23

Representations and Modelling in Problems of Program Formation

S. Amarel

Rutgers University
New Brunswick

Abstract

We consider a class of problems of formation type where the goal is to construct by computer a program – in a given programming language – that satisfies a finite number of conditions in the form of given input-output correspondences. A variety of automatic design problems, as well as problems of theory construction in empirical sciences, are of this general type. The main problems in the design of procedures for the solution of formation problems are: selection of an 'appropriate' grammar for specifying the language in which candidate solutions are to be represented; formulation of evaluation procedures for ordering candidate solutions; and formulation of control strategies for generating 'intelligently' a sequence of candidate solutions which converges efficiently to the desired solution. The problems of representation, evaluation, and control are closely coupled; however, dominant among them is the problem of representation.

A promising approach to the choice of a grammar for the solution language is to find a model corresponding to the grammar, for which it is moderately easy to evaluate the relative merit of candidate solutions (in terms of their satisfying the given problem conditions) by analyzing their representations in the model. In the program formation problem we have found such a model in a mathematical system which is a modified algebra of relations. The atomic elements of the system correspond to elementary commands in the given programming language and its operations correspond to program compositions. The rich structure of relationships in the mathematical model provides a framework for reasoning about the choice of candidate solutions during the problem-solving process. Our work to date suggests that representation systems with considerable algebraic structure are necessary for computer formation procedures of reasonable power.

1. INTRODUCTION

In this paper we are concerned with the design of automatic program writing procedures. In particular, we consider situations where functional properties of a computer program are given in the form of explicit input-output correspondences, and the problem is to synthesize (to form) a program, in a given programming language, that satisfies the given correspondences.

The motivation behind the work is twofold: (1) to explore and elucidate schemes for solving formation problems by machine, and (2) to gain further insight into questions of representation in problem solving, to examine their nature and significance in the context of formation procedures, and to clarify the rôle of models in the solution of formation problems.

The present approach to program formation derives from work done several years ago (Amarel 1962a, 1962b), in which we found it conceptually fruitful to view the automatic formation of computer programs that is based on computational examples as a case of *automatic theory formation.*

A theory in an empirical science emerges from a body of observed correspondences and has the function of an intellectual mechanism for explanation and prediction. The theory evolves from a succession of hypotheses that are tested against experience, and attains a degree of stability once it explains with reasonable consistency the given body of observations. The form of a hypothesis capable of emerging in a scientific culture depends on the language, the basic concepts, and the schemas that are available in the culture.

There appears to be a strong analogy between a scientific theory – its nature and the dynamics of its creation – and a computer program which is formed by a computer (after a process that involves the generation and test of several candidate programs) with the objective of accounting for a body of data correspondences in terms of a language that is available to the machine, and that the machine can effectively use.

There are many aspects of theory formation, however, that are not captured by our approach to program formation. For example, questions of accuracy and error of observation, and questions of induction, are outside the scope of the present work. Nevertheless, any serious attempt to mechanize theory formation processes is likely to encounter problems of the type that we are facing in the context of the present program formation problem; in particular, problems of hypothesis representation and of effective hypothesis generation.

Our problem of program formation can be characterized more directly as a *constraint satisfaction problem.* A large number of functional conditions is imposed on a desired structure, and the machine is asked to synthesize the structure by using a specified repertoire of construction material and of assembly techniques. This is a typical problem of design, of a kind that abounds in the world around us. In section 2 we shall discuss in more detail the nature of formation problems, and their relationship to problems of other types, in particular to derivation problems.

In order to formulate a problem for solution by machine, we must specify at least a language for constructing solutions, and the set of problem conditions that are to be satisfied by the desired solution. Usually, we also provide the machine with a body of knowledge which is relevant to the problem class under consideration, and which can be used to guide the process of constructing candidate solutions in the given language. The knowledge may consist of fragmentary facts and advice about the problem class, or it may represent a well-developed theory.

It is also our responsibility to provide the machine with a procedure which embodies a problem-solving strategy, and which is able to accept the problem formulation and to direct as intelligently as possible the process of searching for a solution.

The specific form in which the problem is formulated for the problem-solving procedure has a strong effect on the nature and efficiency of the solution-finding process. Of particular importance are the ways of describing the solution language, the amount and quality of available knowledge, and the specific forms in which this knowledge is made available to the procedure.

An important aspect of the *problem of representation in problem solving* is to explore processes for shifting problem formulations in the direction of an increase in problem-solving power. We have studied this problem extensively in the context of derivation problems; for recent accounts of this work see Amarel (1970a, 1970b). In the present paper we are making an initial attempt to study problems of representation in the context of formation problems.

Since our main objective is to develop and study design principles and problem-solving schemes for formation procedures, we have decided to continue our work in the context of the relatively simple class of program formation problems that we introduced in Amarel (1962b). This problem class is described in section 3.

The study of representations in program formation will focus on a variety of modes for describing program languages, on the rational for choosing descriptions, and on algebraic models in which formal properties of programs can be obtained.

Our work in these areas has direct relevance to another important domain of activity in computer science where the main concern is with the development of theoretical approaches to the study of programs.

2. A CLASSIFICATION OF PROBLEMS: FORMATION PROBLEMS

In order to facilitate the study of problem-solving processes and to enable the transfer of results from one problem class to another, it is useful to distinguish between families of problem classes for which solution construction processes are significantly different. An examination of the different ways in which specific problem conditions control processes of solution

construction, suggests a system for classifying problems where *derivation problems* and *formation problems* occupy polar positions.

In problems of derivation type the situation is roughly as follows. We are given specific problem conditions in the form of *parts* of a solution description, and we are asked to complete the description by using given rules for solution construction – in such a manner that the initially given parts will be well integrated in the solution structure. Usually, the specific problem conditions specify *boundary* parts of the solution, and the problem-solving process consists of finding a connecting structure between the given boundaries. It is characteristic of a derivation problem that the specific problem conditions are given in the language of solution structures. A typical derivation problem is the problem of finding a proof to a theorem in a formal system.

In formation problems, the situation is, in general, more complex. The problem conditions are in the form of properties that the solution *as a whole* must satisfy, and we are asked to generate a solution description within a given language of solution structures, where no choice of solution element or partial combination of solution elements can be determined directly by the given problem conditions. The solution process cannot proceed by reasoning from the problem conditions to specific parts of the solution – as is possible in derivation problems. The general approach here is to generate candidate solutions in the given language of solutions, and to test them against the problem conditions. The formulation of a formation problem includes the specification of a mechanism (a mapping) that can take the description of a proposed solution in the language of solution structures, and transform it into a statement in the language of conditions, where it can be evaluated in the light of the given problem conditions. However, it is characteristic of 'pure' formation problems that this mechanism cannot be used in reverse; thus problem conditions cannot guide in any substantial way the process of generating candidate solutions. This is the cause of much difficulty in formation problems.

The essence of a successful problem-solving activity in a formation problem is to generate a sequence of candidate solutions which converges efficiently to the desired solution. The process can be guided by the known performance of previously generated candidate solutions, and by certain heuristic assumptions about relationships between structural changes in solutions and the effects of such changes on solution performance.

It is characteristic of a formation problem that the problem conditions are not given in the language of solution structures. An important problem in this area is to find ways of bridging the gap between the language of solution structures and the language of conditions. A promising approach is to find a *theoretical model* where solution structures can be represented, and where it is possible to evaluate the performance of different structures (from the viewpoint of satisfying the problem conditions) by analyzing their relation-

ships in the model. This is the approach that we shall be developing in this paper.

If we augment the representation of a formation problem with a theoretical model where structure-performance relationships can be analyzed, then we are effectively transforming it into a set of derivation problems that are much easier to solve. Such a transformation is possible for the problem presented in this paper, and it is discussed in sections 13, 14, and 15.

Many important real-life problems can be placed in some intermediate position between derivation and 'pure' formation problems; they are quasi-formation problems. The position in the scale depends on the amount of theoretical knowledge which is available about the structure-performance relationships in the space of solutions, and also on the quality of the existing methods for using the available knowledge in the solution-finding process. In quasi-formation problems, theoretical knowledge is not complete, and there are no perfect ways of using it for guiding problem-solving actions. To solve problems in these areas we must combine modes of reasoning that take us from problem conditions to parts of solutions, and also from candidate solutions to the problem conditions. Many examples of such problems exist in engineering and architectural design, in economic planning, in the interpretation of scientific data, and in complex diagnostic tasks.

An interesting example of a computer-based system that is concerned with a quasi-formation problem is the DENDRAL system at Stanford (Buchanan, Sutherland and Feigenbaum 1969). The problem of DENDRAL is to find the molecular structure of a substance given the result of its mass spectrographic analysis as well as of other tests, and also given a language of molecular structures (that is, a language of solutions) and a considerable amount of knowledge in a form which is suitable for controlling the generation of candidate solutions in the light of the given tests on the substance. This problem is basically a formation problem whose representation is much strengthened by added theoretical knowledge about chemistry and mass spectrography. The DENDRAL system provides an excellent vehicle for the study of uses of relevant theoretical knowledge in the context of formation problems.

One of the most effective ways of improving the power of procedures for solving a given class of quasi-formation problems is to increase the theoretical knowledge which is available to the procedure, to find a better (more complete, more relevant) model for the solution space and to formulate more efficient strategies of problem solving on the basis of the new knowledge. The work presented in the following sections is a study of the processes involved in carrying out such an improvement, starting from a 'pure' formation problem.

3. A CLASS OF PROGRAM FORMATION PROBLEMS

We consider a class of problems whose initial representation is as follows:

Given,

(1) a data base in the form of a finite partly ordered set $\langle \mathfrak{S}, I, E \rangle$ where $\mathfrak{S}$ is a set of n elements, I is a proper inclusion relation defined for all the elements of $\mathfrak{S}$ and E is the equality relation;

(2) a language of 2-input *constructible programs*, with data domain $\mathfrak{S} \times \mathfrak{S}$, for performing operations over the data base; the language can handle at the basic operation level the relations I, E, and the converse relation to inclusion ('is included by') which we denote by $\hat{I}$;

(3) a mapping θ: $\mathfrak{S}^2 \rightarrow \mathfrak{S}$ in the form of a finite set of correspondences C_θ,

$$C_\theta = \{((x_1, x_2), x_3) | x_1, x_2, x_3 \in \mathfrak{S}, \ \& \ x_3 = \theta(x_1, x_2)\}$$

find, a 'simple' program p_θ in the given language of constructible programs that computes θ over all the correspondences C_θ, or comes close to computing all the given correspondences.

Figure 1 is an example of a problem from this class:

$\mathfrak{S} = \{a, b, c, d\}$,

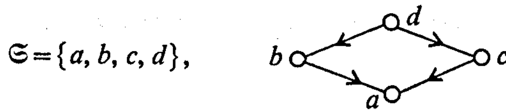


Figure 1

where the arrows in the lattice denote inclusions (that is, dIb is true). The mapping θ is the *Infimum* mapping that produces the greatest lower bound (*glb*) for any two elements in $\mathfrak{S}$. Thus,

$$C_{\text{Infimum}} = \{((a, a), a), ((a, b), a), \ldots, ((b, c), a), \ldots ((d, d), d)\}.$$

The central element in the formulation of the problem class is the specification of the language of constructible programs from which the solution, that is, the 'desired program' that satisfies C_θ, is to be constructed. The power of this language and the specific form in which it is described have an important effect on the ability of a problem-solving system to solve problems in the class. There will be problems for which the given language cannot, in principle, express a perfect solution, that is, no program can be constructed in the language that satisfies the entire set of given correspondences. Under these conditions, the search for solution can never terminate unless a ceiling is imposed on the amount of available effort. According to the problem statement, the non-perfect program that the system produces under these circumstances should satisfy as many correspondences in C_θ as possible.

The initial specification of the language of constructible programs will be given in sections 5 and 6.

The requirement that the solution be simple, comes from reasons of operational efficiency, and it is also desirable from the viewpoint of efficiency of the formation process. A simple program will be easy to run and to test, and it will be also more manageable to manipulate, analyze, and restructure.

The class of problems that we have described are clearly of the 'pure' formation type. The goal of these problems is to synthesize automatically a

program that satisfies a large number of conditions that are specified in a language which is different from the given language of programs. Given a constructible program, it is possible to execute it in a computer over its data domain, and to test how well it satisfies the required conditions. However, we cannot go back from functional conditions to structural properties of programs.

4. PROCEDURES FOR SOLVING PROGRAM FORMATION PROBLEMS; REPRESENTATIONS

We are concerned with the design of procedures that can accept a representation of the program formation problem, and that can generate a solution for any problem in the class as efficiently as possible. We have described procedures of this type in previous papers (Amarel 1962a, 1962b).

The main problems in the design of procedures for solving formation problems are:
(1) selection of appropriate descriptions for the language in which candidate solutions are to be represented;
(2) formulation of evaluation procedures for ordering candidate solutions according to their performance vis-à-vis the desired goal;
(3) formulation of control strategies for directing intelligent searches in solution space.

The problems of representation, evaluation, and control are closely coupled; however, dominant among them is the problem of representation. In our previous work on program formation, we have given little attention to the significance and implications of choices of representation for the formation problem, thus leaving open many questions about the rational and domain of applicability of proposed schemes for evaluation and control. Our main concern here will be with the problem of representation, and with associated problems of modelling.

In the following sections we shall first give an initial definition of the language of constructible programs that will constitute a first step in an evolution of program descriptions. We shall then introduce context-free grammars that can be used for describing the language of programs and for assigning structural descriptions to individual programs. The structural descriptions of programs will then be used for projecting the programs in an algebraic model where they can be manipulated as relational expressions and where structure-performance relationships can be analyzed. The algebraic model will be shown to provide the basis for a radical change in representation of the program formation problem; this change will induce a new approach to the strategy of solution.

5. LANGUAGES OF PROGRAM STATEMENTS

Program statements are to be interpreted as computer instructions in a symbolic assembly language. We shall use two languages for program

statements. The first, called L_l, is in conventional linear form; the other, called L_g, is in a new graphic (two-dimensional) form. The language L_l is well matched to present-day computers, where the flow of control is tightly specified by the program. The other language is better suited for the design and study of programs by people; it is also suitable for program execution in machines where the flow of control is mainly determined by the flow of data and is only minimally specified by the program. The elements in the languages L_l and L_g are closely related, as will be shown below.

5.1 The linear language of program statements L_l

The main elements of a program statement in L_l are names of entities of different kinds: identifiers for different types of data, labels of program statements, and names of basic (executable) operators. We assume that the supply of names for data variables and for labels is inexhaustible, and also that each name carries information about the *type* of entity that it refers to.

Specifically, the vocabulary of L_l is as follows:

(a) For $i=1, 2, \ldots$

u_i: names of variables that can take as values elements in $\mathfrak{S}$ (these are n constants that are identified by a set of reserved symbols) or the symbol ϕ which denotes the empty set.

q_i: names of variables that can take as values subsets of $\mathfrak{S}$. Each subset is defined explicitly as a list of its elements; again, the empty set is denoted by ϕ. Formally, the values of these variables are elements of the power set of $\mathfrak{S}$, that is, of $2^{\mathfrak{S}}$.

g_i: names of variables that can take as values lists of elements in $2^{\mathfrak{S}}$, that is, lists of lists of elements in $\mathfrak{S}$.

α_i: labels of program statements.

(b) η, $\hat{\eta}$, ε: computational operators denoting mappings that correspond to the relations I, $\hat{I}$, E in the following way: if c is the value of a variable u_i, then the application of an operator, say η, on u_i produces the set of all elements z in $\mathfrak{S}$ for which cIz holds.

(c) $\cup$, $\cap$: computational operators denoting set union and intersection respectively.

(d) s, c: list-processing operators denoting *selection* and *collection* respectively; their meaning will be claiified shortly.

(e) $\rightarrow$: assignment operator.

(f) The colon ':' and the comma ',' are used as separators.

It should be pointed out that while the entities u_i, q_i, g_i in (a), and the operators in (b) are specifically related to the universe of discourse that we have chosen, the other entities in this vocabulary are more general and they can be used in a wider context.

The *rules of formation of* L_l are specified by listing the acceptable *statement forms*. These forms have three main parts that are separated by colons. The left and right parts are devoted to control; the left part contains a label

that names the statement, and the right part has one or two labels that constitute 'go to' links. The middle part specifies an assignment operation, and it has two parts that are separated by an assignment arrow. The left part specifies a computational or list-processing operation, and it contains an operator and one or two *input variables* (on which the operator is to be applied); the right part specifies the *output variable* to which the results of the left operation is to be assigned. The specific statement forms of L_t are as follows:

η-st(atement), $\hat{\eta}$-st, ε-st

$$\alpha_t : \theta u_i \rightarrow q_j : \alpha_v$$

Interpretation: when control is at α_t, compute the mapping θ (this can be η, $\hat{\eta}$ or ε) for the current value of u_i, assign the result to q_j and g_0 to α_v. If the value of u_i is ϕ, then use the convention $\theta\phi = \phi$.

S-statements: $\cup$-st or $\cap$-st

Here we have two forms. The first is as follows:

$$\alpha_t : Sg_i \rightarrow q_j : \alpha_v \qquad (1)$$

Interpretation: S stands for $\cup$ or $\cap$. If the current value of g_i is a list with two or more entries (each entry is a subset of $\mathfrak{S}$) then execution of the operation $\cup g_i$ produces the union of these subsets, and the result is to be assigned to q_j; a similar interpretation holds for $\cap g_i$. If the current value of g_i is a list with a single entry, say ψ, then the computational convention $S\psi = \psi$ is to be used. After execution of the set operation, the value of the variable g_i becomes *undefined* (the previous list of lists is erased) and control goes to α_v.

The second form of S-statement is as follows:

$$\alpha_t : Sq_i q_j \rightarrow q_k : \alpha_v \qquad (2)$$

Here the arguments of the set operation are taken explicitly as the values of two variables q_i and q_j, and an appropriate value assignment is made to q_k.

s-st

$$\alpha_t : sq_i \rightarrow u_j : \alpha_v, \alpha_w$$

Interpretation: This is a *selection* operation where a pointer focuses attention on a symbol of the list which constitutes the current value of q_i and assigns it to u_j. The value of q_i can be either a list of symbols denoting elements of $\mathfrak{S}$ or a list with the single symbol ϕ. Initially, the pointer points to the top of the list. Let us consider two cases: (1) when control is at α_t, and a list symbol is under the pointer, then the symbol is assigned to u_j, the pointer is moved to the next list entry, and control goes to α_v; (2) if no list symbol is under the pointer (or if, say, a list termination marker is found) then the pointer is reset to the top of the list, and control goes to α_w.

c-st

$$\alpha_t : cq_i \rightarrow g_j : \alpha_v$$

Interpretation: when control is at α_t, assign the current value of q_i to g_j, as the next entry in the list that constitutes the value of g_j (this is to be the first list entry if the current value of g_j is undefined), and go to α_v. This statement effects a *collection* operation that builds a list of lists (subsets of $\mathfrak{S}$) as a value of g_j, in preparation for a set operation on these lists [*see* the form (1) of $\cup$-st and $\cap$-st].

5.2 The graphic language of program statements, L_g

The graphic language L_g provides the basis for a new representation of programs where the structure of *data flow* is clearly portrayed. This is in contrast to conventional flow charts of programs where the emphasis is on the flow of control. Each program statement in L_l is represented in L_g in an abstracted form where no explicit names are used to identify variables and labels. *Types of variables* are shown, as well as the operators applied to them, the flow of assignments, and the type of control associated with a given statement.

The *vocabulary of* L_g is as follows:

(a) Types of variables, in the form of *nodes* of different kinds;

|o: variable of u-type,
|: variable of q-type,
||: variable of g-type.

(b) There are two kinds of directed branches:

→: for the flow of data,
--→: for the flow of control.

(c) η, $\hat{\eta}$, ε, $\cup$, $\cap$, s, c: The interpretation of these symbols is as in L_l; they are used to label *directed branches* that denote processes on data.

The *statement forms* that are acceptable in L_g are as follows:

η-st, $\hat{\eta}$-st, ε-st. See figure 2(a).
Interpretation: θ can be η, $\hat{\eta}$ or ε. The labeled solid branch denotes a process that takes u-type data at left and transforms it, via application of the operator θ, to q-type data. The dashed arrows denote entry and exit of control.

S-statements
Corresponding to the two forms in L_l, we have the two forms shown in figure 2(b) and 2(c) where S stands for $\cup$ or $\cap$.

s-st. See figure 2(d).
This is the only statement form that has two exits of control. The exit which is farther away from the solid branch is, by convention, that taken after an 'end of list' is recognized by the pointer which scans the q-type input data; it is the second control exit in the description of the statement form in L_l.

c-st. See figure 2(e).

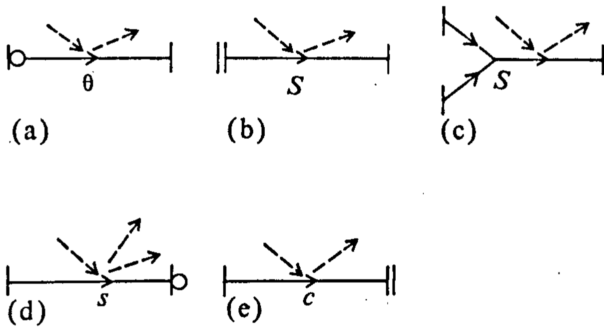


Figure 2

6. PROPERTIES OF CONSTRUCTIBLE PROGRAMS

It can be seen from the definition of the program formation problem that constructible programs are precisely those programs that the program formation system can generate, execute, and manipulate in its search for a solution to the problem.

Constructible programs are made of program statements. The set of constructible programs forms a *language of programs.* The vocabulary for this language is provided by the language for program statements. A specification of the language of programs amounts to specifying possible ways of combining program statements into constructible programs.

A physical analogy may be useful at this point. We can regard the vocabularies of L_l or L_g as corresponding to inventories of sub-atomic particles, a program statement as corresponding to an atom, and a program as corresponding to a molecule that is constructible in certain specified ways from atoms. Thus, a language of program statements corresponds to a set of available atoms, and a language of constructible programs to a set of constructible molecules that can be formed from the available atoms. In this light, our problem of program formation can be seen as analogous to the problem of synthesizing a constructible molecule that satisfies a specified set of functional requirements [or to the problem of discovering the structure of a complex molecule given its mass spectrogram: this is the problem which is currently under investigation in the DENDRAL project at Stanford University (Buchanan, Sutherland and Feigenbaum 1969)].

We intend to characterize the language of programs in terms of generative rules of formation that specify the possible ways of combining elements of the language into valid (constructible) programs. There is no unique way of doing this, and each may result in a different representation of the program formation problem from the point of view of a problem-solving system.

Before discussing the choice of generative rules for the language of programs, we must agree on some 'initial' specification of the set of constructible programs. We shall introduce next such a specification in terms of certain

structural properties of constructible programs. This specification is to be taken as an assumed starting point in the development of generative grammars for the language of programs.

One of the most significant features of a program is its *loop structure.* In our constructible programs we assume that only simple nesting of loops of a given type is permitted. This is summarized in the following property:

Loop property of constructible programs: s-statements and c-statements are used in pairs to form loops in programs. A loop has the following form when expressed in terms of statements in L_l:

loop entry $\rightarrow \alpha_t : sq_i \rightarrow u_j : \alpha_v, \alpha_y$
$\alpha_v : \pi u_j \rightarrow q_k : \alpha_w$
$\alpha_w : cq_k \rightarrow g_l : \alpha_t$
$\alpha_y : Sg_l \rightarrow q_m : \alpha_z \rightarrow$ loop exit

Figure 3

In this program form, π denotes a *variable* whose possible values are constructible programs that specify mappings from u-type variables to q-type variables; S stands for one of the set operations $\cup$ or $\cap$. The same loop can be expressed in terms of statements in L_g as shown in figure 4.

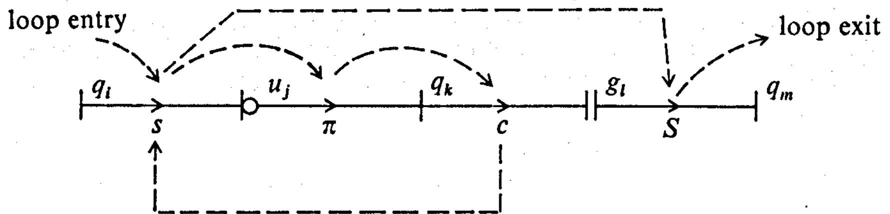


Figure 4

We label data nodes in the graphic representation (figure 4), so that correspondence with the linear representation (figure 3) can be seen more clearly. In general, no labelling of nodes is required for graphic representation of programs.

A loop can have only one input variable (q_i in the previous definition) and one output variable (q_m in the definition). The other variables that appear in the loop are completely local to the loop.

Only loops of this type are permitted in constructible programs.

Note that the program π may itself include one or more s–c loops. However, these loops must be simply nested. Also, no assignment of values to variables of π can be made from outside the s–c loop in which π is included. The entire s–c loop is a one-source, one-destination *program scheme*, which can be regarded as a *macro-statement* form. If the elements π and S of the program scheme are specified, then it becomes a well specified (and executable) macro-statement.

We shall now use the notion of a s–c loop macro-statement in the specification of a new graphic language L_g^* of program statements which provides a a convenient basis for a characterization of constructible programs. The language L_g^* is closely related to L_g but it excludes the description of control flow.

6.1 The modified, graphic language of program statements, L_g^*

The vocabulary of L_g^* includes the vocabulary of L_g, with the exception of the dashed arrow (that designates control flow) and the symbols s, c. In addition, the vocabulary of L_g^* contains the symbols $l_i (i=1, 2, \ldots)$ that serve as identifiers for loop macro-statements. The well-formed statements of L_g^* include the statement forms of L_g, with the exception of the s-st, the c-st, and the (a) form of the S-st, and with the dashed arrows omitted. In addition, the statement form shown in figure 5 is introduced:

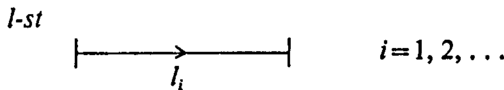


Figure 5

The l-st represents a loop macro-statement. It is an abbreviation of the s–c loop whose representation in L_g is given in figure 4.

6.2 Characterizing constructible programs by river-like graphs

We can now proceed to a global characterization of constructible programs.

The data flow of a constructible program can be represented by a set of directed graphs of a special kind that we call *river-like graphs.*

A river-like graph is constructed from elements of L_g^* in such a manner that the overall graph has the general form of water flow in a river system. In such a system, river channels merge, diverge, and run side by side in a pattern of motions whose general direction is from river sources to the river mouth. Similar systems are found in blood vessel configurations and in nerve assemblies. Warren McCulloch has used for them the term 'anastomatic systems' in the context of his studies of logically stable neural networks (McCulloch 1965).

Two statements in L_g^* can be aggregated to form a part of a river-like graph if the output node type of the one matches an input node type of the other. As an example, the elements $\triangleright\!\xrightarrow[\eta]{}\!\dashv$ and $\vdash\!\xrightarrow[l_1]{}\!\dashv$ can be connected to form the aggregate $\triangleright\!\xrightarrow[\eta]{}\!+\!\xrightarrow[l_1]{}\!\dashv$.

The nodes of a river-like graph that correspond to river sources stand for the input variables of the program that the graph represents. In our case, we permit a river-like graph to have only one or two sources. Each source (input) node has no entering branches, but it has one or more outgoing

branches. The node that corresponds to the river destination stands for the output variable. No 'river delta' structures, resulting in more than one river mouth, are permitted in our graphs. The destination node has a single entering branch and no outgoing branches. For each intermediate node of river-like graphs (excluding the source and destination nodes), only one branch can enter the node, but one or more branches can leave it.

The main feature of river-like graphs is that *they have no loops*. Note that the loops that are being considered here are data flow loops and they should not be confused with program control loops.

Any element l in a river-like graph can be defined separately by specifying its constituents π and S (see the definition in figure 4). The π part can be any one-input (u-type), one-output (q-type) constructible program, with a structure which is also representable by a river-like graph. The S can be specified as $\cup$ or $\cap$.

Thus we can characterize constructible programs as those (and only those) that are representable by a finite set of river-like graphs, one representing the overall input-output data flow, and the others representing the loops in the program (one graph for each loop).

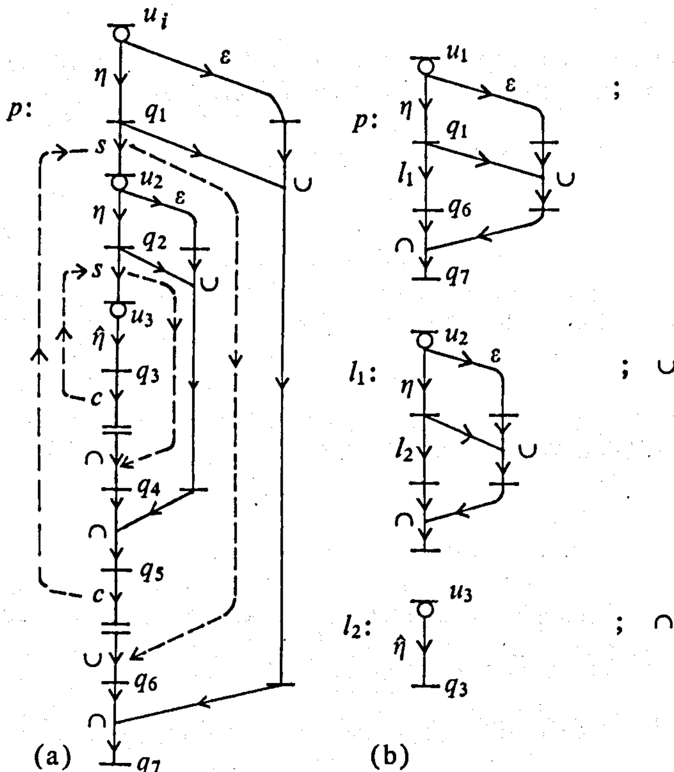


Figure 6. A constructible program p and its representation in terms of river-like graphs: (a) program p in terms of the language L_g; (b) program p in terms of river-like graphs.

The representation of a program by a river-like graph does not include any control information. However, because of the relative simplicity of program loop structures in our case and because of the lack of unconditional program branches, the flow of control can be determined in a straightforward way by the data flow. There are several ways of achieving this but we shall not discuss them here.

Thus we can always go from the representation of a program in terms of river-like graphs to its representation in terms of the language L_g or L_l which is appropriate for execution by computer.

An example of a constructible program and its representation by a set of river-like graphs is shown in figure 6. At left the program p is presented in terms of L_g; most of the control flow is not shown, except for the major lines of control in the two nested loops. It can be seen that this program is composed of 15 program statements, one for each element of L_g that enters in the specification of its structure. At right (top) the overall structure of p is shown as a river-like graph that contains a loop scheme l_1; below this graph, l_1 and l_2 are specified each by a pair, the first element of which is the river-like graph of the part π of l (*see* definition in figure 4), and the second part is the specification of the set operation in the loop. For convenience in establishing correspondence between different points of the program representations, several of the data variables are labelled in these graphs.

6.3 Equivalence classes and normal forms of program structures

An interesting subclass of river-like graphs is characterized by the condition that for each intermediate node (excluding the source and destination nodes) a single branch enters a node and also a *single branch* leaves it (in contrast to the general case, where more-than-one branches can leave an intermediate node). Let us call river-like graphs of this type *decoupled graphs.* A constructible program represented by a decoupled graph is such that the output variable of a program statement (except for the program output) is used precisely once as an input variable in another statement of the program. Under these conditions, it is possible for the program to specify the same computation more than once if the result of the computation is to be used more than once as input of subsequent computations. Clearly, in the latter case it is possible to simplify the program by eliminating the redundant computations and by using intermediate results more than once. The simplified program, however, would not be representable any more by a decoupled river-like graph.

We are using here the notion of program simplification in the sense that the two programs under consideration are *functionally equivalent*, and their structures differ as follows: in the simplified program, some or all of the repetitions of program statements that occur in the other program are eliminated, and this also results in certain obvious differences in 'connections' between intermediate data variables in the two programs.

The converse to the process of simplification, namely, an *expansion* process, is also possible. There exists a relatively simple procedure for expanding a river-like graph which is not a decoupled graph into an equivalent decoupled graph. Again, equivalence is meant here in the sense of functional equivalence between the programs that are represented by the two graphs.

In view of the expansion property of river-like graphs, we can partition the set of constructible programs into *equivalence classes*, such that all the programs in a class are functionally equivalent, and each class has a decoupled graph as its *normal form representative*; the other members in the class are simplifications of the normal form representative of the class.

Let us consider as an example the program p shown in figure 6. The river-like graphs representing p in figure 6 are not decoupled graphs. In figure 7 we show the decoupled graph representations of a program p' which is functionally equivalent to p. It can be seen that p' contains two statements in excess of p; they are repetitions of the ηu_1 and ηu_2 computations.

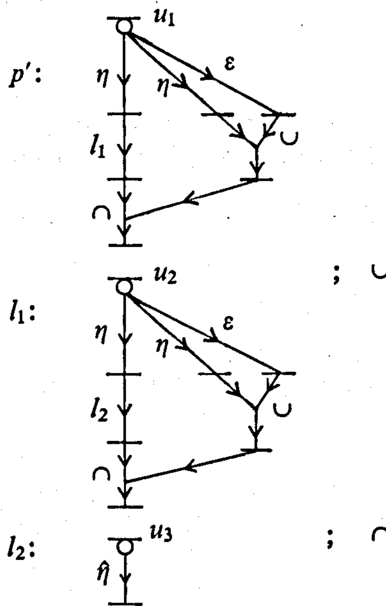


Figure 7. A decoupled graph representation of a program p' which is functionally equivalent to the program p in figure 6

Now each constructible program is either directly representable by a decoupled graph, or it has a functionally equivalent program which is representable by a decoupled graph. Thus, from the point of view of a formation system which is searching for a constructible program that satisfies certain functional conditions, it is sufficient to restrict attention to the set of normal form structures that are characterized by decoupled graphs. We shall con-

sider, therefore, the set of decoupled river-like graphs as the solution language for our formation program.

7. AN APPROACH TO THE DESCRIPTION OF THE LANGUAGE OF PROGRAMS

We are interested in grammatical descriptions of the language of programs that have the form of generative rules for program construction. Descriptions of this type are well suited for problem-solving systems where programs have to be generated systematically, and their descriptions have to be analyzed and manipulated.

Our language of programs can be described in terms of a *context-free* (*CF*) *grammar*. Given such a description, we can assign to each program in the language a *structural description* with respect to the *CF* grammar. A structural description of a program articulates the structure of the program in terms of types of combinations of program statements that are defined in the grammar. Our basic approach to the problem of automatic program formation has been to consider and manipulate programs in terms of their structural descriptions (Amarel 1962b). This approach is guided by the expectation that certain regularities exist between features of a structural description of a program and its functional characteristics, and also that such regularities can be exploited in the formulation of efficient procedures for search over the elements of the program language.

Since there is no unique way of describing the language of programs in terms of a *CF* grammar, then the question arises how to choose a grammar such that it assigns 'appropriate' structural descriptions to programs. Appropriateness here is meant in the sense of leading to the discovery of regularities between program structure and function that can be used in the process of searching for a solution over the elements of the program language. The problem of choosing among possible grammars of the language of programs is at the crux of the *problem of representation* in our present formation problem. An important aspect of this difficult problem involves relationships between different grammars of a solution language (in our present case, it is the program language) and the efforts needed to obtain a solution by a problem-solving system which uses the different grammars. [This problem was discussed in the context of derivation problems in Amarel (1970b).]

In previous work on the present program formation problem [in Amarel (1962b) and in unpublished subsequent explorations] we have specified several program grammars, and we have used them for heuristic search over the set of constructible programs with varying degrees of success. It became increasingly evident from this exploratory work that changes in language description, that is, changes in the framework for representation of the structure of programs, have a strong effect on the ease with which the formation process can proceed to solution. Our initial approach to the problem

was (a) to specify an essentially arbitrary *CF* grammar for the language of programs, (b) to assume that certain types of regularities exist between structural descriptions of programs (based on this *CF* grammar) and the performance of these programs relative to the given functional conditions that a program is to satisfy, and (c) to utilize these assumed regularities in the formulation of a strategy for heuristic search in program space. Our initial emphasis was on problems related to (c). Subsequent attempts to improve strategies of search have forced us to examine the assumptions of regularities made in (b), and in general to establish on a firmer ground relationships between structural descriptions of programs and their corresponding functional properties. As could be expected, for some *CF* grammars it was very difficult to find consistent relationships that could provide a rational basis for search. This result has highlighted the extreme importance of the initial step (a) above, that is, the choice of an 'appropriate' *CF* grammar. The pragmatic, problem-solving, requirements that are imposed on a description of the language of programs led us to formulate the following condition for such a choice:

> The grammar that describes the language of programs should be such that each structural description that is constructed in the grammar must have an interpretation in a mathematical system where there exists a structure of relationships that can be used in relating program structures to the functional conditions of the desired program.

This is in general a strong condition. Variations of this condition, or weaker versions of it, may also be appropriate for some formation problems. In our present case, it is possible to specify *CF* grammars that satisfy this condition. We shall describe these grammars next.

8. GRAMMARS OF PROGRAMS AND THEIR ASSOCIATED LANGUAGES

We classify languages of programs according to the number of distinct inputs in their member programs. We are limiting our considerations here to 1-input and 2-input programs. According to whether a 1-input program appears in a graphic or in a linear form it will belong to the languages $\mathscr{L}_{g,1}$ or $\mathscr{L}_{l,1}$ respectively. Similarly, 2-input programs belong to $\mathscr{L}_{g,2}$ if they appear in graphic form and to $\mathscr{L}_{l,2}$ if they appear in linear form.

8.1 The graphic language of 1-input programs, $\mathscr{L}_{g,1}$

The language $\mathscr{L}_{g,1}$ is the set of decoupled river-like graphs with a single source node (*see* section 6.3 above). This language can be specified in terms of the *CF* grammar $G_{g,1}$;

$$G_{g,1}=\langle T_{g,1}, N_{g,1}, \mathscr{X}_{g,1}, \flat\!\longrightarrow\!\!\dashv\rangle, \tag{1}$$

where $T_{g,1}$ is the terminal vocabulary of the language, $N_{g,1}$ is its non-terminal

vocabulary, $\mathscr{X}_{g,1}$ is the set of replacement rules, and the graph $\triangleright\!\longrightarrow\!\dashv$ stands for the starting non-terminal element.

The terminal elements of the language, that is, the members of the vocabulary $T_{g,1}$, are program statements. Thus,

$$T_{g,1}=L_g \tag{2}$$

In analogy with the notion of a string in ordinary language (that is, a concatenation of elements each of which can be taken from either the terminal or non-terminal vocabularies) we use in our program languages the notion of an *aggregate*. [The present notion of an aggregate is close to the notion of a configuration that we have used for describing solutions in derivation problems (Amarel 1970b).] An aggregate is a combination (an aggregation) of elements in the language, each of which can be taken from either the terminal or non-terminal vocabularies; it is composed of a finite number of elements (possibly a single element), or it may be empty. An aggregate obeys certain rules of aggregation that are defined by $\mathscr{X}_{g,1}$.

The non-terminal vocabulary of $G_{g,1}$ consists of the single element $\triangleright\!\longrightarrow\!\dashv$, which we call a (u, q)-statement.

Non-terminal elements in a program language play the usual syntactic role of representatives for statement types. In our present grammars, we classify statements by their input type and output type. Thus, a (u, q)-statement stands for any statement with an input of u-type and output of q-type. The notion of a non-terminal element here is close to that of a structural shell – an entity that specifies the surface structure of a part and thus determines gross necessary conditions for possible connectives or compositions between parts.

The rules of replacement in $G_{g,1}$, can be regarded as *rules of aggregation* for programs in $\mathscr{L}_{g,1}$. Each rule is assigned a name – a subscripted X', and it is expressed in table 1 as a transition between a non-terminal element and an aggregate (which can replace the non-terminal element in accordance with the rule). Program control lines are not shown explicitly except for the major lines of control in an *s–c* loop.

The application of a rule of replacement (aggregation) amounts to specifying a part of the structure of a graph that represents a program. The structural description of a graph representing a program consists of the record of rule applications that collectively specify the program graph. Thus, every rule application can be regarded as a *unit of structural description.* In table 1 we associate with each rule of replacement a representation of the unit of structural description that corresponds to the application of the rule. This representation is given in a special tree form, to the right of the transition that defines the rule. The upper white node in a tree corresponds to the non-terminal element that forms the left side of the transition; the dark node corresponds to the rule name; the branches that descend from a dark node are ordered; lower white nodes correspond to parts that still remain unspecified after applying the rule (non-terminal elements in the right side of

the transition). The left-to-right order of branches in the tree corresponds to a scan of the right side of the rule of aggregation in a general direction from left to right and from bottom to top. Each white node in a structural description tree is labelled by the type of non-terminal element that it represents. In table 1, the label '1' stands for (u, q)-st. (1-input aggregate).

Table 1. $\mathscr{X}_{g,1}$: Rules of replacement (aggregation) in $G_{g,1}$. [For simplicity, we combine several rules in one row, when no confusion is likely to arise.]

Rule Names	Transitions	Units of structural description
$X^1_\vee$ or $X^1_\wedge$:	→ π_2; ∪ or ∩; π_1	1; $X^1_\vee$ or $X^1_\wedge$; 1, 1; π_1, π_2
$X^1_{\vee'}$ or $X^1_{\wedge'}$:	→ π_1, π_2; s, c, ∪ or ∩	1; $X^1_{\vee'}$ or $X^1_{\wedge'}$; 1, 1; π_1, π_2
X^1_I or $X^1_{\hat{I}}$ or X^1_E:	→ η or $\hat{\eta}$ or ε	1; X^1_I or $X^1_{\hat{I}}$ or X^1_E

Note: The assignment of the names π_1, π_2 to some of the (u, q)-statements has been introduced in order to clarify the correspondences between the graph representations in the transitions and their associated structural descriptions. These notations are not necessary for describing the rules of replacement.

Given any aggregate ω that contains a non-terminal element, then the application of a replacement rule, $\rightarrow\psi$, (where ψ is some aggregate) at the non-terminal amounts to replacing it by ψ. The new aggregate that results from such a replacement is said to be *directly derivable* from ω in $G_{g,1}$. For any two distinct aggregates ρ and ψ, *ψ is derivable from ρ* in the given grammar if there exists a finite sequence of direct derivations (applications of replacement rules) that can take ρ into ψ. We denote this derivability relation by $\rho \Rightarrow \psi$.

We can now define the language of 1-input programs in terms of the grammar $G_{g,1}$. The language $\mathscr{L}_{g,1}$ is the set of terminal aggregates that are

derivable in $G_{g,1}$ from the starting (u, q)-statement, ⊢→⊣. We can also express this as follows:

$$\mathscr{L}_{g,1}=\{p|(p \text{ is a graph in } L_g)\&((⊢→⊣ \Rightarrow p) \text{ holds in } G_{g,1})\} \quad (3)$$

We define the *structural description* of a program p in the grammar $G_{g,2}$ as a tree which can be obtained from the derivation (construction) of p in $G_{g,1}$ and which shows the manner in which replacement rules in the grammar are combined to effect the transition from the starting element ⊢→⊣ to p. We denote the structural description of p by $\delta(p)$. The structural description tree contains only essential information about the construction of p in accordance with the given rules of aggregation; it is built by putting together in an appropriate manner the units of structural description that correspond to rule applications. The root node of a structural description tree is a white node that stands for the starting non-terminal element in the grammar, and all the tip nodes in the tree are dark nodes that correspond to applications of the rules X'_I, $X'_{\hat{I}}$ or X'_E.

It is clear that from a given structural description, and from the definitions of the rules of replacement and their corresponding units of structural description, we can always construct the corresponding program as a graph in L_g. Note that the graphs generated by $G_{g,1}$ are all decoupled river-like graphs. In general, to each structural description in $G_{g,1}$ there corresponds uniquely a constructible (and executable) 1-input program.

In order to clarify the notions of derivation and structural description, let us consider the generation of a simple program p_1. The derivation of p_1, in terms of transitions between aggregates, is shown in figure 8.

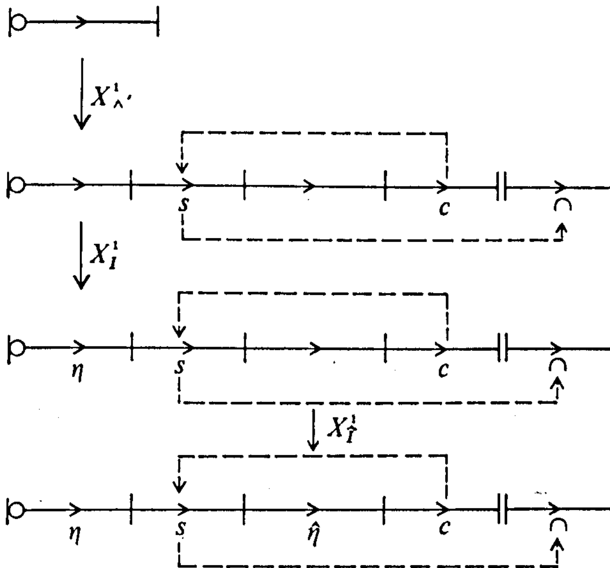


Figure 8

This derivation takes the starting statement of $G_{g,1}$ to the program p_1 in a sequence of three steps, each of which is an application of a replacement rule in $G_{g,1}$.

The structural description of p_1 that corresponds to the previous derivation is shown in figure 9.

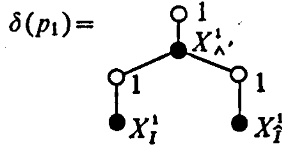


Figure 9

This graphic structural description of p_1 could be read as follows: p_1 consists of a block which is organized according to $X^1\wedge'$ as a cascade of five blocks, three of which are specified in the definition of $X^1\wedge'$; the first unspecified block in the definition performs a η operation, and the second performs a $\hat{\eta}$ operation. Note that $\delta(p_1)$ (together with the information in the table 1) provides a precise specification for building p_1.

It is possible to abbreviate the structural description of a program p, by retaining only the nodes that are labelled by rule names. Such an abbreviated description is a tree which is made out of the dark nodes in the tree $\delta(p)$; we call it an *abstract description*, and we denote it by $\delta_a(p)$. In the case of our previous example, the abstract description of p_1 is shown in figure 10.

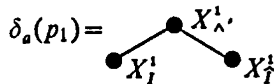


Figure 10

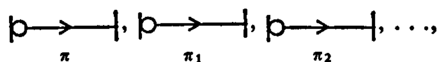
The abstract description of a program captures the essence of a plan for building the program.

8.2 Program schemes: the language $\mathscr{L}^{\pi}_{g,1}$

We now introduce into program languages the notion of *program variables*, that we denote by $\pi, \pi_1, \pi_2, \ldots$. This notion permits us to give a recognizable identity to a program or a program part (a block) that is not completely specified. We can then use the variable that stands for the unspecified program to express some of its properties. Program variables can be used to define the notion of a *program scheme* which is of considerable usefulness in the context of program formation problems. A program scheme is an incompletely specified program that has a given structure, parts of which are well defined while others are identified by program variables.

We define a set of program schemes which is naturally associated with our grammatical description of 1-input programs. This set is specified as a *CF* language which is an *extension of* $\mathscr{L}_{g,1}$; let us call it $\mathscr{L}^{\pi}_{g,1}$.

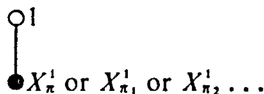
The grammar of $\mathscr{L}^{\pi}{}_{,1}$ is the same as that of $\mathscr{L}_{g,1}$ except for the addition of the $\pi^{(1)}$-statements



to the terminal vocabulary, and the introduction of the following set of replacement rules.

X'_{π} or X'_{π_1} or $X'_{\pi_2}\ldots$: ⊢→⊣ → ⊢→⊣ (π or π₁ or π₂ ...) .

The units of structural description that correspond to these rules are simply:



The language $\mathscr{L}^{\pi}_{g,1}$ is the set of terminal aggregates that are derivable from the starting (u, q)-statement in the extended grammar which we have just described. Clearly, any aggregate that can be formed in the original grammar $G_{g,1}$ at any intermediate stage of derivation, with variables assigned to the non-terminal elements, is a program scheme in $\mathscr{L}^{\pi}_{g,1}$.

The language of program schemes provides more expressive power than the language of programs. It includes the language of programs, and by the use of variables it permits the expression of relationships between two parts of a program without having to specify the actual structure of these parts.

The major usefulness of a variable is its *substitution property*. In our present case, the substitution set of a program variable is the language $\mathscr{L}^{\pi}{}_{,1}$. Thus, the values of program variables can be programs or program schemas which themselves contain other variables. In general, if a program scheme is to be substituted for a program variable, special care must be exercised in the naming of variables, so that circularity can be avoided.

By using the notions of program variables and program schemes we can now define a *multistage process* for deriving (or structuring) programs in $\mathscr{L}_{g,1}$. In such a process (1) a program scheme is first derived in the language of program schemes $\mathscr{L}^{\pi}_{g,1}$, (2) equalities may be established then between some of the variables in the program scheme (indicating that the values that are to be assigned to the variables are identical programs), (3) each independent program variable is then used as a new source for the derivation of a program or a program scheme to be assigned as a value to the variable, and (4) this process of derivation originating from new variables and followed by the (possible) specification of equalities between variables, and so on, is continued (a finite number of times) until all variables are explicitly specified in terms of programs.

Clearly this multistage process of derivation can be used to specify all the programs in $\mathscr{L}_{g,1}$, and to assign to them regular structural description trees. Moreover, it can be used for the specification (structural description) of the entire set of constructible programs that are representable by river-like

graphs – not only the subset of decoupled graphs. Even though this additional expressive power does not add to the functional variety of programs that can be made available to the formation system, it is nevertheless useful for processes of program simplification. The possibility of effecting program simplifications in the context of a multistage derivation comes from the fact that an examination of the record of the derivation process can reveal program blocks that are identical *and* that also process the same data; the elimination of such redundant blocks from a program p yields programs that are structurally simpler and functionally equivalent to p. We shall not further pursue processes of simplification here.

8.3 The linear language of 1-input programs $\mathscr{L}_{l,1}$

The language $\mathscr{L}_{l,1}$ consists of the same set of programs as $\mathscr{L}_{g,1}$, but the programs are expressed in a conventional linear notation. $\mathscr{L}_{l,1}$ can be specified in terms of the *CF* grammar $G_{l,1}$ which is strongly equivalent to $G_{g,1}$ (both grammars assign the same structural descriptions to programs).

$$G_{l,1} = \langle T_{l,1}, N_{l,1}, \mathscr{X}_{l,1}, (u,q)\text{-statement} \rangle \qquad (4)$$

The terminal vocabulary $T_{l,1}$ consists of the linear language of program statements L_l. The non-terminal vocabulary $N_{l,1}$ consists of the single (u, q)-statement in the form:

$$\alpha_t : u_i \rightarrow q_j : \alpha_v \qquad (5)$$

The rules of replacement $\mathscr{X}_{l,1}$ are essentially identical with the rules $\mathscr{X}_{g,1}$ (*see* table 1) and they are easily obtainable from them. As an example, consider the rule $X\wedge'$ in $\mathscr{X}_{l,1}$:

$$X\wedge' : \alpha_t : u_i \rightarrow q_j : \alpha_v \quad \rightarrow \quad \begin{array}{lcll} \alpha_t & : & u_i \rightarrow q_k & : \alpha_w \\ \alpha_w & : & sq_k \rightarrow u_l & : \alpha_{w+1}, \alpha_{w+3} \\ \alpha_{w+1} & : & u_l \rightarrow q_{k+1} & : \alpha_{w+2} \\ \alpha_{w+2} & : & cq_{k+1} \rightarrow g_m & : \alpha_w \\ \alpha_{w+3} & : & \cap g_m \rightarrow q_j & : \alpha_v. \end{array} \qquad (6)$$

The units of structural description that correspond to applications of replacement rules are identical in $G_{l,1}$ and $G_{g,1}$.

The language $\mathscr{L}_{l,1}$ is the set of terminal aggregates that are derivable in $G_{l,1}$ from the starting (u, q)-statement.

A linear language of program schemes, $\mathscr{L}_{l,1}$ is obtained by extending $\mathscr{L}_{l,1}$ in the same manner that $\mathscr{L}_{g,1}$ was extended to form $\mathscr{L}^{\pi}{}_{,1}$. A π-statement in the linear language has the form,

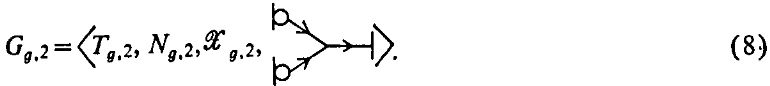
$$\alpha_l : \pi u_i \rightarrow q_j : \alpha_v. \qquad (7)$$

The main difference between the representation of a program (or a program scheme) in the graphic or the linear language lies in the degree of explicitness that is used in the specification of control paths and in the naming of data variables. Since in our program formation problem the focus of attention is on functional (input-output) characteristics of a program, it is sufficient and convenient to work with program representations in the graphic language.

Clearly, in order to execute a program in a conventional digital computer a representation in a linear language is needed.

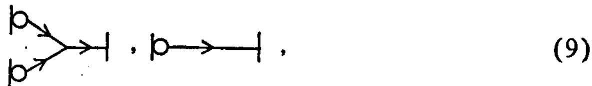
8.4 The languages of 2-input programs $\mathscr{L}_{g,2}$, $\mathscr{L}_{l,2}$ and their extensions

The graphic language $\mathscr{L}_{g,2}$ is the set of decoupled river-like graphs with two source nodes. This language can be specified in terms of the *CF* grammar $G_{g,2}$,



(8)

The terminal vocabulary $T_{g,2}$ is identical with $T_{g,1}$; it consists of the language L_g. The non-terminal vocabulary $N_{g,2}$ consists of the following two elements:



(9)

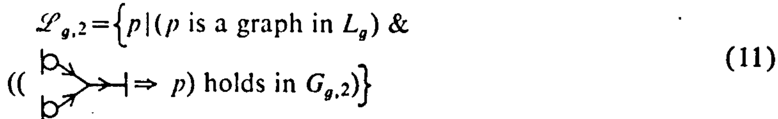
We call the first element a (u_1u_2, q)-statement; the second is a (u, q)-statement.

The set of replacement rules in $G_{g,2}$ is made of the set of rules $\mathscr{X}_{g,1}$ in the 1-input grammar $G_{g,1}$, and of an 'increment' set $\Delta\mathscr{X}_{g,2}$ which specifies the modes of composing 2-input aggregates from 1-input and 2-input aggregates. Thus,

$$\mathscr{X}_{g,2} = \mathscr{X}_{g,1} \cup \Delta\mathscr{X}_{g,2}. \qquad (10)$$

The rules $\mathscr{X}_{g,1}$ are given in table 1. The rules $\Delta\mathscr{X}_{g,2}$, with their corresponding units of structural description are given in table 2. In this table, the label '2' in a node of a structural description stands for the non-terminal (u_1u_2, q)-statement; and (as in table 1) the label '1' stands for the (u, q)-statement.

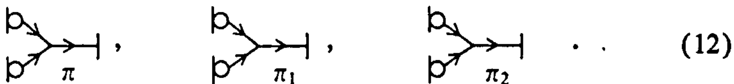
We can now define the language of 2-input programs in terms of derivations in the grammar $G_{g,2}$.



(11)

From our previous discussion of the relationship between the linear and graphic languages, it is easy to obtain a definition of a linear language of 2-input programs $\mathscr{L}_{l,2}$ in terms of the definition of $\mathscr{L}_{g,2}$. We shall not here further discuss the language $\mathscr{L}_{l,2}$.

The language $\mathscr{L}_{g,2}$ of 2-input programs can be extended to a language of program schemes in the manner discussed previously for 1-input programs. Here we have to add to the terminal vocabulary of $\mathscr{L}_{g,2}$ two new sets of elements: (1) the $\pi^{(1)}$-statements already introduced in $\mathscr{L}^{\pi}_{g,1}$, and (2) a new set of statements, the $\pi^{(2)}$-statements, in the form,



(12)

Table 2. $\Delta\mathscr{X}_{g,2}$: Increment of rule of replacement (aggregation) for $G_{g,2}$

Rule Names	Transitions	Units of structural description
$X^2_\vee$ or $X^2_\wedge$:	π_2 ∪ or ∩ π_1	2; $X^2_\vee$ or $X^2_\wedge$; 2 π_1; 2 π_2
$X^2_{\vee,(1)}$ or $X^2_{\wedge,(1)}$:	π_2 ∪ or ∩ π_1	2; $X^2_{\vee,(1)}$ or $X^2_{\wedge,(1)}$; 1 π_1; 2 π_2
$X^2_{\vee,(2)}$ or $X^2_{\wedge,(2)}$:	π_2 ∪ or ∩ π_1	2; $X^2_{\vee,(2)}$ or $X^2_{\wedge,(2)}$; 2 π_1; 1 π_2
$X^2_{\vee'}$ or $X^2_{\wedge'}$:	π_1 π_2 s c ∪ or ∩	2; $X^2_{\vee'}$ or $X^2_{\wedge'}$; 2 π_1; 1 π_2
$X^2_{\vee''}$ or $X^2_{\wedge''}$:	π_1 ∪ or ∩ π_2	2; $X^2_{\vee''}$ or $X^2_{\wedge''}$; 1 π_1; 1 π_2

Note: In this table we combine again several rules in one row when no confusion is likely to arise. Also, the assignment of the names π_1, π_2 to some of the non-terminal elements is introduced to establish correspondences between graph elements in the transitions and their associated structural descriptions; these names are not necessary for describing the rules of replacement.

In addition, the following set of replacement rules must be included in the grammar of $\mathscr{L}^{\pi}_{g,2}$:

$$X^2_\pi \text{ or } X^2_{\pi_1} \text{ or } X^2_{\pi_2} \ldots : \quad \rightarrow \quad \pi \text{ or } \pi_1 \text{ or } \pi_2 . \qquad (13)$$

The units of structural description that correspond to these rules are,

$$\begin{array}{l} \circ 2 \\ | \\ \bullet X^2_\pi \text{ or } X^2_{\pi_1} \text{ or } X^2_{\pi_2} \ \ldots \end{array} \qquad (14)$$

The language of 2-input program schemes includes the language of 2-input programs, and it constitutes the substitution set from which values for 2-input program variables can be taken during multistage processes for deriving (structuring) programs.

9. THE MECHANISM OF PROGRAM GENERATION

In our approach to program formation, the solution candidates are generated in the form of structural descriptions in a given grammar. From the vantage point of the formation process, a candidate program is completely specified by the pattern of choices of aggregation rules that collectively determine the program's structure. After the generation of a structural description, the formation system translates it into an executable program (possibly simplifies it), and then it tests the program by running it over appropriate data in order to obtain an estimate of the extent to which it satisfies the given input-output correspondences. The test provides an evaluation of the program; viewed differently, it provides an evaluation of the set of decisions that co-determine the structural description of the program. This evaluation is then used in subsequent decisions of the formation process.

Consider, for example, a formation problem whose objective is to find a program that expresses the computation of the *Infimum function* (the *glb*) for elements of $\mathfrak{S}^2$. This is a non-trivial problem that we have used for the study of various program formation systems; it was previously discussed in Amarel (1962b). The solution to this problem would be produced by the formation system in the form of a structural description, δ (p_{Infimum}) which we show in figure 11. The program p_{Infimum} which is specified by the structural description in figure 11 can then be represented as a graph in the language $\mathscr{L}_{g,2}$, or as a sequence of instructions in the language $\mathscr{L}_{I,2}$; we show the graph representation of the program in figure 12.

An examination of the structural description of p_{Infimum} or of its corresponding graph representation, shows that it has a redundant frontal block – shown as p_F in figure 12. Clearly a simplification can be obtained by eliminating the redundant block, and by reorganizing the data paths in figure 12 in such a manner that the two data nodes q_1 and q_2 are fused into one.

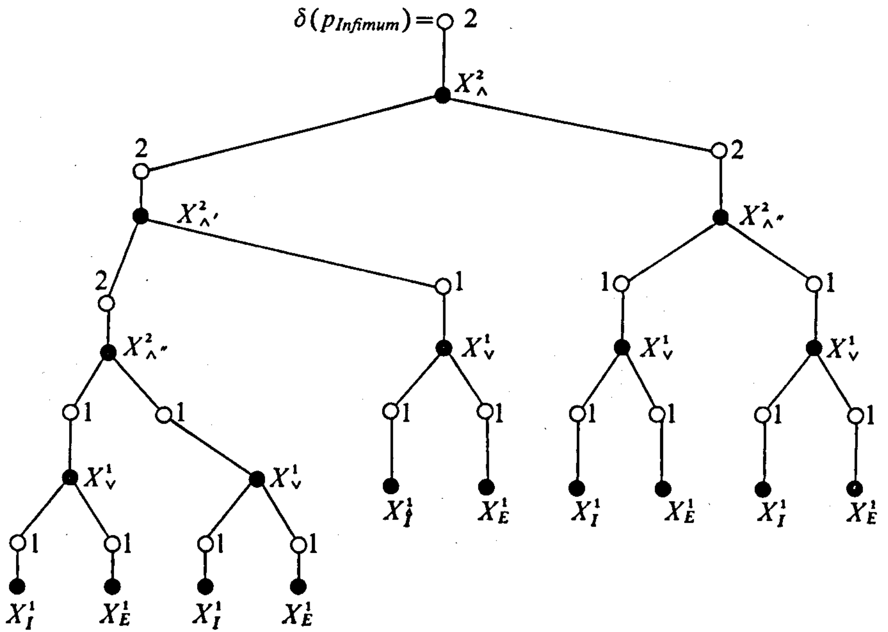


Figure 11. Structural description of the Infimum program

Note that the aggregate obtained by effecting the suggested reorganization of the structure in figure 12 is no longer well formed in the language $\mathscr{L}_{g,2}$. It is still a river-like graph, thus a constructible program, but not a decoupled graph; it is, however, functionally equivalent to the initial program. As far as execution efficiency is concerned, the reorganized program is superior to the initial program that is obtained directly from $\delta(p_{\text{Infimum}})$; the former contains 14 program statements, while the latter contains 21.

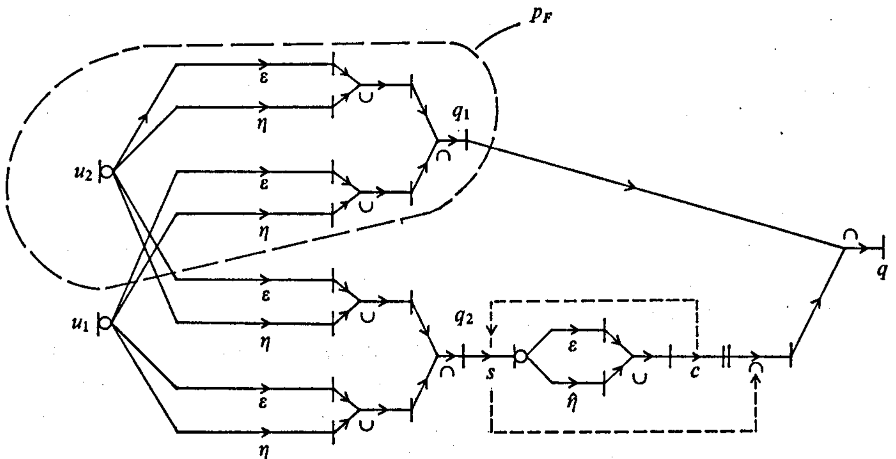


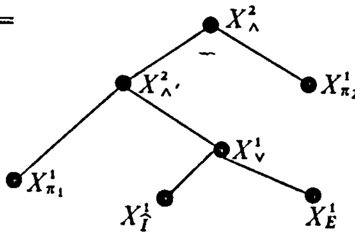
Figure 12. Representation of the Infimum program as a graph in the language $\mathscr{L}_{g,2}$

The importance of program simplification derives from the fact that the testing of candidate programs (running them over a set of data) takes a major part of the effort during a formation process. The existence of good simplification procedures may well determine whether automatic program formation is practicable or not.

In general, there is a need for several representations of the 'same' program in a formation system, each oriented to different uses. A program as a symbolic entity to be manipulated under the direction of a formation strategy has to satisfy different pragmatic conditions from those that have to be satisfied by a program as a computational prescription which has to be efficiently executed.

From the point of view of formation strategy, the relevant representation is the structural description of the program. What really matters is the pattern of aggregation decisions that are made in generating (structuring) the program. This pattern is well captured by the abstract description of the program. A two-stage representation of the abstract description of the program p_{Infimum} is shown in figure 13. In this representation, the sub-program p_F (which is enclosed in dashed lines in figure 12) is factored out and shown separately.

$\delta_a(p_{Infimum}) =$



where $\pi_1 = \pi_2 = p_F$, and

$\delta_a(p_F) =$

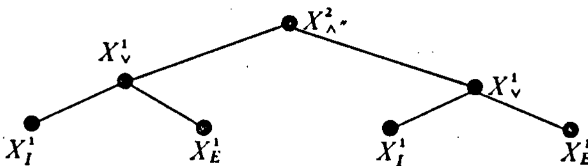


Figure 13. Abstract description of the Infimum program

As indicated previously, the white nodes in a structural description correspond to decision points in the program generation process. Thus, the number of white nodes in the structural description of a program (equivalently, the total number of nodes in the abstract description of the program) is a significant *measure of complexity* of the program from the viewpoint

of the program formation system; we call it the *formation weight* of the program. In our example of the Infimum program, the formation weight is 19. This number is independent of the possible simplifications that may be made on the program structure; it is not related in a simple manner to complexity measures that are based on 'surface structure', such as the number of program statements.

Given the grammar $G_{g,2}$ (or $G_{l,2}$), there are 19 decision points in the specification of a program for computing the Infimum function. Consider the topmost decision point in the structural description shown in figure 11. There are 10 possible rules of aggregation that apply at this point, namely all the rules in table 2. The situation is similar for the other 3 decision points where a rule of aggregation is to be chosen at a (u_1u_2, q)-statement. At each of the 15 remaining decision points there are 7 possible rules of aggregation (in accordance with table 1). If we use a simple exhaustive search strategy to look for the solution we must try roughly 10^{10} programs. Even if the formation system can try 10^4 programs per second (a very optimistic assumption) then we would need a run of over 10 days to solve a single formation problem!

One of the major objectives of our present study is to find methods for *controlling* the process of program generation in such a manner that the amount of search that is needed to reach a solution would be drastically reduced from that of an exhaustive method. A significant step towards effective control of program generation is to formulate a model of program space which can help in planning an intelligent search for a solution-program. We shall next present such a model for our program space. The model is in the form of a mathematical system which is a *modified algebra of relations.* We first discuss a system, $\mathscr{R}(\mathfrak{S})$, which models 1-input programs, and then an extended system, $\mathscr{R}(\mathfrak{S}^2)$ which provides a model for 2-input programs.

10. AN ALGEBRAIC MODEL, $\mathscr{R}(\mathfrak{S})$ FOR 1-INPUT PROGRAMS

10.1. 1-input relations and their matrices; correspondences with program statements

Given a finite set $\mathfrak{S}$ with n elements, a *dyadic relation* R on $\mathfrak{S}$ is a rule that specifies for each ordered pair (x, y) of elements in $\mathfrak{S}$ whether R holds between x and y or not. We call such a relation a 1-*input relation.* It is well known that R can be represented by a $n \times n$ *relation matrix* of 0s and 1s (*see* Birkhoff 1948, Copilowish 1948). We denote the matrix corresponding to R by $\underline{R}$.

The components of the relation matrix are elements of a 2-element Boolean algebra for which the operations $\vee$, $\wedge$, $\sim$, and their associated properties are defined as usual.

To establish the correspondence between R and its matrix representation, we start by associating with each element of $\mathfrak{S}$ a numerical index $i(i \in J_1^n$, where J_1^n is the set of integers from 1 to n). The i-row and the i-column of the

relation matrix are then made to correspond with the i-element of $\mathfrak{S}$. Specifically, the values of the components r_{ij} of $\underline{R}$ are defined as follows:

$$r_{ij}=\begin{cases}1, \text{ if } R \text{ holds between the } i\text{-element and the } j\text{-element of } \mathfrak{S},\\ 0, \text{ otherwise.}\end{cases}$$

We denote by $\tau[\text{row}_i(\underline{R})]$ the set of coordinates in the row vector of $\underline{R}$ where the value is 1.

Consider as an example, the proper inclusion relation I on the partly ordered set $\mathfrak{S}$ of 4 elements shown in figure 14,

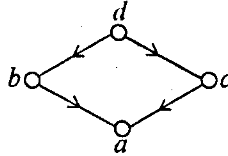


Figure 14

where $\overset{x}{\circ}\!\!\rightarrow\!\!\overset{y}{\circ}$ denotes 'I holds between x and y'. The relation I can be represented by the following matrix:

$$\underline{I}=\begin{pmatrix}0 & 0 & 0 & 0\\ 1 & 0 & 0 & 0\\ 1 & 0 & 0 & 0\\ 1 & 1 & 1 & 0\end{pmatrix} \tag{15}$$

In the specification of this matrix, the index of the element a is 1, of b is 2, of c is 3, and of d is 4. The following is one of the properties of $\underline{I}: \tau[\text{row}_4(\underline{I})] = \{1, 2, 3\}$.

In our specific universe of discourse we have in addition to the proper inclusion relation I, the converse relation to I, that is, $\hat{I}$, and also the identity relation E. In the matrix representation, the matrix $\underline{\hat{I}}$ (that corresponds to $\hat{I}$) is the transpose of $\underline{I}$ (that is, it is obtained by interchanging rows and columns), and the matrix $\underline{E}$ is the identity matrix with 1s in the main diagonal and 0s elsewhere.

Let an *n-dimensional row vector* with a 1 at its i-component and 0 elsewhere represent the element of $\mathfrak{S}$ which is indexed by i. Similarly, let a subset of elements of $\mathfrak{S}$ (an element of $2^{\mathfrak{S}}$) be represented by an n-dimensional row vector with a 1 at each component that corresponds to an element of the subset, and 0 elsewhere. An n-dimensional row vector whose components are all 0 corresponds to the empty set ϕ. Here again, we regard the vector components as elements of a two-element Boolean algebra.

As an example, with reference to the 4-element set given in figure 14, the element d is represented by the vector (0 0 0 1), and the subset $\{a, b, c\}$ by the vector (1 1 1 0).

If u is a variable that takes as values ϕ or elements in $\mathfrak{S}$, then let $\underline{u}$ denote the row vector representation of u. Similarly, if q is a variable with values in

$2^{\mathfrak{S}}$, then let $\underset{\sim}{q}$ denote the vector representation of q. We use the notation $\text{comp}_i(\underset{\sim}{u})$ for the i-component of the Boolean vector $\underset{\sim}{u}$. The expression,

$$\underset{\sim}{q}=\underset{\sim}{u}\underset{\sim}{R}, \tag{16}$$

denotes a matrix multiplication of $\underset{\sim}{u}$ and $\underset{\sim}{R}$ that yields the row vector $\underset{\sim}{q}$ under the rule,

$$\underset{\sim}{q}=\begin{cases}\text{row}_i(\underset{\sim}{R}), \text{ if } \text{comp}_i(\underset{\sim}{u})=1 \text{ and } \text{comp}_j(\underset{\sim}{u})=0, \text{ for all } j\neq i,\\ \text{row}(\underset{\sim}{0}), \text{ if all the components of } \underset{\sim}{u} \text{ are } 0.\end{cases} \tag{17}$$

In eq. (17) $\underset{\sim}{0}$ denotes the $n\times n$ matrix of 0s which corresponds to the *null* relation.

Consider now the following program statement in L_g. [For our present purposes, we assign names to the data nodes of program graphs (here u, q)]:

$$u \vdash\!\!\xrightarrow{\;\eta\;}\!\!\dashv q. \tag{18}$$

If $\underset{\sim}{u}$ is the vector representation of the contents of u when program control points to the given statement, then upon execution of the statement, the vector representation $\underset{\sim}{q}$ of the contents of q is given by the expression

$$\underset{\sim}{q}=\underset{\sim}{u}\underset{\sim}{I}. \tag{19}$$

The correspondence of program (18) and eq. (19) is to be taken in the sense of *functional equivalence*; that is, for all the possible contents of u, the machine execution of program (18) will assign to q a content whose vector representation can be obtained from the evaluation of expression (19).

We have similar interpretations for $\hat{\eta}$ and ε statements in terms of applications of the matrices $\underset{\sim}{\hat{I}}$ and $\underset{\sim}{E}$ respectively.

We shall consider next four operations between relations: two Boolean operations $\vee$, $\wedge$ and two operations $\vee'$, $\wedge'$ that we call *cascade products.* These operations are defined as follows:

For any relations A, B, C, that are representable by $n\times n$ matrices, $\underset{\sim}{A}$, $\underset{\sim}{B}$ and $\underset{\sim}{C}$ respectively, and for all $i\in J_1^n$,

(a) $(C=A\vee B)\equiv(\text{row}_i(\underset{\sim}{C})=\text{row}_i(\underset{\sim}{A})\vee\text{row}_i(\underset{\sim}{B}))$

(b) $(C=A\wedge B)\equiv(\text{row}_i(\underset{\sim}{C})=\text{row}_i(A)\wedge\text{row}_i(\underset{\sim}{B}))$

(c) $(C=A\vee' B)\equiv(\text{row}_i(\underset{\sim}{C})=\bigvee_{k\in\tau[\text{row}_i(A)]}\text{row}_k(\underset{\sim}{B}))$

(d) $(C=A\wedge' B)\equiv(\text{row}_i(\underset{\sim}{C})=\bigwedge_{k\in\tau[\text{row}_i(A)]}\text{row}_k(\underset{\sim}{B}))$ (20)

In eq. (20) (c) and (d), the symbols $\bigvee$, $\bigwedge$ are used to denote iterated disjunction and conjunction respectively, with an index of iteration k which is defined over a non-fixed set $\tau[\ldots]$. The use of this notation assumes the following convention: if $\tau[\ldots]$ has two or more elements, then the iterated operation is interpreted in the ordinary way; if $\tau[\ldots]$ has a single element, say j, then we have in the above expressions $\text{row}_i(\underset{\sim}{C})=\text{row}_j(\underset{\sim}{B})$; if the set $\tau[\ldots]$ is empty, then $\text{row}_i(\underset{\sim}{C})$ is the null vector.

The operations $\vee$, $\wedge$ in eq. (20) are ordinary Boolean operations between relations, and the cascade product $\vee'$ is the ordinary product of relations. With the operation $\wedge'$ we are introducing an unconventional product of

relations that will be mainly responsible for the deviation between our mathematical models and the ordinary algebra of relations. We find the introduction of the new cascade product necessary for a complete modelling of our programming languages by algebraic systems.

10.2 The language of 1-input relational expressions $\mathscr{R}_{R,1}$

The language $\mathscr{L}_{R,1}$ can be defined in terms of a *CF* grammar $G_{R,1}$ whose elements are as follows:

(a) a terminal vocabulary consisting of the constants $I, \hat{I}, E, \vee, \wedge, \vee', \wedge'$, variables $R, A, B, C, \ldots$, and parentheses.

(b) a single non-terminal element ρ_1, which stands for the notion of a well-formed 1-input relational expression; it is also the starting element in the grammar.

(c) a set of rules of replacement as follows:

$$\begin{aligned} \alpha_1, \alpha_2 &: \rho_1 \rightarrow (\rho_1 \vee \rho_1) \text{ or } (\rho_1 \wedge \rho_1) \\ \beta_1, \beta_2 &: \quad \rightarrow (\rho_1 \vee' \rho_1) \text{ or } (\rho_1 \wedge' \rho_1) \\ \gamma_1, \gamma_2, \gamma_3 &: \quad \rightarrow I \text{ or } \hat{I} \text{ or } E \\ \delta_i &: \quad \rightarrow R \text{ or } A \text{ or } B \text{ or } \ldots \end{aligned} \tag{21}$$

The language of 1-input relational expressions is the set of terminal strings that are derivable in the grammar $G_{R,1}$ from ρ_1.

10.3 Modelling programs by 1-input relational expressions

The notion of functional equivalence was used previously [in (18) and (19)] to establish correspondences between specific program statements and relations. Let us now extend this approach to 1-input programs in general.

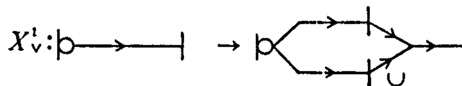
A $\pi^{(1)}$-statement corresponds to the expression,

$$\underline{q} = \underline{u}\underline{R}, \tag{22}$$

where R is a relation variable that corresponds to the program variable π in the $\pi^{(1)}$-statement. If the variable π is assigned a value η, $\hat{\eta}$, or ε, then this assignment corresponds to a valuation of R by I, $\hat{I}$ or E respectively.

A (u, q)-statement in the language of 1-input programs is a syntactic notion that corresponds to ρ_1 in the language of 1-input relational expressions.

Consider now the rule of aggregation X^1 in $G_{g,1}$:



To this rule there corresponds the rule $\rho_1 \rightarrow (\rho_1 \vee \rho_1)$ in the grammar of 1-input relational expressions. The validity of this correspondence can be verified by showing that for any two component programs π_1, π_2 that are assigned to the bottom and top (u, q)-statements respectively, in the right aggregate of the $X^1_\vee$ rule (call it π), and for any two relations A, B that

correspond to these component programs, the program π is functionally equivalent to $A \vee B$.

By similar reasoning it can be seen that: (a) the rule of aggregation $X^1_\wedge$ corresponds to the rule $\rho_1 \rightarrow (\rho_1 \wedge \rho_1)$; (b) the rules $X^1_{\vee'}$ and $X^1_{\wedge'}$ correspond respectively to the rules $\rho_1 \rightarrow (\rho_1 \vee' \rho_1)$ and $\rho_1 \rightarrow (\rho_1 \wedge' \rho_1)$; (c) the rules $X^1_I, X^1_{\hat{I}}, X^1_E$ correspond respectively to $\rho \rightarrow I$ or $\hat{I}$ or E; and (d) the rules X^1_π for assigning program variables to (u, q)-statements correspond to rules $\rho \rightarrow R$ for assigning relation variables.

The correspondences between the rules of replacement in the language of programs and in the language of relational expressions provide a simple *translation basis* from programs to expressions. The translation defines a *mapping* μ from the language of 1-input programs to the language of 1-input relational expressions.

Consider a program p whose abstract description is $\delta_a(p)$ and let $P = \mu(p)$ denote its corresponding relational expression. P can be obtained from the tree $\delta_a(p)$ by (a) replacing each label of type X^1_π in the tree by a relation variable; (b) replacing each label $X^\prime_\alpha$ where α is $\vee$, $\wedge$, $\vee'$, $\wedge'$, $I, \hat{I}, E$ by α itself; and (c) expressing the re-labelled tree in an infix linear form. This translation can be achieved by a fairly simple computer procedure.

As an example of the translation, consider the program p_1 whose derivation is shown in figure 8 and whose abstract description is given in figure 10. The relational expression that corresponds to p_1 is easily obtained as $p_1 = (I \wedge' \hat{I})$; it can be verified that this expression is functionally equivalent to p_1.

Let us summarize the correspondence between programs and relational expressions in the following theorem.

Theorem 1

(Projecting 1-input programs into the algebraic model)

There exists a mapping μ that takes any 1-input constructible program into a functionally equivalent 1-input relational expression; furthermore, there exists a simple translation procedure for computing μ.

The proof is based on the validity of the correspondences that provide the translation basis, and on the matrix interpretation of relational expressions.

From our previous discussion it can be seen that the mapping μ is one–one. The translation basis between programs and relational expressions is *bilateral*, and it can be used both for projecting programs into relational expressions, and inversely for projecting relational expressions into programs. Thus, it will be possible to manipulate within an algebraic system relational expressions that correspond to programs, and then to interpret the results of these manipulations in terms of actions on executable programs.

10.4 The algebra of 1-input relational expressions $\mathscr{R}(\mathfrak{S})$

After having established a clear correspondence between 1-input programs and relational expressions, we can now proceed to the development of an

algebra $\mathscr{R}(\mathfrak{S})$ which will provide a mathematical model for the set of 1-input programs that are constructible in our system.

The system $\mathscr{R}(\mathfrak{S})$ is a *modified algebra of dyadic relations.* The atomic elements (and also the constants) of $\mathscr{R}(\mathfrak{S})$ are the relations I, $\hat{I}$, E, and O, for which we have specific $n \times n$ matrix interpretations. The operations of $\mathscr{R}(\mathfrak{S})$ are $\vee$, $\wedge$, $\vee'$, $\wedge'$; these operations were interpreted previously in terms of relation matrices.

The terms of $\mathscr{R}(\mathfrak{S})$ are the relational expressions that are contained in the language $\mathscr{L}_{R,1}$ (except that in the algebraic system we shall use as an additional terminal element the null relation O).

In view of the previously given definitions for the atomic elements and the operations of $\mathscr{R}(\mathfrak{S})$, it should be clear that every term in $\mathscr{R}(\mathfrak{S})$, has a corresponding $n \times n$ relation matrix.

The terms of the algebra $\mathscr{R}(\mathfrak{S})$ are partially ordered under an *implication* relation that we denote by $\rightarrow$. [In the remainder of this paper, the arrow will be used only for implication, so that confusion is not likely to arise with the other interpretations of the arrow that we have used previously.] For any two terms A, B, the implication $A \rightarrow B$ holds (A implies B) when the relation matrix corresponding to A *is included in* the relation matrix corresponding to B. We denote this inclusion by $A \leqslant B$. More explicitly,

$$(A \rightarrow B) \equiv (\underline{A} \leqslant \underline{B}) \equiv (\tau[\text{row}_i(\underline{A})] \subseteq \tau[\text{row}_i(\underline{B})], \text{ for all } i \in J_1^n). \quad (23)$$

The logical interpretation of $(A \rightarrow B)$ is that for every pair of elements in $\mathfrak{S}$ (denoted by i, j), if the relation A holds between i and j, then the relation B also holds between these two elements.

We shall consider next properties of $\mathscr{R}(\mathfrak{S})$ that are obtainable as consequences of the previously given matrix interpretations of its elements.

The operations $\vee$, $\wedge$ have the usual associativity and distributivity properties that are known from Boolean algebra. Let us concentrate then on properties of the products $\vee'$, $\wedge'$ and the interactions of these products with the Boolean operations. We present first general properties (not especially tied to the constants I, $\hat{I}$) and then more specific properties that depend on the atomic elements.

Theorem 2

(General properties of cascade products)

(1) For any two terms (relational expressions) A, B,

$$(A \wedge' B) \rightarrow (A \vee' B).$$

(2) For any term A,

$$(O \vee' A) = (A \vee' O) = (O \wedge' A) = (A \wedge' O) = O$$

(3) For any term A,

$$(E \vee' A) = (A \vee' E) = (E \wedge' A) = A.$$

(4) For any term A,

$$(A \wedge' E) = A;$$

if the matrix $\underline{A}$ corresponding to A has no more than a single 1 in each row, then

$$(A \wedge' E) = A.$$

(5) For any three terms A, B, C, if $A \rightarrow B$ then the following implications hold:

(a) $(C \vee' A) \rightarrow (C \vee' B)$

(b) $(A \vee' C) \rightarrow (B \vee' C)$

(c) $(C \wedge' A) \rightarrow (C \wedge' B)$

(d) $(A \wedge' C) \leftarrow (B \wedge' C)$.

Proof. These properties can be easily derived from the definition of the cascade products and of the constants E, O. We prove here the property (5d) in order to illustrate the reasoning involved.

The supposition $A \rightarrow B$ is equivalent to the following:

$$\tau[\text{row}_i(\underline{A})] \subseteq \tau[\text{row}_i(\underline{B})] \text{ for all } i \in J_1^n; \tag{24}$$

In the cases where the two index sets τ are identical, then the relationship $\text{row}_i(\underline{A} \wedge' \underline{C}) = \text{row}_i(\underline{B} \wedge' \underline{C})$ holds. If for some i the set inclusion in eq. (24) is proper, then there exist indices, say $k_1, \ldots, k_m$, that are in the set $\tau[\text{row}_i(\underline{B})]$ but not in $\tau[\text{row}_i(\underline{A})]$. Thus, in view of the definition of $\wedge'$ [*see* eq. 20(d)] we can write,

$$\text{row}_i(\underline{B} \wedge' \underline{C}) = \text{row}_i(\underline{A} \wedge' \underline{C}) \wedge (\text{row}_{k_1}(\underline{C}) \wedge \ldots \wedge \text{row}_{k_m}(\underline{C})), \tag{25}$$

from which we can obtain,

$$\text{row}_i(\underline{B} \wedge' \underline{C}) \rightarrow \text{row}_i(\underline{A} \wedge' \underline{C}). \tag{26}$$

Hence the property (5d) of the theorem is proved. This property is interesting in that it provides a means for inverting the implication relationship between two relational expressions A, B by forming the cascades $(A \wedge' C)$, $(B \wedge' C)$ for some C.

Theorem 3

(Associative properties of cascade products)

In the following pattern of relationships [each relationship is labelled by a number from (1) to (10)] full arrows denote implications that hold for any three terms A, B, C in $\mathscr{R}(\mathfrak{S})$ and dotted arrows denote implications that hold under certain conditions imposed on the terms A, B, C. This is made clear in figure 15 opposite.

Conditions on (9), (10): The implication (9) holds if the relation matrix corresponding to B does not have any null row. The implication (10) holds if the relation matrix corresponding to B has a column of 1s.

Proof. The equality (1) (which is equivalent to a two-directional implication) expresses the well-known associativity of ordinary relation products. The validity of the implications (2), (3), and (4) is obtained directly from the property (1) of theorem 2. The implications (5) and (6) can be obtained in two steps from the properties (1) and (5) of theorem 2. The properties (7), (8), (9), and (10) require a lengthier derivation. We prove here the relationships (8) and (9) in order to illustrate the method of reasoning.

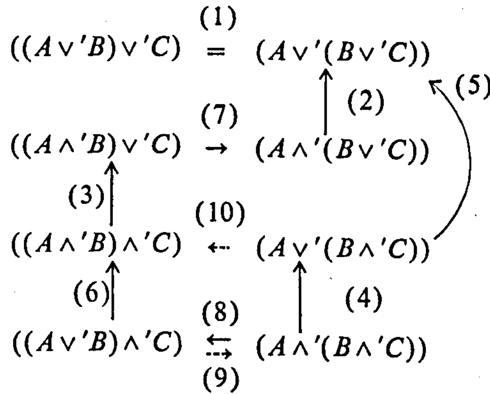


Figure 15

Let

$$(A \wedge'(B \wedge' C)) \stackrel{\text{def}}{=} X; \quad ((A \vee' B) \wedge' C) \stackrel{\text{def}}{=} Y \tag{27}$$

In view of the definitions of $\vee'$ and $\wedge'$ in eqs. (20c) and (20d) respectively, we have for any $i \in J_1^n$,

$$\text{row}_i(\underline{X}) = \bigwedge_{k \in \tau[\text{row}_i(\underline{A})]} \; \bigwedge_{j \in \tau[\text{row}_k(\underline{B})]} \text{row}_j(\underline{C}) \tag{28}$$

$$\text{row}_i(\underline{Y}) = \bigwedge_{j \in \tau\left[\bigvee_{k \in \tau[\text{row}_i(\underline{A})]} \text{row}_k(\underline{B})\right]} \text{row}_j(\underline{C}) \tag{29}$$

If $\tau[\text{row}_i(\underline{A})]$ is empty, or if it contains a single element, then $\text{row}_i(\underline{X}) = \text{row}_i(\underline{Y})$. Suppose now that $\tau[\text{row}_i(\underline{A})]$ contains two or more elements $k_1, \ldots, k_m$. These elements act as selectors for rows of $\underline{B}$, namely, the rows $\text{row}_{k_1}(\underline{B}), \ldots, \text{row}_{k_m}(\underline{B})$. In turn, these rows determine the sets whose elements act as selectors for rows of $\underline{C}$. The selectors of rows of $\underline{C}$ in eq. (28) are the sets $\tau[\text{row}_{k_\nu}(\underline{B})]$, $\nu = 1, \ldots, m$; for simplicity let us denote these sets by $\tau(k_\nu)$, $\nu = 1, \ldots, m$. The selectors of rows of $\underline{C}$ in eq. (29) are members of the set $\tau[\text{row}_{k_1}(\underline{B}) \vee \ldots \vee \text{row}_{k_m}(\underline{B})]$ which we abbreviate by $\tau(k_1 \vee \ldots \vee k_m)$.

Note that

$$\tau(k_\nu) \subset \tau(k_1 \vee \ldots \vee k_m), \text{ for } \nu = 1, \ldots, m. \tag{30}$$

Suppose that for all ν, $\tau(k_\nu) \neq \phi$. Then we can write,

$$\text{row}_i(\underline{X}) = \bigwedge_{j \in (\tau(k_1) \cup \ldots \cup \tau(k_m))} \text{row}_j(\underline{C}),$$

$$\text{row}_i(\underline{Y}) = \bigwedge_{j \in \tau(k_1 \vee \ldots \vee k_m)} \text{row}_j(\underline{C}) \tag{31}$$

Since $\tau(k_1 \vee \ldots \vee k_m) = \tau(k_1) \cup \ldots \cup \tau(k_m)$, then in this case we obtain $\text{row}_i(\underline{X}) = \text{row}_i(\underline{Y})$.

Suppose now that for some ν, $\tau(k_\nu) = \phi$; then $\text{row}_i(\underline{X})$ is null, but $\text{row}_i(\underline{Y})$ is not necessarily null. On the other hand, if $\tau(k_1 \vee \ldots \vee k_m)$ is null, then both $\text{row}_i(\underline{Y})$ and $\text{row}_i(X)$ are null. Hence, in all situations where there is *no k*

such that $\tau[\text{row}_k(\underline{B})]=\phi$, we can assert that $\text{row}_i(\underline{X})\rightleftarrows\text{row}_i(\underline{Y})$; otherwise $\text{row}_i(\underline{X})\rightarrow\text{row}_i(\underline{Y})$. This proves the relationships (8) and (9).

A simple example where the implications (8) and (9) both hold is when the value of B is the identity relation E (where each row has one 1). This can be directly verified via the property (3) of theorem 2. Thus

$$(A\wedge'(E\wedge'C))=(A\wedge'C);\ ((A\vee'E)\wedge'C)=(A\wedge'C).$$

A simple example of a case where the property (10) holds is when $B=(I\vee E)$. In this case, the $\underline{B}$ matrix has one column of 1s.

It is evident from theorem 3 that the introduction of the product $\wedge'$ destroys the semigroup property of the algebra of relations, which relies on the associativity of the ordinary product of relations $\vee'$. Our system is therefore more complex than the ordinary algebra of relations, and it lacks the tight structure of that algebra. However, the system $\mathscr{R}(\mathfrak{S})$ has a fair amount of interesting implicational structure, as evidenced by the previous theorem as well as the next one.

Theorem 4

(Distributive properties of cascade products with Boolean operations)

We present separately two patterns of distributivity relationships; one where the cascade products are distributed from the right and another where they are distributed from the left.

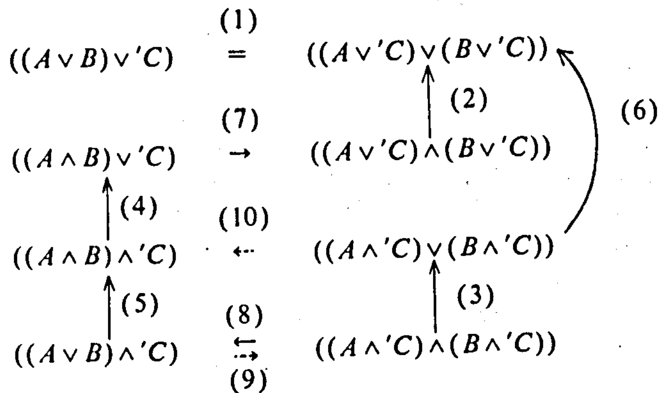


Figure 16

(R) *Right distributivities.* In the pattern shown in figure 16 [where relationships are labelled by numbers from (1) to (10)] we are using the same notational conventions as in theorem 3.

Conditions on (9), (10). The implications (9), (10) hold if there is no i, $i\in J_1^n$, such that one of the rows, $\text{row}_i(\underline{A})$, $\text{row}_i(\underline{B})$, is null (but not both).

(L) *Left distributivities.* For any three terms A, B, C in $\mathscr{R}(\mathfrak{S})$ the pattern of relationships shown in figure 17 [labelled by numbers from (1) to (8)] holds:

Proof. The equalities (R1) and (L1) express the well-known distributivities of Boolean operations and ordinary matrix products.

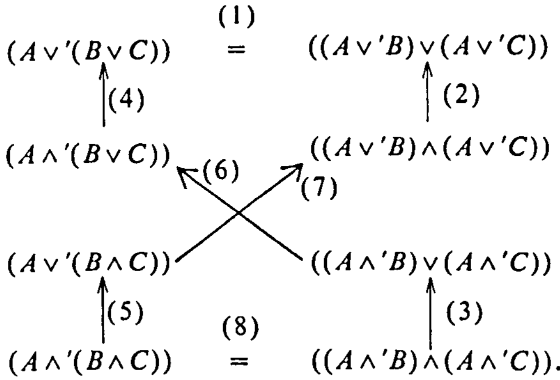


Figure 17

The validity of the implications (R2), (R3), (L2), and (L3) is obtained directly from the properties of Boolean operations. The validity of (R4), (R5), (L4), and (L5) is obtained from properties given in theorem 2. The validity of (R6) is based on the following theorem in logic:

$$((p\to q)\ \&\ (r\to s))\to((p\vee r)\to(q\vee s)).$$

The method of proving the remaining relationships is similar to the one used in the previous theorem. Here we shall present the proofs of only (R8), (R9), and (R10).

Consider first the relationships (R8) and (R9). Let

$$((A\vee B)\wedge'C)\overset{\text{def}}{=}X;\quad ((A\wedge'C)\wedge(B\wedge'C))\overset{\text{def}}{=}Y. \tag{32}$$

We can write, for any $i\in J_1^n$,

$$\text{row}_i(\underline{X})=\bigwedge_{k\in(\tau[\text{row}_i(\underline{A})]\cup\tau[\text{row}_i(\underline{B})])}\text{row}_k(\underline{C}),$$

$$\text{row}_i(\underline{Y})=\Big(\bigwedge_{k\in\tau[\text{row}_i(\underline{A})]}\text{row}_k(\underline{C})\Big)\wedge\Big(\bigwedge_{k\in\tau[\text{row}_i(\underline{B})]}\text{row}_k(\underline{C})\Big) \tag{33}$$

If $\tau[\text{row}_i(\underline{A})]=\tau[\text{row}_i(\underline{B})]=\phi$, then $\text{row}_i(\underline{X})$ and $\text{row}_i(\underline{Y})$ are both null. If neither $\tau[\text{row}_i(\underline{A})]$ nor $\tau[\text{row}_i(\underline{B})]$ are empty, then it is easy to verify that $\text{row}_i(\underline{X})=\text{row}_i(\underline{Y})$. If one of the sets $\tau[\text{row}_i(\underline{A})]$, $\tau[\text{row}_i(\underline{B})]$ is empty (but not both), then $\text{row}_i(\underline{Y})$ is null, but $\text{row}_i(\underline{X})$ is not necessarily null. Thus, with the exception of the case '$\tau[\text{row}_i(\underline{A})]=\phi$ or $\tau[\text{row}_i(\underline{B})]=\phi$ but not both', we can assert $\text{row}_i(\underline{X})\leftrightarrows\text{row}_i(\underline{Y})$. Otherwise, we have that $\text{row}_i(\underline{X})\leftarrow\text{row}_i(\underline{Y})$. This proves (R8) and (R9).

Consider now the relationship (R10). Let

$$((A\wedge B)\wedge'C)\overset{\text{def}}{=}X;\quad ((A\wedge'C)\vee(B\wedge'C))\overset{\text{def}}{=}Y. \tag{34}$$

For any $i\in J_1^n$,

$$\text{row}_i(\underline{X})=\bigwedge_{k\in(\tau[\text{row}_i(\underline{A})]\cap\tau[\text{row}_i(\underline{B})])}\text{row}_k(\underline{C})$$

$$\text{row}_i(\underline{Y})=\Big(\bigwedge_{k\in\tau[\text{row}_i(\underline{A})]}\text{row}_k(\underline{C})\Big)\vee\Big(\bigwedge_{k\in\tau[\text{row}_i(\underline{B})]}\text{row}_k(\underline{C})\Big). \tag{35}$$

By the condition of this relationship, we exclude the case where '$\tau[\text{row}_i(\underline{A})]=\phi$ or $\tau[\text{row}_i(\underline{B})]=\phi$ but not both'. If both $\tau[\text{row}_i(\underline{A})]$, $\tau[\text{row}_i(\underline{B})]$ are empty, then $\text{row}_i(\underline{X})$ and $\text{row}_i(\underline{Y})$ are both null. Consider now the case that neither $\tau[\text{row}_i(\underline{A})]$ nor $\tau[\text{row}_i(\underline{B})]$ are empty. We can then write

$$\begin{aligned}\tau[\text{row}_i(\underline{A})]&=(\tau[\text{row}_i(\underline{A})]\cap\tau[\text{row}_i(\underline{B})])\cup\alpha,\\ \tau[\text{row}_i(\underline{B})]&=(\tau[\text{row}_i(\underline{A})]\cap\tau[\text{row}_i(\underline{B})])\cup\beta,\end{aligned} \tag{36}$$

for some (perhaps empty) sets α, β. Thus,

$$\text{row}_i(\underline{Y})=\left[\text{row}_i(\underline{X})\wedge\left(\bigwedge_{k\in\alpha}\text{row}_k(\underline{C})\right)\right]\vee\left[\text{row}_i(\underline{X})\wedge\left(\bigwedge_{k\in\beta}\text{row}_k(\underline{C})\right)\right], \tag{37}$$

which is equivalent to

$$\text{row}_i(\underline{Y})=\text{row}_i(\underline{X})\wedge\left[\bigwedge_{k\in\alpha}\ldots\vee\bigwedge_{k\in\beta}\ldots\right]. \tag{38}$$

Hence $\text{row}_i(\underline{Y})\rightarrow\text{row}_i(\underline{X})$ in this case. This proves the implication (R10).

It is interesting to note the similarity of the pattern of relationships (R) in theorem 4 with the pattern of associativities in theorem 3. Note also that the pattern of left distributivities (L) is 'better structured' than the pattern (R); of special interest is the new pair of distributive properties (R8) and (R9).

Properties of the atomic elements in $\mathscr{R}(\mathfrak{S})$

(1) *Intersections*

$$(I\wedge\hat{I})=(I\wedge E)=(\hat{I}\wedge E)=O$$

(2) *Converses.* Since, I and $\hat{I}$ are converse relations, and E is self-converse we have for all $i\in J_1^n$

$$\begin{aligned}\text{row}_i(\underline{I})&=\text{col}_i(\underline{\hat{I}})\\ \text{row}_i(\underline{E})&=\text{col}_i(\underline{E})\end{aligned}$$

(3) *Transitivities*

$$(I\vee'I)\rightarrow I;\quad(\hat{I}\vee'\hat{I})\rightarrow\hat{I};\quad(E\vee'E)=E.$$

(4) *Restricted transitivities*

$$(I\wedge'I)=(\hat{I}\wedge'\hat{I})=O;\quad(E\wedge'E)=E.$$

(5) *Antisymmetry*

$$((I\vee E)\wedge(\hat{I}\vee E))=E.$$

This follows directly from distributivity of Boolean connectives and the intersection properties of atomic elements given in (1) above.

(6) *Equipotencies (Symmetric stable products)*

$$\begin{aligned}((I\vee E)\vee'(I\vee E))&=(I\vee E)\\ ((\hat{I}\vee E)\vee'(\hat{I}\vee E))&=(\hat{I}\vee E).\end{aligned}$$

This property is a corollary of the following, more general property of the

ordinary algebra of relations: If A is a transitive relation, then the following equality holds:

$$((A \vee E) \vee' (A \vee E)) = (A \vee E). \tag{39}$$

The validity of this property rests on the distributivity of $\vee'$ and $\vee$, on theorem 2 (3) and on the hypothesis of transitivity, that is, $(\underline{A} \vee' \underline{A}) \rightarrow A$.

The next (seventh) property is a corollary of the following theorem.

Theorem 5

Consider any two terms A, B in $\mathscr{R}(\mathfrak{S})$ such that B is transitive, $\hat{B}$ denotes the converse of B, and $A \rightarrow B$ holds; then the following equality holds:

$$((A \vee E) \wedge' (\hat{B} \vee E)) = (\hat{B} \vee E).$$

Proof. Let

$$((A \vee E) \wedge' (\hat{B} \vee E)) \stackrel{\text{def}}{=} X. \tag{40}$$

For any $i \in J_1^n$ we can write

$$\text{row}_i(\underline{X}) = \bigwedge_{k \in \tau[\text{row}_i(\underline{A} \vee \underline{E})]} \text{row}_k(\underline{\hat{B}} \vee \underline{E}) = \bigwedge_{k \in (\tau[\text{row}_i(\underline{A})] \cup \tau[\text{row}_i(\underline{E})])} \text{row}_k(\underline{\hat{B}} \vee \underline{E}) \tag{41}$$

Note that by the definition of E for all i, $\tau[\text{row}_i(\underline{E})]$ is not empty, and it contains a single element, namely i.

Suppose first that $\tau[\text{row}_i(\underline{A})]$ is empty; then $\text{row}_i(\underline{X}) = \text{row}_i(\underline{\hat{B}} \vee \underline{E})$.

Suppose alternatively that $\tau[\text{row}_i(\underline{A})]$ is not empty; specifically let

$$\tau[\text{row}_i(\underline{A})] = \{k_1, \ldots, k_m\}. \tag{42}$$

We may then write

$$\text{row}_i(\underline{X}) = [\text{row}_{k_1}(\underline{\hat{B}} \vee \underline{E}) \wedge \ldots \wedge \text{row}_{k_m}(\underline{\hat{B}} \vee \underline{E})] \wedge \text{row}_i(\underline{\hat{B}} \vee \underline{E}), \tag{43}$$

which can be put in the form,

$$\text{row}_i(\underline{X}) = [(\text{row}_{k_1}(\underline{\hat{B}}) \wedge \ldots \wedge \text{row}_{k_m}(\underline{\hat{B}})) \vee \alpha] \wedge \text{row}_i(\underline{\hat{B}} \vee \underline{E}), \tag{44}$$

for an appropriate α; let

$$(\text{row}_{k_1}(\underline{\hat{B}}) \wedge \ldots \wedge \text{row}_{k_m}(\underline{\hat{B}})) \stackrel{\text{def}}{=} \text{row}_i(\underline{Y}). \tag{45}$$

We shall now attempt to prove the following implication:

$$\text{row}_i(\underline{\hat{B}} \vee \underline{E}) \rightarrow \text{row}_i(\underline{Y}), \tag{46}$$

since if this implication holds then we may write:

$$[\text{row}_i(\underline{Y}) \vee \alpha] \wedge \text{row}_i(\underline{\hat{B}} \vee \underline{E}) = \text{row}_i(\underline{\hat{B}} \vee \underline{E}),$$

and hence we may obtain,

$$\text{row}_i(\underline{X}) = \text{row}_i(\underline{\hat{B}} \vee \underline{E}).$$

To prove that the implication in eq. (46) holds, we show that if any component of $\text{row}_i(\underline{\hat{B}} \vee \underline{E})$ is 1, then the corresponding component of $\text{row}_i(\underline{Y})$ is also 1. Note that $\text{row}_i(\underline{\hat{B}} \vee \underline{E}) = \text{row}_i(\underline{\hat{B}}) \vee \text{row}_i(\underline{E})$.

Consider first the case where the i-component of $\text{row}_i(\underline{\hat{B}} \vee \underline{E})$ is 1. Since by hypothesis $A \rightarrow B$, then

$$\tau[\text{row}_i(\underline{A})] \subseteq \tau[\text{row}_i(\underline{B})]. \tag{47}$$

Furthermore, since $\hat{B}$ is the converse of B, we may write

$$\tau[\text{row}_i(\underline{A})] \subseteq \tau[\text{col}_i(\underline{\hat{B}})]; \tag{48}$$

thus, in view of eqs. (42) and (48), we have that the i-components in $\text{row}_{k_1}(\underline{\hat{B}}), \ldots, \text{row}_{k_m}(\underline{\hat{B}})$ are all 1. Hence, by the definition (45) the i-component of $\text{row}_i(\underline{Y})$ is 1.

Now suppose that for any $j \in J_1^n$, $(j \neq i)$, the jth component in $\text{row}_i(\underline{\hat{B}})$ is 1. This means that the relation $\hat{B}$ holds between i and j. Then the converse relation B holds between j and i. By our supposition, the relation A holds between i and k_1, i and $k_2, \ldots, i$ and k_m. Since $A \rightarrow B$, then B also holds between i and k_1, i and $k_2, \ldots, i$ and k_m. Since B is transitive, and since it holds between j and i then B also holds between j and k_1, j and $k_2, \ldots, j$ and k_m. But then $\hat{B}$ holds between k_1 and j, k_2 and $j, \ldots, k_m$ and j. Thus, the j-components in $\text{row}_{k_1}(\underline{\hat{B}})$, $\text{row}_{k_2}(\underline{\hat{B}}), \ldots, \text{row}_{k_m}(\underline{\hat{B}})$ are all 1. Hence the j-component of $\text{row}_i(\underline{Y})$ is also 1.

From the above two cases we obtain that the implication $\text{row}_i(\underline{\hat{B}} \vee E) \rightarrow \text{row}_i(\underline{Y})$ holds, and hence $\text{row}_i(\underline{X}) = \text{row}_i(\underline{\hat{B}} \vee \underline{E})$. This proves the theorem.

(7) *Antisymmetric stable products*

$$((\hat{I} \vee E) \wedge'(I \vee E)) = (I \vee E),$$
$$((I \vee E) \wedge'(\hat{I} \vee E)) = (\hat{I} \vee E).$$

11. AN ALGEBRAIC MODEL $\mathscr{R}(\sigma^2)$ FOR 2-INPUT PROGRAMS

11.1 2-input relations and their matrices

We shall consider now *triadic relations* on our finite set $\mathfrak{S}$. These relations will be called 2-*input relations*, and they will be denoted by a capital letter with a parenthesized superscript 2; for example, $R^{(2)}$. A triadic (or 2-input) relation $R^{(2)}$ is a rule that specifies for each ordered pair $((x_1, x_2)y)$ [where $x_1, x_2, y \in \mathfrak{S}$, and the first element of the pair, (x_1, x_2), is itself an ordered pair] whether $R^{(2)}$ holds between (x_1, x_2) and y or not. For a finite set $\mathfrak{S}$ with n elements, we can represent a 2-input relation $R^{(2)}$ by a $n^2 \times n$ matrix of 0s and 1s. We denote this matrix by $R^{(2)}$.

The value of a component $r_{ij,k}$ of $\underline{R}^{(2)}$ (where $i, j, k \in J_1^n$) is defined as follows:

$$r_{ij,k} = \begin{cases} 1, \text{ if } R^{(2)} \text{ holds between the pair } (i,j) \text{ of elements in } \mathfrak{S} \text{ and} \\ \quad \text{the } k\text{-element in } \mathfrak{S}, \\ 0, \text{ otherwise.} \end{cases}$$

For a given indexing of the elements in $\mathfrak{S}$ let us arrange pairs of elements according to the ordering $((1, 1), (1, 2), \ldots, (1, n), (2, 1), \ldots, (n, 1), \ldots, (n, n))$, and let us establish a correspondence between the components of n^2-*dimensional row vectors* and this ordering. Now, in analogy to the 1-input

case, let an n^2-dimensional row vector with a 1 at its component ij and 0 elsewhere correspond to the pair (i, j) of elements in $\mathfrak{S}$.

If u_1, u_2 are variables that take as values elements in $\mathfrak{S}$, then let $\underline{u_1u_2}$ denote the representation of the pair (u_1, u_2) as an n^2-dimensional row vector. We denote by the expression

$$q = \underline{u_1u_2}\, \underline{R}^{(2)},$$

an ordinary multiplication of the n^2-dimensional row vector $\underline{u_1u_2}$ and the $n^2 \times n$ matrix $\underline{R}^{(2)}$ that yields the n-dimensional row vector q under the rule:

$$q = \mathrm{row}_{ij}(\underline{R}^{(2)}), \text{ if } ij \text{ is the component of } \underline{u_1u_2} \text{ whose value is 1.}$$

We shall now introduce two operations $\vee''$ and $\wedge''$ for composing triadic relations from dyadic relations. These operations will be called *cross-products*; they are defined as follows:
For any 1-input relations A, B, any 2-input relation $C^{(2)}$, and also for all $i, j \in J_1^n$,

$$(C^{(2)} = A \vee'' B) \equiv (\mathrm{row}_{ij}(\underline{C}^{(2)}) = \mathrm{row}_i(\underline{A}) \vee \mathrm{row}_j(\underline{B})),$$
$$(C^{(2)} = A \wedge'' B) \equiv (\mathrm{row}_{ij}(\underline{C}^{(2)}) = \mathrm{row}_i(\underline{A}) \wedge \mathrm{row}_j(\underline{B})). \tag{48}$$

By using these definitions, we may now introduce the notion of extensions of 1-input relation matrices to 2-input relation matrices. Given a 1-input matrix $\underline{A}$, its *left extension* is denoted by $\underline{A}_1^{(2)}$ and is defined as follows:

$$\underline{A}_1^{(2)} \equiv (A \vee'' 0) \equiv (\mathrm{row}_{ij}(\underline{A}_1^{(2)}) = \mathrm{row}_i(\underline{A})), \tag{49}$$

where 0 denotes the 'null' relation.
The *right extension of* $\underline{A}$ is denoted by $\underline{A}_2^{(2)}$ and is defined as follows:

$$\underline{A}_2^{(2)} \equiv (0 \vee'' A) \equiv (\mathrm{row}_{ij}(\underline{A}_2^{(2)}) = \mathrm{row}_j(\underline{A})). \tag{50}$$

As an example, the left and right extensions of the proper inclusion matrix $\underline{I}$ for a set of 4 elements [given in eq. (15)] are as shown in eq. (51):

$$\underline{I}_1^{(2)} = \begin{pmatrix} 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&1&1&0\\ 1&1&1&0\\ 1&1&1&0\\ 1&1&1&0 \end{pmatrix} \qquad \underline{I}_2^{(2)} = \begin{pmatrix} 0&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&1&1&0\\ 0&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&1&1&0\\ 0&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&1&1&0\\ 0&0&0&0\\ 1&0&0&0\\ 1&0&0&0\\ 1&1&1&0 \end{pmatrix} \tag{51}$$

On the basis of the definition of extensions from 1-input to 2-input relation matrices, we can now express cross-products in terms of Boolean operations as follows:
For any pair of 1-input relations A, B,

$$\begin{aligned}(A \vee '' B) &\equiv (\underline{A}_1^{(2)} \vee \underline{B}_2^{(2)}),\\ (A \wedge '' B) &\equiv (\underline{A}_1^{(2)} \wedge \underline{B}_2^{(2)}).\end{aligned} \tag{52}$$

Note that cross-product operations can be used to compose relations of higher order than triadic from lower-order relations. As in the present case, these compositions can be shown to be equivalent to Boolean operations between matrices that are appropriate extensions of the matrices that represent the composing relations.

The notion of cascade products is extended in a natural way from 1-input relations to 2-input relations. Thus, we have

$$(C^{(2)} = A^{(2)} \vee ' B) \equiv (\text{row}_{ij}(\underline{C}^{(2)}) = \bigvee_{k \in \tau[\text{row}_{ij}(\underline{A}^{(2)})]} \text{row}_k(\underline{B}), \text{ for all } i, j \in J_1^n),$$

$$(C^{(2)} = A^{(2)} \wedge ' B) \equiv (\text{row}_{ij}(\underline{C}^{(2)}) = \bigwedge_{k \in \tau[\text{row}_{ij}(\underline{A}^{(2)})]} \text{row}_k(\underline{B}), \text{ for all } i, j \in J_1^n).$$

In these operations, a $n^2 \times n$ relation matrix is obtained from the product of a $n^2 \times n$ matrix and a $n \times n$ matrix.

Boolean operations on 2-input relations are defined in the ordinary way: For any 2-input relations $A^{(2)}$, $B^{(2)}$, $C^{(2)}$ and for all $i, j \in J_1^n$,

$$\begin{aligned}(C^{(2)} = A^{(2)} \vee B^{(2)}) &\equiv (\text{row}_{ij}(\underline{C}^{(2)}) = \text{row}_{ij}(\underline{A}^{(2)}) \vee \text{row}_{ij}(\underline{B}^{(2)}))\\ (C^{(2)} = A^{(2)} \wedge B^{(2)}) &\equiv (\text{row}_{ij}(\underline{C}^{(2)}) = \text{row}_{ij}(\underline{A}^{(2)}) \wedge \text{row}_{ij}(\underline{B}^{(2)})).\end{aligned} \tag{53}$$

11.2 The language of 2-input relational expressions $\mathscr{L}_{R,2}$

$\mathscr{L}_{R,2}$ is an extension of $\mathscr{L}_{R,1}$. It can be defined in terms of a *CF* grammar, $G_{R,2}$, which includes all the elements of the grammar of $\mathscr{L}_{R,1}$ that were given in section 10.2 (except that ρ_1 is not the starting element in the present case), and it has the following additions:

(a) The non-terminal vocabulary contains the constants 0 (signifying the 'null' relation), $\vee''$, $\wedge''$, and variables for 2-input relations, $\underline{R}^{(2)}$, $\underline{A}^{(2)}$, $\underline{B}^{(2)}$, and so on.

(b) A non-terminal element ρ_2, which stands for a well-formed 2-input relational expression; it is also the starting element of the grammar.

(c) the following rules of replacement:

$$\begin{array}{ll}\alpha_1, \alpha_2\text{:} & \rho_2 \to (\rho_2 \vee \rho_2) \text{ or } (\rho_2 \wedge \rho_2)\\ \beta_1, \beta_2\text{:} & \to ((\rho_1 \vee '' 0) \vee \rho_2) \text{ or } ((\rho_1 \vee '' 0) \wedge \rho_2)\\ \gamma_1, \gamma_2\text{:} & \to (\rho_2 \vee (\rho_1 \vee '' 0)) \text{ or } (\rho_2 \wedge (\rho_1 \vee '' 0))\\ \delta_1, \delta_2\text{:} & \to (\rho_2 \vee ' \rho_1) \text{ or } (\rho_2 \wedge ' \rho_1)\\ \varepsilon_1, \varepsilon_2\text{:} & \to (\rho_1 \vee '' \rho_1) \text{ or } (\rho_1 \wedge '' \rho_1)\\ \zeta_i\text{:} & \to R^{(2)} \text{ or } A^{(2)} \text{ or } B^{(2)} \text{ or} \ldots\end{array} \tag{54}$$

The language of 2-input relational expressions is the set of terminal strings that are derivable in the grammar $G_{R,2}$ from ρ_2.

11.3 Modelling programs by 2-input relational expressions

As in the case of 1-input programs, correspondences between programs and relational expressions will be based on the notion of functional equivalence.

A $\pi^{(2)}$-statement corresponds to the expression

$$q = \underline{u_1 u_2}\, \underline{R}^{(2)} \tag{55}$$

where $R^{(2)}$ is a relation variable that corresponds to the program variable π in the $\pi^{(2)}$-statement.

A (u_1u_2, q)-statement in the language of 2-input programs corresponds to the non-terminal ρ_2 in the language $\mathscr{L}_{R,2}$.

Consider now correspondences between rules of aggregation for 2-input programs (*see* table 2) and the new rules of replacement for relational expressions that are given in eq. (54). The rules of aggregation $X^2_\vee$ and $X^2_\wedge$ correspond to the rules α_1 and α_2; the rules $X^2_{\vee,(1)}$ and $X^2_{\wedge,(1)}$ correspond to the rules β_1 and β_2; the rules $X^2_{\vee,(2)}$ and $X^2_{\wedge,(2)}$ correspond to the rules γ_1 and γ_2; the rules $X^2_{\vee'}$ and $X^2_{\wedge'}$ correspond to the rules δ_1 and δ_2; the rules $X^2_{\vee''}$ and $X^2_{\wedge''}$ correspond to the rules ε_1 and ε_2; the rules X^2_π [*see* eq. (13)] for assigning variables to (u_1u_2, q)-statements correspond to the rules ζ in eq. (54).

To examine the reasoning behind these correspondences, let us consider two specific cases: the rule $X^2_{\vee''}$, and the rule $X^2_{\vee,(1)}$. The rule $X^2_{\vee''}$ is shown in figure 18.

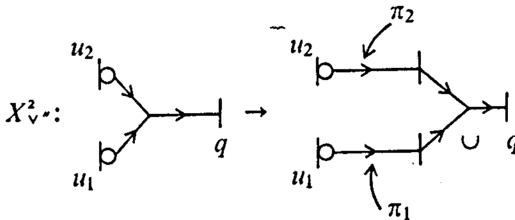


Figure 18

We label the aggregates in the rule $X^2_{\vee''}$ in order to facilitate the exposition. Let $\underline{u}_1$, $\underline{u}_2$ be the n-dimensional vectors that correspond to u_1, u_2 respectively, and furthermore let i and j be the components of $\underline{u}_1$ and $\underline{u}_2$ respectively whose value is 1. Also suppose that the 1-input relations A, B, correspond to π_1, π_2 respectively. We may then write for the right aggregate in figure 18,

$$q = \underline{u}_1\underline{A} \vee \underline{u}_2\underline{B} = \text{row}_i(\underline{A}) \vee \text{row}_j(\underline{B}). \tag{56}$$

Let π stand for the right aggregate, and suppose that $C^{(2)}$ is its corresponding 2-input relation; then we may write,

$$q = \underline{u_1 u_2}\,\underline{C}^{(2)} = \text{row}_{ij}(\underline{C}^{(2)}). \tag{57}$$

From eqs. (56) and (57), and in view of the definition of $\vee''$, in eq. (48) it is evident that the 2-input program π at the right is functionally equivalent to $(A \vee'' B)$. Thus, the correspondence between $X^2_{\vee''}$ and the rule ε_1 in eq. (54) is justified.

Consider now the rule of aggregation $X^2_{\vee,(1)}$ shown in figure 19.

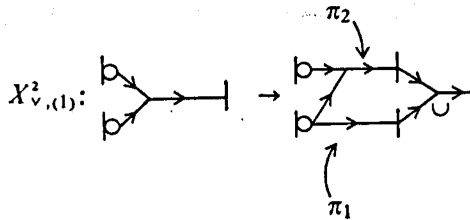


Figure 19

The aggregate at the right side of the rule is functionally equivalent to the aggregate shown in figure 20.
Suppose that A, $B^{(2)}$ are relation variables that correspond to π_1, π_2 respectively. Then, the aggregate in figure 20 can be shown to be functionally equivalent to $(A \vee {''}0) \vee B^{(2)}$. Thus, the correspondence between $X^2_{\vee,(1)}$ and the rule β_1 in eq. (54) is justified.

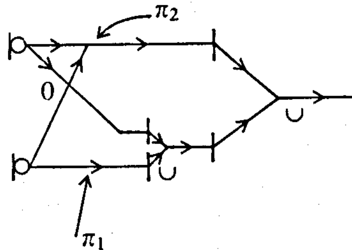


Figure 20

Given the correspondences between $\mathscr{L}_{R,2}$ and the language of 2-input programs, we now have a broader translation basis from programs to expressions. The translation defines an extended one-one mapping $\mu^{(2)}$ from programs to relational expressions. Starting from the abstract description of a program p, we can easily obtain the translation via a syntax-directed procedure whose operation is determined by the correspondences between the rules of the grammar of programs and the rules of $G_{R,2}$. We can now state the following theorem.

Theorem 6
(Projecting 2-input programs into a model)
There exists a mapping $\mu^{(2)}$, which is an extension of μ, that takes any 2-input constructible program into a functionally equivalent 2-input relational expression; furthermore, there exists an effective translation procedure for computing $\mu^{(2)}$.

11.4 The algebra of 2-input relational expressions $\mathscr{R}(\mathfrak{S}^2)$

The system $\mathscr{R}(\mathfrak{S}^2)$ is a *modified algebra of triadic relations* that extends the system $\mathscr{R}(\mathfrak{S})$. The atomic elements of $\mathscr{R}(\mathfrak{S}^2)$ are as in $\mathscr{R}(\mathfrak{S})$; there are two new operations, $\vee''$ and $\wedge''$.

The terms of $\mathscr{R}(\mathfrak{S}^2)$ are the relational expressions in $\mathscr{L}_{R,2}$. Every term in $\mathscr{R}(\mathfrak{S}^2)$ can be interpreted as an $n^2 \times n$ matrix, as discussed previously. The terms of the algebra $\mathscr{R}(\mathfrak{S}^2)$ are partly ordered under an implication relation $\rightarrow$.

The logical interpretation of $(A^{(2)} \rightarrow B^{(2)})$ is that for every ordered triple of elements in $\mathfrak{S}$ (denoted by i, j, k), if the relation $A^{(2)}$ holds between the pair (i, j) and k, then the relation $B^{(2)}$ also holds between them. We have

$$(A^{(2)} \rightarrow B^{(2)}) \equiv (\underline{A}^{(2)} \leqslant \underline{B}^{(2)}) \equiv (\tau(\text{row}_{ij}(\underline{A}^{(2)})) \subseteq \tau(\text{row}_{ij}(\underline{B}^{(2)})) \text{ for all } i, j \in J_1^n) \qquad (58)$$

The operations $\vee$, $\wedge$ have the usual Boolean associativities and distributivities when operating on 2-input relations.

All the properties of $\vee'$, $\wedge'$ presented in the theorems 2, 3 and 4 carry over to $R(\mathfrak{S}^2)$, with appropriate replacements of 1-input relation variables by 2-input relation variables. Let us consider a few examples:

From theorem 2 (5d): for any $A^{(2)}$, $B^{(2)}$, C, if $A^{(2)} \rightarrow B^{(2)}$, then $(A^{(2)} \wedge' C) \leftarrow (B^{(2)} \wedge' C)$ holds.

From theorem 3: for any $A^{(2)}$, B, C, $((A^{(2)} \vee' B) \vee' C) = (A^{(2)} \vee' (B \vee' C))$.

From theorem 4 (R8, 9): for any $A^{(2)}$, $B^{(2)}$, C such that there is no pair (i, j), $i, j \in J_1^n$, for which one of the rows $\text{row}_{ij}(\underline{A}^{(2)})$, $\text{row}_{ij}(\underline{B}^{(2)})$ is null (but not both), the following equality holds:

$$((A^{(2)} \vee B^{(2)}) \wedge' C) = ((A^{(2)} \wedge' C) \wedge (B^{(2)} \wedge' C)).$$

Let us consider now the properties of the cross products $\vee''$, $\wedge''$ and the interactions of these products with the Boolean operations and the cascade products $\vee'$, $\wedge'$.

The associativities of $\vee''$, $\wedge''$ are the same as the associativities of the Boolean operations $\vee$, $\wedge$ respectively. The operations $\vee''$, $\wedge''$ distribute with Boolean operations in the same manner as $\vee$ and $\wedge$ respectively. One should be careful to apply Boolean operations to relations that are of the same type (that is, 1-input relations or 2-input relations); cross products apply only to 1-input relations, and they produce 2-input relations. For example, the following are valid properties:

$$(A \wedge C) \vee'' B = (A \vee'' B) \wedge (C \vee'' B)$$
$$A \vee'' (B \wedge C) = (A \vee'' B) \wedge (A \vee'' C),$$

for any 1-input relations A, B, C.

The right distributivities of the cascade products of $\vee'$, $\wedge'$ with respect to the cross products $\vee''$, $\wedge''$ are the same as those between $\vee'$, $\wedge'$ and the Boolean operations $\vee$, $\wedge$; they can be obtained directly from theorem 4 (R), by exchanging throughout $\vee$, $\wedge$ by $\vee''$, $\wedge''$ respectively. Let us consider two examples:

For any 1-input relations A, B, C,

From (R1): $((A \vee'' B) \vee' C) = ((A \vee' C) \vee'' (B \vee' C))$.

From (R2): $((A \wedge'' B) \vee' C) \rightarrow ((A \vee' C) \wedge'' (B \vee' C))$.

In figure 21 we show a part of the implicational structure of terms in $\mathscr{R}(\mathfrak{S}^2)$.

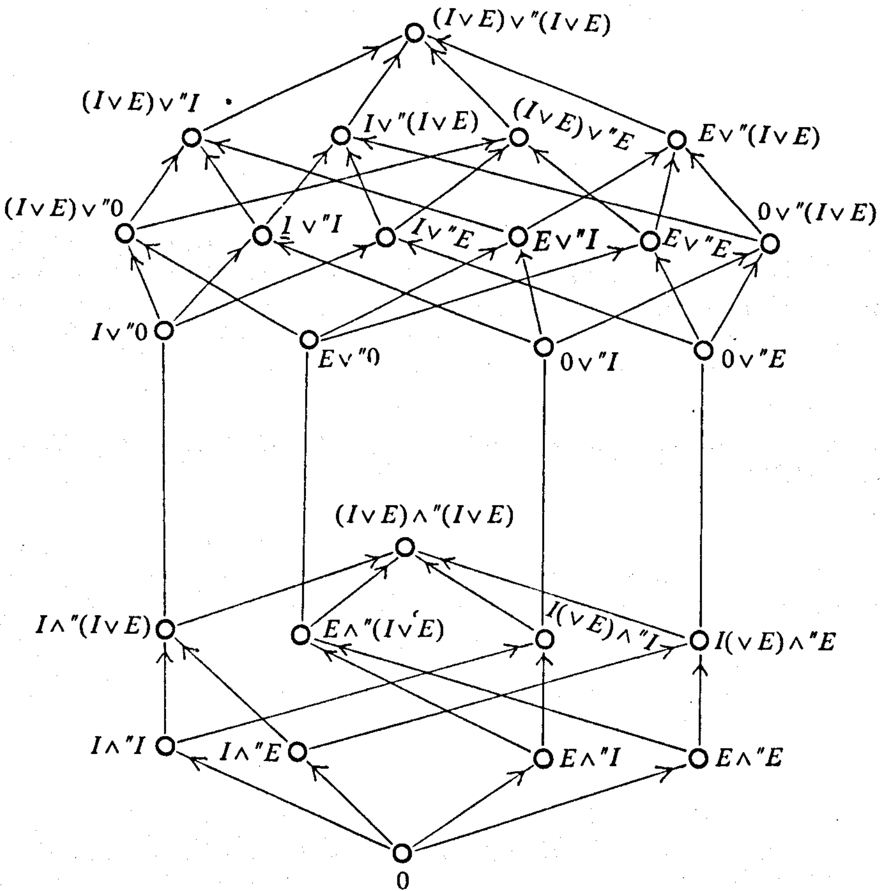


Figure 21. Part of the implicational structure of $\mathscr{R}(\mathfrak{S}^2)$.

12. A PERFORMANCE SPACE FOR PROGRAMS

The mapping $\mu^{(2)}$ induces the implicational structure of $\mathscr{R}(\mathfrak{S}^2)$ on the language of 2-input constructible programs. For any two 2-input programs p_1, p_2 with projections $P_1^{(2)}$ and $P_2^{(2)}$ respectively in the system $\mathscr{R}(\mathfrak{S}^2)$, if the implication $P_1^{(2)} \rightarrow P_2^{(2)}$ holds, then we may also write $p_1 \rightarrow p_2$. This means that for any pair of inputs u_1, u_2 the output computed by p_1 (this can be ϕ or a subset of $\mathfrak{S}$) is included in the output computed by p_2. An alternative way of describing the situation is that the $n^2 \times n$ relation matrix that corresponds to p_1 is included in the relation matrix that corresponds to p_2.

The system $\mathscr{R}(\mathfrak{S}^2)$ gives us a calculus of programs which can be used for manipulating programs and for reasoning about equivalence and inclusion of programs.

As an example of using $\mathscr{R}(\mathfrak{S}^2)$ in a formation procedure, consider predict-

ing the effect of carrying out the structural move shown in figure 22 which changes the program p_1 into p_2. By projecting the two programs into $\mathscr{R}(\mathfrak{S}2)$ we obtain,

$$P_1^{(2)}=(I\wedge''I)\wedge'\hat{I}$$
$$P_2^{(2)}=(I\wedge''I)\wedge'(\hat{I}\wedge E). \tag{59}$$

By applying the left distributivity property of $\wedge'$ [*see* theorem 4(L8)] on $P_2^{(2)}$ we obtain,

$$P_2^{(2)}=((I\wedge''I)\wedge'\hat{I})\wedge((I\wedge''I)\wedge'E)$$
$$=P_1^{(2)}\wedge((I\wedge''I)\wedge'E). \tag{60}$$

Thus, the inclusion relation $p_2\rightarrow p_1$ holds; also the exact nature of the functional variation between p_1 and p_2 that results from the given structural variation of p_1 is clearly obtained in the analysis.

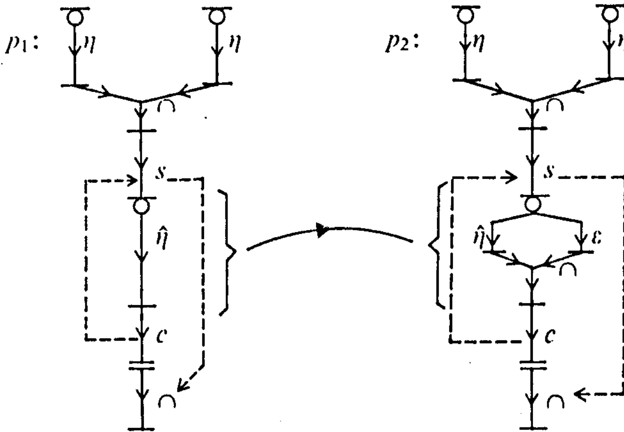


Figure 22. A structural move that changes program p_1 and p_2.

The number of terms in $\mathscr{R}(\mathfrak{S}2)$ is not finite. Correspondingly, there is an infinite number of 2-input programs in the language $\mathscr{L}_{g,2}$. However, the functional variety which is possible in the set of 2-input programs is finite, since there is a finite number of $n^2\times n$ relation matrices that correspond to the terms of $\mathscr{R}(\mathfrak{S}2)$; this number is certainly less than 2^{n^3}. We can visualize all the 2-input programs as organized in a finite number of equivalence classes, sorted by $n^2\times n$ relation matrices – one distinct matrix per class. To each equivalence class of programs there corresponds a point in a space; we call this space the *performance space* for the set of constructible programs, and we denote it by $\Pi(\mathfrak{S}2)$.

The points of $\Pi(\mathfrak{S}2)$ are partly ordered under relation inclusion $\leqslant$. They form a sublattice of the full Boolean lattice of triadic relations of 2^{n^3} elements which we denote here by $B(\mathfrak{S}2)$.

Each constructible program is in one of the equivalence classes represented

in $\Pi(\mathfrak{S}^2)$. The functional specification of the 'desired program' which is given in the statement of the formation problem can always be represented as a point in the Boolean lattice of triadic relations. Furthermore, if the desired program is realizable in the given language of constructible programs, then it is representable by a point in $\Pi(\mathfrak{S}^2)$. Thus, the lattice $B(\mathfrak{S}^2)$ provides a space where we can represent both the functional conditions that the 'desired program' (the solution) must satisfy and also the functional properties of all the possible solution candidates. In this space we can now develop strategies for going from points that represent known program structures to the goal point that represents the desired program.

Clearly, there are cases where a given program formation problem has no solution. In these cases the function that we are asked to implement is not among the points of $\Pi(\mathfrak{S}^2)$. In general, the problem-solving system has no way of knowing this fact by analyzing the problem statement. It can only try to come as close as possible to the point in the Boolean lattice that represents the desired function. In order to avoid an endless search in the 'no solution' cases, a predetermined ceiling on available computing effort is set in the system, and the system stops when the ceiling is reached.

In order to give operational meaning to the notion of 'closeness' between programs, we introduce a *distance function* in the Boolean lattice $B(\mathfrak{S}^2)$.

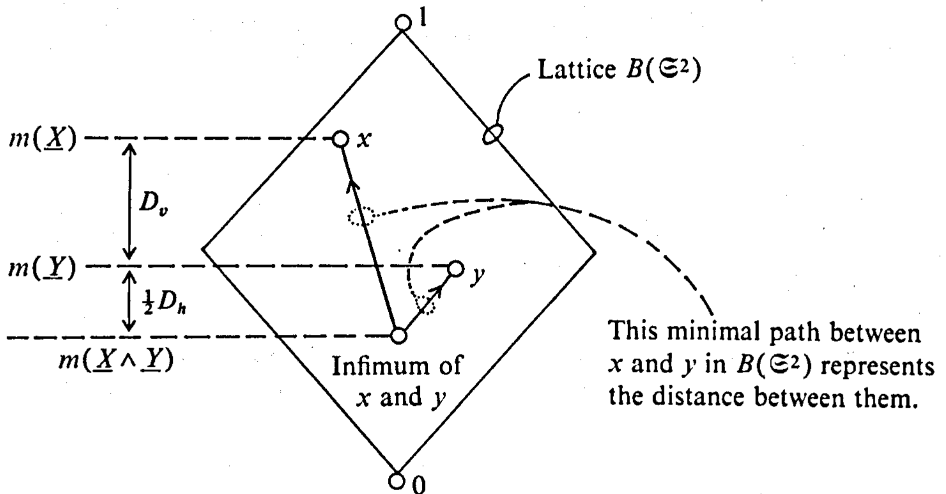


Figure 23. Distance between two points in the Boolean lattice $B(\mathfrak{S}^2)$

Let $\underline{X}$ be the $n^2 \times n$ relation matrix that corresponds to a point x in $B(\mathfrak{S}^2)$ and let us denote by $m(\underline{X})$ the number of 1s in the matrix $\underline{X}$. Then the distance $D(x, y)$ between two points x, y in the lattice is defined as follows:

$$D(x, y) = m(\underline{X} \vee \underline{Y}) - m(\underline{X} \wedge \underline{Y}). \qquad (61)$$

D is a measure of the symmetric difference between the relation matrices of the two points in $B(\mathfrak{S}^2)$.

It is useful to think of D as made of two additive components, a 'vertical' component D_v and a 'horizontal' component D_h as follows:

$$D(x, y) \overset{\text{def}}{=} D_v(x, y) + D_h(x, y),$$

where, if we assume $m(\underline{X}) \geqslant m(\underline{Y})$, then we have,

$$D_v(x, y) = m(\underline{X}) - m(\underline{Y})$$
$$D_h(x, y) = 2(m(\underline{Y}) - m(\underline{X} \wedge \underline{Y})). \tag{62}$$

In a graphic representation of $B(\mathfrak{S}^2)$ the null relation is shown at the bottom, and for any pair of points x, y if $x < y$, then x is shown below y in the lattice. As we move up in the lattice the values of m increase. In figure 23 we show schematically the distance relationships between points in $B(\mathfrak{S}^2)$.

For any pair of points x, y in $B(\mathfrak{S}^2)$, if $x < y$ then the 'horizontal' distance between x and y is zero, and the vertical distance is a measure of the strength of inclusion.

13. AN APPROACH TO A STRATEGY OF FORMATION

The notions that we have introduced so far induce important changes in the representation of the program formation problem. The problem can now be presented to a problem-solving system in the following form:

Given,

(1) a data base $\langle \mathfrak{S}, I, E \rangle$;

(2) a language $\mathscr{L}_{l,2}$ (or $\mathscr{L}_{g,2}$) of 2-input constructible programs which is generated by the grammar, $G_{l,2}$ (or $G_{g,2}$), and for which there is an algebraic model $\mathscr{R}(\mathfrak{S}^2)$;

(3) a performance space $\Pi(\mathfrak{S}^2)$ which is a subset of the Boolean lattice $B(\mathfrak{S}^2)$ and it is such that for each constructible program there is a corresponding point in $\Pi(\mathfrak{S}^2)$ (call it a *performance point*) which can be determined by executing the program over its data domain $\mathfrak{S} \times \mathfrak{S}$; furthermore, a distance function is defined in $B(\mathfrak{S}^2)$;

(4) a *goal point* g in the lattice $B(\mathfrak{S}^2)$ which corresponds to the given set of correspondences C_θ.

Find a constructible program whose corresponding performance point in $\Pi(\mathfrak{S}^2)$ is either the goal point or comes as close as possible to the point g.

The goal point represents the functional conditions that the 'desired program' must satisfy. Consider a program p that has been generated and tested by the formation system. Let us represent p by a pair $(P^{(2)}, (D_v(p, g), D_h(p, g)))$, which we call the *characteristic description* of p relative to g. The first element in the pair is the structural description of p in $G_{g,2}$ or its corresponding relational expression; the second element consists of the pair of distance components (the vertical and the horizontal) between p and g.

At any stage of the problem-solving process, the *problem state* is given by the set of characteristic descriptions relative to g of the programs already generated.

Given a problem state, the problem solver must decide what 'next program' to generate; after making a choice, the new program is executed and tested relative to g; if a zero distance is obtained between the new program and g, a solution has been found, and the process stops; if this is not the case, and the given ceiling of computing effort has not been reached yet, then the process is repeated from the new problem state.

The objective of a good problem solver is to reach a solution after a small number of program generations. The key problem is how to best use the specific information in the current problem state, together with general knowledge about the program space, in deciding what move to take, that is, what next program to generate.

We can regard the problem as a *navigation* problem in the lattice $B(\mathfrak{S}^2)$. Appropriate navigating moves can be formulated on the basis of known relationships (from the algebraic model) between program structures and the relative positions of their performance points in the lattice. A dominant direction in the lattice is the 'vertical', which is determined by inclusion relations. Thus a reasonable plan of navigation would be to first move into a rough 'vertical' alignment with the given point g, by trying to reduce the 'horizontal' distance, and then to slide into g by using the rich set of implication relationships in the algebraic model.

A more specific formation strategy which is based on this overall navigation plan is outlined next.

(1) Generate and test a set of simple programs in the form $(A \vee {}''B)$, where A, B can take as values the constants I, $\hat{I}$, E. These programs will create scattered points in performance space. Consider for further processing a subset of these programs whose 'horizontal' distance to g is small.

(2) Use the subset of programs produced in (1) for forming Boolean combinations until a program is obtained with zero 'horizontal' distance to g.

(3) Change Boolean operators in the last program so that 'vertical' distance to g will be reduced. Let us denote the program produced in this step by p_c.

(4) Form other programs with a zero 'horizontal' distance to g *and* a large 'horizontal' distance to p_c. A promising form for such a program is $P_c^{(2)} \wedge {}'C$; try several simple structures for the block C until the desired properties are obtained. Note that $P_c^{(2)}$ is the relational expression corresponding to p_c. Let us denote the program produced in this step by p_d.

(5) Form a program $P_c^{(2)} \wedge P_d^{(2)}$. This program must be closer to the point g than p_c or p_d are, and possibly it is the 'desired program'. If not, go back to step (2) and develop a new line of programs by proceeding again as indicated in (3), (4), and (5).

In this strategy we are specifying an evolutionary process which starts by generating and testing an initial variety of structures, it proceeds by combining successful structures and subsequently refining them, and finally it ends with a terminal manoeuvre where widely different structures, each individually evolved and refined, are intersected.

Note that the formation strategy that we have just outlined transforms the formation problem into an ordered set of much simpler subproblems. The only subproblems that require search are (2), (3), and (4). These are *derivation* problems where the reasoning can proceed from the problem statement to a solution via the use of knowledge about the implicational structure of $\mathscr{R}(\mathfrak{S}^2)$.

14. AN EXAMPLE OF A FORMATION PROCESS

Let us illustrate the formation strategy with the problem of finding a program for the Infimum function. (A program for the Infimum is shown in figure 12, and its associated structural description in figure 11.) Suppose that the specification for the Infimum function is given to the system in terms of 16 correspondences that are defined for a lattice of 4 elements. A trace of the problem-solving activity in a (hand simulated) formation system that uses the proposed strategy is as follows:

(1) A set of simple programs are generated and tested. The following are the characteristic descriptions of some of the programs:

$$((\hat{I} \vee {}''\hat{I}), (13, 32))$$
$$((\hat{I} \vee {}''I), (23, 16))$$
$$((I \vee {}''I), (13, 8))$$
$$((E \vee {}''E), (12, 4))$$

Note that the pair of numbers at the right stand for 'vertical' and 'horizontal' distance from g. The programs $(I \vee {}''I)$ and $(E \vee {}''E)$ have the smallest 'horizontal' distance from g and they are chosen for the next step.

(2) A Boolean combination of the programs obtained from step (1) yields immediately a zero 'horizontal' distance. We have the characteristic description,

$$(((I \vee {}''I) \vee (E \vee {}''E)), (29, 0)).$$

This program is equivalent to $((I \vee E) \vee {}''(I \vee E))$.

(3) To 'slide down vertically' in the direction of g, a program is sought which must be included in the program of step (2) and it should also include the 'desired' program. By changing the cross-product, we obtain,

$$(((I \vee E) \wedge {}''(I \vee E)), (9, 0));$$

let us call this program p_c.

(4) By using a cascade form based on p_c, we obtain

$$(((I \vee E) \wedge {}''(I \vee E)) \wedge {}'(\hat{I} \vee E)), (33, 0);$$

let us call this program p_d.

(5) By intersecting the programs p_c and p_d we obtain the desired solution,

$$(((I \vee E) \wedge {}''(I \vee E)) \wedge (((I \vee E) \wedge {}''(I \vee E)) \wedge {}'(\hat{I} \vee E))), (0, 0).$$

It can be seen that the relational expression obtained in step (5) represents exactly the Infimum program shown in figure 12.

The number of decisions made during this formation process is about 20. The last two types of formation action in (4) and (5) involve the combination of relatively large blocks that were obtained from previous steps. In general the process is tightly guided and extremely efficient; almost no search is involved.

It should be noted that during the formation process a structural description of the desired program is gradually built 'from the bottom up'. First some of the terminal subblocks are built, and then they are combined into bigger blocks. The process differs appreciably from the 'top to bottom' generation process that was studied in our previous work (Amarel 1962b).

While the grammar of the program language determines the program units and their possible modes of aggregation, it is the strategy of formation – which grows from reasoning in the model – that determines the specific manner in which the system constructs a structural description of the desired solution-program.

15. CONCLUDING COMMENTS

The formation strategy which is based on our algebraic model of program space specifies processes of different types for different stages of the problem-solving activity. There is an 'opening' process which is exploratory in character, and which uses relatively simple navigation techniques – based on distance measurements – to reach a desired region in program space. This is followed by a 'middle' process whose goal is more sharply defined, and whose methods rely on distance measurements from the desired destination in program space, and also on known algebraic properties of the space. Finally, there is a 'terminal' process, where specific coordinated moves are used to reach the solution point, starting from appropriate jump-off points that were obtained during the 'middle' process. The decisions in this stage are based on order properties in program space and on other explicit algebraic properties of the model.

The major effect of the algebraic model is to permit a transformation of the formation problem into a set of derivation problems where it is possible to reason directly from goals to choices of relevant problem-solving moves. The effect of this type of transformation is strongly to reduce the amount of effort needed to solve the initial formation problem.

The steps that are required in the transition from the initial representation of the formation problem to its improved representation are the following:

(1) find a theoretical model of program space,

(2) find (extend) properties of the model,

(3) use the model in formulating an overall problem-solving strategy, and in building a repertoire of problem-solving moves.

The mechanization of (3) is within sight at present; yet it is a challenging problem for workers in artificial intelligence. The mechanization of (2) is more difficult. It involves the *finding* of 'interesting' theorems in a mathema-

tical system, and it assumes a fairly well-developed machinery for theorem-proving. The notion of an 'interesting' theorem in the present context is to be taken in the sense of contributing to the formulation of a powerful problem-solving procedure.

At present we know very little about possible approaches to the mechanization of (1). Finding a mathematical system that models a set of phenomena is a difficult intellectual task that often involves not only choosing among available 'shelf' systems the one which is most appropriate, but also trying to extend existing mathematical models in such a way that they will fit the phenomenon under consideration. In our program formation problem we had to extend the existing algebra of relations in order to account for a certain type of cascading that occurs in our given world of programs.

One of the great difficulties in establishing a modelling relationship between two systems is the fact that the systems may be described in terms of completely different concepts. At least one of the descriptions must be manipulated and translated into a form where modelling correspondences with the second system are easily recognized. In our present problem it was necessary to change the initial description of constructible programs as river-like graphs into a grammatical formulation that assigns to each program a structural description. Given the structural description of a program, its correspondence with an expression in the algebra of relations can be 'easily seen'. It is possible, however, to have grammatical formulations of the program language that do not assign structural descriptions to programs that are isomorphic to relational expressions. The problem then is how to find a grammar for the program language that permits the establishment of easily recognizable correspondences with a mathematical model. This was precisely the modelling problem that we faced in the context of the program formation problem.

Processes of model finding are exclusively in the domain of humans at present. In order better to understand these processes and to advance their possible mechanization, it would be most helpful to study them in a man-machine interactive environment. This would force us to clarify the nature of models and of their relationship with the phenomena that they are intended to model; it would also induce the development of appropriate descriptive tools for computer manipulation of models, and the identification of key processes that occur in model-finding activities.

A model is a symbolic entity with a pragmatic requirement attached to it; it must be useful in the solution of problems in a given class. Therefore, model finding and model utilization should be studied jointly in the context of specific problem-solving processes. The present paper is an attempt in this direction.

In our formation problem the objects to be constructed are relatively simple computer programs. It is clear to us now that in order to have a formation procedure of reasonable power, we must furnish the procedure

with a problem representation which includes not only a good grammatical description of the language of constructible programs but also a mathematical system (a theory) within which the procedure can conduct *a priori* tests of possible functional effects of structural changes in programs. The question arises how much theory should be made available to the formation procedure, and, more generally, what would be the relationship between its performance and the quality/completeness of the theoretical knowledge that it possesses. Much more work is needed to clarify these questions.

The modified algebra of relations presented in this paper provides a good theoretical framework for the study of formal properties of our programs, such as functional equivalence. This framework is directly applicable to program languages that are extensions of the language of programs presented here (with a larger vocabulary of primitive relations and with possible extensions beyond triadic relations) and that may be designed typically for fact-retrieval applications in the context of structured data bases.

In program languages where our theoretical framework is not directly applicable it may still be possible to obtain useful results by using our general approach. Specifically, the idea is to find for a set of programs (it may be only a part of a program language) a representation which will permit them to be modelled in a mathematical system where formal properties of the programs could be studied. It is not unlikely that this type of activity may stimulate the development of some new mathematics that would be of special significance to computer science.

Acknowledgement

The research reported in this paper was sponsored by the Air Force Office of Scientific Research of the Office of Aerospace Research under Grant No. AFOSR 70–1863.

REFERENCES

Amarel, S. (1962a) An approach to automatic theory formation. *Principles of Self Organization* (eds Von Foerster & Zopf). London: Pergamon Press.

Amarel, S. (1962b) On the automatic formation of a computer program which represents a theory. *Self Organizing Systems* – 1962 (eds Yovits, Jacobi & Goldstein). New York: Spartan Books.

Amarel, S. (1970a) On the representation of problems and goal-directed procedures for computers. *Formal Systems and Non-Numerical Problem Solving by Computers* (eds Banerji & Mesarovic). New York: American Elsevier.

Amarel, S. (1970b) Problem solving and decision making by computer: an overview. *Cognition – A Multiple View* (ed. Garvin). New York: Spartan Books.

Birkhoff, G.S. (1948) *Lattice Theory*. New York: American Mathematical Society.

Buchanan, B.G., Sutherland, G.L., Feigenbaum, E.A. (1969) Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry. *Machine Intelligence 4*, pp. 209–54 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Copilowish, I.M. (1948) Matrix development of the calculus of relations. *J. Sym. Logic*, **13**, (4).

McCulloch, W.S. (1965) Agatha tyche: Of nervous nets – the lucky reckoners. *Embodiments of Mind*. Cambridge, Mass: MIT Press.