RELATIONAL EQUATIONS, GRAMMARS, AND PROGRAMS *)

M.H. van Emden
Research Report CS-77-17
Department of Computer Science
University of Waterloo

June, 1977

*) an abridged version of this paper appears in the proceedings of the Conference on Theoretical Computer Science held at the University of Waterloo, 15-17 August, 1977.

RELATIONAL EQUATIONS, GRAMMARS, AND PROGRAMS

Abstract The theory and some applications of equations in terms of binary relations are studied in this paper. Such an equation can be regarded as a set of axioms of first-order predicate logic such as used in logic programming, but simplified so as to suit the situation where the logic program is a monadic recursive program schema.

A proof theory, a model theory, and a "fixed-point theory" is established for certain relational equations. As a result such an equation determines a vector of relations equivalently syntactically, as minimal solution, and as minimal fixed point. The union characterization of the minimal solution allows the application of Scott's induction rule to establish a property of the minimal solution.

The properties of relational equations can be applied both to formal language theory, and to the semantics of monadic recursive program schemas, by a suitable choice of constants. In the case of formal grammars the constants are determined by the "quotient representation" of a language as a binary relation. For relational equations of type 3 we obtain a matrix-vector calculus similar to the one used for type-3 grammars.

Flowgraphs, certain programs according to the "grammar model", are described together with an operational semantics, which includes the possibility of indeterminacy, the notions of success and failure, and backtracking execution. Successful computations are related to the least solution and failed computations are related to the greatest solution of the relational equation defining the nonoperational semantics of a flowgraph. We apply computation rules for predicate transformers to proofs of correctness of flowgraphs. The semantics and proof method are applied to obtain an alternative foundation of Dijkstra's sequencing primitives.

1. Introduction

1.1 Overview and Motivation

The research reported in this paper is a continuation of [17], where three equivalent kinds of semantics are defined for predicate-logic programs: an operational semantics based on proof theory and two varieties of nonoperational semantics, one based on model theory and one on the fixed-point characterization. In [17] no attention is paid to proving properties of programs. In logic programming, a result of a procedure call is a logical implication of the procedure definitions, hence true in all models. A property of a program is typically true in the minimal model, but not in all models, and is hence not a logical implication.

It is, then, not obvious what is to be the logical basis for proving properties of logic programs. One approach is to add axioms to a logic program so that unwanted models are eliminated and the property to be proved becomes a logical implication. The axioms to be added often take the form of instances of induction schemas. Another approach is to leave the logic programs as they are and to discover rules of proof that derive statements true in the minimal model, rather than in all models.

If one takes the second approach it seems hard to avoid algebraic manipulation with explicit expressions for the minimal model. It is encouraging that the procedure definitions of logic programs have a natural interpretation as equations in a relational algebra. However, this algebra is rather complicated, which is not surprising in view of the advanced features of logic programs as compared to the toy languages usually modelled in semantics.

We therefore study a restriction of logic programs to the level of complexity of one such toy language, namely the monadic recursive program schemas of de Bakker and de Roever [3]. This restriction suggested the "grammar-modelled programs" introduced in [16]. The grammar model turned out to be interesting in its own right because (in the case of a type-3 grammar) the program is itself a set of verification conditions proving partial correctness [16], because of the suitability for systematic program development from specification to code [14], and because of the interesting relationships between grammars and program schemas in general [1, 8, 18].

Because of the intrinsic interest of grammar-modelled programs and because of the extremely modest demands they make on the rich structure of first-order predicate logic, we think that they deserve a logic of their own, with matching simplicity. This "logic" is presented here as a method of defining binary relations over an unstructured domain by means of equations involving relational product as explicit operation. We use the terminology of equations rather than the one associated with deductive theories. In this way the basic concepts are those already assimilated at the pre-university level of education. No formal system is used in deriving properties of relations defined: reasoning is informal, precise, and rigorous, which is the way mathematics has always been done. Yet those familiar with first-order predicate logic will recognize the distinction between proof theory and model theory, the distinction between truth in all models ("implication") and truth in the minimal model ("weak implication"), and the soundness and completeness (for a restricted type of equation) of the derivation mechanism.

The applications of relational equations studied here are to formal grammars and to programs. Many workers have used in one way or another the analogies between formal grammars and program schemas [1, 8, 18]. In our approach both are obtained by suitable definition of the constants in the equation.

The key concept used in the case of grammars is that of what we call the "quotient representation" of strings and languages, which we owe to Colmerauer and Kowalski [21], who used such a representation in logic programs for parsing.

For relational equations of type 3 we obtain a matrix-vector calculus similar to the well-known calculus for regular languages. Because the problem of determining an automaton recognizing the language denoted by a given regular expression is also of interest in relational equations, we apply Brzozowski's [9] notion of the derivative of a regular expression in the context of binary relations.

There are several points of interest in the application of relational equations to programs. The programs share with those of Scott [24] the suitability of being used instead of automata in the study of formal languages. They differ from Scott's in being possibly indeterminate, a property recently discovered [13,14] as useful in the systematic development of programs for practical use. The denotational semantics of a program with a type-3 schema (a "flowgraph") is characterized by either a "forward" or a "backward" relational equation. The forward equation expresses the usual verification conditions in the sense of Floyd's method of proof. The backward equation does the same but with respect to an inverse form of the usual partial correctness; both forms are expressed in terms of "(predicate) transformers". Total correctness is expressed by means of the backward transformer.

An interesting point of flowgraphs is the notion of a failed computation and the fact that the interpreter backtracks upon failure, as if searching for a successful computation. Both the least and the greatest fixpoint play a role in the nonoperational semantics of the input-output behaviour resulting from such backtracking execution.

When formalism F_1 is stated to be a special case of formalism F_2, it is often understood that F_1 is claimed to be therefore superior to F_2. That this is a misunderstanding must be clear before observing that Dijkstra's sequencing primitives [12,13] are special cases of flowgraphs. The observation is useful because it shows that the widely applicable properties of relational equations can be used to define the nonoperational semantics of these primitives; this instead of the somewhat ad-hoc semantics they were endowed with originally. The observation also suggests backtracking upon "abort" into the execution of Dijkstra's programs, thus enhancing their utility for goal-directed programming.

1.2 Related work

The present paper is justified partly by the diversity of applications derived from a single elementary concept and partly by the fact that flowgraphs are useful in the systematic construction of programs of practical significance [14,15]. Probably none of the results in this paper is new when considered in isolation. As a consequence I have not attempted to list exhaustively publications where similar results have been derived.

The work of Mazurkiewicz [22,8] and Blikle [5,6,7] is most closely related. We have been influenced by Scott's suggestions in [24] and by their elaboration by Clark and Cowell[11]. Several publications with related work have already been mentioned in the previous section.

2. Relational Equations

2.1 Equation schemas

An equation schema is an ordered triple (V,C,F) where V is a set of symbols called variables, C is a set constant symbols, and F is a set of inclusions. An inclusion is an ordered pair (t_1,t_2), where t_1 is a term, the greater term of the inclusion and where t_2 is a term, the lesser term of the inclusion. A term is a constant symbol or a variable or a product. A product is an ordered pair of terms. We will write

an inclusion (t_1,t_2) as $t_1 \supseteq t_2$ and

a product (t_1,t_2) as $t_1 \circ t_2$.

2.2 Equations

An equation schema is an entirely syntactic entity. Before we can speak of solutions we must associate with the constant symbols certain relations called constants. The means for doing this are attached to an equation schema and give an equation.

An equation E is an ordered triple (E',D,B), where E' is an equation schema (V,C,F), D is a set (the domain of E), and B (the base of E) is a subset of C x (D x D). B associates by means of the valuation function with each constant symbol of E' a binary relation over D.

A valuation is a function, denoted by "val", that associates with each term constructible from constant symbols and variables of E', a binary relation over D. The result of the valuation function depends in general on its base, a subset X of V x (D x D).

$val(X,t) = \{ (d_1,d_2): (c,(d_1,d_2)) \in B \ \& \ d_1 \in D \ \& \ d_2 \in D\}$
if the term t is the constant symbol c. Note that in this case the valuation does not depend on X.

$= \{ (d_1,d_2): (v,(d_1,d_2)) \in X \ \& \ d_1 \in D \ \& \ d_2 \in D\}$
if the term t is the variable v.

= the relational product of $val(X,t_1)$ and $val(X,t_2)$ if the term t is $t_1 \circ t_2$.

In view of the associative property of product we will regard a product as a sequence rather than a pair, i.e. we leave out any parentheses.

An inclusion $t_1 \supseteq t_2$ is satisfied by a valuation with base X iff $val(X,t_1) \supseteq val(X,t_2)$ as subsets of DxD. An equation E is satisfied by a valuation with base X iff each of its inclusions is satisfied by that valuation; X is then called a solution of E. Let S(E) be the set of solutions of E. Because each solution is a set, it makes (the usual) sense to speak of $\cap S(E)$, the intersection of solutions.

Suppose equations E_1 and E_2 have the same set of variables and the same domain. Then E_1 implies E_2 ($E_1 \models E_2$) iff $S(E_1) \subseteq S(E_2)$, and E_1 weakly implies E_2 ($E_1 \Rightarrow E_2$) iff $\cap S(E_2) \subseteq \cap S(E_1)$. Note that implication implies weak implication. E_1 and E_2 are said to be <u>equivalent</u> ($E_1 \models\!\dashv E_2$) iff $E_1 \models E_2$ and $E_2 \models E_1$. E_1 and E_2 are said to be <u>weakly equivalent</u> ($E_1 <=> E_2$) iff $E_1 => E_2$ and $E_2 => E_1$.

Let us now suppose that E_1 and E_2 differ only in their sets of inclusions F_1 and F_2. Notice that $E_1 \models E_2$ if $F_2 \subseteq F_1$. It is more interesting to have $E_1 \models E_2$ and $F_1 \subseteq F_2$. Then we would also have $E_1 \models\!\dashv E_2$. If we regard the difference between F_2 and F_1 as a function of F_1, then we say that E_2 has been <u>derived</u> from E_1. We define $E_1 \,|\!\!\vdash E_2$ if E_2 is derived from E_1 by an application of the following rule of inference:

$F_2 = F_1 \cup \{t_1 \subseteq t_2\}$ whenever $t_1 \subseteq t_2'$ and $t_3 \subseteq t_4$ are in F_1 such that t_2 is the result of replacing t_3 in t_2' by t_4

Also, $E_1 \vdash E_2$ if $E_1 = E_2$ or if there exists an E such that $E_1 \vdash E$ and $E \vdash E_2$. A variable v <u>generates with respect to E</u> a term t written ($v \overset{E}{\to} t$) iff $E \vdash E'$ where $v \supseteq t$ is an inclusion of E'.

It should be clear that $E_1 \models\!\dashv E_2$ if $E_1 \vdash E_2$, which we could express by saying that the derivation mechanism is "sound".

We will use the following notational conventions. A possibly subscripted γ will stand for a product of constants. If the product is empty it stands for the identity relation. The constant symbols ϕ and I stand for the empty relation and for the identity relation, respectively. That is, whenever they occur in an equation schema, the base of the equation is supposed to contain $\{(I,(d,d)) : d \in D\}$ and not to contain any pair with ϕ as first element.

An equation is of type 2 if its schema is of type 2, and this is so whenever the greater term in each inclusion is a single variable. An equation of type 2 is also of type 3 if its schema is (also called linear, following Blikle [5]), and this is so whenever either all lesser terms only have a variable (if at all) as the first term of a product (left-linear), or all lesser terms have a variable (if at all) as the last term of a product (right-linear).

2.3 Existence and characterization of the least solution

For a given type-2 equation $((V,C,F),D,B)$, let T be a function from subsets of $V\times(D\times D)$ to subsets of $V\times(D\times D)$ defined as follows:

$$(v,(x,y)) \in T(X) \quad \text{iff} \quad \exists\, (v \supseteq t) \in F \text{ such that } (x,y) \in val(X,t)$$

It follows immediately that T is monotone: $X_1 \subseteq X_2$ implies $T(X_1) \subseteq T(X_2)$.

$X \supseteq T(X)$ can be regarded as an equation which is equivalent with E in the following sense.

Theorem 2.1 $\qquad X \in S(E) \quad \text{iff} \quad X \supseteq T(X)$

Proof Suppose $X \in S(E)$.

$(v,(x,y)) \in T(X) \xrightarrow{\text{def. of } T} (x,y) \in val(X,t)$ where t in some $v \supseteq t$ $\xrightarrow{X \text{ is a solution}} (x,y) \in val(X,v) \xrightarrow{\text{def. of val}} (v,(x,y)) \in X.$

Suppose $X \supseteq T(X)$. We must now prove that for any $(v \supseteq t) \in F$, $val(X,v) \supseteq val(X,t)$.

$(x,y) \in val(X,t) \xrightarrow{\text{def. of } T} (v,(x,y)) \in T(X) \longrightarrow (v,(x,y)) \in X$ $\xrightarrow{\text{def. of val}} (x,y) \in val(X,v)$ □

According to the Knaster-Tarski theorem, $X \supseteq T(X)$ has a unique minimal solution which is also the unique minimal solution of $X = T(X)$. We have just seen that this solution must be $\cap S(E)$.

We define $T^o(X)$ to be X, $T^{n+1}(X)$ to be $T(T^n(X))$, and $T^*(X)$ to be $\cup_{n=o}^{\infty} T^n(X)$.

We define $L(E) = \{ (v,(x,y)) : \exists\, \gamma$ such that $(x,y) \in \gamma$ and $v \xrightarrow{E} \gamma\}$

It may be helpful to see that

$$val\ (L(E),v) = \cup\{ \gamma : v \xrightarrow{E} \gamma\}$$

that is, with $L(E)$ as basis, a valuation assigns to v the relation which is the union of the products of constants that would be generated with v as start symbol by the grammar corresponding (in the sense explained in the next chapter) to E.

It is of course easy to prove that the minimal solution $\cap S(E)$ of $X \supseteq T(X)$ is $T^*(\phi)$ by showing the so-called "continuity" of T. But we prefer to work in a more concrete manner and to show this directly. Besides, we also have to prove that $L(E) = \cap S(E)$.

Theorem 2.2
a) $L(E) \subseteq \cap S(E)$
b) $\cap S(E) \subseteq T^*(\phi)$
c) $T^*(\phi) \subseteq L(E)$

hold for a type-2 equation E.

Proof Let γ be a product of constants.

a) If $v \xrightarrow{E} \gamma$ then $v \supseteq \gamma$ is true in every solution of E and hence $val(\cap S(E),v) \supseteq \gamma$.

Suppose $(v,(x,y)) \in L(E)$. By the definition of L, there exists a γ such that $(x,y) \in \gamma$ and $v \xrightarrow{E} \gamma$. Hence $(x,y) \in val(\cap S(E),v)$ which is $(v,(x,y)) \in \cap S(E)$. The conclusion $L(E) \subseteq \cap S(E)$ can be understood as the "soundness" of the derivation mechanism determining L.

b) To show that $\cap S(E) \subseteq T^*(\phi)$ we show that $T^*(\phi)$ is closed under T, because then (Theorem 2.1) $T^*(\phi) \in S(E)$.

Suppose that $(v,(x,y)) \in T(T^*(\phi))$. By the definition of T, there exists a $(v \supseteq t) \in F$ such that $(x,y) \in val(T^*(\phi),t)$. As t is a finite product of constants or variables, there must exist an N such that $(x,y) \in val(T^N(\phi),t)$. But then $(v,(x,y)) \in T^{N+1}(\phi) \subseteq T^*(\phi)$.

c) Let us prove that $T^n(\phi) \subseteq L(E)$ for $n \geq 0$. By the definition of L we have to prove that $(x,y) \in val(T^n(\phi),v)$ implies that $(x,y) \in \gamma$ for some γ such that $v \xrightarrow{E} \gamma$. We proceed by induction on n.

Suppose $(x,y) \in val(T^{n+1}(\phi),v)$. We must now find a γ such that $v \xrightarrow{E} \gamma$ and $(x,y) \in \gamma$. By the definition of T there exists a $v \supseteq t$ such $(x,y) \in val(T^n(\phi),t)$. It t contains no variables, it is the γ we need.

Otherwise γ can be found by eliminating the variables in t. Let v' be one such; $(x,y) \in val(T^n(\phi),t)$ implies some $(x',y') \in val(T^n(\phi),v')$. By the induction hypothesis there must be a γ' such that $(x',y') \in \gamma'$ and $v' \xrightarrow{E} \gamma'$ in at most n steps. We conclude that $(x,y) \in val(T^{n+1}(\phi),t')$, with $v \xrightarrow{E} t'$ in at most $n+1$ steps, where t' is the result of replacing v' by γ' in t. All variables in t can be replaced simultaneously in this way. To verify the basis of the induction, observe that by the definition of T, $(x,y) \in val(T(\phi),v)$ iff $(x,y) \in val(\phi,t)$ for some $v \supseteq t$; this can only be the case if t is a product of constants only.

The conclusion $T^*(\phi) \subseteq L(E)$ can be understood as the completeness of the derivation mechanism: whenever $(v,(x,y))$ is the minimal solution of E, this can be derived by a suitable $v \xrightarrow{E} \gamma$ □

According to theorem 2.1 solutions X of relational equations are characterized by $X \supseteq T(X)$, of which only the least and not the greatest solution is of interest. We will, however, make use of the greatest solution of $X \subseteq T(X)$, which can be proved to be equal to $\bigcap_{n=o}^{\infty} T^n(Vx(DxD))$ in a similar way to the proof of theorem 2.2. Note that we do not consider any form of relational equation of which the solutions X are characterized by $X \subseteq T(X)$.

3. Application to Languages

3.1 The quotient representation

Relational equations look very much like formal grammars: constants are like terminals, variables like nonterminals, terms like strings, inclusions like productions. This similarity suggests the following definition. A grammar G is an associated grammar of an equation schema $E = (V,C,F)$ if G has C as set of terminals, V as set of nonterminals, any start symbol of V, and a set of productions containing $s_1 \rightarrow s_2$ for every inclusion $t_1 \supseteq t_2$ in F, where

$s_i (i=1,2)$ is the string of symbols of t_i in the order as they appear in t_i. Thus, for every choice of start symbol there is an associated grammar.

We shall use type-2 relational equations to characterize context-free languages as least solutions. A relational equation is widely applicable because, independently of its schema, the domain may be chosen to give any binary relations over the domain as values to the constant symbols. However, for a given schema (V,C,F) there is one equation of special interest, namely the <u>grammar equation</u> of the schema:

$$((V,C,F),C^*,\{(c,(cs,s)):\ c \in C\ \&\ s \in C^*\})$$

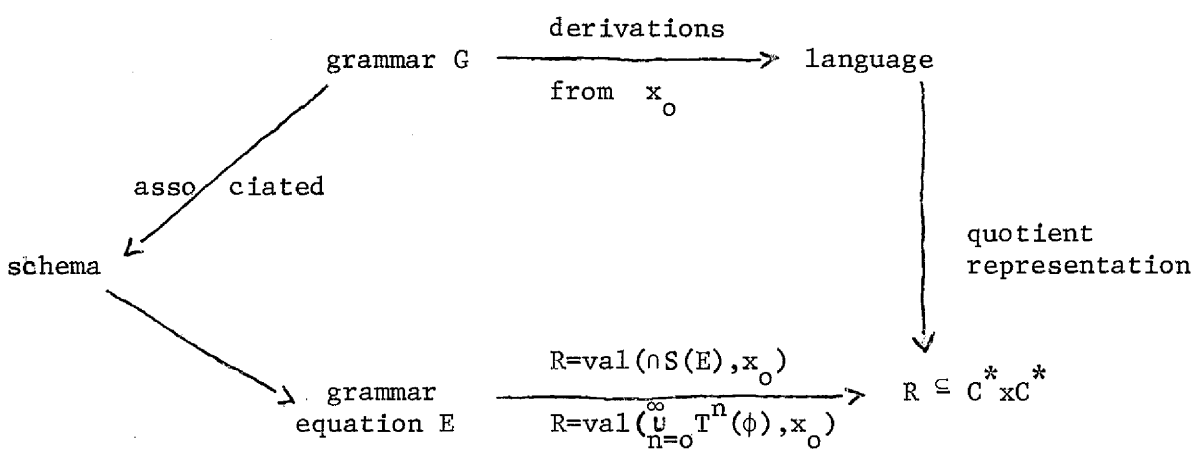
where the domain is the set of strings of constants and where the base is chosen to be such that $val(\phi,c) = \{(cs,s):\ s \in C^*\}$. This construction, where meaning is determined by syntactic objects only, is familiar in logic as the "Herbrand interpretation".

The reason for this choice of domain and base is the utility of the "quotient representation" of strings and languages as binary relations over strings. Suppose string t is the catenation t_1t_2. Then t_1 is the left-quotient of t with respect to t_2 and may be represented by the set of pairs $\{(t_1t_2,t_2):\ t_2 \in C^*\}$; this set, a binary relation over C^*, is the <u>quotient representation</u> of t_1.

It should be clear that the relational product of $\{(ss_1,s_1):\ s_1 \in C^*\}$ and $\{(tt_1,t_1):\ t_1 \in C^*\}$, the quotient representations of s and t, is $\{(stt_1,t_1):\ t_1 \in C^*\}$ which is the quotient representation of the catenation st. Because $val(B,t_1 \circ t_2)$ is the relational product of $val(B,t_1)$ and $val(B,t_2)$, we see that $val(B,c_1 \circ \ldots \circ c_n) = \{(c_1 \ldots c_n s,s):\ s \in C^*\}$: for a grammar equation the value of a product of constants is the quotient representation of the string of constant symbols. The quotient representation of a language $L \subseteq C^*$ is $\{(st,t):\ s \in L\ \&\ t \in C^*\}$.

Now theorem 2.2 can be used to give a nonoperational ("mathematical", "denotational") characterization of a context-free language, and to show it equivalent to the usual operational characterization. The operational characterization of a language generated by a grammar G is as the set of strings of terminal symbols such that a derivation exists from the start symbol x_o to s.

Let $R \subseteq C^* x C^*$ be the quotient representation of the language. Every derivation of the grammar can be simulated by an inference from E, hence $R = val(L(E),x_o)$. It follows that the nonoperational characterization $val(\cap S(E),x_o)$ also equals R. The situation can be shown in the following diagram, of which the commutativity expresses a theorem of Ginsburg and Rice [20].



Anything that is true of the minimal solution of a relational equation with type-2 schema S, for any domain and for any base, also applies to a grammar associated with the equation schema, because the grammar equation of S is obtained by a special choice of domain and base. More interestingly, we can proceed from the particular to the general (at least so in appearance) by means of what we shall call the <u>correspondence principle</u>: if something is proved of a formal language by reasoning about sets of strings and derivations, then the same can be proved of $val(L(E),x_o)$ by parallel reasoning about unions of products of constants and inferences, and the result is true independently of the choice of domain and base.

Of course, the solutions about which such things are proved are unions of products of constants, and hence limited by the vocabulary of the constant symbols. This need not be restrictive: we can make the constants as "small" as we like. Suppose we are interested in expressing as minimal solution binary relations over a given domain D. Then we can take as set of constant symbols a set DD containing a name for each single pair of DxD. In an equation ((V,DD,F),D,B) the base B can contain $(dd,(d_1,d_2))$ whenever $dd \in DD$ is the name for $(d_1,d_2) \in DxD$.

3.2 Solving and unsolving right-linear equations

To solve an equation usually means to find its solutions. In the case of a relational equation of type 2, solving will mean finding the minimal solution, which is a binary relation, and which we must somehow denote by an expression possibly involving several relational operations, rather than just the product, which is the only one used up till now. It should not be taken for granted that such an expression (an explicit form) is more useful than the equation itself which may be regarded as an implicit way of denoting a relation. In some situations an equation is more useful than an explicit expression, so that "unsolving", a process inverse to solving, is called for.

A type-3 relational equation in the form $X \supseteq T(X)$ can usefully be interpreted in terms of vectors and a matrix, with formally the same result as for regular languages.

An $X \subseteq Vx(DxD)$ can be regarded as a vector of binary relations indexed by the variables $v_1,v_2,\ldots$ of V: the i-th component of X is $X_i = val(X,v_i)$. Apparently, the transformation T associated with an equation E can be regarded as a vector transformation. And if E is right-linear, T can be regarded as a matrix of binary relations, as will now be explained. Let $v_i \supseteq \gamma'_{ij}v_j,\ldots,v_i \supseteq \gamma'''_{ij}v_j$ be all inclusions beginning with v_i and ending with v_j, and let $\gamma_{ij} = \gamma'_{ij} \cup \ldots \cup \gamma'''_{ij}$. Let $v_i \supseteq \beta'_i,\ldots,v_i \supseteq \beta'''_i$ be all inclusions beginning with v_i and not ending with a variable, and let $\beta_i = \beta_i' \cup \ldots \cup \beta_i'''$.

Of the i-th component of $T(X)$ we know that, by the definition of T,

$$val(T(X),v_i) \supseteq val(X,\gamma_{ij}'' v_j) \quad \text{for each inclusion} \quad v_i \supseteq \gamma_{ij}'' v_j, \text{ and}$$

$$val(T(X),v_i) \supseteq val(X,\beta_i'') \quad \text{for each inclusion} \quad v_i \supseteq \beta_i'' , \text{ hence}$$

$$val(T(X),v_i) \supseteq val(X,\gamma_{ij}' \circ v_j) \cup \ldots \cup val(X,\gamma_{ij}''' \circ v_j) \cup \beta_i' \cup \ldots \cup \beta_i'''$$

$$val(T(X),v_i) \supseteq \gamma_{ij} \circ val(X,v_j) \cup \beta_i$$

A right-linear equation $X \supseteq T(X)$ can apparently be written in terms of vectors and a matrix as $X \supseteq AX \cup B$ if in the usual matrix or vector operations addition is replaced by union and multiplication by relational product, if the (i,j)-element of the matrix A is γ_{ij}, and if the i-element of B is β_i.

It should be clear that the matrix form of $T^n(X)$ is $A^n X \cup (A^{n-1} \cup \ldots \cup A \cup I)B$ and that $T^*(\phi)$, the least solution of $X \supseteq T(X)$, is A^*B (see [9] for the corresponding result for regular languages). The result of solving a right-linear equation is an explicit expression for $val(T^*(\phi),v_i)$, the i-th component of the least solution $T^*(\phi)$. The expression is the i-th component of A^*B, in general an infinite expression in the γ_{ij}'s and the β_i's. An application, via the above-mentioned correspondence principle, of Kleene's theorem on the "representability of regular events", says that this expression can always be written in finite form, using product, union and star. This completes our remarks on solving relational equations of type 3.

For regular languages the method of derivatives [9] is a process inverse to solving an equation: given an expression for a regular language, the method finds a finite automaton that recognizes the language; such an automaton is just another form for a grammar generating the language, and such a grammar is nothing but a relational equation in disguise. Because regular expressions have meaning (the same, except that catenation for languages is product for relations) in terms of binary relations, the method of derivatives also applies in the context of relational equations: given a regular relational expression, the method finds an equation that has as minimal solution the relation denoted by the expression.

The key notion of Brzozowski's method is that of a derivative of a regular expression. The method can be applied without interpreting derivatives in terms of relations: product of constants are treated as strings of (constant) symbols, and the usual definition of derivative can be applied to such strings. When the constants are the quotient-representation of single-symbol strings, we can also give a relational interpretation of the derivative. Let L be a relation representing a language L' and γ a relation representing a string γ', both over an alphabet α. Let P be the binary relation $\{(t_1,t_2):(\exists t.\ t_1=tt_2)$ and $t_1, t_2 \in \alpha^*\}$, that is, P is the quotient representation of α^*. Then the derivative of L' with respect to γ' is a set of strings of constant symbols of which the products have as union $(\gamma^{-1}\circ L)\cap P$, where $(x,y)\in\gamma^{-1}$ iff $(y,x)\in\gamma$. It is easy to see that $\gamma^{-1}\circ L$ would already be the derivative, were it not for the fact that L' may contain prefixes of γ' shorter than γ'. In that case $\gamma^{-1}\circ L$ contains products of inverses of constants (strings of negative length, or even "anti-strings" in some analogy to anti-matter) and these are not the denotation of any string in the quotient-representation. Hence the need for intersection with P to eliminate such anomalies.

4. Application to Programs

4.1 Grammar-modelled programs

Predicate-logic programs, especially when modelling sequential state-transition processes, look very much like formal grammars. This observation inspired the definition in [16] of grammar-modelled programs.

A grammar-modelled program consists of two parts: a program schema, which is a grammar, and a machine, which has a set of states and a set of commands and these are binary relations over the states. The relations specify the input-output behaviour of the commands. The program also associates with each terminal symbol of its schema a command of its machine. The strings generated by the grammar play the role of computations and are then associated with products of commands which are binary relations between (input) states and (output) states.

More precisely, a <u>program schema</u> is a formal grammar (N,T,P,S) with N a set of nonterminals, T a set of terminals, P a set of productions, and S a start symbol. A <u>program</u> is a pair (PS,(D,B)) with PS a program schema and (D,B) a <u>machine</u>; D is the <u>memory set</u> (with elements called <u>states</u>) of the machine and B is the <u>command definition</u> of the machine. B is a subset of Tx(DxD), where T is the set of terminals of PS. The command associated by the program with the terminal symbol t is the binary relation

$$val(B,t) = \{(d_1,d_2): \ (t,(d_1,d_2)) \in B\}$$

For a given state x_o, a <u>computation</u> of a program is a sequence of pairs $(t_1,x_1),\ldots,(t_n,x_n)$ where $t_1\ldots t_n$ is a string generated by the program schema of the program and where $x_1,\ldots,x_n$ are states such that $(x_{i-1},x_i) \in val(B,t_i)$, for $i=1,\ldots,n$. The <u>start state</u> of the computation is x_o; its <u>halt state</u> is x_n. An <u>interpreter</u> for a program is a procedure for constructing a computation for any start state where one exists.

<u>Example 4.1</u> Program schema =

(nonterminals: { S, P}

,terminals: { a,b,c,d,e,f}

,productions: { S → aP, P → bP, P → cdP, P → ef}

,start symbol: S

)

The set of states of the machine is $\{(u,v,w)\} \cup \{(u,v)\} \cup \{(w)\}$ where u,v,w range over the rationals, and may be thought of as registers. In different states different sets of registers may be in use. The command definition B is such that

val(B,a) = { ((u,v) , (u,v,1))}

(real w:=1 in Algol notation)

val(B,b) = { ((u,v,w), (u,v-1,uxw))}

(v:=v-1;w:=uxw)

val(B,c) = { ((u,v,w),(u,v,w)) : v is an even integer}

(if even(v) then else 1:goto 1)

val(B,d) = { ((u,v,w),(uxu,v/2,w))}

(u:=uxu; v:=v/2)

val(B,e) = { ((u,0,w),(u,0,w))}

(if v=0 then else 1: goto 1)

val(B,f) = { ((u,v,w),(w))}

(deallocate u,v)

i	t_i	x_i	t_i	x_i	t_i	x_i
0		(2,10)		(2,10)		(2,10)
1	a	(2,10,1)	a	(2,10,1)	a	(2,10,1)
2	d	(4,5,1)	d	(4,5,1)	d	(4,5,1)
3	b	(4,4,4)	b	(4,4,4)	b	(4,4,4)
4	d	(16,2,4)	d	(16,2,4)	d	(16,2,4)
5	d	(256,1,4)	b	(16,1,64)	b	(16,1,64)
6	b	(256,0,1024)	b	(16,0,1024)	b	(16,0,1024)
7	e	(256,0,1024)	e	(16,0,1024)	b	$(16,-1,10^{-14})$
8	f	(1024)	f	(1024)	b	$(16,-2,10^{-18})$
					⋮	⋮
		Sequence I		Sequence II		Sequence III

Sequences I and II are finite computations; sequence III can not be completed to one □

Example 4.2: An Alphabet Machine

Let the program schema be the same as in the previous example. Let the set of states of the machine be the set T^* of strings over the alphabet $T = \{a,b,c,d,e,f\}$. Let the command definition $B = \{(t,(tx,x)): t \in T \ \& \ x \in T^*\}$. With this machine the program is the grammar-equation of the last chapter.

Intuitively, the information environment of the machine consists only of an input tape represented by the string which is the state: there are no internal registers and no output tape. The command associated with a $t \in T$ is defined only if the input tape starts with t and the action caused by the command is that the tape is advanced. It may be verified that a pair (x,y) of states is in the input-output relation computed by the program iff $x = \sigma y$ where σ is a string produced by the program schema. This fact is no more than a curiosity for programs with a schema of type≠3, because the accepting algorithm does nothing but comparing the input string with successively generated strings of the language, as performed in an unspecified manner by the interpreter. It seems reasonable to require that the interpreter needs only a finite amount of of memory, so that we consider programs with a schema of type 3, as in the following example.

Example 4.3: A Pushdown-Store Machine

We consider the problem of accepting with a program $((N',T',P',S'), (D,B))$ a language generated by a grammar $G = (N,T,P,S)$, where P' is of type 3 and P is of type 2 with the restriction that no terminal follows a nonterminal in a right-hand side of a production. We describe the accepting program in terms of the grammar G;

$N' = \{S', Q, R, F, H\}$

$T' = T \cup \{push(x): x \in N\} \cup \{pop(x): x \in N\}$

$\cup \{$stack empty, tape empty, tape nonempty$\}$

We assume that T does not already contain any of the symbols added here.

$P' = \{S' \to push(S)\ Q,\ Q \to$ stack empty R

, R $\to$ tape nonempty F, R $\to$ tape empty H

$\} \cup \{ Q \to pop(n)\ t_1 \ldots t_j\ push(n_1) \ldots push(n_k)\ Q$

$: n \to t_1 \ldots t_j\ n_1 \ldots n_k \in P$ with the t's terminals

in T and the n's nonterminals in N

$\}$

$D = N^* \times T^*$, i.e. the memory set consists of all pairs of a string of nonterminals (the "stack") and a string of terminals (the "input tape").

B is such that:

$(x,y) \in val(B, push(n))$ iff y is the state resulting from pushing $n \in N$ on the stack in state x;

$(x,y) \in val(B, pop(n))$ iff y is the state resulting from popping the stack in a state x where $n \in N$ is the top of a nonempty stack;

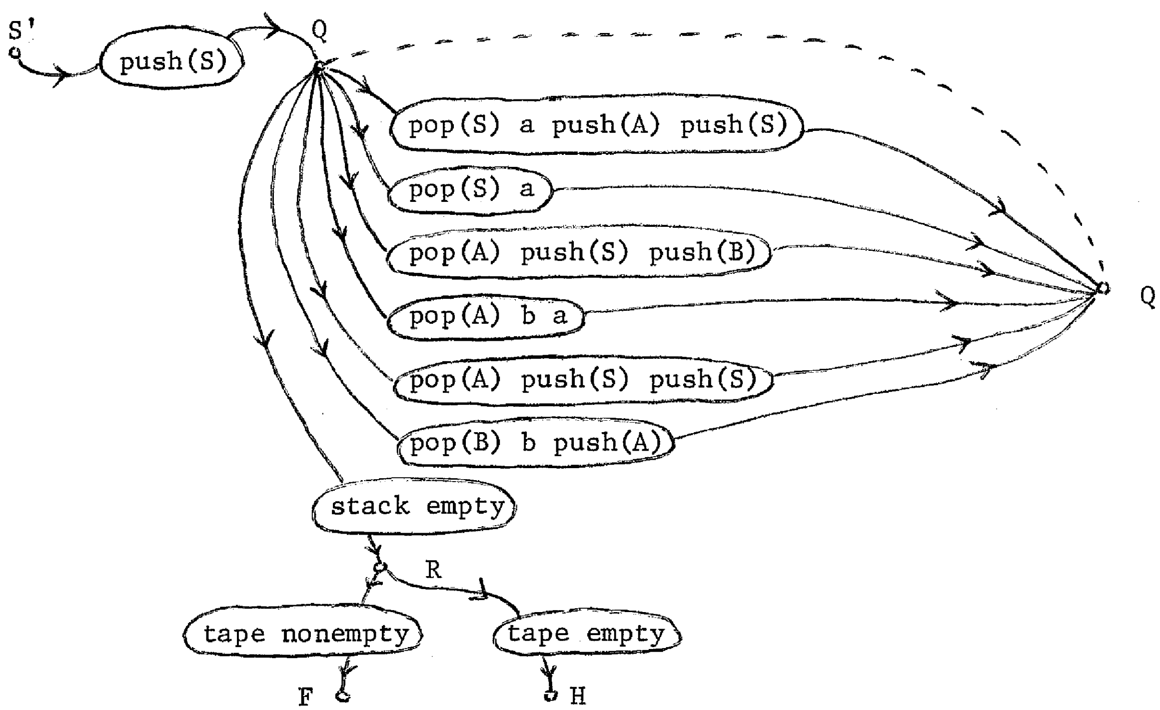
$(x,y) \in val(B, t)$ iff y is the result of advancing the tape one symbol in a state x where $t \in T$ is the first symbol on the tape;

$(x,y) \in val(B,$ tape empty$)$ iff the tape is empty in state x and x=y; similarly for "stack empty" and "tape nonempty".

For example, the program with pushdown store machine accepting the language generated by

```
G = ( { S,A,B}
    , { a,b}
    , { S → aAS,  S → a
      ,A → SB,  A → ba
      ,A → SS,  B → bA
      }
    , S
    )
```

is, in graph notation (see Section 4.2):



Example 4.4

Let D be the set of triples (x,y,z), where x is a left-infinite and z is a right-infinite string on the alphabet $\Sigma = \{0,1,\text{blank}\}$; $y \in \Sigma$. Let B equal

$$\begin{aligned}&\{(L,((x,y,uz),(xy,u,z)))\}\cup\\&\{(R,((xu,y,z),(x,u,yz)))\}\cup\\ \cup_{i=0,1,\text{blank}}&\{(P_i,((x,y,z),(x,i,z)))\}\cup\\ \cup_{i=0,1,\text{blank}}&\{(i,((x,i,z),(x,i,z)))\}\end{aligned}$$

where x and z range over Σ^* and u and y over Σ.

This machine is Turing's own [25]; together with any program schema it makes up a grammar-modelled program. Any terminal symbol of the schema not L,R,Pi, or i is assigned by B the totally undefined command. As argued by Scott [24], it is advantageous to separate programs from machines, so that instead of having different "Turing machines" to compute different sets of sequences, there are different programs for the same machine, such as the one described in this example □

4.2 Operational semantics of flowgraphs

The operational semantics of programs has in principle been defined already by means of the finite computations. However, this is not satisfactory because it has not been specified how the interpreter constructs the strings generated by the program schema. We can be more specific about this when the schema is of type 3: in a right-hand side of a production a nonterminal can only occur as last symbol.

For the purposes of grammar-modelled programs, type-3 grammars have an unpleasant lack of symmetry: there is a start symbol, but not a halt symbol.

We shall assume that our type-3 program schemas are modified as follows: all right-hand sides of productions have to end in a nonterminal. Where there was none, we add one, say H, which is different from the previously present nonterminals, and call it the halt symbol. Moreover, without loss of generality we can assume that the start symbol does not occur in a right hand side. It is clear that the halt symbol cannot be a left-hand side. The language defined by a grammar G is now the set of strings t of terminals such that $S \overset{G}{\rightarrow} tH$ where S is the start symbol and H the halt symbol.

For type-3 schemas the strings can be defined by means of paths though a labelled directed graph, as follows. The graph contains a node for every nonterminal, and an arc from N_1 to N_2 labelled with a string of terminals t for every production $N_1 \rightarrow tN_2$. Because of this representation, we will refer to programs with a type-3 schema as a flowgraph: something which is much like the traditional flowchart, but truly a graph because only one kind of node exists. Computations now correspond to paths from S to H through the graph, and they are represented by the labels of the arcs traversed. It will be more convenient to consider instead the sequences of nodes. Therefore we define a path as a sequence

$$\ldots,(N_i,x_i),\ldots \qquad i = \ldots,-2,-1,0,1,2,\ldots$$

where (N_{i+1},x_{i+1}) is a successor of (N_i,x_i) for all i in the sequence. (N',x') is a successor of (N,x) iff there is an arc labelled t from N to N' and if $(x,x') \in val(B,t_1\circ\ldots\circ t_n)$ where $t_1,\ldots,t_n$ are the symbols (in that order) of t. It is understood that a pair can only be the last (first) of a path if it has no successor (is not the successor of any pair).

Note that if S occurs in a (node,state)-pair, then the sequence must have a first pair, and is called a forward path. If H occurs, then the sequence must have a last pair, and is called a <u>backward path</u>. The operational semantics of a program with type-3 schema can now be equivalently, and more specifically, be defined as

$$\{ (x,y) : \text{ there exists a path } (S,x),\ldots,(H,y)\}$$

which is the <u>input-output relation</u> computed by the program.

A forward path which is also a backward path is a finite path. There may exist finite forward paths which are not backward; we will call these <u>failed</u>: reaching H is thought of as fulfilling the program's goal, so that a forward and backward computation is called <u>successful</u>. Note that only successful paths correspond to computations. Suppose

$$(S,x_o),\ldots,(P,x_n)$$

is a failed path. If the program is indeterminate then it may happen that a successful path

$$(S,x_o),\ldots,(H,x_m)$$

also exists. Another way of viewing an interpreter is as a procedure for constructing a successful path whenever one exists for a given start state.

Consider the tree with (S,x_o) as root and (N',x') a descendant of (N,x) iff (N',x') is a successor of (N,x). Forward paths are paths in this tree from the root. An interpreter may be viewed as a procedure searching for a pair with the halt symbol. One of the several algorithms the interpreter can use is depth-first search with backtracking upon failure, that is, encountering a terminal node of the tree without halt symbol. Such an algorithm is commonly employed in interpreters for indeterminate programs. Note that backtracking over an input command implies regurgitating input already ingested. This is essential in the programs in examples 4.2 and 4.3.

Example 4.5

Suppose we want to find a way of paying an amount of n cents when n_i coins of denomination i cents are available, i = 10,5,1. The following program ((N,T,P,S),(D,B)) performs the required computation.

$N = \{S,Q,H\}$

$T = \{(p_{10},p_5,p_1:=0,0,0),(n=0)\} \cup$
$\quad \{(n,n_i,p_i:=n-i,n_i-1,p_i+1:\ i=10,5,\text{or } 1\}$

$P = \{S \rightarrow (p_{10},p_5,p_1:=0,0,0)\ Q$
$\quad ,Q \rightarrow (n=0)\ H$
$\quad \} \cup \{Q \rightarrow (n,n_i,p_i:=n-i,n_i-1,p_i+1)\ Q:\ i=10,5,\text{ or } 1\}$

D = the set of 7-tuples of nonnegative integers, representing the values of the variables $n,n_1,n_2,n_3,p_1,p_2,p_3$

B is such that each of the nonterminals is assigned the relation as usual in programming. Note that only nonnegative integers exist, so any assignment that would lead to a negative integer as component of a tuple is undefined.

4.3 Floyd's proof method for flowgraphs

Suppose p and q are subsets of D. Such sets will be, improperly, called assertions, although this term will also be used properly for the statement that a state belongs to such a set. A flowgraph is partially correct with respect to assertions p and q if for each terminated path with start state in p, the final state is in q. Note that $x \in p$ does not imply that there exists a terminated path with x as start state; it is this deficiency that the "partial" in "partial correctness" refers to.

If R is the input-output relation computed by flowgraph p, then its partial correctness with respect to p and q can be expressed as an inclusion among binary relations:

$$p' \circ R \subseteq R \circ q'$$

where p' is the partial identity containing just those pairs (x,x) such that $x \in p$, and similarly for q. A traditional notation (convenient when R is not very short) for the partial correctness of p is

$$\{p\}\ P\ \{q\}$$

The purpose of the method of Floyd[19] is to prove partial correctness for a program written as a flowdiagram. The method applies to flowgraphs as well, as will now be explained [10]. Let S be the start node and H the halt node of a flowgraph. According to the method, there is associated with each node an assertion which is denoted here by the same symbol as the associated node; the context should make it clear which type of object meant. The assertions are said to <u>verify</u> the flowgraph if for each arc the <u>verification condition</u>

$$L_1 \circ C \subseteq C \circ L_2$$

holds; $L_1 (L_2)$ is the assertion associated with the initial (final) node of the arc, and C is the command labelling the arc.

The premiss of Floyd's rule of proof is that the flowgraph be verified. The conclusion is its partial correctness with respect to any assertions $p \subseteq S$ and $q \supseteq H$:

$$p' \circ R \subseteq R \circ q'$$

where R is the input-output relation computed by the flowgraph.

<u>Theorem 4.1</u> If in a path there is a (node,state) pair (L,x) such $x \in L$, where L is the associated assertion, then the same holds for all subsequent pairs in the path.

<u>Proof</u> Let (L_{i-1},x_{i-1}), (L_i,x_i) be two successive pairs in the path. By the definition of a path, $(x_{i-1},x_i) \in C_i$, the command labelling an arc from L_{i-1} to L_i in the flowgraph. By the supposition that the assertions verify the flowgraph, $L_{i-1};C_i \subseteq C_i;L_i$. Suppose now that $x_{i-1} \in L_{i-1}$, then $x_i \in L_i$. Apparently, if in a path of a verified flowgraph $x \in L$, then the same holds for all subsequent pairs. It was assumed that $x_o \in S$, the assertion associated with the node in the first pair of a path □

It is easy to see that Floyd's rule of proof is justified by the special case of this theorem for finite paths. It should be noted that Floyd's method may also be usefully applied to algorithms that do not terminate (operating systems, or programs controlling telephone exchanges may be designed never to terminate).

P may have the property that whenever a backward path ends in a state in q and the backward path also has a beginning, then the start state must be in p' in relational notation: $p' \circ R \supseteq R \circ q'$. The above remarks on partial correctness and theorem 4.1 apply, with obvious modifications, to this backward analog of partial correctness, which has been used in connection with the method of subgoal induction of Morris and Wegbreit [23].

4.4 <u>Nonoperational semantics</u>

4.4.1 <u>Forward and backward equations</u>

The nonoperational semantics is defined by means of the least solution of either of two relational equations associated with the program. One is called the forward equation, the other the backward equation.

Let $((N,T,P,S),(D,B))$ be a program. The <u>forward equation</u> associated with the program is $((N,T,F_1),D,B)$ where F_1 contains the inclusion $n_2 \supseteq n_1 \circ c_1 \circ \ldots \circ c_k$ if P contains the production $n_1 \to c_1 \ldots c_k n_2$. To the inclusions there is added $S \supseteq I$, where I is the identity.

The <u>backward equation</u> associated with the program is $((N,T,F_2),D,B)$ where F_2 contains the inclusion $n_1 \supseteq c_1 \circ \ldots \circ c_k \circ n_2$ if P contains the production $n_1 \to c_1 \ldots c_k n_2$. To the inclusions there is added $H \supseteq I$, where I is the identity relation.

The forward and backward equations correspond to the two different ways de Bakker [2] gives for expressing a given flow diagram as an equivalent set of procedure declarations.

<u>Example 4.6</u>

After adding the halt symbol H, the program schema of example 4.1 becomes

(nonterminals: $\{S,P,H\}$
, terminals: $\{a,b,c,d,e,f\}$
, productions: $\{S \to aP,\ P \to bP,\ P \to cdP,\ P \to efH\}$
, start symbol: S
)

The associated forward equation has as set of inclusions

$$\{S \supseteq I,\ P \supseteq S \circ a,\ P \supseteq P \circ b,\ P \supseteq P \circ c \circ d,\ H \supseteq P \circ e \circ f\}$$

The associated backward equation has as set of inclusions

$$\{S \supseteq a \circ P,\ P \supseteq b \circ P,\ P \supseteq c \circ dP,\ P \supseteq e \circ f \circ H,\ H \supseteq I\}$$

□

<u>Theorem 4.2a</u> Let P be a program with forward equation E. Then $(x,y) \in \mathrm{val}(\cap S(E),t)$ iff $(n_o,x),\ldots,(t,y),\ldots$ is a path of P, where n_o is the start symbol.

<u>Proof</u> Suppose $(x,y) \in val(\cap S(E),t)$. Then $(x,y) \in val(L(E),t)$ and there must exist products of constants γ_i such that $E \vdash t \supseteq \gamma_1 \circ \ldots \circ \gamma_k$ such that for $i=1,\ldots,k$ $n_i \supseteq n_{i-1} \circ \gamma_i$ is an inclusion of E. The existence of the path follows. The converse of this reasoning proves the converse □

Note that the path need not be finite. Apparently the forward equation characterizes by its least solution the paths that have a beginning but not necessarily an end. In case t is the halt symbol H the path is finite; the theorem states that $val(\cap S(E),H)$ is the input-output relation as defined according to the operational semantics.

<u>Theorem 4.2b</u> Let P be a program with backward equation E. Then

$(x,y) \in val(\cap S(E),t)$ iff

$\ldots,(t,x),\ldots,(H,y)$

is a path of P.

Note that the path need not be finite. Apparently the backward equation characterizes by its least solution the paths that have an end but not necessarily a beginning. In case t is the start symbol, the path is finite; the theorem states that $val(\cap S(E),S)$ is the input-output relation as defined according to the operational semantics.

Let E_1 be the forward equation of a program P and E_2 its backward equation. By "coupling" E_1 and E_2 into a single equation E we can characterize the computations that have both a beginning and an end. E contains as inclusions those of E_1 and those of E_2, where we suppose that the variables from $E_1(E_2)$ are distinguished by an extra subscript 1 (2).

Let n be a nonterminal from P.

$(x,y) \in val(\cap S(E), n_1)$ implies (Theorem 4.2a) that (n,y) is in a forward path. $(y,z) \in val(\cap S(E), n_2)$ implies (Theorem 4.2b) that (n,y) is in a backward path. Hence $val(\cap S(E), n_1 \circ n_2)$ is the contribution to the input-output relation by the finite paths through n. It follows (de Bakker [2]) that the input-output relation computed by P is $\cup_n val(\cap S(E), n_1 \circ n_2)$. Let s be the start symbol of P, and h its halt symbol. Then we see that $val(\cap S(E), h_1 \circ s_2^{-1})$ has as domain the domain of convergence of P.

4.4.2 The significance of greatest solutions

Let T be the transformation associated with the forward equation of a program $((N,T,P,S),(D,B))$. T maps the power-set of $N \times (D \times D)$ to itself. The forward equation can be written as $x \supseteq T(x)$. We saw that $(H,(u,v))$ is in the least solution of the forward equation iff there exists a (successful) path $(S,u),\ldots,(H,v)$. The greatest solution of $x \supseteq T(x)$ is of no interest. However, the least solution of $x \supseteq T(x)$ happens to be a solution (in fact, the least) of $x = T(x)$ and this equation has an interesting greatest solution, which happens to be the greatest solution of $x \subseteq T(x)$.

Theorem 4.3 If all paths starting with (S,x_o) are failed then for no y is (x_o,y) in the greatest solution of $x \subseteq T(x)$.

Proof We recall that the forward paths can be regarded as paths from the root in a "path tree", where (S,x_o) is the root and where (N',x') is a descendant of (N,x) iff (N',x') is a successor of (N,x). Theorem 2.2 suggests that $\overset{\infty}{\underset{n}{\cap}} T^k(N\times(D\times D))$ is the greatest solution of $x \subseteq T(x)$, which may be proved in a similar way.

We will prove by induction on k that for all y, $(S,(x_o,y))$ is not in $T^k(N\times(D\times D)$ if (S,x_o) is the root of a path tree containing failed paths only and having a length $\leq k$.

For the basis of the induction we assume that the path tree consists of the root only, i.e., (S,x_o) has no successor, so that for any inclusion $S \supseteq \gamma_i \circ V_i$ with S as greater term, x_o is not in the domain of γ_i. Hence $(S,(x_o,y)) \in T(Nx(DxD))$ for no y.

For the induction step, suppose that (S,x_o) is the root of a failed search tree of depth $k+1$. For any successor of the root, say (V_i,x_1^i) there must exist an inclusion $S \supseteq \gamma_i \circ V_i$ such that $(x_o,x_1^i) \in \gamma_i$. (V_i,x_1^i) is the root of a failed search tree of depth k; by the induction hypothesis $(V_i,(x_1^i,y))$ is not in $T^k(Nx(DxD))$ for any y. For $(S,(x_o,y))$ to be in $T^{k+1}(Nx(DxD))$ for some y, it is necessary that there exist an inclusion $S \supseteq \gamma_i \circ V_i$ with $(x_o,x_1^i) \in \gamma_i$ and $(V,(x_1^i,y))$ in $T^k(Nx(DxD))$; this has just been found to be impossible. Hence $(S,(x_o,y))$ not in $T^{k+1}(Nx(DxD))$ for any y, which completes the induction step □

To summarize the operational significance of least and greatest solutions, let the domain of $val(u_o,H)$ be equal to A when u_o is the least solution of $u \supseteq T(u)$ and equal to B when u_o is the greatest solution $u \subseteq T(u)$. For $x \in A$, (S,x) begins at least one successful path and possibly also failed and infinite paths. For $x \in B$ and $x \notin A$, (S,x) begins no successful path, at least one infinite path, and possibly failed paths. If (S,x) begins failed paths only, x must be in the complement of B.

4.5 Equations and verification conditions

Correctness will be expressed by means of properties of states; a subset p of D will be thought of as the set of states having a particular property. As before, a subset R of DxD will be thought of as the input-output relation of a program or a command.

$p \rightarrow R$ is defined as the subset of D containing y iff there exists an x such that x in p and (x,y) in R. Read in $p \rightarrow R$ "$\rightarrow$", the forward transformer, as a function symbol in infix notation with p and R as arguments. Note that $D \rightarrow R$ is the range of R.

Similarly, $x \in (R \leftarrow p)$ iff there exists a y such that $(x,y) \in R$ and $y \in p$. The function symbol "$\leftarrow$" is the <u>backward transformer</u>. $R \leftarrow D$ is the domain of R. These transformers are due to de Bakker and de Roever [3] who wrote $\varepsilon_R(P)$ for $p \rightarrow R$ and $R \circ p$ for $R \leftarrow p$. Note that the backward transformer only coincides with Dijkstra's weakest precondition [13] for determinate programs. For an indeterminate program, R may be such that $x \in (R \leftarrow p)$ and yet there may exist a y such that $(x,y) \in R$ and $y \notin p$.

Partial correctness of R with respect to properties p and q, as used in Floyd's method of proof, was expressed as $p' \circ R \subseteq R \circ q'$ and defined as

$$x \in p \quad \text{and} \quad (x,y) \in R \Rightarrow y \in q,$$

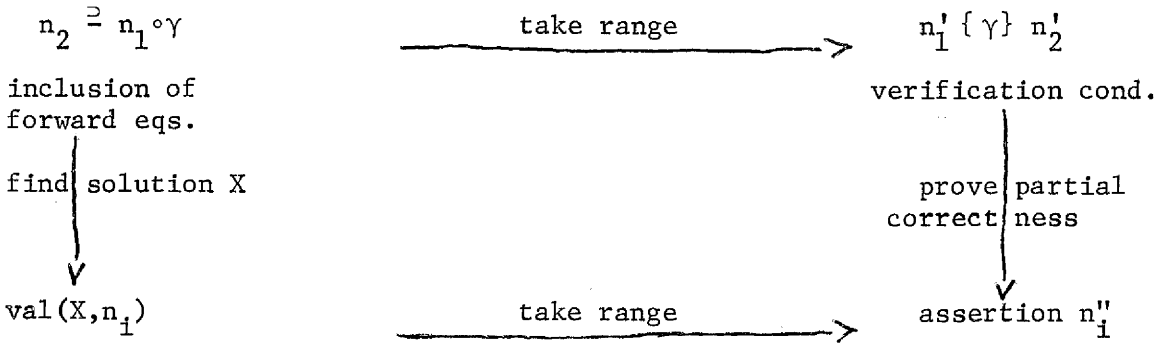
can be expressed as $(p \rightarrow R) \subseteq q$. The backward analog $p' \circ R \supseteq R \circ q'$, that is

$$y \in q \quad \text{and} \quad (x,y) \in R \Rightarrow x \in p$$

can be expressed as $p \supseteq (R \leftarrow q)$.

Let P be a flowgraph $((N,T,P,S),(D,B))$. The premiss in Floyd's rule of proof is the conjunction of verification conditions, one, namely $a_2 \supseteq (a_1 \rightarrow \gamma)$, for each production $n_1 \rightarrow tn_2$ of P, where a_1 (a_2) is the assertion associated with node n_1 (n_2) of the flowgraph and γ is the product of constants corresponding to the string t of terminals.

Now the inclusion in the forward equation associated with $n_1 \rightarrow tn_2$ is $n_2 \supseteq n_1 \circ \gamma$, hence we derive an inclusion among sets of states $(D \rightarrow n_2) \supseteq (D \rightarrow n_1 \circ \gamma)$, which is $a_2 \supseteq (a_1 \rightarrow \gamma)$ if $a_1 = D \rightarrow n_1$ and $a_2 = D \rightarrow n_2$. Apparently, the inclusions of the forward equation are in a disguised form the premiss in Floyd's rule of proof. Proving partial correctness is solving the verification conditions where the assertions are the unknowns, which is much like solving the forward equation. The diagram below illustrates two alternative equivalent ways of proving partial correctness.



Floyd's method of proof can also be justified, in a very direct way, by the properties of relational equations. Let R be the input-output relation of a program with forward equation E_1. The partial correctness of the program with respect to properties p and q is $(p \to R) \subseteq q$; it may be expressed in terms of binary relations as $p' \circ R \subseteq R \circ q'$ where p' (q') are the relational analogs of p and q: $(x,x) \in p'$ iff $x \in p$, and similarly for q. Because $R = val(\cap S(E_1), H)$, it is plausible that Floyd's method of proof can be justified by showing that $p' \circ H \subseteq H \circ q'$ is in some sense a consequence of the forward equation. Indeed we have:

<u>Theorem 4.4</u> A program P has the partial correctness property with respect to p and q iff $p' \circ H \subseteq H \circ q'$ is a weak implication of the forward equation E_1 of P. P has the backward partial correctness property with respect to p and q iff $S \circ q' \subseteq p' \circ S$ is a weak implication of the backward equation E_2 of P.

<u>Proof</u> $\quad p' \circ R \subseteq R \circ q'$ iff

$p' \circ val(\cap S(E_1), H) \subseteq val(\cap S(E_1), H) \circ q'$ iff

$val(\cap S(E_1), p' \circ H) \subseteq val(\cap S(E_1), H \circ q')$ iff

$p' \circ H \subseteq H \circ q'$ satisfied by the least solution of E_1. A similar reasoning justifies the backward part of the theorem □

4.6 A semantics for Dijkstra's sequencing primitives

An interesting application of the semantics of flowgraphs is to express the semantics of programs in Dijkstra's [12,13] programming language. We effect the application by exhibiting an equivalent flowgraph for each construct of Dijkstra and by then giving the input-output relation for each flowgraph as obtained, for instance, by the value of the halt symbol in the forward equation.

Let the flowgraphs again be ((N,T,P,S),(D,B)), where only the set P of productions is a different one for each of the Dijkstra constructs.

$N = \{S,Q,H\}$

$T = \{\text{skip},\text{abort},\ldots\}$

The set D of states is left unspecified and assumed to be the one in which the primitive statements of the Dijkstra constructs act. The subset B of Tx(DxD) assigns binary relations to the terminal symbol of T:

val(B,skip) = I, the identity relation

val(B,abort) = ϕ, the empty relation

Whenever b is a boolean expression occuring as a "guard":

$$\text{val}(B,b) = \{(x,x): \ b \text{ is true in } x,\ x \in D\} \overset{df}{=} b'$$

T also includes assignment statements with expressions in their right-hand sides; these statements will not be discussed here: we assume that B will be such that these terminals get the correct relation as value.

The construct $S_1;S_2$ corresponds to a flowgraph with $P = \{ S \to S_1S_2H\}$. The value of H in the least solution of the forward equation is $S_1' \circ S_2'$ where S_i' is the input-output relation of S_i, $i = 1,2$.

The construct

$$\underline{\text{if}}\ b_1 \to SL_1 \ |\ldots|\ b_n \to SL_n\ \underline{\text{fi}}$$

corresponds to a flowgraph with

$$P = \{ S \to b_1SL_1H,\ldots,S \to b_nSL_nH\} .$$

The value of H in the least solution of the forward equation is

$$b_1' \circ SL_1' \cup \ldots \cup b_n' \circ SL_n' \qquad \ldots(4.6.1)$$

where SL_i' is the input-output relation of SL_i, $i=1,\ldots,n$.

The construct

$$\underline{\text{do}}\ b_1 \to SL_1 \ |\ldots|\ b_n \to SL_n\ \underline{\text{od}}$$

corresponds to a flowgraph with

$$P = \{ S \to Q,\ Q \to b_1SL_1Q,\ldots,Q \to b_nSL_nQ,\ Q \to bH\} .$$

The value of H in the least solution of the forward equation is

$$(b_1' \circ SL_1' \cup \ldots \cup b_n' \circ SL_n')^* \circ b' \qquad \ldots(4.6.2)$$

where b' is the complement in I of $b_1' \cup \ldots \cup b_n'$.

It is curious that Dijkstra does not consider backtracking as part of the execution mechanism for his programs. There are several circumstances that suggest backtracking. Firstly, one of the advantages claimed by Dijkstra for the do...od construct is its goal-directed nature: the complement b of the union of the guards $b_1,\ldots,b_n$ is the goal achieved by execution of

$$\underline{\text{do}}\ b_1 \to SL_1 \ |\ldots|\ b_n \to SL_n\ \underline{\text{od}}$$

The indeterminacy of this construct makes possible the flexibility of adding each $b_i \to SL_i$ independently of the others, whenever SL_i is discovered as an action useful under condition b_i for bringing the state nearer the goal.

However, the situation, where the goal happens to coincide with the complement of the union of the guards, is a rather special case. Consider for instance the problem of example 4.5 where an amount of n cents has to be paid with dimes, nickels, and cents. It would be most straightforward if we could use

$$(n \geq 10 \wedge n_{10} > 0) \rightarrow n, n_{10}, p_{10} := n-10, n_{10}-1, p_{10}+1$$
$$\mid (n \geq 5 \wedge n_5 > 0) \rightarrow n, n_5, p_5 := n-5, n_5-1, p_5+1$$
$$\mid (n \geq 1 \wedge n_1 > 0) \rightarrow n, n_1, p_1 := n-1, n_1-1, p_1+1$$

But our goal is n=0, which not the negation of the disjunction of the guards. This is an example of a case where

```
do ¬goal → if b1 → SL1
           | ...
           ...
           | bn → SLn
           fi
 od
```

is a clear expression of the programmers intention. And the only thing needed to make this work in Dijkstra's programming language is to assume that the executing mechanism backtracks upon "abort". Dijkstra has already specified that, if a statement if...fi is executed in a state with all guards false, then the statement will be equivalent to "abort".

Take in the coins example $n=7$, $n_5=1$, $n_1=3$. After a few indeterminate choices we may have $n=4$, $n_5=1, n_1=0$. All guards are false, but the goal is not achieved. Dijkstra [13] has indeed identified "abort" with failure, but it is hard to see the use of this without backtracking. In the above example backtracking would return to the point where $n=5$, $n_5=1$, $n_1=1$, and the other choice will now lead to the goal where $n=0$.

The semantics of flowgraphs supposes a backtracking interpreter. We have shown that the usual fixpoint semantics is in no way made more complicated as a result. In fact, our theorem 4.3 about failed paths may well be regarded as a contribution to the understanding of greatest fixpoints.

It should be clear that we do not usually advocate running indeterminate programs. Goal-directed programming is facilitated when indeterminacy is allowed; a first approximation to an algorithm is thus obtained, which can then be refined to a determinate equivalent [14,15].

4.7 Correctness proofs with transformers

Floyd's method for proving partial correctness of a program with input-output relation R with respect to assertions p and q requires that invariants and other intermediate assertions be invented. This is not in principle necessary: we might be able to compute $p \rightarrow R$ and compare the result with q. However, we should not be surprised if the computation is only feasible with some special representation of the $p \rightarrow R$, and that the invariants required by Floyd's method are useful for the evaluation of $p \rightarrow R$.

The same considerations apply to backward transformers and to the backward version of partial correctness. Backward transformers are also, or perhaps primarily, useful for proving total correctness of determinate programs: if $p \subseteq (R \leftarrow q)$ then for every state in p there exists at least one path terminating in q. As every flowgraph can be expressed by means of $\cup$, $\circ$, and $*$ we will use the following properties of transformers:

$$(R_1 \cup R_2) \leftarrow q = (R_1 \leftarrow q) \cup (R_2 \leftarrow q)$$

$$(R_1 \circ R_2) \leftarrow q = R_1 \leftarrow (R_2 \leftarrow q)$$

$$R^* \leftarrow q = \bigcup_{n=0}^{\infty} f^n(q) \ \underline{\underline{df}}\ f^*(q)$$

where $f^0(q) = q$, $f^{n+1}(q) = f(f^n(q))$, $i=0,1,\ldots$, and

where $f = \lambda x(R \leftarrow x)$

The analogous properties hold for the forward transformer.

The backward transformer gives the following results for some of Dijkstra's statements:

 37.

$$\text{skip} \leftarrow q = q$$

$$\text{abort} \leftarrow q = \phi$$

$$(S_1;S_2) \leftarrow q = S_1' \leftarrow (S_2' \leftarrow q)$$

$$(\underline{\text{if}}\ b_1 \rightarrow SL_1 \quad \ldots \quad SL_n\ \underline{\text{fi}}) \leftarrow q = \qquad \text{(by(4.6.1))}$$

$$(b_1' \circ SL_1' \cup \ \ldots \cup \ b_n' \circ SL_n') \leftarrow q =$$

$$(b_1' \leftarrow (SL_1' \leftarrow q)) \cup \ \ldots \cup \ (b_n' \leftarrow (SL_n' \leftarrow q)) = f(q), \quad \text{where}$$

$$f = \lambda x \cdot ((b_1' \cap f_1(x)) \cup \ \ldots \cup \ (b_n' \cap f_n(x))), \quad \text{where}$$

$$f_i = \lambda x \cdot (SL_i' \leftarrow x), \ i=1,\ldots,n$$

$$(\underline{\text{do}}\ b_1 \rightarrow SL_1 \mid \ldots \mid b_n \rightarrow SL_n\ \underline{\text{od}}) \leftarrow q = \qquad \text{(by(4.6.2))}$$

$$((b_1' \circ SL_1' \cup \ \ldots \cup \ b_n' \circ SL_n')^* \circ b') \leftarrow q =$$

$$(b_1' \circ SL_1' \cup \ \ldots \cup \ b_n' \circ SL_n')^* \leftarrow (b' \cap q) = f^*(b' \cap q)$$

with f as above.

As an example we shall prove the correct termination of an exponentiation algorithm. Consider a program $((N,T,P,S),(D,B))$ with

$$N = \{S,Q,R,H\}$$

$$T = \{a,b,c,c',d,e,e'\}$$

$$P = \{ S \rightarrow aP,\ Q \rightarrow e'R,\ Q \rightarrow eH,\ R \rightarrow cdR,\ R \rightarrow c'bQ \}$$

D is as in Example 4.1. B assigns to a,b,c,d,e the same (as in Example 4.1) binary relations as values; B assigns to c' and e' the complements in I of c and e respectively.

We have to determine a useful set of input states such that termination is guaranteed and the condition q: $w=u_o^{v_o}$ is assured to hold, where u_o and v_o are the values of u and v in the input state. Let R be the input-output relation of the program; any subset of

$$R \leftarrow q$$

will do. An expression for a suitable set of input states is therefore

$$(a\circ(e'\circ(c\circ d)^{*}\circ c'\circ b)^{*}\circ e) \leftarrow (w=u_o{}^{v_o})$$

which is not useful because it is not easy to tell (for us at least) whether a given input state is in it. In order to be able to evaluate the expression we will need to know something about the functions $\lambda x\cdot d \leftarrow x$ and $\lambda x\cdot b \leftarrow x$. It is not clear whether an easily evaluated formula can be found which is valid for all arguments. But we may only need the value for special cases of the argument and then it may easily be calculated. Consider the case where $x = (wxu^{v} = u_o{}^{v_o})$; then both $x \subseteq (d \leftarrow x)$ and $x \subseteq (b \leftarrow x)$. The condition $wxu^{v} = u_o{}^{v_o}$ is an invariant of d and of b. Let us call it "inv". We only need this special form of argument because the desired terminal condition $w = u_o{}^{v_o}$ is implied by $\text{inv} \cap (v=0)$.

We can now derive a useful set of input states with respect to which there is total correctness:

$$\begin{aligned}
&(a\circ(e'\circ(c\circ d)^{*}\circ c'\circ b)^{*}\circ e) \leftarrow (w=u_o{}^{v_o}) \supseteq \\
&(a\circ(e'\circ(c\circ d)^{*}\circ c'\circ b)^{*}\circ e) \leftarrow (\textbf{inv} \cap (v=0)) = \\
&(a\circ(e'\circ(c\circ d)^{*}\circ c'\circ b)^{*}) \leftarrow (\text{inv} \cup (v=0))
\end{aligned}$$

Note that $(e'\circ(c\circ d)^{*}\circ c'\circ b)^{d(i)} \leftarrow (\text{inv} \cap (v=0)) \supseteq \text{inv} \cap (v = i)$, where $d(i)$ is the number of ones in the binary representation of i. Hence

$$\begin{aligned}
&(a\circ(e'\circ(c\circ d)^{*}\circ c'\circ b)^{*}) \leftarrow (\text{inv} \cap (v=0)) \supseteq \\
&a \leftarrow \bigcup_{i=0}^{\infty} (\text{inv} \cap (v = i)) = \\
&a \leftarrow (\text{inv} \cap (v\geq 0)) \supseteq \\
&v\geq 0
\end{aligned}$$

Thus correct termination is guaranteed for $v\geq 0$.

REFERENCES

1. Ashcroft, E., Z. Manna, A. Pnueli: Decidable properties of monadic functional schemas. J.ACM 20 (1973) 489-499

2. de Bakker, J.W.: The fixed point approach in semantics: theory and applications. J.W. de Bakker (ed.): Foundations of computer science. Tract 63, Mathematical Centre, Amsterdam, 1975

3. de Bakker, J.W., W.P. de Roever: A calculus of recursive program schemes. M. Nivat (ed.): Automata, Languages, and Programming North Holland, Amsterdam, 1973

4. de Bakker, J.W., D. Scott: A theory of programs. IBM Seminar, Vienna, August 1969

5. Blikle, A.: An algebraic approach to the mathemtical thoery of programs. CCPAS Report 119, Computation Centre Polish Academy of Sciences, Warsaw, 1973

6. Blikle, A.: An extended approach to the mathematical analysis of programs. CCPAS Report 169, Computation Centre Polish Academy of Sciences, Warswa, 1974

7. Blikle, A.: Floyd's method of program verification with strongest inductive assertions. Proc. IFIP 1977

8. Blikle, A., A. Mazurkiewicz: An algebraic approach to the theory of programs, algorithms, languages, and recursiveness. International Symposium and Summer School on Mathematical Foundations of Computer Science, Warsaw, 1972

9. Brzozowski, J.A.: Derivatives of regular expressions. J.ACM 11 (1964) 481-494

10. Burstall, R.M.: An algebraic description of programs with assertions, verification, and simulation, SIGPLAN Notices 7 (1972), pp. 7-14

11. Clark, K.L., D.F. Cowell: Programs, Machines, and Computation. McGraw-Hill, 1976

12. Dijkstra, E.W.: Guarded commands, nondeterminacy, and formal derivation of programs. CACM, 18 (1975), 453-457

REFERENCES (CONT'D.)

13. Dijkstra, E.W.:
A discipline of programming.
Prentice Hall, 1976

14. van Emden, M.H.:
Unstructured Systematic Programming.
Report CS-76-09, Dept. of Computer Science, University of Waterloo

15. van Emden, M.H.:
A worked example in unstructured systematic programming.
Report CS-76-27, Dept. of Computer Science, University of Waterloo

16. van Emden, M.H.:
Verification conditions as programs.
S. Michaelson, R. Milner (eds.), Automata, Languages, and Programming,
Edinburgh University Press, 1976

17. van Emden, M.H., R.A. Kowalski:
The semantics of predicate logic as a programming languages
J.ACM, 23 (1976) 733-742

18. Engelfriet, J.:
Simple program schemes and formal languages.
Lecture Notes in Computer Science 20, Springer, 1974

19. Floyd, R.W.:
Assigning meanings to programs, pp. 19-32 in:
Proc. Symposium App. Math. Vol. XIX
J.T. Schwartz (ed.), A.M.S., Providence, R.I., 1967

20. Ginsburg, S., H.G. Rice:
Two families of languages related to Algol.
J.ACM 9 (1962) 350-371

21. Kowalski, R.A.:
Logic for problem-solving. Memo 75,
Dept. of Computational Logic, University of Edinburgh, 1974

22. Mazurkiewicz, A.:
Proving properties of processes.
CCPAS Report 134, Computation Centre, Polish Academy of Sciences, 1973

23. Morris, J.H., B. Wegbreit:
Subgoal induction; CACM 20 (1977), 209-222

24. Scott, D.:
Some definitional suggestions for automata theory.
J. Computer and Systems Sciences, 1 (1967) 187-212

25. Turing, A.M.:
On computable numbers, with an application to the Entscheidungsproblem.
Proc. London Math. Soc. Ser. 2, 42 (1936-1937), 230-265