FLOYD'S METHOD OF PROGRAM VERIFICATION WITH THE STRONGEST INDUCTIVE ASSERTIONS

A. Blikle*

Computation Center
Polish Academy of Sciences
PKiN, P.O. Box 22
00-901 Warsaw, Poland

Research Report CS-76-36

July 1976

* This work was done when the author was visiting the University of Waterloo and was supported by the National Research Council of Canada under Grant A-1617.

Abstract

Dealing with the strongest Floyd assertions of programs allows a simultaneous verification of the strongest partial correctness, the termination and the nontermination of programs. No auxiliary constructs such as well-founded sets or loop counters are required. The method works equally for correct and incorrect programs since it provides a way of the "computation" of the actual properties of programs.

Mathematically the method uses the algebra of binary relations. It is proved that for any iterative program the strongest Floyd assertions exist, are unique and constitute the least solution of a fixed-point set of equations. A problem-oriented calculus allows to find the least solutions and to transform them into such a form where the termination domain and the input-output processing are described explicitly. The method was tested on several hardware microprograms of a floating-point arithmetical unit of a computer.

1. Introduction

For a better explanation of the present approach let us start with a short review of the original Floyd's method (Floyd [6]; see also Manna [9] for a formal exposition). Given a program Π operating on a vector y of variables one first replaces it by the program

$$\Pi_1 = y := x;\ \Pi;\ z := y$$

where x and z are vectors of variables which do not appear in Π and which are called the input vector and the output vector respectively. These two auxiliary vectors of variables are necessary in order to formulate partial-correctness properties which say that if some intput predicate $\phi(x)$ is satisfied and the program eventually terminates then some output predicate $\psi(x,z)$ must also be satisfied. To carry out proofs of such properties one introduces so-called cut-points $\alpha_1,\ldots,\alpha_n$ of Π_1, which are in fact some selected control states of this program, and one associates with each of them a so-called inductive assertion $p_{\alpha_i}(x,y)$. These assertions are auxiliary (intermediate) propositions about the relationship between the initial and the current state of variables and allow to proceed from the assumption $\phi(x)$ to the conclusion $\psi(x,z)$. For technical detail see Sec.4.

The method of Floyd, although quite widely used in experimental program-verifiers, has some inconveniences pointed out by other authors (cf. Katz and Manna [8]; Grief and Waldinger [7]):

1) The method requires that the predicates $\phi(x), \psi(x,z), p_{\alpha_1}(x,y), \ldots, p_{\alpha_n}(x,y)$ be all known in advance before we proceed to the proof of correctness. This is especially cumbersome if our program is not correct, i.e. if it does not satisfy the "intended" properties.

2) Proofs of termination require the introduction of so-called well-founded sets and of additional predicates. These sets and predicates also must be known in advance which offer the same type of inconvenience as in 1). Moreover, proofs of termination are totally different from proofs of partial correctness which require new mathematical techniques in a program-verification system.

3) The technique described in 2) cannot be used to prove non-termination of a program.

In [8] Katz and Manna propose a solution of the problems mentioned above. First of all they show several heuristic methods for generating inductive-assertions. Next they show that each of the following problems:

a) partial correctness w.r.t. $\phi(x)$ and $\psi(x,z)$

b) incorrectness w.r.t. $\phi(x)$ and $\psi(x,z)$

c) termination w.r.t. $\phi_1(x)$

d) nontermination w.r.t. $\phi_2(x)$

can be reduced to the problem of the existence of an appropriate set of assertions (which satisfies appropriate conditions). In each case the set of assertions is different, but in each case the assertion-generation method is helpful in its construction. To summarize, Katz and Manna's method allows the use of just one assertion-generation vehicle for all the four

problems a)-d). But the proofs must be carried out independently. In particular, the failure of proving a) does not prove b) and vice-versa; the failure of proving c) does not show that $\phi(x)$ is not a termination condition. Also, if we succeed to prove that $\phi(x)$ implies termination it does not necessarily mean that $\sim\phi(x)$ implies nontermination. In effect, a full verification of a program requires three independent proofs: of a), of c) and of d). And in each of the three cases another set of assertions must be generated.

Concerning our approach to the Floyd method we must stress first of all that its philosophy is slightly different from that developed in other approaches. Its main point is that the application of mathematics to computer science (to the theory of programs in particular) can be similar to the application of mathematics to other sciences or fields of engineering. For instance, if one wants to mathematically verify an electrical circuit, one shall probably start with the corresponding Kirchoff equations, one shall solve them and obtain formulas which describe several physical properties of this circuit. One shall proceed in a similar way in verifying any other technical object: a car engine, a bridge, an optical device etc. No engineer would mathematically verify a technical device in such a way that he assumes some properties of this device, then tries to prove them and in the case of failure he modifies the assumed properties and tries to prove them again. The actual properties of the device "comes out" in the process of mathematical analysis and only when we know them we make the comparison with the required properties and decide if our device is acceptable or not. It is our belief

that the mathematical verification of programs can frequently be developed along the same lines. Given a program, we do not assume any properties of it but simply "compute" these properties and only when we know them we decide if our program is the required one. Precisely speaking, we attempt to get an explicit description of the input-output function F of our program. The domain Dom(F) of this function describes exactly the domain of termination, i.e. we know that our program terminates everywhere in Dom(F) and that it does not terminate everywhere outside of Dom(F). The "body" of F describes exactly what the program does. In this way, the process of "computing" the formula which describes F provides at the same time the proofs of partial correctness, termination and nontermination. Theproblem of proving incorrectness simply disappears since the very concept of incorrectness makes sense only with respect to a program property assumed in advance.

Technically, we deal in our approach with the minimal (the strongest) Floyd's assertions. Since these assertions are unique we can develop a systematic way of "computing" them. Of course, this way is not free of heuristics but these heuristics are restricted now to the search of an appropriate description of F, while F itself is unique. Observe that in the traditional approach to Floyd's method the predicates $\phi(x)$ and $\psi(x,z)$ are not unique for the program, nor are the inductive assertions $p_{\alpha_1}(x,y),\dots,p_{\alpha_n}(x,y)$ unique for $\phi(x)$ and $\psi(x,z)$.

The mathematical vehicle used in the present approach is the algebra of binary relations equipped with fixed-point methods. The set of minimal Floyd inductive assertions turns out to be the least solution of a left-linear

system of fixed-point equations. Coefficients in this system of equations are relations (functions if the program is deterministic) describing the input-output meanings of "boxes" (assignment statements, tests etc.). By a simple and standard method we can find now the minimal Floyd's assertions described in terms of these coefficients and three standard operations of our algebra (precisely speaking, we get these assertions described by regular-like expressions). In this way, we get the assertions in an implicit form. To transform them into an explicit form (i.e. where the domain of termination and the data processing are "evident") we use the problem-oriented calculus developed in our algebra and some standard mathematical techniques such as proofs by induction. Of course, this last step is the most difficult and crucial in the process of analysis. In particular, all the heuristics appearing in our approach appear there. These heuristics do not require, however, any auxiliary constructs such as well-founded sets [6] or loop-counters [8] and is strictly limited to the search of the form rather than the "content" of assertions.

Concerning the applicability of our approach to the construction of program-verification systems it should not offer more technical difficulties than the traditional approach (for more details, see Sec.6). On the other hand, it is our opinion that the construction of program-verification systems is just one of many other goals of the mathematical theory of programs. Two other evident goals are the better understanding of phenomena appearing in programs and the improvement of the process of design and documentation. The more we can know about programs the better we can understand, design and document them. And once we deal with the strongest Floyd's assertions we get a complete information about the input-output properties of programs.

To complete this section let us mention that the practical aspects of the method were quite successfully tested by a group of hardware engineers designing a floating-point arithmetical unit of a computer (see Blikle and Budkowski [4]). Of course, the method is not free of technical inconveniences which are briefly described in Sec.6.

2. The algebra of binary relations

The algebra of binary relations offers a natural mathematical language of dealing with the input-output properties of programs. Below we describe some preliminaries of this algebra together with the notation used in the sequel. Although we are going to apply our approach to deterministic programs, whose input-output relations are functions, we shall also use extensively nonfunctional relations to describe Floyd's inductive assertions.

Let D be an arbitrary nonempty set called the domain and interpreted as the set of all possible states of the vector of variables in a program. By Rel(D) we denote the set of all binary relations in D, i.e.

$$\mathrm{Rel}(D) = \{R \mid R \subseteq D \times D\}$$

For any a,b in D and R in Rel(D) we shall write aRb for $(a,b) \in R$. By ϕ we shall denote the empty relation, and by I the identity relation, i.e. $I = \{(a,a) \mid a \in D\}$.

Basic operations in the set Rel(D), which we shall use in the sequel, are defined below. Let $R_1, R_2 \in \mathrm{Rel}(D)$.

$R_1 \cup R_2 = \{(a,b) \mid aR_1b \vee aR_2b\}$		- union
$R_1 \circ R_2$	$= \{(a,b) \mid (\exists c)aR_1c \,\&\, cR_2b\}$	- composition
R_1^0	$= I$	- 0-th power
R_1^n	$= R_1^{n-1} \circ R_1$	- n-th power
R_1^*	$= I \cup R_1 \cup R_1 \circ R_1 \cup R_1 \circ R_1 \circ R_1 \cup \ldots = \bigcup_{n=0}^{\infty} R_1^n$	- *-iteration
R_1^+	$= R_1 \cup R_1 \circ R_1 \cup R_1 \circ R_1 \circ R_1 \cup \ldots = \bigcup_{n=1}^{\infty} R_1^n$	- +-iteration

Fig.1 shows the interpretation of these operations in terms of flow-chart constructs. Below we list the most important properties of these

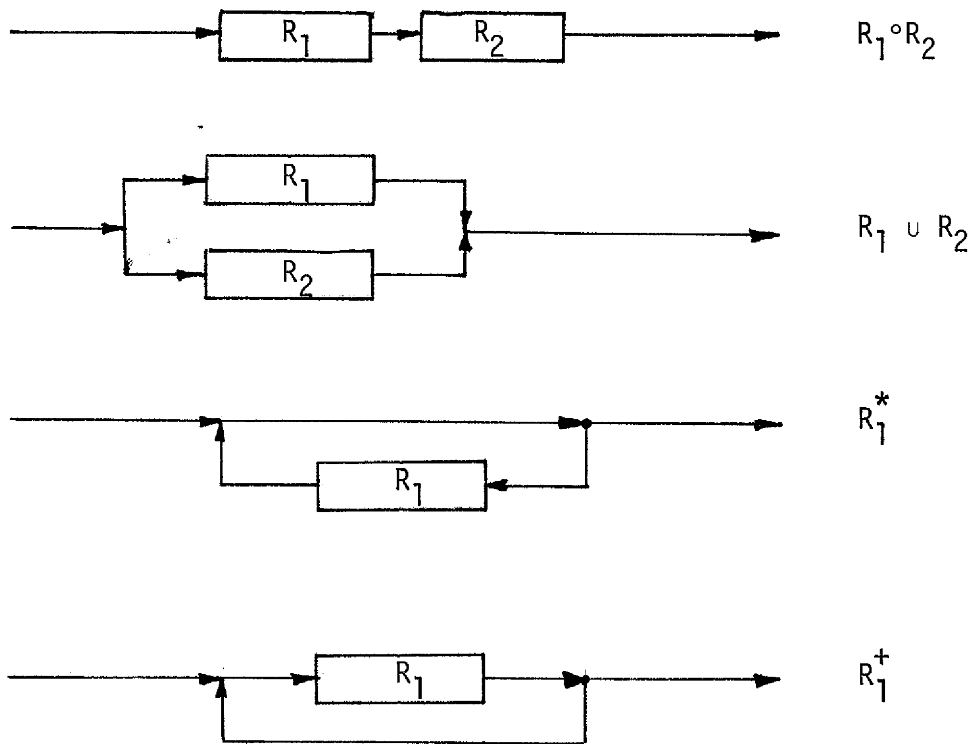

Fig.1

operations. Here and in the sequel we shall omit the symbol "$\circ$" of composition and write R_iR_j instead of $R_i \circ R_j$.

1) $R_1(R_2R_3) = (R_1R_2)R_3$ - associativity

2) $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$ - finite distributivity

$(R_2 \cup R_3)R_1 = R_2R_1 \cup R_3R_1$

3) $R_0(\bigcup_{i=1}^{\infty} R_i) = \bigcup_{i=1}^{\infty} R_0R_i$ - infinite distributivity

$(\bigcup_{i=1}^{\infty} R_i)R_0 = \bigcup_{i=1}^{\infty} R_iR_0$

4) $RI = IR = R$ - the unit property of I

5) $R\phi = \phi R = \phi$ - the zero property of ϕ

6) $R^* = I \cup R^+$

$R^+ = RR^* = R^*R$

In dealing with concrete programs and carrying out their analysis we shall need an explicit notation to specify concrete relations in Rel(D). We shall distinguish between two cases: the general case of arbitrary relations and the particular case of partial functions. Of course, the former case covers the latter, but practical experience shows that calculations on functions are easier than these on relations, and therefore it is desirable to have the "functional" case distinguished. We shall start with the notation for functions introduced by Mazurkiewicz [5].

Let $f:D \to D$ be an arbitrary partial function and let $p:D \to \{true, false\}$ be an arbitrary partial unary predicate such that if $p(d) = true$ then $f(d)$ is defined. We denote by

$$[p(x)|x := f(x)] = \{(d_1,d_2)|p(d_1) = true \;\&\; d_2 = f(d_1)\} \qquad (2.1)$$

Of course, $[p(x)|x := f(x)]$ is a partial function whose domain is $\{d|d \in D \,\&\, p(d) = true\}$. The variable x in (2.1) is bound (as "bound by a quantifier") which means that $[p(x)|x := f(x)]$ and $[p(y)|y := f(y)]$ denote the same function. For the sake of simplicity we shall also write

$$[x := f(x)] \quad \text{for} \quad [true|x := f(x)]$$

and

$$[p(x)] \qquad \text{for} \quad [p(x)|x := x].$$

Of course, $[p(x)][x := f(x)] = [p(x)|x := f(x)]$. Also note that every function of the form $[p(x)]$ is a subset of the identity function I. Consequently, these functions are used in our approach to describe tests in programs. The following rules are useful in program verification (all can be easily proved from (2.1)):

$$\begin{aligned}
&1)\quad [p(x)|x := f(x)][q(x)|x := g(x)] = \\
&\qquad\qquad = [p(x) \,\&\, q(f(x))|x := g(f(x))] \\
&2)\quad [p(x)|x := f(x)] \cup [q(x)|x := f(x)] = \\
&\qquad\qquad = [p(x) \vee q(x)|x := f(x)] \qquad\qquad (2.2) \\
&3)\quad [p(x)][q(x)] = [q(x)][p(x)] = [p(x) \,\&\, q(x)] \\
&4)\quad \text{If } p(x) \Rightarrow q(f(x)) \text{ then } [p(x)|x := f(x)][q(x)] = \\
&\qquad\qquad = [p(x)|x := f(x)]. \\
&5)\quad [p(x)|x := f(x)]^n = \\
&\qquad\qquad = [(\forall 1 \le i \le n)p(f^{i-1}(x))|x := f^n(x)] \text{ for } n \ge 1
\end{aligned}$$

As we shall see in Sec.5 (Examples 5.1 and 5.3) the rule 5) allows the elimination of induction proofs in the consideration of loops.

Now we introduce the more general "nonfunctional" notation. Let $p: D \times D \rightarrow \{true, false\}$ be an arbitrary partial binary predicate. We denote by:

$$[R|p(\bar{x},x)] = \{(d_1,d_2)|p(d_1,d_2) = true\}. \qquad (2.3)$$

The character R appears here to indicate that we are dealing with the new notation. In fact, we must be able to distinguish between the three following relations:

$$[p(x)] \quad = \{(d_1,d_2)|d_1 = d_2 \;\&\; p(d_1) = true\}$$
$$[R|p(x)] = \{(d_1,d_2)|p(d_2) = true\}$$
$$[R|p(\bar{x})] \quad \{(d_1,d_2)|p(d_1) = true\}$$

The variables $\bar{x}$ and x in (2.3) are bound in the same sense as x is bound in (2.1) and can be interpreted as the initial and the terminal data vector of the corresponding program. Of course, the former notation can be defined in terms of the latter:

$$[p(x)|x := f(x)] = [R|p(\bar{x}) \;\&\; x = f(\bar{x})] \qquad (2.4)$$

where we still must assume that $p(d)$ implies $d \in Dom(f)$. We also have the obvious rules for calculation:

1) $[R|p(\bar{x},x)][R|q(\bar{x},x)] = [R|(\exists y)(p(\bar{x},y) \;\&\; q(y,x))]$

2) $[R|p(\bar{x},x)] \cup [R|q(\bar{x},x)] = [R|p(\bar{x},x) \vee q(\bar{x},x)]$ (2.5)

3) $\bigcup_{c \in C} [R|p(\bar{x},x,c)] = [R|(\exists c \in C)p(\bar{x},x,c)]$

4) $[q(x)|x := f(x)][R|p(\bar{x},x)] = [R|q(\bar{x}) \& p(f(\bar{x}),x)]$

5) if f is reversible, i.e. if f^{-1} is a function then:

$$[R|p(\bar{x},x)][q(x)|x := f(x)] =$$
$$= [R|p(\bar{x},f^{-1}(x)) \& q(f^{-1}(x)) \& x \in Dom(f^{-1})]$$

where $Dom(f^{-1})$ denotes the domain of f^{-1}. The rules 1)-4) are fairly obvious. We prove 5) below, mainly to show how our calculus works:

$$[R|p(\bar{x},x)][q(x)|x := f(x)] =$$
$$= [R|(\exists y)(p(\bar{x},y) \& q(y) \& x = f(y))] =$$
$$= [R|(\exists y)(p(\bar{x},y) \& q(y) \& y = f^{-1}(x)] =$$
$$= [R|(\exists y)(p(\bar{x},f^{-1}(x)) \& q(f^{-1}(x)) \& y = f^{-1}(x)] =$$
$$= [R|p(\bar{x},f^{-1}(x) \& q(f^{-1}(x)) \& x \in Dom(f^{-1})]$$

In the last step we eliminate the quantifier $(\exists y)$ by substituting "$x \in Dom(f^{-1}(x))$" for " $(\exists y)(y = f^{-1}(x))$".

3. The mathematical models of programs

In order to describe the Floyd's method in terms of our algebra we shall need a rigorous mathematical concept of a program and of its input-output relation. To this effect we shall use here the Mazurkiewicz algorithms [5] and shall define their operational semantics. In the general case these algorithms are nondeterministic devices but the deterministic programs appear simply as the particular case.

By an <u>algorithm</u> we shall mean any system $A = (D,V,\alpha_1,\mathfrak{J})$ where

D is an arbitrary nonempty set called the <u>domain</u> of the algorithm and is interpreted as in Sec.2,

$V = \{\alpha_1,\ldots,\alpha_n\}$ is a finite nonempty set of elements called <u>labels</u> of the algorithm,

α_1 is a distinguished element of V called the <u>initial label</u> of the algorithm,

$\mathfrak{J} = \{(\alpha_i,R_{ij},\alpha_j) | R_{ij} \in Rel(D);\ i,j \leq n\}$ is a set of $p = n^2$ triples called <u>instructions</u>. It is implicit in the notation above that for any α_i and α_j there is exactly one R_{ij} such that $(\alpha_i,R_{ij},\alpha_j) \in \mathfrak{J}$. Usually many of the R_{ij} relations will be empty. An instruction with $R_{ij} = \phi$ describes the fact that there is no direct "arc" between α_i and α_j in the program. Given an instruction, the corresponding α_i, R_{ij} and α_j are called the <u>entrance label</u>, the <u>action</u> and the <u>exit label</u>.

<u>Interpretation</u>. As mentioned before, the algorithms of Mazurkiewicz are used to formalize flowchart programs. The labels α_i's are usually interpreted as control states, but in this case we can interpret them as the Floyd's cutpoints. Each action R_{ij} describes the data processing effected between α_i and α_j. The choice of cutpoints in a flowchart is arbitrary, but once these cutpoints has been chosen the corresponding algorithm is unique. Consider the flowchart of Fig.2. The set of cutpoints $\{\alpha_1,\ldots,\alpha_7\}$ is "maximal", in a sense, for this flowchart. The corresponding set of instructions is the following (we omit, of course, all the instructions with $R_{ij} = \phi$):

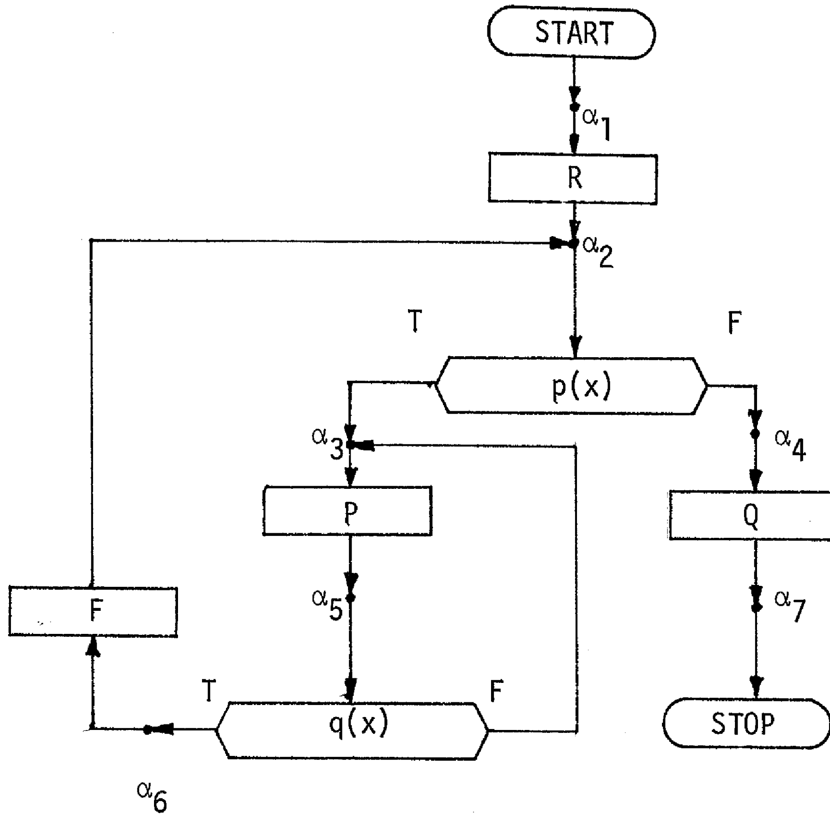

Fig. 2

(α_1,R,α_2)

$(\alpha_2,[p(x)],\alpha_3)$, $(\alpha_2,[\sim p(x)],\alpha_4)$

(α_3,P,α_5)

(α_4,Q,α_7)

$(\alpha_5,[q(x)],\alpha_6)$, $(\alpha_5,[\sim q(x)],\alpha_3)$

(α_6,F,α_2)

The same flowchart can be described with a smaller set of cutpoints $\{\alpha_1,\alpha_2,\alpha_3,\alpha_7\}$. The corresponding algorithm will be, of course, different from the former:

(α_1,R,α_2)

$(\alpha_2,[p(x)],\alpha_3)$, $(\alpha_2,[\sim p(x)]Q,\alpha_7)$

$(\alpha_3,P[q(x)]F,\alpha_2)$, $(\alpha_3,P[\sim q(x)],\alpha_3)$. □

Now, we shall establish the operational semantics of Mazurkiewicz algorithms. Consider an arbitrary algorithm $A = (D,V,\alpha_1,\mathfrak{I})$ where $V = \{\alpha_1,\ldots,\alpha_n\}$. For any $\alpha_i,\alpha_j \in V$, by an (α_i,α_j)-<u>run</u> we shall mean any sequence of instructions of $\mathfrak{I}$:

$$(\alpha_{i_1},R_1,\alpha_{j_1});\ \ldots;\ (\alpha_{i_k},R_k,\alpha_{j_k}) \qquad (3.1)$$

such that $\alpha_{i_1} = \alpha_i$; $\alpha_{j_k} = \alpha_j$ and $\alpha_{j_p} = \alpha_{i_{p+1}}$ for $p \le k-1$. Of course, an (α_i,α_j)-run is simply a path in the graph of A. The corresponding sequence of actions $(R_1,\ldots,R_k)$ will be called an (α_i,α_j)-<u>symbolic execution</u> (abbreviated: s-execution). Let $\mathrm{Exec}(\alpha_i,\alpha_j)$ denote the set of all the (α_i,α_j)-s-executions in A. The (α_i,α_j)-<u>resulting relation</u> is defined as follows:

$$\mathrm{Res}(\alpha_i,\alpha_j) = \cup\{R_1\circ\ldots\circ R_k \ (R_1,\ldots,R_k) \in \mathrm{Exec}(\alpha_i,\alpha_j)\} \qquad (3.2)$$

This is of course the input-output relation of A under the assumption that α_i is the input label and α_j is the output label. Indeed, $d_1 \mathrm{Res}(\alpha_i,\alpha_j) d_2$ iff there exists an (α_i,α_j)-s-execution $(R_1,\ldots,R_k)$ such that $d_1 R_1\circ\ldots\circ R_k d_2$. Observe that in any (α_i,α_j)-run the control of the algorithm may pass through α_i and α_j many times.

By the definition of A the label α_1 is assumed to be initial which means that we shall be especially interested in the relations $R(\alpha_1,\alpha_j)$ for $j = 1,\ldots,n$. Nevertheless, we shall also use the other $R(\alpha_i,\alpha_j)$ in our approach.

Now suppose we are considering an algorithm A where α_n has been chosen to be the terminal label and suppose that we have proved

$$\mathrm{Res}(\alpha_1,\alpha_n) = [p(x) \mid x := f(x)]. \qquad (3.3)$$

By the definition of $\mathrm{Res}(\alpha_i,\alpha_j)$ this implies the following about A:

1) for every initial $d \in D$ the algorithm terminates if and only if $p(d)$ is satisfied,

2) for every initial $d \in D$, if the algorithm terminates, then the output value is $f(d)$.

Of course, 2) is the strongest partial correctness property of A and 1) defines exactly the domain of termination. Consequently, (3.3) is the strongest possible total-correctness property of A (see also the next section). Observe that $p(d)$ implies termination and $\sim p(d)$ implies non-termination. In the next section we show how to prove properties of the form (3.3) in a systematic way.

4. An algebraic approach to Floyd's method

In this section we describe and discuss the Floyd's method in terms of our algebra of relations.

Let $A = (D,V,\alpha_1,\mathfrak{I})$ be an arbitrary algorithm with $V = \{\alpha_1,\ldots,\alpha_n\}$ and let $\alpha_i \in V$. By an $\underline{\alpha_i}$-invariant of A we shall mean any relation $Q \in \mathrm{Rel}(D)$ which satisfies

$$\mathrm{Res}(\alpha_1,\alpha_i) \subseteq Q \tag{4.1}$$

The inclusion (4.1) says, that given an arbitrary input d, if this input leads to some output d_1 at α_i (i.e. if $d\,\mathrm{Res}(\alpha_1,\alpha_i)d_1$), then the input d and the output d_1 are in the relation Q (i.e. dQd_1). Of course, Q is simply a Floyd assertion corresponding to the cutpoint α_i.

Now, consider a vector of relations $(Q_1,\ldots,Q_n)$ such that each Q_i is an α_i-invariant of A. This vector will be called a consistent vector of invariants (abbreviated CVI) iff

$$Q_i\mathrm{Res}(\alpha_i,\alpha_j) \subseteq Q_j \quad \text{for } i,j \le n. \tag{4.2}$$

The following theorem describes the Floyd method in our terms:

Theorem 1. A vector of relations $(Q_1,\ldots,Q_n)$ is a CVI iff

1) $R_{1i} \subseteq Q_i$ for $i \le n$
2) $Q_iR_{ij} \subseteq Q_j$ for $i,j \le n$ □

Proof. The "if" part. Let $(R_{i_1j_1},\ldots,R_{i_mj_m}) \in \mathrm{Exec}(\alpha_k,\alpha_p)$. By 1) and 2) we get therefore

$$R_{i_1j_1}\circ\ldots\circ R_{i_mj_m} \subseteq Q_{j_1}R_{i_2j_2}\circ\ldots\circ R_{i_mj_m} \subseteq \ldots \subseteq Q_{j_m} = Q_p$$

If this is true for any execution in Exec(α_p,α_k) it must also be true for Res(α_k,α_p) (cf. (3.3)). This proves (4.1). The proof of (4.2) is similar.

The "only if" part. Since the one-element sequence (R_{1i}) is an element of Exec(1,i) we have

$$R_{1i} \subseteq \text{Res}(\alpha_1,\alpha_i)$$

which proves 1) by (4.1). The proof of 2) by (4.2) is similar. □

Observe that the inclusions 1) and 2) are Floyd's verification conditions. In particular, 1) corresponds to the verification condition with the input predicate in the premise. In this case the input predicate is true everywhere in D, which means that we are considering all possible inputs. If we wish to restrict the set of inputs to these satisfying some predicate p(x), we have to modify the algorithm A by adding new label α_0 and new instruction (α_0,[p(x)],α_1) and by assuming that α_0 is the initial label in the new algorithm. In this case we shall have, of course, Res(α_0,α_i) = [p(x)]Res(α_1,α_i) for all $1 \le i \le n$.

Now, consider the set of all CVI's. It is not difficult to prove that this set is closed under arbitrary (componentwise) union and intersection. In particular, therefore, the greatest and the least element in this set must exist. The greatest element is of course the vector

$(D\times D,\ldots,D\times D)$, which is - evidently - of no interest for the purpose of proving correctness. The least element is $(\mathrm{Res}(\alpha_1,\alpha_1),\ldots,\mathrm{Res}(\alpha_1,\alpha_n))$. The proof is very simple. This vector is a VCI since $\mathrm{Res}(\alpha_1,\alpha_i)\mathrm{Res}(\alpha_i,\alpha_j) = \mathrm{Res}(\alpha_1,\alpha_j)$ and it is smaller than any other CVI by (4.1).

The least CVI is of course of particular interest for the proofs of correctness since it offers the set of strongest Floyd's assertions. Moreover, by the definition of $\mathrm{Res}(\alpha_1,\alpha_i)$, it defines unambiguously the domain of termination. We shall show below how to get the $\mathrm{Res}(\alpha_1,\alpha_i)$ relations expressed in terms of R_{ij}'s and the operations of union, composition and iteration.

Lemma 1. A vector of relations $(Q_1,\ldots,Q_n)$ is a CVI iff it satisfies the following set of inclusions:

$$Q_1R_{1i} \cup \ldots \cup Q_nR_{ni} \cup R_{1i} \subseteq Q_i \quad \text{for } i \leq n \tag{4.3}$$

□

Proof is immediate by Theorem 1.

Theorem 2. The vector of relations $(\mathrm{Res}(\alpha_1,\alpha_1),\ldots,\mathrm{Res}(\alpha_1,\alpha_n))$ is the least solution of the following set of equations:

$$\begin{aligned} X_1 &= X_1R_{11} \cup \ldots \cup X_nR_{n1} \cup R_{11} \\ &\ldots \\ X_n &= X_1R_{1n} \cup \ldots \cup X_nR_{nn} \cup R_{1n} \end{aligned} \tag{4.4}$$

□

Proof. The set (4.4) results from (4.3) by replacing inclusions by equalities. It is a well known mathematical fact (see Tarski [10]) that the least solutions of (4.4) and (4.3) are the same. □

Now, we shall describe a systematic way of solving (4.4) and getting the (least) solution expressed by the coefficients R_{ij} and the operations "∪", "∘" and "*". This way is based on two following variable-elimination transformations:

1) Substitution: The substitution of $X_1R_{1i} \cup \ldots \cup X_nR_{n1} \cup R_{1i}$ for an arbitrary occurrence of X_i on the right side of (4.4).

2) Iteration: The replacement of the equation

$$X_i = X_1R_{1i} \cup \ldots \cup X_nR_{ni} \cup R_{1i}$$

by the equation

$$X_i = (X_1R_{1i} \cup \ldots \cup X_{i-1}R_{i-1i} \cup X_{i+1}R_{i+1i} \cup \ldots \cup X_nR_{ni} \cup R_{1i})R^*_{ii}$$

Each of these transformations is applicable to any set of equations like (4.4) and yields another set of equations of the same form. As can be proved (see Blikle [3]) the new set of equations has exactly the same least solution as the former. To solve a given set we keep applying our transformations as long as there are some unknowns (variables) on the right side.

Example 1. Consider the set of equations corresponding to the program of Fig.2 (empty components are of course omitted):

$$
\begin{aligned}
X_1 &= \phi \\
X_2 &= X_6 F \cup R \\
X_3 &= X_2[p(x)] \cup X_5[\sim q(x)] \\
X_4 &= X_2[\sim p(x)] \\
X_5 &= X_3 P \\
X_6 &= X_5[q(x)] \\
X_7 &= X_4 Q
\end{aligned}
\tag{4.5}
$$

Observe that the number of components in the i-th equation equals the number of arrows entering the label α_i. Therefore, $X_1 = \phi$ in our case.

Now, suppose that we only wish to find $Res(\alpha_1,\alpha_7)$. In such a case the process of solving (4.5) can be simplified by omitting some unnecessary equations. First, let us perform substitutions for X_6, X_5 and X_4:

$$
\begin{aligned}
X_1 &= \phi \\
X_2 &= X_3 P[q(x)]F \cup R \\
X_3 &= X_2[p(x)] \cup X_3 P[\sim q(x)] \\
X_7 &= X_2[\sim p(x)]Q
\end{aligned}
$$

This set of equations corresponds precisely to the "reduced" algorithm with the cutpoints $\alpha_1,\alpha_2,\alpha_3,\alpha_7$. As we see, therefore, the process of solving (4.4) is equivalent - in a sense - to the process of removing cutpoints from the program. Now, by the iteration for X_3 we get (we can also omit $X_1 = \phi$):

$$X_2 = X_3P[q(x)]F \cup R$$
$$X_3 = X_2[p(x)](P[\sim q(x)])^*$$
$$X_7 = X_2[\sim p(x)]Q.$$

Therefore,

$$X_2 = X_2[p(x)](P[\sim q(x)])^*P[q(x)]F \cup R$$
$$X_7 = X_2[\sim p(x)]Q.$$

Hence by iteration and substitution:

$$X_7 = R([p(x)](P[\sim q(x)])^*P[q(x))F)^*[\sim p(x)]Q.$$

By Theorem 2, the right side of this equation equals $Res(\alpha_1,\alpha_7)$. □

5. Examples of program verification

In this section we discuss three examples of the verifications of programs. The first and the third example were taken from [9] and [8] respectively. This should allow the reader to compare the "traditional" Floyd method with our approach. For a more "practical" example the reader is referred to [4]. In our examples we show two different techniques of calculations. The first is based on the relational notation (2.3) and expresses the input-output functions implicitly (first and second example) the second uses the functional notation (2.1) and expresses the input-output functions explicitly (third example). It is difficult to say at the present stage which of the techniques is more efficient. Most probably a combination of them would be an optimum.

Example 5.1 (integer division [9])

Consider the program given by the flowchart of Fig.3. The corresponding algorithm is defined by the set of instructions

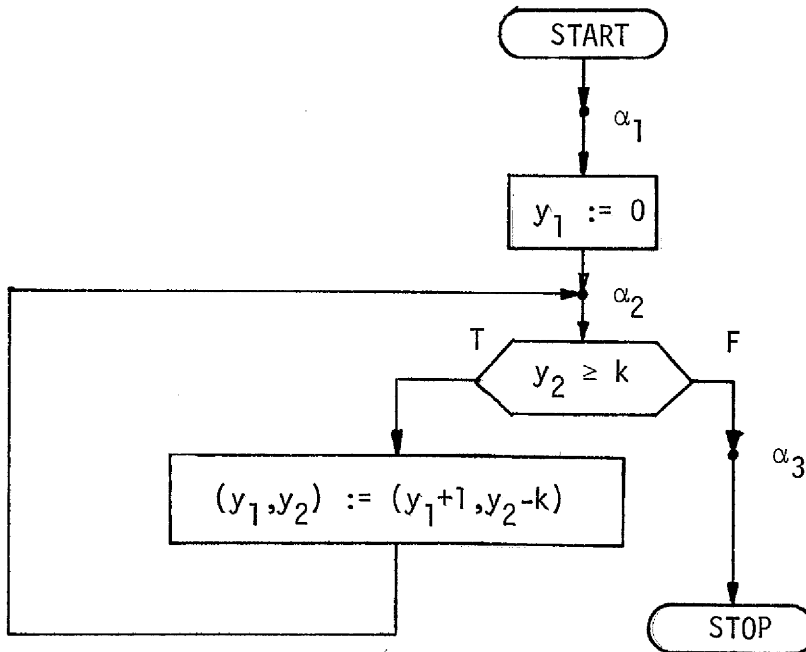

Fig. 3

(note that we use here simultaneous assignment statements):

$$(\alpha_1,[y_1 := 0],\alpha_2)$$
$$(\alpha_2,[y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)],\alpha_2)$$
$$(\alpha_2,[y_2 < k],\alpha_3)$$

where $[y_1 := 0]$ stays of course for $[(y_1,y_2) := (0,y_2)]$. The corresponding set of equations is:

$$X_1 = \phi$$
$$X_2 = X_1[y_1 := 0] \cup X_2[y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)] \cup [y_1 := 0]$$
$$X_3 = X_2[y_2 < k].$$

Therefore, we get immediately

$$\mathrm{Res}(\alpha_1,\alpha_2) = [y_1 := 0][y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)]^*$$
$$\mathrm{Res}(\alpha_1,\alpha_3) = \mathrm{Res}(\alpha_1,\alpha_2)[y_2 < k].$$

Now we have to transform the formulas above into an explicit form $[R|p(\bar{x},x)]$. To do so we shall use the rules for calculation described in Sec.2. In the following we shall assume that y_1, y_2 and k range over arbitrary integers. Comments in parentheses explain the performed transformations:

$\mathrm{Res}(\alpha_1,\alpha_2) =$

(the definition of "*")

$$= [y_1 := 0](I \cup \bigcup_{n=1}^{\infty}[y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)]^n) =$$

(the distributivity of "∘")

$$= [y_1 := 0] \cup \bigcup_{n=1}^{\infty}[y_1 := 0][y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)]^n =$$

(the rule 5) of (2.2) and composition with $[y_1 := 0]$)

$$= [y_1 := 0] \cup \bigcup_{n=1}^{\infty}[y_2-(n-1)k \geq k|(y_1,y_2) := (n,y_2-nk)] =$$

(the same in the relational notation)

$$= [y_1 := 0] \cup \bigcup_{n=1}^{\infty}[R|\bar{y}_2 \geq nk \,\&\, y_1 = n \,\&\, y_2 = \bar{y}_2-nk] =$$

(elimination of "∪")

$$= [y_1 := 0] \cup [R|(\exists n \geq 1)(\bar{y}_2 \geq nk \,\&\, y_1 = n \,\&\, y_2 = \bar{y}_2-nk)] =$$

(the substitution of y_1 for n)

$$= [y_1 := 0] \cup [R|(\exists n \geq 1)(\bar{y}_2 \geq y_1 k \ \& \ y_1 = n \ \& \ y_2 = \bar{y}_2 - y_1 k)] =$$

(quantifier elimination: "$(\exists n \geq 1)(y_1 = n)$" is equivalent to "$y_1 \geq 1$")

$$= [y_1 := 0] \cup [R|\bar{y}_2 \geq y_1 k \ \& \ y_1 \geq 1 \ \& \ y_2 = \bar{y}_2 - y_1 k] =$$

(the substitution of $y_2 + y_1 k$ for $\bar{y}_2$)

$$= [y_1 := 0] \cup [R|y_2 \geq 0 \ \& \ y_1 \geq 1 \ \& \ \bar{y}_2 = y_2 + y_1 k]$$

The formula above gives us an explicit specification for the strongest Floyd invariant associated with the cutpoint α_2. This invariant says that at any time the control of the program arrives in α_2, one of the two conditions must be satisfied:

1) either the current value of y_1 equals 0 in which case the current value of y_2 equals the initial value of y_2.

2) or the current value of y_1 is greater or equal to 1 in which case the current value of y_2 is non-negative and $\bar{y}_2 = y_2 + y_1 k$ holds.

Observe that the initial value $\bar{y}_1$ does not appear in our formula which actually means that this value is irrelevant for the values of y_1 and y_2 at α_2.

Now, we have one easy step left to get an explicit form of $Res(\alpha_1, \alpha_3)$:

$$Res(\alpha_1, \alpha_3) = Res(\alpha_1, \alpha_2)[y_2 < k] =$$
$$= [y_2 < k|y_1 := 0] \cup [R|0 \leq y_2 < k \ \& \ y_1 \geq 1 \ \& \ \bar{y}_2 = y_2 + y_1 k]$$

This formula gives the complete description of the input-output properties of our program. We shall analyse it in two separate cases:

1) $k \leq 0$. In this case $[R|0 \leq y_2 < k \ \& \ y_1 \geq 1 \ \& \ \bar{y}_2 = y_2+y_1k] = \phi$ since there are no y_2 with $0 \leq y_2 < 0$. Therefore

$$\mathrm{Res}(\alpha_1,\alpha_3) = [y_2 < k|y_1 := 0].$$

This means the following:

i) the program terminates if and only if initially $y_2 < k$ (hence if $y_2 \geq k$ at α_1 then the program does not terminate);

ii) if the program terminates, then the data processing can be described by the assignment $(y_1,y_2) := (0,y_2)$.

2) $k > 0$. In this case

$$\mathrm{Res}(\alpha_1,\alpha_3) = [y_2 < k|y_1 := 0] \cup [R|0 \leq y_2 < k \ \& \ y_1 \geq 1 \ \& \ \bar{y}_2 = y_2+y_1k]$$

The domain of this function equals the domain of the first component - which is $\{(\bar{y}_1,\bar{y}_2)|\bar{y}_2 < k\}$ - plus the domain of the second component - which is $\{(\bar{y}_1,\bar{y}_2)|(\exists y_1,y_2)(0 \leq y_2 < k \ \& \ y_1 \geq 1 \ \& \ \bar{y}_2 = y_2+y_1k)\}= \{(\bar{y}_1,\bar{y}_2)|\bar{y}_2 \geq k\}$. Consequently, the domain of $\mathrm{Res}(\alpha_1,\alpha_3)$ equals $\mathrm{Int}\times\mathrm{Int}$ where Int denotes the set of all the integers. In terms of our program we have therefore the following:

i) the program always terminates,

ii) if the program terminates then the terminal value of y_1 is the required integer quotient and the terminal y_2 is the corresponding remainder.

This accomplishes the full proof of the strongest total correctness (and nontermination) of our program. Of course, our method can also be used to prove partial correctness of the program, in which case the calculations will be much simpler. E.g. if we wish to show that the relations

$$Q_2 = [y_1 := 0] \cup [R|y_2 \geq 0 \;\&\; y_1 \geq 1 \;\&\; \bar{y}_2 = y_2+y_1k]$$

and

$$Q_3 = [y_2 < k|y_1 := 0] \cup [R|0 \leq y_2 < k \;\&\; y_1 \geq 1 \;\&\; \bar{y}_2 = y_2+y_1k]$$

are Floyd's assertion in our program, then all we have to do is to show that they satisfy the following inclusions (Lemma 1):

$$Q_2[y_2 \geq k|(y_1,y_2) := (y_1+1,y_2-k)] \cup [y_1 := 0] \subseteq Q_2$$

$$Q_2[y_2 < k] \subseteq Q_3$$

This is an easy task, however, if we usethe rule 5) of (2.5).

Example 5.2 (integer square root)

Consider the program given by the flowchart of Fig.4. This example will be discussed with fewer comments. We shall only show the most

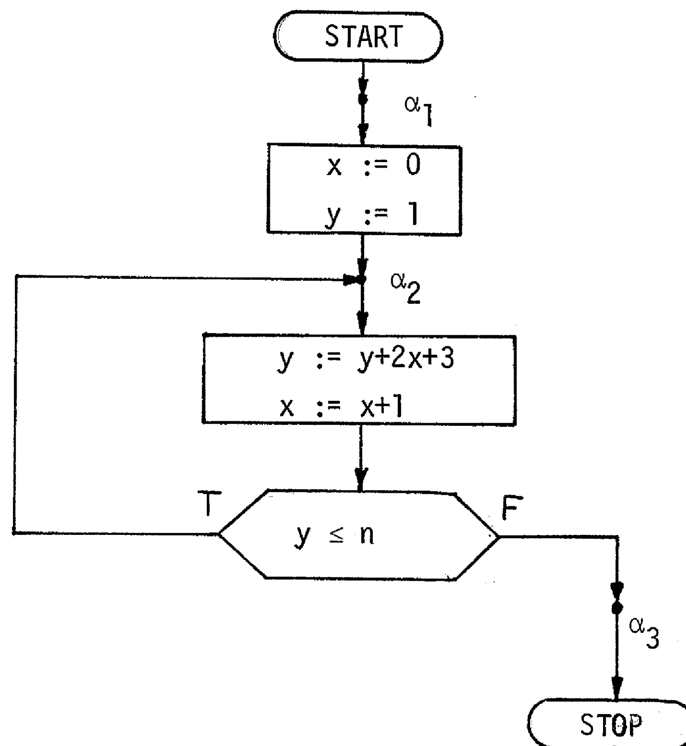

Fig. 4

important points. First of all, we can omit the specification of the Mazurkiewicz algorithm and proceed directly to the equations:

$$X_1 = \phi$$
$$X_2 = X_1R_{12} \cup X_2R_{22} \cup R_{12}$$
$$X_3 = X_2R_{23}$$

where
$$R_{12} = [(x,y) := (0,1)]$$
$$R_{22} = [(x,y) := (x+1,y+2x+3)][y \le n]$$
$$R_{23} = [(x,y) := (x+1,y+2x+3)][y > n]$$

Therefore, we get

$$Res(\alpha_1,\alpha_3) = R_{12}R_{22}^{*}R_{23}$$

and we proceed to the main step of our analysis:

$$Res(\alpha_1,\alpha_3) = \bigcup_{i=0}^{\infty} R_{12}R_{22}^{i}R_{23} = R_{12}R_{23} \cup \bigcup_{i=1}^{\infty} R_{12}R_{22}^{i}R_{23}$$

We shall consider both components separately. First

$$R_{12}R_{23} = [(x,y) := (1,4)][y > n] = [n < 4|(x,y) := (1,4)]$$

Next

$$\bigcup_{i=1}^{\infty} R_{12}R_{22}^{i}R_{23} =$$

(by induction on i)

$$= \bigcup_{i=1}^{\infty} [(x,y) := (i,(i+1)^2)][y \le n]R_{23} =$$

$$= [R|(\exists i \ge 1)(x = i \;\&\; y = (i+1)^2 \;\&\; y \le n]R_{23} =$$

(the substitution of x for i and the elimination of the quantifier)

$$= [R|x \geq 1 \ \& \ y = (x+1)^2 \ \& \ y \leq n]R_{23} =$$

(the application of 5) in (2.5): here $f^{-1}(x,y) = (x-1,y-2x-1)$)

$$= [R|x-1 \geq 1 \ \& \ y-2x-1 = x^2 \ \& \ y-2x-1 \leq n \ \& \ y > n] =$$

(arithmetical transformations)

$$= [R|x \geq 2 \ \& \ y = (x+1)^2 \ \& \ x^2 \leq n \ \& \ (x+1)^2 > n] =$$

$$= [R|y = (x+1)^2 \ \& \ x \geq 2 \ \& \ x^2 \leq n < (x+1)^2].$$

Therefore,

$$Res(\alpha_1,\alpha_3) = [n < 4|(x,y) := (1,4)] \cup [R|y = (x+1)^2 \ \& \ x \geq 2 \ \& \ x^2 \leq n < (x+1)^2]$$

Now, if $n < 4$, then the second component above is empty since in this case $x \geq 2 \ \& \ x^2 < n$ is never satisfied. This proves that for $n < 4$ the program always terminates and performs $(x,y) := (1,4)$. If, in turn, $n \geq 4$ then the first component is obviously empty and the second component says that the terminal value of x is the integer approximation of $\sqrt{n}$. Since for every $n \geq 2$ such an integer approximation exists and is unique, the second component is a total function. This proves that our program always terminates.

Example 5.3 (hardware integer division [8])

Consider the program given by the flowchart of Fig.5 (for short, we use simultaneous assignments already in the flowchart). This program performs a well known integer-division algorithm which is widely applied in electronical and mechanical arithmometers. It divides y_1 by y_2 and stores the

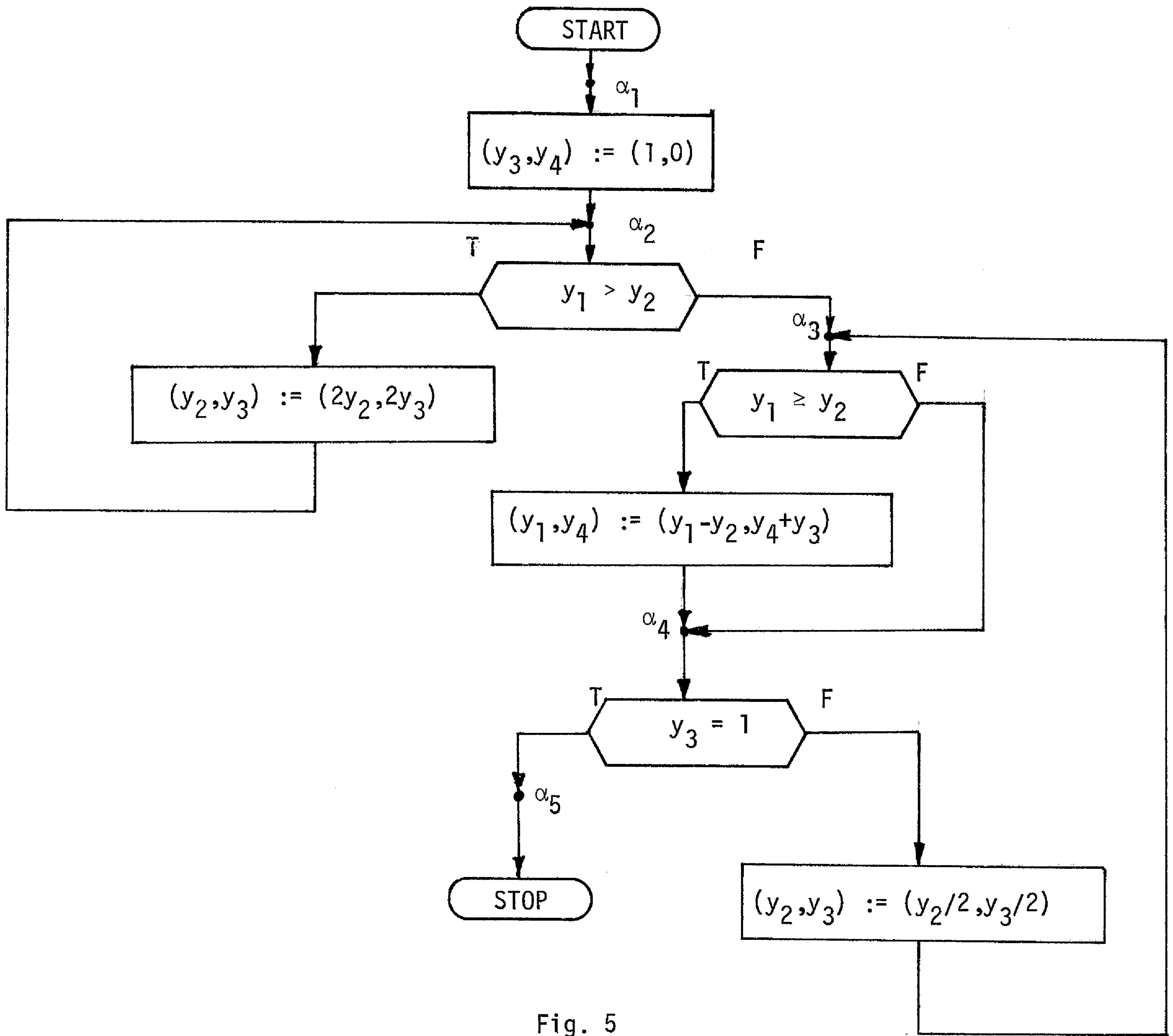

Fig. 5

result $div(y_1,y_2)$ and the remainder $res(y_1,y_2)$ in y_4 and y_1 respectively. It computes the same function as the program of Example 5.1, but now the computation time is much shorter. To explain the idea of this algorithm let $\bar{y}_1$ and $\bar{y}_2$ denote the initial values of y_1 and y_2. At the first step, after $(y_3,y_4) := (1,0)$ has been performed, the algorithm enters the upper loop which is iterated some number $n \geq 0$ of times (this corresponds to the left shift of the binary representation of y_2). At the exit of this loop we have $y_1 = \bar{y}_1 \leq 2^n\bar{y}_2 = y_2$ and $y_3 = 2^n$. Now, the algorithms enters the lower loop where it "tries" to subtract y_2 from y_1. If it succeeds to do so (i.e. if $y_1 \geq y_2$), then this subtraction is "worth" 2^n subtractions of $\bar{y}_2$ from y_1. Consequently, the number 2^n is "loaded" to y_4 and y_1 becomes $res(\bar{y}_1,2^n\bar{y}_2)$. If it does not succeed (i.e. if $y_1 < y_2$) then y_2 and y_3 are divided by 2 (shift right) and the algorithm "tries" to subtract y_2 from y_1 again. In the k+1-th iteration of the lower loop the algorithm "tries" to subtract $2^{n-k}\bar{y}_2$ from $y_1 = res(\bar{y}_1,2^{n-k+1}\bar{y}_2)$ and in the case of success adds 2^{n-k} to the current value of y_4. After exactly n iterations (y_3 must descend from 2^n to 1) the algorithm tries to subtract $\bar{y}_2$ from y_1 and exits the loop at α_4. As we see therefore, this is the well known shift-and-subtract algorithm.

For the sake of our program's verification we shall define the functions "div" and "res" and shall establish some of their properties. We assume that the arguments of these functions range over non-negative integers and that the function themselves satisfy the following four axioms:

A.1 $y_1 = div(y_1,y_2)y_2 + res(y_1,y_2)$ for $y_1 \geq 0$, $y_2 > 0$

A.2 $div(y_1,y_2) \geq 0$ for $y_1 \geq 0$, $y_2 > 0$

A.3 $div(0,0) = 0$

A.4 $0 \leq res(y_1,y_2) < y_2$

It is a well known mathematical fact that the functions satisfying A.1-A.4 exist and are unique. The following properties of these functions will be used in the sequel.

1) $y_1 = y_2 \neq 0 \Rightarrow div(y_1,y_2) = 1 \ \& \ res(y_1,y_2) = 0$

2) $y_1 < y_2 \Rightarrow div(y_1,y_2) = 0 \ \& \ res(y_1,y_2) = y_1$

3) $div(res(y_1,2y_2),y_2) = \begin{cases} 0; & res(y_1,2y_2) < y_2 \\ 1; & res(y_1,2y_2) \geq y_2 \end{cases}$

4) $res(y_1,2y_2) \geq y_2 \Rightarrow res(y_1,2y_2)-y_2 = res(y_1,y_2)$

5) $res(y_1,2y_2) < y_2 \Rightarrow res(y_1,2y_2) = res(y_1,y_2)$

6) for all $n \geq 1$, if $2^{n-1}y_2 < y_1 \leq 2^n y_2$ then

$$div(y_1,y_2) = \sum_{i=0}^{n} div(res(y_1,2^{n-i+1}y_2),2^{n-i}y_2)2^{n-i}$$

These properties are general mathematical facts which must be known in order to design our program. In fact, 3) and 6) describe the main point of our algorithm.

For the sake of calculations that follow we shall slightly modify our notation. First we assume to omit trivial assignment statements $y_i := y_i$. E.g. we shall write $[(y_1,y_4) := (y_1-y_2,y_4+y_3)]$ instead of $[(y_1,y_2,y_3,y_4) := (y_1-y_2;y_2,y_3,y_4+y_3)]$. Next, dealing with longer formulas, we shall write expression (2.1) vertically, i.e.

$$\begin{aligned}
Res(\alpha_1,\alpha_2) &= R_{12}R_{22}^{*} \\
Res(\alpha_1,\alpha_3) &= Res(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^{*} \\
Res(\alpha_1,\alpha_4) &= Res(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^{*}R_{34} \\
Res(\alpha_1,\alpha_5) &= Res(\alpha_1,\alpha_4)R_{45}
\end{aligned}$$

We shall elaborate our $R(\alpha_1,\alpha_i)$'s successively.

$$Res(\alpha_1,\alpha_2) = R_{12} \cup \bigcup_{n=1}^{\infty} R_{12}R_{22}^{n} =$$

(rule 5) of (2.2))

$$= R_{12} \cup \bigcup_{n=1}^{\infty} R_{12}[y_1 > 2^{n-1}y_2 | (y_2,y_3) := (2^n y_2, 2^n y_3)] =$$

$$= R_{12} \cup \bigcup_{n=1}^{\infty} [y_1 > 2^{n-1}y_2 | (y_2,y_3,y_4) := (2^n y_2, 2^n, 0)] \tag{5.1}$$

Now, we proceed to the main step of our verification which deals with the lower loop. Let

$$(y_1,y_2,y_3,y_4) := f(y_1,y_2,y_3,y_4,n,k)$$

denote the following (simultaneous) assignment statement:

$$\begin{aligned}
y_1 &:= res(y_1, 2^{n-k}y_2) \\
y_2 &:= 2^{n-k}y_2 \\
y_3 &:= 2^{n-k} \\
y_4 &:= \sum_{i=0}^{k} div(res(y_1, 2^{n-i+1}y_2), 2^{n-i}y_2)2^{n-i}
\end{aligned} \tag{5.2}$$

We shall prove that for all $k \geq 0$

$$\mathrm{Res}(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^k R_{34} =$$

$$= [y_1 \leq y_2 \ \&\ k=0 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,0,k)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [2^{n-1}y_2 < y_1 \leq 2^n y_2 \ \&\ k \leq n | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,n,k)] \quad (5.3)$$

For $k = 0$ compute first

$$R_{23}R_{34} = [y_1 = y_2 | (y_1,y_4) := (0,y_4+y_3)] \cup [y_1 < y_2]$$

and now

$$\mathrm{Res}(\alpha_1,\alpha_2)R_{23}R_{34} =$$

$$= [y_1 = y_2 | (y_1,y_3,y_4) := (0,1,1)] \cup [y_1 < y_2 | (y_3,y_4) := (1,0)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [y_1 > 2^{n-1}y_2 \ \&\ y_1 = 2^n y_2 | (y_1,y_2,y_3,y_4) := (0,2^n y_2,2^n,2^n)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [y_1 > 2^{n-1}y_2 \ \&\ y_1 < 2^n y_2 | (y_2,y_3,y_4) := (2^n y_2,2^n,0)] =$$

$$= [y_1 = y_2 \ \&\ 0 = 0 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,0,0)] \ \cup$$

$$[y_1 < y_2 \ \&\ 0 = 0 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,0,0)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [y_1 = 2^n y_2 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,n,0)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [2^{n-1}y_2 < y_1 < 2^n y_2 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,n,0)] =$$

$$= [y_1 \leq y_2 \ \&\ 0 = 0 | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,0,0)] \ \cup$$

$$\bigcup_{n=1}^{\infty} [2^{n-1}y_2 < y_1 \leq 2^n y_2 \ \&\ 0 \leq n | (y_1,\ldots,y_4) := f(y_1,\ldots,y_4,n,0)]$$

Now, let (5.3) be satisfied for some $k \geq 0$:

$$\mathrm{Res}(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^{k+1}R_{34} =$$

$$= \mathrm{Res}(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^{k}R_{34}(R_{43}R_{34}).$$

Compute first

$$R_{43}R_{34} =$$

$$= [y_3 \neq 1 \ \& \ y_1 \geq y_2/2 | (y_1,y_2,y_3,y_4) :=$$

$$:= (y_1-y_2/2, y_2/2, y_3/2, y_4+y_3/2)] \cup$$

$$[y_3 \neq 1 \ \& \ y_1 < y_2/2 | (y_2,y_3) := (y_2/2, y_3/2)]$$

Now

$$\mathrm{Res}(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^{k+1}R_{34} =$$

$= [y_1 \leq y_2 \ \& \ k=0 \ \& \ 1 \neq 1 \ \& \ \ldots] \cup$ these two components vanish

$[y_1 \leq y_2 \ \& \ k=0 \ \& \ 1 \neq 1 \ \& \ \ldots] \cup$ because $1 \neq 1$ is false

$$\bigcup_{n=1}^{\infty} [2^{n-1}y_2 < y_1 \leq 2^n y_2 \ \& \ k \leq n \ \& \ 2^{n-k} \neq 1 \ \& \ \mathrm{res}(y_1, 2^{n-k}y_2) \geq 2^{n-k-1}y_2 |$$

$$y_1 := \mathrm{res}(y_1, 2^{n-k}y_2) - 2^{n-k-1}y_2$$

$$y_2 := 2^{n-k-1}y_2$$

$$y_3 := 2^{n-k-1}$$

$$y_4 = \sum_{i=0}^{k} \mathrm{div}(\mathrm{res}(y_1, 2^{n-i+1}y_2), 2^{n-i}y_2)2^{n-i} + 2^{n-k-1}] \cup$$

$$\bigcup_{n=1}^{\infty} [2^{n-1}y_2 < y_1 \leq 2^n y_2 \ \& \ k \leq n \ \& \ 2^{n-k} \neq 1 \ \& \ \mathrm{res}(y_1, 2^{n-k}y_2) < 2^{n-k-1}y_2 |$$

$$y_1 := res(y_1, 2^{n-k}y_2)$$

$$y_2 := 2^{n-k-1}y_2$$

$$y_3 := 2^{n-k-1}$$

$$y_4 := \sum_{i=0}^{k} div(res(y_1, 2^{n-i+1}y_2), 2^{n-i}y_2)2^{n-i}] =$$

(by the properties 1)-5) of "div" and "res")

$$= \bigcup_{n=1}^{\infty}[2^{n-1}y_2 < y_1 \le 2^n y_2 \;\&\; k+1 \le n \;\&\; res(y_1, 2^{n-k}y_2) \ge 2^{n-k-1}y_2 |$$
$$(y_1,\dots,y_4) := f(y_1,\dots,y_4,n,k+1)] \cup$$
$$\bigcup_{n=1}^{\infty}[2^{n-1}y_2 < y_1 \le 2^n y_2 \;\&\; k+1 \le n \;\&\; res(y_1, 2^{n-k}y_2) < 2^{n-k-1}y_2 |$$
$$(y_1,\dots,y_4) := f(y_1,\dots,y_4,n,k+1)] =$$
$$= \bigcup_{n=1}^{\infty}[2^{n-1}y_2 < y_1 \le 2^n y_2 \;\&\; k+1 \le n |$$
$$(y_1,\dots,y_4) := f(y_1,\dots,y_4,n,k+1)]$$

This coincides with (5.3) since $k+1 \ne 0$ and therefore the first component of (5.3) vanishes. To get the formula for $Res(\alpha_1,\alpha_5)$ we compute

$$Res(\alpha_1,\alpha_5) = \bigcup_{k=0}^{\infty} Res(\alpha_1,\alpha_2)R_{23}(R_{34}R_{43})^k R_{34}R_{45} =$$
$$= \bigcup_{k=0}^{\infty}([y_1 \le y_2 \;\&\; k=0 \;\&\; 1=1 | (y_1,\dots,y_4) := f(y_1,\dots,y_4,0,0)] \cup$$
$$\bigcup_{n=1}^{\infty}[2^{n-1}y_2 < y_1 \le 2^n y_2 \;\&\; k \le n \;\&\; 2^{n-k} = 1 |$$
$$(y_1,\dots,y_4) := f(y_1,\dots,y_4,n,k)]) =$$

(since $k \le n \;\&\; 2^{n-1} = 1 \equiv n = k$)

$$= [y_1 \le y_2 \; (y_1,\dots,y_4) := f(y_1,\dots,y_4,0,0)] \cup$$
$$= \bigcup_{n=1}^{\infty}[2^{n-1}y_2 < y_1 \le 2^n y_2 \; (y_1,\dots,y_4) := f(y_1,\dots,y_4,n,n) =$$

(the application of (5.2) and of property 6))

$$
\begin{aligned}
= [y_1 \le y_2 | \; & y_1 := res(y_1, y_2) \\
& y_2 := y_2 \\
& y_3 := 1 \\
& y_4 := div(y_1, y_2)] \cup
\end{aligned}
$$

$$
\begin{aligned}
\bigcup_{n=1}^{\infty} [2^{n-1} y_2 < y_1 \le 2^n y_2 | \; & y_1 := res(y_1, y_2) \\
& y_2 := y_2 \\
& y_3 := 1 \\
& y_4 := div(y_1, y_2)]
\end{aligned}
$$

Now observe that

$$
\begin{aligned}
& [y_1 \le y_2] \cup \bigcup_{n=1}^{\infty} [2^{n-1} y_2 < y_1 \le 2^n y_2] = \\
& = [y_1 \le y_2] \cup [y_2 < y_1 \; \& \; y_2 \ne 0] = \\
& = [y_1 = y_2 = 0 \vee y_2 \ne 0]
\end{aligned}
$$

Finally,

$$
\begin{aligned}
Res(\alpha_1, \alpha_5) = [y_1 = y_2 = 0 \vee y_2 \ne 0 | \; & y_1 := res(y_1, y_2) \\
& y_2 := y_2 \\
& y_3 := 1 \\
& y_4 := div(y_1, y_2)]
\end{aligned}
\tag{5.4}
$$

This equation describes completely the input-output properties of our program. It says in particular that the program terminates for all (y_1, y_2, y_3, y_4) except the case where $y_1 \ne 0$ & $y_2 = 0$. In the latter case the program does not terminate. Observe that the equations (5.1)-(5.4) give quite complete documentation of our program.

6. Final remarks

As mentioned in the introduction the method was tested on several hardware microprograms of a floating-point arithmetical unit of a computer [4]. This proved that the method is applicable to at least some "practical" examples (the largest program consisted of 20 assignment statements and 20 tests), even in the case all the calculations are performed manually. Despite this positive experience, the method is not free of many technical inconveniences. First of all if the number of variables in the program is large, then the formulas $[R|p(\bar{x},x)]$ become long and therefore cumbersome for manual calculations. Also large programs offer technical problems since the corresponding set of equations is usually as large as the program itself. On the other hand, our approach is compatible with the techniques of structuralization. In fact, once we can structure our program into smaller modulus we can analyse each of them separately and then represent each of them by the corresponding relation $Res(\alpha_1,\alpha_n)$. Since these relations provide complete descriptions of input-output properties they are always sufficiently strong to perform the analysis on the higher level.

Of course, to deal with really large programs a computerised system is required. Our present experience shows that such a system is not unrealistic. No doubt that this must be an interactive system since many heuristics certainly cannot be eliminated from the approach. What the system can be expected to do (at the present stage) is of course the transformation of programs into equations and the solving of these equations. It is also hopeful that many simplifications of formulas (e.g. substitutions, elimination of quantifiers, standard induction applying 5) of (2.2))

could be performed by such a system. What the system should leave for humans are certainly the nonstandard induction proofs (like in Examples 5.2 and 5.3) and the choice of functions (or relations) which are going to be used in these proofs (like "div" and "res" in Example 5.3). Also the general strategy of the process of verification cannot be expected to be established by a system.

References

[1] A. Blikle, "Complex iterative systems", Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 20 (1972), pp.57-61.

[2] A. Blikle, "Proving programs by δ-relations", Formalization of Semantics of Programming Languages and Writing of Compilers (Proc. Symp. Frankfurt am oder 1974), Elektronische Informationsverarbeitung und Kybernetik (to appear in 1976).

[3] A. Blikle, "An analysis of programs by algebraic means", The Mathematical Foundations in Computer Science (Proc. MFCS Semester of the Int. Math. S. Banach Center in Warsaw, 1974), Polish Scientific Publishers, Warsaw (to appear in 1976).

[4] A. Blikle and S. Budkowski, "An algebraic program-verification method applied to microprograms", University of Waterloo Research Report CS-76-31, June 1976.

[5] A. Blikle and A. Mazurkiewicz, "An algebraic approach to the theory of programs, algorithms, languages and recursiveness", Mathematical Foundations of Computer Science (Proc. Symp. Warsaw-Iablonna 1972), Warsaw 1972.

[6] R.W. Floyd, "Assigning meanings to programs", Mathematical Aspects of Computer Science (Proc. Symp. Appl. Math. vol.19, I.T. Schwartz (ed.)), American Mathematical Society, New York 1967, pp.19-32.

[7] I. Greif and R. Waldinger, "A more mechanical approach to program verification", Programming Symp., Proc. Colloque Paris, Lecture Notes Computer Sci. 19, (1974), pp.109-119.

[8] S. Katz and Z. Manna, "Logical analysis of programs", Communication of the ACM, vol.19, No.4, April 1976, pp.188-206.

[9] Z. Manna, "Mathematical Theory of Computation", McGraw-Hill Book Company, New York, 1974.

[10] A. Tarski, "A lattice-theoretic fixpoint theorem and its applications", Pacific Journal of Mathematics, vol.5 (1955), pp.285-309.