)\right)\right]$

23. $X_i$ =df $L^{(h_i)}\overline{(Q_t Y_t)}$

24. $G_i$ =df $T\left(S_{[h_i]}(KL^{(h_i)})\right)$

25. $F$ =df $L^{(\ell)}\overline{(G_i X_i)}$

26. $A$ =df $T\left(S_{[\ell]}(KL^{(\ell)})\right)$

27. $P$ =df $AF$

Six parameters are identified in the above list of explicit definitions as follows: (1) $k$ is the length of an input sentence, (2) $n_j$ is the length of a jth word of an input sentence, (3) $\ell$ is the length of an output sentence, (4) $h_i$ is the length of an ith output word in an output sentence, (5) $t$ is the tth bit position in a word of an output sentence, and (6) $m_t$ is the number of min-terms of which the tth bit of the output is a function.

It is clear from Theorem 1 that if in explicit definition 16 the $n_j$ are constant, then D is the schema for elements of the $(0^n)^k$ algebra. The following theorem is then proved in [19].

Theorem 2. Every transformation from $(0^n)^k$ to $(0^n)^k$ is represented by a combinator of the form P provided $k = 1$ and $n_1 = \ldots = n_k = h_1 = \ldots = h_k$.

If a hardware facility, i.e., a computer, is taken to be a set of states together with a set of state transition functions, then the Theorems 1 and 2 combined are the hardware definability theorem.

Theorem 3 (Hardware Definability Theorem). Every hardware facility is uniquely definable in a set of combinators $G^{(k,n)}$ which represent the states and in combinators of the form P, with the parameters appropriately restricted, which represent state transition functions.

Software structure refers collectively to data structures and programs. Intuitively, trees, lists, matrices, etc. are particular data structures. These are included in the more general notion adopted here that data structures are relations on finite sets the fields of which may again be data structures. In particular, we are interested in data structures on the Cartesian product $0^{n_1} \times 0^{n_2} \times \ldots \times 0^{n_m}$ where $0 = \{0,1\}$. The extensions of programs are functions from one data structure to another. These concepts are defined in [20] as follows.

Definition 1 (Data Structures). Let J be any finite subset of the set of all d for which there are positive integers $n_1, n_2, \ldots, n_m$ such that d is a subset of $0^{n_1} \times 0^{n_2} \times \ldots \times 0^{n_m}$. Then the set of all data structures of depth two, T, is given by

$$T = \{D : D = \cup J \text{ for some } J\}.$$

Definition 2 (Programs). The extensions of programs on data structures of depth two is given by

$$P = \{p : p : D_1 \rightarrow D_2, \text{ for some } D_1 \text{ and } D_2 \text{ in } T\}$$

If p is the extension of a program with domain $D_1$ and range $D_2$, then the length $\ell$ of the longest sentence and the length m of the longest word occurring in $D_1 \cup D_2$ can be found. There exists a one-to-one code assignment c which maps $D_1 \cup D_2$ into the set of states $(0^w)^{\ell}$. Then $c(D_1)$ and $c(D_2)$ are subsets of $(0^w)^{\ell}$ and the function p is mirrored in the partial state transition function $cpc^{-1}$. Clearly, the combinators of explicit definition 27, with the parameters unrestricted, represent programs in extension while the combinators of 16 represent elements of data structures.

<u>Theorem 4 (Software Definability Theorem)</u>. Every data structure in T has a unique combinatory representation and every extension of a program is uniquely represented by a combinator of the form P and conversely.

# CHAPTER III

# THE ALGEBRAS OF COMPUTATION AND COMPUTERS

## Introduction

The work summarized in the last section shows that the combinators of interest are tied to Boolean functions which are ultimately definable in terms of the one-bit reset functions and the bit Boolean operations. We there took hardware structures to be pairs $\left((0^n)^k, G\right)$ formed by all the k-long vectors of 0,1-valued functions of n-variables together with a selection G from all the mappings of $(0^n)^k$ into $(0^n)^k$. Software structures differed from hardware structures in that they could be composed of vectors of non-uniform lengths of 0,1-valued functions of non-uniform numbers of variables and, as a consequence, the set of mappings that represented extensions of programs would be defined on such non-uniform vectors.

Hardware and software structures so conceived are related by a two-dimensional abstraction to the real world entities that concern us. Along one dimension in the case of hardware we abstract from the switching theoretic structures that implement the state transitions, the wired in operations in G. Along the same dimension in the case of software we abstract from the syntactical devices that are used to specify the actions commanded by a program. Along the second dimension, hardware structures abstract from the set of behaviors that characterize a computer. Similarly, along this dimension software structures abstract from the fact that programs are not merely mappings but are also imperatives calling for a behavior.

In this section we wish to show how these abstractions, hardware and software structures, are related to what we call computation algebras. Also we shall develop further the abstract concept of the computer so as to be able to encompass in an algebraic setting some of the aspects of computer behavior.

While all the sets obtained from $\{0,1\}$ by dint of iterating Cartesian products are either product Boolean algebras or are allied to such algebras by relatively simple coding expediencies, it does not follow that this Boolean structure is germane to either the wired in operations of a computer or the operatives present in a language. Let us suppose that we are dealing with a hardware structure $(0^n,G)$. It may be the case that some or all of the elements of G are neither recoverable with the Boolean operators in the product algebra $0^n$ nor do they preserve any of the Boolean structure of $0^n$.

A computation algebra is a Boolean algebra together with transformation operators, which are endomorphisms, and reset operators, which are hemimorphisms. Having given the states of a hardware facility the structure of a Boolean algebra, our objective is to recover arbitrary state transition functions in terms of structure preserving functions on the Boolean algebra in question or on algebras systematically related to them. A larger algebra is required to recover arbitrary functions in terms of endomorphisms than that required to recover such functions in terms of the operations that preserve only half the Boolean structure, and both of these algebras are larger than the one originally given.

A simple example will show the necessity of seeking structure preserving operators in an algebra larger than the one formed by the states. The nature of such larger algebras is developed in a later section. Let

$$0^2 = \{f_1=00,\ f_2=01,\ f_3=10,\ f_4=11\}$$

and let g map $0^2$ into $0^2$ as defined by

$$g(f_1) = f_2$$

$$g(f_2) = f_3$$

$$g(f_3) = f_4$$

and

$$g(f_4) = f_1$$

Thus, we seek an input-output relation of

$$\big((00,01),(01,10),(10,11),(11,00)\big)$$

We have in the set and reset operations the ability to map bits into

{1} and {0}, respectively, and we have in the transformations the ability to permute bits and to copy the bit at one position into another position. A reset of one bit applied to either column equates two of the arguments and this equality must stand under subsequent application of any function. Therefore, in view of the one-one character of g, neither sets nor resets can be used in the recovery of g. Any transformation that performs a copying function will also equate two arguments. Thus, we are left with transformations of a cyclic nature. But such transformations perform cycles of order two whereas the problem demands a cyclic permutation of order four. Clearly, we must seek operators in a larger algebra than that of the states. One problem of this section is how we must conceive of a computation algebra such that the operations in G are mirrored by the operations in the algebra. This problem may be only of theoretical importance in the case of computers if in fact the preponderance of wired in instructions is intimately related to Boolean operators. The following table of wired in actions illustrates some of these relationships.

Table 1. Boolean Character of Computer Actions

| Computer Action | Boolean Operations and Elements Preserved | | | | | |
|---|---|---|---|---|---|---|
| | • | + | - | 0 | 1 | one-one |
| Word Shift | yes | yes | no | yes | no | no |
| Word Rotate | yes | yes | yes | yes | yes | yes |
| Move (Load, Store, Jump) | yes | yes | yes | yes | yes | no |
| Increment | no | no | no | no | no | yes |
| Complement | no | no | yes | no | no | yes |

In this short list of typical computer instructions we find isomorphisms, homomorphisms, hemimorphisms, and functions that do not preserve any of the Boolean structure.

Another central problem of this section is what to ally to hardware structures in order to represent some important aspects of computer behavior. It is known or it is assumed that digital computers are a variety of finite state machines. Thus a possible first step is to represent such machines in computers. The concept that seems to be reasonable is that of a control unit.

A control unit CG for a hardware structure $(0^n, G)$ is a mapping of $0^n$ into G. The behavior that carries the computer from any state $f_1$ in $0^n$ at some time $t_0$ into a state $f_2$ in $0^n$ at time $t_1$ is the $CG(f_1)$, where $t_1$ is the immediate successor of $t_0$. Thus the state $f_2$ at $t_1$ is $CG(f_1)f_1$. So conceived, the computer is a very restricted machine whose state at any time $t_n$, the nth power of the immediate successor time function applied to $t_0$, is uniquely determined by the initial state, the state at time $t_0$. Many different states may call the same behavior. The set of states that call on the same behavior as the state f is $CG^{-1}\big(CG(f)\big)$. Thus, CG defines a partition on $0^n$, $0^n/E$, where Efh if and only if, $CG(f) = CG(h)$. If $CG(f) = CG(h)$ then $CG(f)h = CG(h)h$ and $CG(h)f = CG(f)f$.

In the very restricted machine realized by the computer it makes limited sense to talk about the sequencing of states relative to a sequence of inputs. Two inputs $f_1, f_2$ in the sequence $f_1f_2$ yield a sequence of states:

$$CG(f_1f_2)f_0 = CG(f_2)\left(CG(f_1)f_0\right)$$

$CG(f_1)f_0$ is the "first" state obtained from the state $f_0$ on the input $f_1$ and $CG(f_2)\left(CG(f_1)f_0\right)$ is the "second" state obtained from the "first" state on the input $f_2$. But it is most important to note that the above equation makes sense only if $f_1$ is in the equivalence class of $f_0$ and $f_2$ is in the equivalence class of $CG(f_1)f_0$. In other words the sequence of inputs to a computer is uniquely determined up to an equivalence by the initial input and the initial input is fixed up to an equivalence by the initial state.

Nevertheless, computers are able to model machines that do not have such restrictions provided they are large enough. That is to say that the equivalence class in $0^n/E$ must have at least as many members as the unrestricted machine has states. Thus for example inputs could be coded in the leftmost segments of the vectors in $0^n$ and states of the finite state machine could be coded to the right. Thereby, the Cartesian product of the set of coded inputs with the set of coded states would yield $0^n$. The control unit would call for the proper state transition function for each input-state pair. Such a coding procedure would model a linear bounded automaton. Infinite and unbounded automata could be recovered in hardware structures of the appropriate cardinality.

## Computation Algebras and Computers

The intuitive ideas of the preceding section will now be made precise.

Definition 3. Let $\mathcal{O}$ be the simple Boolean algebra and let I denote the set of positive integers. Also consider the Boolean algebra

$$A = (\mathcal{O}^I, +, 0, \cdot, 1)$$

where the operations are the coordinate-wise induced operations and the 0 and 1 of $A$ are the constant functions on I whose values are 0 and 1, respectively.

We now define two sets of transformations, one on the index set I and the other on the Boolean algebra $A$.

Definition 4. Let E be the set of all mappings $e: I \to I$ for which there exists a finite subset J of I such that e is equal to the identity transformation d outside of J.

Definition 5. Let T be the set of all transformations T(e) from A into A such that for all e in E and p in A,

$$\big(T(e)\big)(p) = p \circ e$$

Theorem 5. The elements of T are Boolean endomorphisms on $A$.

Proof. Let T(e) be in T and let p be an arbitrary element in A. Then

$$T(e)(p) = p \circ e$$

and clearly $p \circ e$ maps I into $\{0,1\}$. Hence T(e)(p) is a member of A and the range of T(e) is a subset of A. Again, let p be an arbitrary element of A. Then

$$T(e)(p') = p'\circ e = (p\circ e)' = \big[T(e)(p)\big]'$$

The inner equality holds since the 'not' is defined coordinate-wise and

$$(p'\circ e)(i) = p'\big[e(i)\big] = \Big[p\big[e(i)\big]\Big]'$$

for i in I. Similarly, for arbitrary p and q in A

$$T(e)(p+q) = (p+q)\circ e = (p\circ e) + (q\circ e)$$

and

$$(p\circ e) + (q\circ e) = T(e)(p) + T(e)(q)$$

Hence T(e) is a homomorphism. We have at this point what is called a transformation algebra.

Definition 6. A *transformation algebra* is a triple (*A*,I,S), where *A* is a Boolean algebra, I is a set, and S is a function from transformations on I to Boolean endomorphisms on *A*, such that

$$S(d)p = p$$

whenever p belongs to A and d is the identity transformation on I, and

$$S(s)\circ S(t) = S(s\circ t)$$

whenever s and t are transformations on I.

<u>Theorem 6</u>. The triple ($A$,I,T) is a transformation algebra, where $A$ and I are as in definition 3 and T is a mapping from transformations on I to Boolean endomorphisms on A as in definition 5.

<u>Proof</u>. Since d is the identity in E,

$$T(d)(p) = p\circ d = p$$

for arbitrary p in A; therefore, T(d) is the identity endomorphism. Let p be an arbitrary element of A and let s and t be arbitrary elements of E. We have

$$T(s\circ t)\big(p(i)\big) = p\big((s\circ t)(i)\big) = p\Big[s\big(t(i)\big)\Big]$$

and

$$\big(T(s)\circ T(t)\big)\big(p(i)\big) = T(s)\Big[T(t)\big(p(i)\big)\Big]$$

$$T(s)\Big[T(t)\big(p(i)\big)\Big] = T(s)\Big[p\big(t(i)\big)\Big] = p\Big[s\big(t(i)\big)\Big]$$

Thus, the second axiom of a transformation algebra,

$$T(s\circ t) = \big(T(s)\circ T(t)\big)$$

holds in the present instance.

Definition 7. A mapping f from transformation algebra $(\mathcal{A},I,T)$ onto transformation algebra $(\mathcal{B},I,S)$ is a *transformation homomorphism* if f is a Boolean homomorphism from $\mathcal{A}$ to $\mathcal{B}$ and

$$f\big(T(s)p\big) = S(s)fp$$

whenever p belongs to A.

Definition 8. A subset M of A is a *transformation ideal* of $(\mathcal{A},I,T)$ if M is a Boolean ideal of $\mathcal{A}$ and $T(s)p$ belongs to M whenever p belongs to M.

In addition to the transformation operations we consider a new kind of operator that is normed, idempotent, additive, multiplicative, and non-increasing. Such an operator is called a reset operator.

Definition 9. A *reset operator* is a mapping from a Boolean algebra into itself such that:

(1) $R\circ R = R$

(2) $R(p+q) = R(p) + R(q)$

(3) $R(p) \leq p$

(4) $R(p') = R\big(R(p)\big)'$

(5) $R(p\cdot q) = Rp\cdot Rq$

for p and q elements of the Boolean algebra.

Of course, definition 9 is a generalization of the specific class of operators that suggest themselves for computation. All that is needed for computation is a single position reset operator, transformations, and composition in order to get general reset operations. However, some tedium can be avoided by taking the higher definition that follows.

Definition 10. Let J be a finite subset of I and define $R(J):A\to A$ such that on I - J

$$R(J)p(i) = p(i)$$

and on J

$$R(J)p(i) = 0$$

for p in A.

Theorem 7. The R(J) of definition 10 are reset operators.

Proof. It is obvious that R(J) is idempotent, multiplicative, and additive. Since the domain of R is the Boolean algebra *A*, the relation $\le$ is the coordinate-wise induced relation. Therefore,

$$p(1),p(2),\ldots,p(i),\ldots \le q(1),q(2),\ldots,q(i),\ldots$$

if and only if $p(i) \le q(i)$ for every i in I. Now it is equally obvious that $R(J)p \le p$. $\{R(J)p\}'$ has the value $p'(i)$ on I-J and 1 on J; therefore, $R(J)\{R(J)p\}'$ has the value $p'(i)$ on I-J and 0 on J, which is the same as $R(J)p'$. This establishes property 4.

A computation algebra can now be defined as a Boolean algebra together with transformations and reset operators by specifying the basic properties of the interaction between these two types of operators. We now define the central concept of this research.

Definition 11 (Computation Algebra). A *computation algebra* is a quadruple $(C,I,T,R)$ where $(C,I,T)$ is an atomic transformation Boolean algebra, I is the set of positive integers, and R is a mapping from finite subsets of I to reset operators on C such that:

(1) There is an atom a in $C$ and a transformation

$$u(n) = n + m$$

for finite m such that

$$T(u)a' = 1$$

(2) $R(J)p = p\cdot T(s)(a')$ where p is in C, a is the atom mentioned in (1), and

$$s(i) = \left\{\begin{matrix} 1 \text{ for } i \text{ in } J \\ i \text{ for } i \text{ in } I-J \end{matrix}\right\} \text{ for } i > 1$$

and

$$s(1) = \left\{\begin{matrix} 1 \text{ for } 1 \text{ in } J \\ k, \text{ where } k \text{ is the least integer} > 1 \text{ in } I-J \text{ if } 1 \notin J \end{matrix}\right\}$$

Theorem 8. $(A,I,T,R)$ is a computation algebra where $A$ and I, T, and R are as given in definitions 3, 5, and 10, respectively.

Proof. Clearly $A$ is an atomic Boolean algebra and I is the set of positive integers. Theorem 6 provides that $(\mathcal{A},I,T)$ is an atomic transformation algebra. Theorem 7 shows that the $R(J)$ are reset operators. In order to show that property (1) holds we exhibit the distinguished atom a in $0^I$, namely,

$$a(1) = 1$$

$$a(i) = 0 \text{ for } i > 1$$

and the distinguished transformation

$$u(i) = i + 1$$

To show that property (2) holds we notice that

$$a' = 011 \ldots 1 \ldots$$

and that $T(s)a'$, with s as defined in 11, yields the element in $0^I$ which has the value 0 on J and 1 on I-J. The conjunction of such an element with an element p yields the same result as $R(J)p$. Thus, $(\mathcal{A},I,T,R)$ is a computation algebra.

In order to gain familiarity with the reset operators, a number of elementary properties will now be derived. The most important of these are the normality and monotonicity of reset operators.

Theorem 9. The reset operator is: (1) normed, (2) quasi-additive, (3) monotone, and (4) quasi-multiplicative.

Proof. (1) is established by putting $p = 0$ in item (3) of definition 9. Since R is additive and idempotent

$$R(p+Rp) = Rp + RRq = Rp + Rq$$

establishing (2). We have $p \leq q$ if and only if $p+q = q$. Let $p+q = q$ then using successively the fact that R is a function, R is additive, and the biconditional just stated, we have

$$R(p+q) = Rq$$

$$Rp + Rq = Rq$$

$$Rp \leq Rq$$

showing that R is monotonic. Since R is multiplicative and idempotent

$$R(p\cdot Rq) = Rp\cdot RRq = Rp\cdot Rq$$

and R is quasi-multiplicative.

As it will be shown, the bit Boolean operations, the bit reset operations, and the bit transformations are sufficient to recover in a piecemeal fashion to be explained, all mappings of *A* into *A*. However, in computation algebras it is not necessary to postulate both the bit reset and complement. As the next definition shows, finite complement operators can be obtained from reset operators, transformations, and the operations that are available in any Boolean algebra.

Definition 12. For each finite subset J of I the complement operator is defined by

$$C(J)p = \left[p'\cdot\left(R(J)1\right)'\right] + R(J)p$$

The elementary properties of the complement operator are established in the following theorem.

Theorem 10. The complement operator has the properties for J and K finite subsets of I:

(1) $C(J)\circ C(J) = C(\emptyset)$

(2) $C(J)\circ C(K) = C(J-K)$

(3) $C(J)p' = \left(C(J)p\right)'$

(Proof omitted.)

Several relationships exist among the Boolean operations, the reset operations, the set operations (an obvious counterpart for the reset operator), and the transformations, that are available in computation algebras. Familiarity with these interconnections often facilitates the work. It is clear from the definition of a computation algebra that every reset operator can be obtained from the Boolean 'and', a distinguished atom of the algebra, and a transformation. A similar remark would hold true for the set-operator. The unary 'and' operator and the unary 'or' operator for each element can be defined, in the finite case, in terms of reset and set by

$$p \cdot q = R\left(p^{-1}(0)\right)q \quad \text{and} \quad p+q = S\left(p^{-1}(1)\right)q$$

respectively. And if p and q are in $0^I$, we need to allow that R(J) and S(J) be defined for infinite J. The relationship between reset operators and set operators is first of all that the latter may be defined in terms of the former per

$$S(J)p = \left(R(J)p'\right)'$$

and secondly that any composition of set with reset, for example, $S(I) \circ R(J)$, can be expressed in terms of an operationally equivalent composition which is furthermore commutative, namely, $S(I) \circ R(J-I)$.

Definition 13. A computer $(0^n, G, CG)$ is a hardware structure $(0^n, G)$ together with a mapping CG from $0^n$ into G.

Computers that are of particular interest are those built upon computation algebras. Such computers have control units which have as their ranges reset and transformation operations or operations composed of resets and transformations. Consider the computation algebra $\mathcal{A} = (\mathcal{O}^I, I, T, R)$ and the set $P_A$ which is the closure of the set of finite resets, finite transformations, and finite Boolean operations under functional composition. This algebra together with a mapping CU from $0^I$ into $P_A$ forms a computer.

Definition 14. A *universal computer* is a computer $(A,P_A,\mathrm{CU})$ where CU is an onto mapping.

We say that a control unit is universal if it is an onto mapping. Clearly, the intuitive appeal of universal control is that every possible procedure can be called. In general, computers have very restricted initial sets of actions, and therefore do not have universal control in the sense of being able to call every procedure.

A control unit, CU, is *localized* if and only if there is a J, a finite subset of I, such that for any element g in $O^I$:

$$\mathrm{CU}\big(S(J)O\cdot g\big) = \mathrm{CU}(g)$$

Localized control says that only a finite and fixed portion of every state determines the procedure assigned to states. Obviously, all conventional computers have localized control, which is usually identified as the program counter, instruction analyzer, etc. Indeed, this is precisely the concept of buffering. Insofar as all memory-register operations are actually buffer-register operations, the circuitry going to all of memory being limited to fetch and store circuitry, the control is localized to the buffer, rather than the memory, and the other central processor control units.

Two observations are made concerning localized control. First, there is no localized control unit for the universal computer. That is, if CU is a universal control unit for the $(O^I,I,T,R)$ computation algebra then CU is not localized. Since CU is a function with a range of cardinality Aleph-zero, at least this many elements of the algebra must be discriminated by CU. Clearly, such a discrimination cannot be made on the basis of a finite portion of the vectors in the algebra. Thus, the CU is not a localized control unit.

The second observation is that every control unit for a conventional computer is localized. This is obviously the case since the state vectors are of finite length.

Gorn [5] has noted that one reason for the digital computer being such a versatile tool is the presence of a certain ambiguity in the use of memory. A word in memory may contain the code for a piece of data or the code for an instruction depending, respectively, upon whether the contents of the word are sent to the arithmetic unit or the control unit. The same code may on one occasion play the role of data and on another occasion play the role of an instruction.

Professor Gorn's statement concerning the ambiguity of process and control may be paraphrased algebraically as follows. If f is an element of $0^n$ in the computer $(0^n,G,CG)$ and $CG(f)f \epsilon CG^{-1}(CG(f))$, then the instruction segment of f is unmodified by the operation of CG(f). In our earlier terminology, f and CG(f)f belong to the same equivalence class. If on the other hand $CG(f)f \notin CG^{-1}(CG(f))$, if f and CG(f)f do not belong to the same equivalence class, then the instruction segment of f has been modified by the operation CG(f). In the first case the instruction segment of f has not played the role of data; in the second case it has. Thus sometimes the instruction segment may both call an operation and be operated on. It is in Gorn's sense ambiguous. It is clear that not every segment of every f in $0^n$ need be ambiguous in this sense. It is also clear that in order to obtain at least one transition from an equivalence class of states to another some ambiguity is unavoidable.

It may be noted in passing that Gorn's sense of ambiguity, or at least our interpretation of it, has nothing to do with indeterminacy. In any computer as is here conceived each state determines a unique procedure and therefore a unique next state. It may be also noted that computers without some ambiguity are very uninteresting since they are limited to the iteration of just one operation. The limit of ambiguity is perhaps achieved by the universal computer since this machine cannot have localized control, hence instruction segments cannot be localized. Nevertheless, it is worthy of note that even this computer can contain segments that never function as repositories of instructions.

Three distinct concepts have thus far evolved which may be viewed in a hierarchy of computational power. These are, in ascending order, the computer, the universal computer, and the computation algebra itself.

Computers have the power of finite state machines. The concatenatability of inputs to restricted semiautomata in terms of the control unit concept has been put forth in the introduction. We have also noted that the input set is the same as the state set and that the usual concept of time is available to computers as we know them here.

A universal computer is a triple $(A, P_A, CU)$ as introduced above. We may view the set of states of this computer as the set of tapes possible in a Turing machine of two symbols with a tape infinite in one direction. For every finite n the set of all mappings from $O^n$ into $O^n$ will be shown in the next section to be recoverable via coding and embedding theorems as functions in $P_A$. Thus viewed the universal computer has at least the power of a two symbol universal Turing machine.

## Coding and Embedding Theorems

Hardware structures in which the actions are arbitrary functions can be coded in hardware structures in which the actions in question are mirrored in hemimorphisms (sets and resets) or they may be coded in structures in which the actions are mirrored in endomorphisms (transformations). Arbitrary computers can be coded in computers of the same size in which all the actions are only set and reset operations. This mode of coding will not generally recapture the elementary operations that are "wired in" the arbitrary computer but it is capable of a piecemeal specification of every transition from one state to another. Hardware structures in which the actions preserve Boolean structure can be homomorphically embedded in the computation algebra $(O^I, I, T, R)$. Computers that call only structure preserving actions can be homomorphically embedded in the universal computer. In this section we prove theorems supporting the preceding statements. The strategy is first to code, then to embed. First we code and embed hardware structures, then we code and embed computers.

<u>Theorem 11</u>. For every hardware structure $(0^n,G)$ there exists a hardware structure $(0^k,T)$, where T is a set of transformations, and a coding function C from $0^n$ into $0^k$ such that for any f in $0^n$

$$Cgf = T(t)Cf$$

for $g\varepsilon G$ and some $T(t)\varepsilon T$.

<u>Proof</u>. Choose $k = 2^{(2^n)}$ and define $2^n$ generators of $0^k$ as follows: Let $W_0$ belong to $0^k$ such that $W_0(1) = 1$ and $W_0(k) = 0$. Then for $i = 1,\ldots,2^n$, the generators are

$$W_i = T(t_i)W_0$$

where

$$t_i(m) = 1 \text{ if } (m-1)(\text{mod } 2^i) \leq 2^{(i-1)} - 1$$

and

$$t_i(m) = k \text{ otherwise.}$$

It is clear that the set $\{W_1,W_2,\ldots,W_{2^n}\}$ freely generates $0^k$ since it is an independent set.

Let C be an arbitrary one-one function from $0^n$ onto the generators of $0^k$ and order the elements of $0^n$ such that

$$C(f_i) = W_i$$

Let g be a mapping from $O^n$ into $O^n$ defined by

$$g(f_j) = f_{i_j}$$

What follows is an algorithm for constructing the t of T(t) mentioned in the theorem. Form a matrix having the generators $W_1,\ldots,W_{2^n}$, in that order, as its rows. Let h be a mapping from $C(O^n)$ into $C(O^n)$ that corresponds to g. That is,

$$h(W_j) = W_{i_j}$$

given g as above. For each j in $\{1,\ldots,2^{(2^n)}\}$ we have the requirement that

$$\begin{aligned} W_1(j) &= W_{i_1}(j) \\ &\vdots \\ W_{2^n}(j) &= W_{i_{2^n}}(j) \end{aligned}$$

The column on the right of the equals sign is a column of 0's and 1's that summarizes the values h must yield and is perforce a column j' of the matrix formed above. Now define the transformation t on the index set by

$$t(j) = j'$$

According to this construction, $h = T(t)$ and

$$\bigl(T(t)W_x\bigr)(j) = (W_x \circ t)(j) = W_x(j) = W_{i_x}(j')$$

Finally, we have

$$Cgf_x = Cf_{i_x} = W_{i_x} = T(t)W_x = T(t)Cf_x$$

as required.

Theorem 12. For every hardware structure $(O^n,G)$ there exists a hardware structure $(O^k,S\circ R)$, where $S\circ R$ is the closure under composition of sets and resets on $O^k$, and a coding function $C$ from $O^n$ into $O^k$ such that for any $f$ in $O^n$

$$gf_r = P_r R(J)(Cf_r)$$

for $g$ belonging to $G$ and some $I,J$ subsets of $\{1,\ldots,k\}$, and $P_r$ a projection of the rth coordinate of $O^k$ into $O^n$.

Proof. Let $g$ belong to $G$, a mapping from $O^n$ into $O^n$, where

$$O^n = \{f_1,\ldots,f_{2^n}\}$$

and define a mapping $g^*$ belonging to the set of all mappings from $\{1,\ldots,2^n\}$ into $O^n$ by

$$g^*(i) = g(f_i)$$

We note that the algebra of all mappings from $\{1,\ldots,2^n\}$ into $0^n$ is isomorphic to $0^{n\cdot 2^n}$ and define the isomorphism I in the obvious way:

$$I\big(g^*(i)\big)(j) = g\big(f_i(j)\big)$$

Let $\{a_1,\ldots,a_n\}$ be the atoms of $0^n$. Then the atoms of the algebra of mappings from $\{1,\ldots,2^n\}$ into $0^n$ may be defined by

$$A_{ij}(k) = 0 \text{ of } 0^n \text{ for } k \neq j$$

and

$$A_{ij}(k) = a_i \text{ of } 0^n \text{ for } k = j$$

where $j$, $k = 1,\ldots,2^n$ and $i = 1,\ldots,n$. As a notational expedient let us write

$$I(A_{ij}) = (0,\ldots,0,1_u,0,\ldots,0)$$

Then

$$g(f_r) = g^*(r) = \Big(\sum_{\substack{i\varepsilon I\\ j\varepsilon J}} A_{ij}\Big)(r) = \sum_{\substack{i\varepsilon I\\ j\varepsilon J}} A_{ij}(r)$$

and

$$g^* = \Big\{S_u\big(\cup(IA_{ij})^{-1}(1)\big)0\Big\}$$

where 1 belongs to the simple algebra O and O belongs to the product algebra $O^{n\cdot 2^n}$. Therefore

$$g(f_r) = g^*(r) = \left[S_u\left(u(IA_{ij})^{-1}(1)\right)0\right](r)$$

We now define the coding function

$$C(f_r) = R(\{1,\ldots,2^n\} - \{r\})(f_1,\ldots,f_{2^n})$$

$$gf_r = P_r(g^*\cdot Cf_r) = P_rR\left(g^{*-1}(0)\right)Cf_r$$

Hence, the J required in the theorem is $g^{*-1}(0)$.

<u>Theorem 13</u>. Every hardware structure $(O^n,TRS)$, where TRS is a set of transformations, reset operators, and set operators, is a quotient of the computation algebra $(\mathcal{O}^I,I,T,R)$.

<u>Proof</u>. Let E be an equivalence relation defined on $O^I$ as follows:

$$Epq \equiv p(i) = q(i) \text{ for } i = 1,\ldots,n$$

Then E induces a partition on $O^I$ and a homomorphism

$$h: O^I \to O^I/E$$

such that $h(p) = |p|$. The quotient $O^I/E$ is isomorphic to $O^n$.

Furthermore, we may define induced reset, set, and transformation operators on $0^I/E$ as follows:

$$R(J)|p| = |R(J)p|$$

$$S(J)|p| = |S(J)p|$$

$$T(t)|p| = |T(t)p|$$

where if J intersects {1,...,n} in the null set then

$$R(J) = S(J) = R(\emptyset)$$

and if the range of t is outside {1,...,n} then similarly T(t) is the identity transformation. From these definitions it follows that:

$$R(J)hp = R(J)|p| = |R(J)p| = h\big(R(J)p\big)$$

$$S(J)hp = S(J)|p| = |S(J)p| = h\big(S(J)p\big)$$

$$T(t)hp = T(t)|p| = |T(t)p| = h\big(T(t)p\big)$$

Thus, the homomorphism preserves sets, resets, and transformations. (Consult the next section for justification of the definitions of the induced operators.)

<u>Theorem 14</u>. Every computer $(0^n,G,CG)$ is isomorphic to a set-reset computer $(0^n,SR,CSR)$. That is, for every f in $0^n$,

$$CG(f)f = CSR(f)f$$

<u>Proof</u>. The conclusion of the theorem is that there exist $R(J)$ and $S(I)$ such that $R(J)\circ S(I)$ is in $S\circ R$ and such that

$$CG(f)f = R(J)\circ S(I)f$$

For $CG(f) = g$ we have

$$I = (gf)^{-1}(1) \text{ and } J = (gf)^{-1}(0)$$

so that since CG is a function there is only one set I and only one set J for each g belonging to G and f belonging to $0^n$. The new control unit is defined:

$$CSR(f) = R\left((gf)^{-1}(0)\right) \circ S\left((gf)^{-1}(1)\right)$$

Since

$$\left[R\left((gf)^{-1}(0)\right) \circ S\left((gf)^{-1}(1)\right)\right]h = gf$$

for any h in $0^n$ the conclusion of the theorem follows.

<u>Theorem 15</u>. There exists a universal computer $(A,P_A,CU)$, as

defined in 14, in which every set-reset computer can be embedded via a coding.

Proof. The coding K is achieved by way of a Gödel numbering as follows: For each set of states $0^n$ we assign the number n. This is an ordering of the set of all finite memories. Next we assign an index c to each control unit for a memory $0^n$. That is to say we order the set of mappings from $0^n$ into the set of sets and resets. As was noted earlier, every arbitrary composition of sets and resets is equal to $S(I) \circ R(J)$ for some I and J disjoint subsets of $\{1,\ldots,n\}$. The number of distinct control units for computers with memory $0^n$ is a finite number which is a function of n and the index c is unique. Finally we assign a unique index s to each state of $0^n$. Through the described assignments a unique ordered triplet (n,c,s) is associated with each state of each computer. The coding K is then the binary representation of the Gödel number of triplets (n,c,s), which we denote by G(n,c,s). More explicitly, if f is in $0^n$, the index of f is s, and the index of the control unit is c, then

$$K(f) = G(n,c,s)$$

where G(n,c,s) is in A and we have left justified the binary representation appropriately.

We define a universal control unit for the universal computer

$$CU\big(G(n,c,s)\big)\big(G(n,c,s)\big) = G\big(n,c,c(s)s\big)$$

where c(s)s is the index of the state which ensues when the operation called by the control unit with index c at the state with index s is applied to the state with index s. For any set-reset computer $(O^n, SR, CSR)$, f in $O^n$ with index s, CSR with index c, then

$$K\big(CSR(f)(f)\big) = G\big(n,c,c(s)s\big) = CU\big(G(n,c,s)\big)\big(G(n,c,s)\big) = CU(Kf)(Kf)$$

Thus K is a proper embedding function in that it preserves control units.

Theorem 16. There exists a universal computer $(A,P_A,CU)$ in which every arbitrary computer can be embedded by a coding.

Proof. Follows from Theorems 14 and 15.

It seems reasonably clear that we may not obtain homomorphic embedding theorems on the style of Theorems 15 and 16 because of the presence of control units. Computers may have the same hardware structure but differ in control units. Theorem 13 shows that every hardware structure whose set of actions is composed of transformation, set, and reset operators is a quotient of the computation algebra. Theorem 14 states that every computer is isomorphic to a set-reset computer. Nevertheless, even in the light of these two theorems it does not follow that there exists a universal computer of which every other computer is a quotient. In fact, the denial of this statement may be shown by considering that homomorphisms must preserve the zero element and that two computers may differ in that their control units may call for different actions on the zero element. But the image of the zero element under a homomorphism must be the zero of the universal computer; hence, it is not possible to homomorphically embed these two computers in the universal one. The strongest homomorphism theorem to be expected is as follows.

Theorem 17. Every arbitrary computer is a quotient of *some* universal computer.

Proof. Since we have Theorem 14 it is sufficient to show that for every set-reset computer there exists a universal computer of which the set-reset computer is a quotient. In order to show this last step we use Theorem 13 and construct the appropriate control unit. Computer $(0^n, TRS, CTRS)$ is a quotient of $(A, P_A, CU)$ provided CU is defined such that

$$CU(p) = CTRS(hp)$$

for p in A.

The theorems stated above summarize what might be viewed as ten different theorems arising from five separate situations. Beginning with a hardware structure we have two ways to go: (i) We can go to the structure based on transformations, (ii) we can go to the structure based on set-reset operators. Each of these we then embed in the $0^I$ algebras. This accounts for four theorems--two for encoding and two for embedding. Beginning with a computer, we first choose whether to go to a larger algebra or to a piecemeal recovery of the actions as appears in Theorem 14. If we choose to go to the higher algebras, then we have a choice of going to hemimorphisms or to endomorphisms and from either to go on to a quotient of some universal computer. Thus, we may follow a path to two more encoding theorems and two more embedding theorems. This accounts for eight of the theorems. If we choose the piecemeal recovery of the actions, then we encode to the set-reset computer and embed in some universal computer. Thus, we have a total of five encodings and five embeddings.

On the basis of these results we hereinafter understand by a computer the appropriate quotient of some universal computer, and similarly for hardware structures.

## Algebraic Theory

The concepts of subalgebra, homomorphism, and ideal are basic concepts of universal algebraic theory. Generally, a computation subalgebra is a transformation subalgebra *B* of a computation algebra *A* that is a computation algebra with respect to the operators on *A*. A computation homomorphism is a transformation homomorphism that preserves reset operators. A computation ideal is a transformation ideal that is closed under resets. These are the general, intuitive descriptions of the basic concepts. In this section we pull together various universal algebraic facts that were used without proof or justification in the previous section.

Definition 15 (Computation Subalgebra). If *A* is a computation algebra and *B* is a transformation subalgebra of *A* such that for all p in B, R(J)p belongs to B for all finite subsets J of I, then *B* is a computation subalgebra of *A*.

Definition 16 (Computation Homomorphism). If $A$ and $B$ are computation algebras, a computation homomorphism is a mapping $h:A \to B$ such that h is a transformation homomorphism and

$$h\big(R(J)p\big) = R(J)h(p)$$

where J is a finite subset of I.

Definition 17 (Computation Ideal). A subset M of C of $(\mathcal{C},I,T,R)$ is a computation ideal if it is a transformation ideal of $(\mathcal{C},I,T)$.

Clearly, there is no novelty introduced to a transformation ideal by a reset operator since resets are non-increasing. We may say that given an element of an ideal, all smaller elements (in the Boolean sense) are in the ideal. The reset of an element is either equal to or smaller than the element and therefore is in the transformation ideal already.

Theorem 18 (Homomorphism Theorem). A subset M of a computation algebra $A$ is the kernel of a computation homomorphism if and only if it is a proper computation ideal.

Proof. Suppose that M is a subset of A and the kernel of computation homomorphism h. Obviously, the 0 of $A$ is in M. If p and q are in M then

$$h(p) = 0 \text{ and } h(q) = 0$$

and since

$$h(p+q) = h(p) + h(q) = 0 + 0 = 0$$

we have p + q belongs to M. Let p be in M and q be in A. Since

$$h(p \cdot q) = h(p) \cdot h(q) = 0 \cdot h(q) = 0$$

p • q belongs to M. Again, let p be in M. Then

$$h\big(T(s)p\big) = T(s)h(p) = T(s)0 = 0$$

so that T(s)p belong to M. This proves that M is a computation ideal. To show that M is also proper one simply notes that M does not contain the 1 of *A* since in particular h is a Boolean homomorphism.

In order to show that every proper computation ideal M of the algebra *A* = (A,I,T,R) is the kernel of a computation homomorphism, consider the Boolean quotient algebra

$$\mathcal{B} = (A/M,+,0,\cdot,1)$$

and the natural Boolean homomorphism h from A onto B. The task is to convert B uniquely into a computation algebra in such a way that h becomes a computation homomorphism with kernel M. In order to do this, we define T on B and R on B and prove that they are transformations and resets, respectively.

For (hp) an element of *B*, define the induced transformations on *B* by

$$T(s)(hp) = h\big(T(s)p\big)$$

In order to prove that this definition is unambiguous it is necessary to prove that

$$(hp) = (hq) \text{ implies } T(s)(hp) = T(s)(hq)$$

Assuming the antecedent, p is congruent to q and their symmetric difference, p - q, belongs to the computation ideal M. Transformations are endomorphisms and therefore

$$T(s)(p-q) = T(s)p - T(s)q$$

Since M is also a transformation ideal it contains T(s)(p-q) and T(s)p - T(s)q. Thus, T(s)p is congruent to T(s)q, or equivalently

$$h\big(T(s)p\big) = h\big(T(s)q\big)$$

By definition then

$$T(s)(hp) = T(s)(hq)$$

and the induced transformations (so-called) are well-defined. That these operations are indeed transformations is shown by

$$T(d)(hp) = h\left(T(d)p\right) = hp$$

and

$$T(s \circ t)(hp) = h\left[\left(T(s) \circ T(t)\right)p\right] = h\left[T(s)\left(T(t)p\right)\right]$$

$$h\left[T(s)\left(T(t)p\right)\right] = T(s)\left(hT(t)p\right) = \left(T(s) \circ T(t)\right)(hp)$$

Analogously, we define

$$R(J)(hp) = h\left(R(J)p\right)$$

and show that this definition is unambiguous by showing that

$$(hp) = (hq) \text{ implies } T(s)(hp) = T(s)(hq)$$

By hypothesis, $p - q$ is in $M$ and it follows that $R(J)(p-q)$ is in $M$. We have

$$\begin{aligned} R(J)(p-q) &= (p-q) \cdot T(s)(a') \\ &= \left(p \cdot T(s)a'\right) - \left(q \cdot T(s)a'\right) \\ &= R(J)p - R(J)q \end{aligned}$$

which implies that

$$h\big(R(J)p\big) = h\big(R(J)q\big)$$

Thus, the induced reset operators are well defined. It is mechanical to show that the operator in question meets the conditions set forth in Definition 9 for reset operators. The details are as follows with the index set J omitted for brevity. It must be understood that the index set is fixed throughout.

(1) $RR(hp) = R\big(h(Rp)\big) = hRRp = hRp = R(hp)$

(2) $R(hp+hq) = R\big(h(p+q)\big) = hR(p+q) = h(Rp+Rq) = hRp+hRq =$ $Rhp+Rhq$

(3) $R(hp) \leq hp$ iff $R(hp)+hp = hp$

$R(hp)+hp = hRp+hp = h(Rp+p) = hp$

(4) $R(hp)' = Rhp' = hRp' = hR(Rp)' = Rh(Rp)' =$ $R(hRp)' = R(Rhp)'$

(5) $R(hp \cdot hq) = R\big(h(p \cdot q)\big) = hR(p \cdot q) = h(Rp \cdot Rq) =$ $hRp \cdot hRq = Rhp \cdot Rhq$

In summary, we have constructed a computation algebra *B* from *A* in such a way that h maps *A* onto *B* and the kernel of h is M.

The only concrete example of a computation algebra is the one based on the 0-valued Boolean algebra. Representation theory proceeds by showing to what extent every computation algebra is representable by this 0-valued, or simple, computation algebra. Thus, the main theorem of this section is the following representation theorem.

Theorem 19 (Representation Theorem). Every computation algebra $(\mathcal{C},I,T,R)$ with $(\mathcal{C},I,T)$ isomorphic to $(\mathcal{O}^I,I,T)$ is isomorphic to $(\mathcal{O}^I,I,T,R)$.

Proof. Since $(\mathcal{C},I,T)$ is isomorphic to $(\mathcal{O}^I,I,T)$, the transformation isomorphism determines a transformation ideal which is 0. But this is also a computation ideal and by Theorem 18 it determines a computation isomorphism from $(\mathcal{C},I,T,R)$ to $(\mathcal{O}^I,I,T,R)$.

The question of whether or not one can study computers algebraically (and if so, to what extent, and in what algebra) has now been reduced to a manageable question. Given that the memory structure of a computer, its set of actions, and its control unit constitute an interesting and studyworthy portion of a computer, the question becomes one of whether or not there is a theory of morphisms for such entities, and if so, what is the extent of the theory, and what are the models of the theory. The answer is in the affirmative and the algebras in question are computation algebras. We now extend the theory of morphisms of computation algebras to computers in the following definitions and theorems.

Definition 18. $(B,P_B,CB)$ is a subcomputer of $(C,P_C,CC)$ if and only if $(B,P_B)$ is a computation subalgebra of $(C,P_C)$ and CC restricted to B is equal to CB.

Definition 19. A mapping h from $(B,P_B,CB)$ into $(C,P_C,CC)$ is a computer homomorphism if and only if h is a computation homomorphism and

$$hCB(f)f = CC(hf)hf$$

Definition 20. A subset M of B in the computer $(B,P_B,CB)$ is a computer ideal if and only if M is a computation ideal and for all f,g in B such that f - g is in M, then

$$CB(f)f = CB(g)g$$

We note that a computer ideal is more than a computation ideal, and, therefore, more than a transformation ideal. The novelty is introduced by the control unit.

Theorem 20. A subset M of a computer is the kernel of a computer homomorphism if and only if it is a proper computer ideal.

Proof. If M is the kernel of a computer homomorphism

$$h:(B,P_B,CB) \to (C,P_C,CC)$$

then it is the kernel of a computation homomorphism and

$$h\big(CB(f)f\big) = CC(hf)hf$$

for every f in C. If f-g belongs to M, that is, if hg = hf then

$$CC(hf)hf = h\big(CB(f)f\big) = CC(hg)(hg) = hCB(g)g$$

Let M be a computer ideal and consider

$$h:(B,P_B) \rightarrow (B/M,P_B)$$

which is a computation homomorphism. Define a computer $(B/M,P_B,CM)$ such that

$$CM(|f|)|f| = |CB(f)f|$$

for all f in B and $|f|$ in B/M. If g belongs to $|f|$, then $CB(g)g = CB(f)f$ since M is a computer ideal. Let

$$h(f) = |f|$$

for f in B then

$$h\big(CB(f)f\big) = |CB(f)f| = CM(|f|)|f| = CM(hf)hf$$

That completes the proof of the theorem.

<u>Theorem 21</u>. Every computer $(B,P_B,CB)$ with $(\mathcal{B},I,T,R)$ isomorphic to $(\mathcal{O}^I,I,T,R)$ is isomorphic to some universal computer.

<u>Proof</u>. Since $(B,I,T,R)$ is isomorphic to $(\mathcal{O}^I,I,T,R)$, the computation isomorphism determines a computation ideal which is $\{0\}$. Define the control unit $CU:O^I \rightarrow P_A$ such that

$$CU\big(Iso(f)\big)\big(Iso(f)\big) = Iso\big(CB(f)f\big)$$

for all f in B. The isomorphism so extended is a computer isomorphism since {0} is indeed a proper computer ideal.

The field of computation can indeed be approached algebraically and the algebra is at times quite illuminating. However, we have noted at least one very substantial limitation to further algebraic developments. Namely, it is not possible to homomorphically embed every computer in one and the same universal computer. Thus, the comparison of computers cannot be carried out as hoped in terms of the comparison of different quotients of a fixed algebra, in general. Such a limitation was anticipated to the author by Professor S. Gorn in private discussions. The source of the limitation is the nature of control units, as was further anticipated by Professor Gorn.

## Computation Spaces

The topological version of Stone's theorem says that there is a one-to-one correspondence between Boolean algebras and totally disconnected compact Hausdorff spaces, i.e. Boolean spaces. Since homeomorphism is the topological counterpart of the algebraic concept of isomorphism, this one-one-ness means that an algebra $A$ determines a space X to within homeomorphism and a space X determines an algebra $A$ to within isomorphism. The algebra $A$ corresponding to the space X is called the dual of X and is notated $X^*$. The space X corresponding to the algebra $A$ is called the dual of $A$ and is notated $A^*$. The theory of duality makes possible a somewhat more unified treatment of transformations and resets than is possible in the algebraic theory in that all structural details are dualized to relations and relational properties.

It has already been noted that transformations and reset operators are hemimorphisms. The topological concept that is the dual of algebraic hemimorphisms is that of a Boolean relation. Before defining Boolean relations we introduce a limited amount of notation. Elements of a binary relation $\phi$ on Boolean spaces Y and X are written in the prefix style $\phi yx$ for y in Y and x in X. $\phi^{-1}$ denotes the inverse of $\phi$. If Q is a subset of Y, the direct image of Q under $\phi$, $\phi Q$, is the set of all points x in X for which there exists a point y in Q such that $\phi yx$. The inverse image of a subset P of X under $\phi$, $\phi^{-1}P$, is the set of points y in Y for which there exists a point x in P such that $\phi yx$.

Definition 21. A *Boolean relation* $\phi$ is a relation on $Y \times X$, where X and Y are Boolean spaces, such that the inverse image of every open and closed set in X is an open and closed set in Y and such that the direct $\phi$ image of every point in Y is a closed set in X.

The topology corresponding to the simple Boolean algebra is the one in which every member of the power set of 0 is an open set. There is a so-called natural isomorphism between $X^*$, for X a Boolean space, and the set of all continuous functions from X into the topological space on 0. Every Boolean algebra is to be identified with the algebra of all 0-valued continuous functions on its dual space. Accordingly, assume that an element p in A is a continuous function from X into 0. Then p(x), for x in X, is the characteristic function for the open set $p^*$ in the space X. Thus, $p(x) = 1$ if x belongs to $p^*$ and $p(x) = 0$ if x does not belong to $p^*$.

Let *A* and *B* be Boolean algebras with dual spaces X and Y, respectively. If f is a hemimorphism from $A$ into $B$, its dual, $f^*$, is the relation on $Y \times X$ defined by

$$f^*yx \text{ if and only if } p(x) \leq fp(y)$$

for all p in $A$. If $\phi$ is a Boolean relation on $Y \times X$, its dual, $\phi^*$, is the mapping that assigns to every p in A a function $\phi^*p$ from Y to O. $\phi^*p$ is defined by

$$\phi^*p(y) = \sum \{p(x):\phi yx\}$$

We can now state the principal theorem in the theory of Boolean duality.

<u>Theorem 22</u> (Halmos [8], Page 54). If f is a hemimorphism, then $f^*$ is a Boolean relation, and $f^{**} = f$. If $\phi$ is a Boolean relation, then $\phi^*$ is a hemimorphism and $\phi^{**} = \phi$.

All transformation and reset operators are, in the light of Theorem 22, the duals of Boolean relations, and conversely. However, transformations are more than hemimorphisms; they are endomorphisms. The following theorem shows that these Boolean relations must be functions.

<u>Theorem 23</u> (Halmos [8], Page 57). If f is a hemimorphism from $A$ to $B$ and $\phi$ is the corresponding Boolean relation on $Y \times X$, then f is a homomorphism if and only if $\phi$ is a function with domain Y.

Clearly, one Boolean relation of the dual of a computation algebra must be the identity relation on the dual space of the algebra. The following theorem will help in obtaining the dual concept of the requirement that the composition of two transformations must be a transformation.

Theorem 24 (Halmos [8], Page 56). If *A*, *B*, and *C* are Boolean algebras with dual spaces X, Y, and Z, respectively, and if f and g are hemimorphisms from *A* into *B* and *B* into *C*, respectively, then

$$(gf)^* = f^* | g^*$$

The appropriate concept is that the set of Boolean relations on computation spaces must be closed under relational product.

Theorem 23 also shows that the idempotence of reset operators dualizes to a requirement of transitivity and density on the corresponding Boolean relation, $R^*$. $R^*$ is dense since the intersection of $R^*$ with the difference relation $x \neq y$ is a Boolean relation that is a subset of its square. According to the theorem that follows, the reset Boolean relations must be functions.

Theorem 25 (Halmos [8], Page 57). If a hemimorphism f from *A* to *B* and a Boolean relation $\phi$ from Y to X are each other's duals, then f is multiplicative if and only if $\phi$ is a function.

Note that this theorem does not make multiplicative hemimorphisms into homomorphisms since there is no requirement that the dual of a multiplicative hemimorphism be a function with domain Y. Indeed it can be shown that since the multiplicative hemimorphisms of interest, the sets and resets, do not preserve 0 and 1, respectively, their duals will have as domains proper subsets of Y. For $J \neq \emptyset$, $R(J)1 \neq 1$, then

$$\{y:R(J)1(y) = 1\} = \left((R(J))^{*}\right)^{-1}X$$

is a proper subset of Y, and similarly for set.

Reset operators are non-increasing so their duals must have $R^{*}x \subseteq x$, for open set x. The dual of

$$Rp' = R(Rp)'$$

is

$$R^{*}x' = R^{*}(R^{*}x)'$$

for open set x.

Since it is well known that every atomic Boolean algebra is isomorphic to the field of sets of all subsets of some set, we have developed the following representation theorem.

Theorem 26 (Representation Theorem). Every computation algebra $(\mathcal{C},I,T,R)$, which may be written

$$(\mathcal{C},+,0,\cdot,1,T_1,\ldots,T_n,\ldots,R_1,\ldots,R_n,\ldots)$$

for $T_i$ in the range of T and $R_i$ in the range of R, is isomorphic to

$$(P(U), \cup, \emptyset, \cap, U, T_1^*, \ldots, T_n^*, \ldots, R_1^*, \ldots, R_n^*, \ldots)$$

where:

(1) All the $T_i^*$ are functions on $U \times U$.

(2) For arbitrary $T_i^*$ and $T_j^*$, there exists a $T_k^*$ such that $T_i^* \mid T_j^* = T_k^*$.

(3) One $T_i^*$ is the identity in $U \times U$.

(4) All $R_i^*$ are functions from $U$ into $U$.

(5) The $R_i^*$ are transitive and dense.

(6) $R_i^*(u) \subseteq u$.

(7) $R_i^*(u') = R_i^*(R_i^* u)'$.

(8) There exists a closed set $v$ and a relation $T_i^*$ such that $T_i^*(v) = U$.

(9) For each $R_i^*$ there exists a $T_j^*$ such that

$$R_i^*(u) = u \cap T_j^*(v)$$

with $v$ as in (8).

Urysohn's metrization theorem states that a topological space that is $T_3$ and for which the second axiom of countability holds is metrizable. A topological space that is $T_1$ and normal is also $T_4$. If a space is $T_4$ then it is $T_3$. In order to prove that a computation space is metrizable, it is necessary to show simply that it is normal and has a denumerable base.

A topological space is normal if and only if for each disjoint pair of closed sets, A and B, there are disjoint open sets, U and V, such that A is a subset of U and B is a subset of V. Computation spaces are obviously normal since every subset of the underlying set is both open and closed.

Computation algebras of interest have cardinality Aleph-one, are atomic, and have Aleph-zero atoms. An element of an atomic Boolean algebra is the supremum of the elements it dominates. In the dual space of such an algebra, an element is the union of the unit sets of its members. The set of all unit sets forms a denumerable base. Thus these spaces are metrizable and discrete.

The space U together with the discrete topology P(U) and the Boolean relations $T_1^*,\ldots,T_n^*,\ldots,R_1^*,\ldots,R_n^*,\ldots$ of Theorem 26 is the dual of the computation algebra (C,I,T,R) and hence may be appropriately called the computation space of this algebra. An entirely similar concept of hardware space is also available.

Let h be a computational homomorphism with kernel M from the computation algebra A onto A/M. By Theorem 14 $h^*$ is a Boolean relation and furthermore a function. Since each function from one Boolean space to another is a Boolean relation if and only if it is continuous, $h^*$ is such a continuous mapping from $A^*$ to $(A/M)^*$. From this it follows that the dual of a hardware structure, a hardware space, is always the range space of some continuous mapping from the computation space. This fact is the dual of Theorem 13 which states that every hardware structure of the transformation-reset-set type is a quotient of the computation algebra.

If we dualize beyond computation algebras to computers we may consider that the control unit dually defines a function from the clopen sets of the hardware space into the Boolean relations on the space. The direct image of a clopen set under the Boolean relation which the control unit assigns to it is the clopen set which is the dual of the next state of the computer.

# CHAPTER IV

# CONCLUDING REMARKS

## Limitations and Advantages of the Algebraic Approach

This paper represents one algebraic approach to the study of computation. In particular, we studied Boolean algebras with operators. This choice suggested itself for several reasons. A good deal of success has been achieved by Halmos and Tarski in finding in Boolean algebras with operators the structures suitable for the explication of the lower predicative calculus. Prior to this, of course, Boolean algebras were shown to be suitable for the study of the sentential logic. More recently, Boolean algebras have been used to explicate matters of set theory that have heretofore been rather opaque [21]. It was felt that Boolean algebras would prove similarly illuminating in the field of computation.

More specifically, we recognize the seemingly natural affinity between matters concerning computers and Boolean algebras. A Boolean algebra is readily definable on the set of states of a computer. Further, many computer operations, i.e. instructions, are Boolean in nature. For these reasons it seemed reasonable to initiate an algebraic study of computation along Boolean lines.

Certain limitations are inherent in this choice, however. The inherent limitations stem from the fact that Boolean algebras have so much structure that a considerable burden is placed on making definitions and theorems. For example, we found in an earlier section that a certain hoped-for homomorphic embedding could not be achieved. It could not be achieved because of an overage of structure, namely distinguished elements, that had to be preserved and the preservation of these elements was incongruent with the nature of control units. Another example of such limitations is imminent in the section on computation spaces. The metric associated with such spaces is not defined on the clopen sets; however, the duals of the items on which we want metrics are in terms of the clopen sets. In general, the limitations on the present approach are manifest in the algebra's being too rich in structure.

Other algebraic approaches might avoid these difficulties. Lattices have less structure than do Boolean algebras, and semi-lattices have even less structure. Therefore, the extension of algebraic theory to hardware structures and computers would be less demanding of an algebra of computation based on lattices or semi-lattices. On the other hand, important contact with other theories, e.g. polyadic algebras and Boolean spaces would be lost.

The duality considerations in this approach suggest an avenue of investigation of measures, through metrics. Difficulties with this avenue of investigation have already been anticipated; however, some suggestions also come to mind for overcoming some of them.

The present approach does have the merit that it fits nicely into the general schema of Boolean algebras with operators advanced by Tarski.

Thus, those general developments accrue naturally to computation algebras. Computation algebras are very similar to polyadic algebras, underlining the close relationship between logic and computation. A complete rapproachment between polyadic algebras and computation algebras should prove interesting and profitable to the theory of computation. At the present stage of the rapproachment, the two algebras have the same topologies. The theory of duality of polyadic algebras banks only on the fact that quantifiers are hemimorphisms. This much they have in common with reset operators. Such close contact with an existing algebraic theory is valuable for perspective, if for nothing else.

We have made the study of computation an object of universal algebraic theory. Computation algebra is in this sense clearly more algebra than computation. In essence, this means that some problems in the field of computation may now be studied with widely-known algebraic methods, which may make the field more palatable to professional mathematicians.

In dealing with contemporary computers, we are able to study them at any degree of detail or generality. The states and state transition functions have been given a great deal of structure so that their characteristics are reflected in the algebra. Further, the basic algebraic concepts pertaining to such facilities have duals in the topology.

## Further Developments

Many questions along the present line of development remain to be investigated. Since most of the theorems in the section on coding and embedding are existence theorems, questions of optimality arise.

It has not been proved that the algebra in which all operations are recovered in transformations is the smallest. A similar statement holds for the algebra based on sets and resets. It is also likely that the constructions given for control units are not in any sense the most desirable. It remains to develop criteria of desirability and optimality and to state and prove the appropriate theorems.

The detailed investigation of several special universal computers would also be useful. Those universal computers whose control units are uniquely determined by their definition on a subalgebra of the universal algebra, would perhaps be of special interest. Similarly, universal computers whose control units are specified by the set of atoms, the set of generators, proper ideals, or proper filters would appear to be of special interest. Control units determined in ways just suggested are systematically related to control units that are preserved under a distinguished set of morphisms.

An alternative to seeking refuge in algebras of less structure than a Boolean algebra would be to narrow the concepts we wish to recover. The study of special universal computers would furnish ways of narrowing such concepts. For example, if we agree that all control units are to call the identity mapping on the states that are the distinguished elements of the algebra, then homomorphic embedding is possible. Such a convention is not unrealistic.

More generally, the question can be posed as to what are the characteristics of a maximal set of computers, all of which can be in some standard way embedded in a particular universal computer. This standard way of embedding may be by monomorphisms, homomorphisms, or hemimorphisms. In the present paper we investigated only homomorphic embedding. In the event that an embedding was not a homomorphism we did not make inquiry as to what lesser sort of embedding was possible. It is clear however that an exhaustive study of embedding possibilities is desirable.

Such a study could result in an algebraic theory of the classification of computers. All computers "morphic" to a given universal computer, which has its control unit prescribed in a special way, would be in the same class. Admittedly, such a classification would be from the point of view of theoretical expediency rather than from the point of view of mirroring modes of classification that exist in practice.

In the theory of duality there is a large number of questions to explore. The previous chapter concluded with a comment to the effect that the dual of a control unit defines a function from clopen sets of the hardware space into Boolean relations on the space. Thus, the obvious effect of the control unit on the topology of the space is that it sends clopen sets into clopen sets. However, this is a first observation and the question remains open as to what is the appropriate dual, if any, of a control unit in a computation space.

The duals of the states of a computer are the clopen sets of the underlying space of the topology. Metrics are defined on the space rather than on the clopen sets of the space. Since in quantitative studies we may be interested in assigning weights to the states of the computer, the question of how to pass from a metric on the space to measures on its clopen sets needs to be explored. This question also bears on the possibility of having a notion of distance between states of a computer. In this case, an alternative question to ask is how to appropriately associate states of the computer (clopen sets) to points in the space such that the duals of the state transitions are preserved.

A final suggestion is the question of how to temporally deploy computers in computation algebras. That is, what are the algebraic facts relevant to all notions of time that preserve the sequentiality of computers?

These suggestions are not exhaustive; they are only indicative of the sorts of algebraic investigations of computers which appear to be viable.

## Summary

We set out to establish a base from which to view broad and general problems in the theory of computation. The combinatory setting provides a formalism in which these problems can be uniformly represented. Since representation proceeds in a piecemeal fashion, any program of representation has limitations as to the nature of the problems that can be solved and the nature of the results that can be obtained. In order to circumvent the process of representation, we obtained the algebraic structure of a portion of combinatory logic that appears to be adequate for the representation of many problems of interest. The appropriate algebraic structures were found to be transformational Boolean algebras with a distinguished element and a distinguished operator.

The universal algebraic concepts of subalgebra, ideal, homomorphism theorem, and representation theorem were developed for computation algebras. These developments proceeded without novelty as straightforward applications of the general algebraic concepts.

Digital computers are at the center of many problems in the field of computation. In order to show that the algebraic theory of computation algebras is capable of dealing with such devices, we represented in the algebra the three key aspects of a digital computer, its memory structure, its elementary actions, and its control unit. Computation algebras directly give the set of states the structure of a Boolean product algebra, with the memory structure being given by the manner in which the product is formed. The elementary actions of the computer were regarded generally as arbitrary mappings of the set of states into itself. In order to recover these actions algebraically, the set of states was coded into a larger set of states in which the operations that mirror the actions preserved half or all of the Boolean structure, depending upon the construction of the larger algebra. Finally, the control unit was represented as a mapping from the set of states into the set of actions. We then homomorphically embedded the computer, that is, the algebra together with the structure preserving mappings and control unit, into a universal computer. The algebraic theory of computation algebras was then extended to computers.

By way of Stone spaces of Boolean algebras and Boolean relations, we were able to obtain the topological dual of a computation algebra. The hemimorphisms on the algebra dualized to Boolean relations on the topological space. The important relationship between computation algebras and their quotients was dualized to continuous mappings from computation spaces to their quotients. Finally, the computation spaces were proved metrizable and it is this property that may point a way to the development of a quantitative theory of computation.

## REFERENCES

1. Church, A. 1937. "Combinatory Logic as a Semigroup." (Abstract) *Bulletin of the American Mathematical Society* 43 : 333.

2. Curry, H. B. and Feys, R. 1958. *Combinatory Logic* Vol. 1. Amsterdam : North Holland.

3. Fitch, F. B. 1942. "A Basic Logic." *Journal of Symbolic Logic* 7 : 105-114.

4. ____________ 1948. "An Extension of Basic Logic." *JSL* 13 : 95-106.

5. Gorn, S. 1968. "The Identification of the Computer and Information Sciences : Their Fundamental Semiotic Concepts and Relationships." *Foundations of Language* 4 : 339-372.

6. Halmos, P. R. 1954. "Polyadic Boolean Algebras." In *Proceedings of the National Academy of Sciences* 40 : 296-301. Reprinted in P. R. Halmos; *Algebraic Logic*.

7. ____________ 1956. "Algebraic Logic II." *Fundamenta Mathematicae* 43 : 225-325. Reprinted in P. R. Halmos, *Algebraic Logic*.

8. ____________ 1966. *Algebraic Logic*. New York : Chelsea.

9. Henkin, L. and Tarski, A. 1957. "Cylindric Algebras." In *Summaries of Talks Presented at the Summer Institute of Symbolic Logic*. 3 : 332-340.

10. ____________ 1960. "Cylindric Algebras." In *Proceedings of Symposium in Pure Mathematics, II*. pp. 83-113.

11. Jonsson, B., and Tarski, A. 1951. "Boolean Algebras with Operators I." *American Journal of Mathematics* 73 : 891-939.

12. ____________ 1952. "Boolean Algebras with Operators II." *Amer. J. of Math.* 74 : 127-162.

13. McCarthy, J. 1963. "A Basis for a Mathematical Theory of Computation." *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. Amsterdam : North Holland.

14. ____________ 1963. "Towards a Mathematical Science of Computation." *Proceedings of the IFIP Congress Munich, 1962*. Amsterdam : North Holland.

15. Orgass, R. J. 1967. "A Mathematical Theory of Computing Machine Structure and Programming." Doctoral Dissertation, Yale University, New Haven, Connecticut. Available as IBM Research Paper RC-1863, IBM Corp., Yorktown Heights, N. Y.

16. ____________ 1968. *Problems in the Theory of Programming.* IBM Research Paper RC-2236, IBM Corp., Yorktown Heights, N.Y.

17. ____________ and Fitch, F. B. 1969. "A Theory of Computing Machines." *Studium Generale* 22 : 83-104.

18. ____________ 1969. "A Theory of Programming Languages." *Studium Generale* 22 : 113-136.

19. Poore, J. H., Baralt-Torrijos, J., and Chiaraviglio, L. 1970. "On the Combinatory Definability of Hardware." (Forthcoming.)

20. ____________ 1970. "On the Combinatory Definability of Software." (Forthcoming.)

21. Rosser, J. B. 1969. *Simplified Independence Proofs: Boolean Valued Models of Set Theory.* New York : Academic Press.

22. Sanderson, J. G. 1966. "The Theory of Programming Languages--A Survey." In *Proceedings of the Australian Computer Conference.*

23. Tarski, A. 1954. "Contributions to the Theory of Models." *Proceedings of the Royal Academy of Science of the Netherlands, ser. A.* 57 : 572-588.

## VITA

Jesse H. Poore, Jr. was born in Louisville, Kentucky on July 20, 1942. After graduating from Campbellsville High School, Campbellsville, Kentucky, in 1960, he attended the University of Kentucky, 1960-1961, the Louisiana Polytechnic Institute, 1962-1965, Memphis State University, 1966-1967, and the Georgia Institute of Technology, 1968-1970. The degrees of Bachelor of Science in Mathematics, Master of Science in Mathematics, and Doctor of Philosophy in Information and Computer Science were earned at the last three institutions named, respectively.

Mr. Poore was employed as a programmer in the computing centers of the University of Kentucky and the Louisiana Polytechnic Institute, as a Mathematics Programmer for the Operations Research Department of Armour and Company, Chicago, Illinois, as Director of the Memphis State University Computing Center, and as a Research Associate in the School of Information and Computer Science, Georgia Institute of Technology, while attending the universities.

Mr. Poore has five publications, has read two papers before regional conferences, and has three publications pending. He has been a member of the Association for Computing Machinery since 1961, is founder and former chairman of its Memphis Chapter, is a former faculty advisor of its Georgia Tech Student Chapter, and is a reviewer for the *Computing Reviews*.