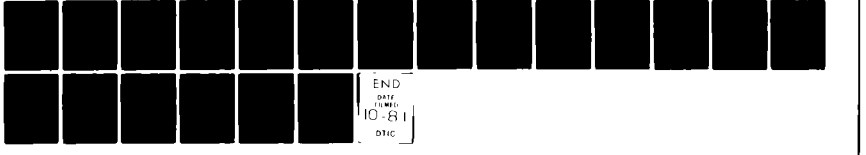
AD-A103 748 RENSSELAER POLYTECHNIC INST TROY NY DEPT OF MATHEMAT--ETC F/G 12/1
SYNTHESIS OF INVARIANT ASSERTIONS FOR ARITHMETICAL PROGRAMS.(U)
JUN 81 S K ABDALI, J VYTOPIL N00014-75-C-1026
UNCLASSIFIED RPI-CS-8101 NL



END
DATE FILMED
10-81
DTIC

(9) Technical Rept.

(14) RPI

Technical Report CS-8101

(6) SYNTHESIS OF INVARIANT ASSERTIONS FOR ARITHMETICAL PROGRAMS.

(10) S. Kamal Abdali
Jan Vytopil

(11) June 1981

(12) 39

Prepared for

U.S. Office of Naval Research
Contract Number (15) N00014-75-C-1026

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By Per Ltr. on File	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

Mathematical Sciences Department
Rensselaer Polytechnic Institute
Troy, New York 12181

DTIC ELECTE SEP 4 1981 S D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

411816

1. INTRODUCTION

Most of the work in mechanical program verification is based on the method of invariant assertions [5]. The automatic discovery of invariant assertions from given program specifications has thus been a matter of much research, e.g. [1,2,3,6,7, 12,13,15]. The general problem of synthesizing invariant assertions is deemed unsolvable [12], and, as Wegbreit [16] implies, even when the problem is solvable, it may require an exponential amount of time in the worst case. Our result is that for the restricted class of arithmetical programs, the problem is solvable as well as quite simple. Informally, arithmetical programs are those in which values and operations range over non-negative integers. Most data types of practical use can be represented by non-negative integers by proper mapping, and hence the class of arithmetical programs is quite large.

As an alternative to the second-order predicate-calculus formalization of flowchart programs, developed by Cooper [2] and Manna (see [9]), we formalize arithmetical flowchart programs in terms of Gödel's classical arithmetical predicates (see [8]) -- a first-order theory containing equality, zero, successor, addition, and multiplication. We then present a construction for the minimal predicate [9] associated with any cut-point in a given arithmetical program. (A minimal predicate for a cut-point is in a sense the strongest invariant assertion at that point.) We also show that two alternative definitions of invariant assertions, namely by minimal predicates [9] and optimal predicates [3] are equivalent; our explicit solution for a minimal predicate also satisfies the definition of optimal predicate.

2. ARITHMETICAL PREDICATES AND PROGRAMS

This section defines and relates together the concepts of arithmetical predicates and arithmetical flowchart programs. Arithmetical predicates are defined following the treatment in [8]. Informally, arithmetical predicates are properly formed formulas which contain: zero, individual variables, successor,

addition, multiplication, equality, logical connectives, quantifiers over individuals. The formulas may also contain other symbols that can be defined in the system. For example, one may abbreviate 0' (the successor of zero), 0'', ... by 1, 2, ..., and $\exists z(x+z')=y)$ by $x<y$, etc. A function f is arithmetical if the predicate $f(x)=y$ is arithmetical.

The notions of 1-recursive schemas and 1-recursive program are defined about the same way as recursive schemas and recursive programs have been defined in the literature (e.g. [9], except that the number of function variables is restricted to one, i.e., the schemas and programs are restricted to a single use of recursion. A 1-recursive program is arithmetical if it is obtained by interpreting a 1-recursive schema over the domain of non-negative integers. It is then shown that the computation associated with an arithmetical 1-recursive program can be stated in terms of an arithmetical predicate.

The result is then extended to flowchart programs in general. A flowchart program is called arithmetical if it is obtained from a flowchart schema under an arithmetical interpretation. By using computation-preserving translations to obtain a 1-recursive program from any flowchart program, it is shown that the computation associated with a flowchart program can be characterized by an arithmetical predicate.

2.1 Basic notation and definitions

The symbols from which our formulas are constructed are the following:

1. Punctuation marks: , ( )
2. Truth symbols: T F
3. Logical symbols: $\sim$ $\supset$ & $\vee$ $\exists$ $\forall$
4. Constants:
   - 2-adic function constants: * +
   - 1-adic function constant: Succ
   - 2-adic predicate constant: =
   - constant: 0
5. Variables:
   - individual variables: $v_1, v_2, v_3, \ldots$

(For simplicity, we may use additional symbols such as x,y,u to denote the formal individual variables v_i.)

We define recursively three classes of expressions as follows:

Definition 1

a) p-terms:

1. 0 is a p-term.
2. Each individual variable is a p-term.
3. If t_1 and t_2 are p-terms, then so are $(t_1 + t_2)$, $(t_1 * t_2)$ and $\mathrm{Succ}(t_1)$.

b) Atomic formulas:

1. T and F are atomic formulas.
2. If t_1 and t_2 are p-terms, then the expression $(t_1 = t_2)$ is an atomic formula.

c) Arithmetical predicates:

1. Each atomic formula is an arithmetical predicate.
2. If R is an arithmetical predicate and x is a variable, then $\sim(R)$, $\exists x(R)$ and $\forall x(R)$ are arithmetical predicates.
3. If R and S are arithmetical predicates, then so are $(R \supset S)$, $(R \,\&\, S)$ and $(R \vee S)$

In defining an atomic formula, term and arithmetical predicate, we use more parentheses than is strictly necessary to indicate the scope of operators. We can omit some of them by employing the usual rank conventions. We list operators in the order of increasing precedence as follows:

$$\supset \quad \& \quad \vee \quad \sim \quad \forall \quad \exists \quad = \quad + \quad * \quad \mathrm{Succ}$$

This allows us to shorten the length of the formulas. Another kind of abbreviation is provided by introducing a new symbol with a method for translating an expression containing the new symbol back into one without it. For example, we abbreviate the expression

$$\text{"}\exists v_3(\mathrm{Succ}(v_3)+v_1) = v_2\text{" by "}v_1 < v_2\text{"}$$

and

"Succ(0)", "Succ(Succ(0))", ... by " 1 ", " 2 ", ...

Definition 2

A predicate p is arithmetical if there exists an arithmetical predicate logically equivalent to p. A function f of n arguments (n>0) is arithmetical if there exists an arithmetical predicate equivalent to $f(v_1,\ldots,v_n) = v_{n+1}$.

The notation next used has been employed by Manna [9, pp. 319-321] to define more general schemas. The 1-recursive schemas used here are syntactically simpler than general schemas, yet, as will be shown later, they define the same class of functions. The syntax for 1-recursive schemas makes use of the following symbols:

1. punctuation marks , ( )
2. definition symbol ←
3. conditional symbols IF THEN ELSE
4. Constants:
   - n-adic function constants f_i^n , $i\geq 1$, $n\geq 0$
   - n-adic predicate constants p_i^n , $i\geq 1$, $n\geq 0$
   - undefined value $\perp$

A 0-adic function constnat f_i^0 is called an individual constant, and a 0-adic predicate constant p_i^0 is called a propositional constant.

5. Variables
   - individual variables $x_1, x_2, \ldots, y_1, y_2, \ldots$
   - output variable z
   - function variable G

Instead of f_i^n or p_i^n, we simply write f_i or p_i when the number n of arguments is clear from the context. For simplicity, we use additional symbols to denote the formal ones, e. g., f, g, h for function constants, p, q for predicate constants, a, b for individual constants, etc. The context will make such useage clear. We also use the vector notation for conciseness. For example, we write $\underset{\sim}{x}$ for a vector of variables $x_1, \ldots, x_n$ for some fixed n.

We define recursively three classes of expressions as follows:

Definition 3

a) s-terms:

1. Each individual variable is an s-term.
2. If $t_1,\ldots,t_n$ and t_1',t_2' are s-terms, then so are

$$f_i(t_1,\ldots,t_n)$$

and

$$\text{IF } p_i(t_1,\ldots,t_n) \text{ THEN } t_1' \text{ ELSE } t_2' .$$

b) conditional terms

1. $\perp$ is a conditional term.
2. Each s-term is a conditional term.
3. If $t_1,t_2,\ldots,t_n$ are s-terms and w_1,w_2 are conditional terms

then

$$\text{IF } p_i(t_1,\ldots,t_n) \text{ THEN } w_1 \text{ ELSE } w_2$$

is a conditional term.

c) 1-recursive schemas

A 1-recursive schema is an expression of the form

$$z=G(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x})) \text{ where}$$
$$G(\underset{\sim}{x},\underset{\sim}{y}) \leftarrow \text{IF } p(\underset{\sim}{x},\underset{\sim}{y}) \text{ THEN } w(\underset{\sim}{x},\underset{\sim}{y}) \text{ ELSE } G(\underset{\sim}{x},\underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{y}))$$

Here $\underset{\sim}{t}(\underset{\sim}{x}),\underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{y})$ are vectors of s-terms, with $\underset{\sim}{t}(\underset{\sim}{x})$ having the same number of components as $\underset{\sim}{y}$, and $w(\underset{\sim}{x},\underset{\sim}{y})$ is a conditional term. The variables $\underset{\sim}{x}$ and $\underset{\sim}{y}$ are called input and program variables, respectively.

Given a 1-recursive schema S, we can specify an interpretation I of the schema in terms of:

1. A set D called the domain of the inerpretation.
2. Assignments to the constants:

To each function constants f_i^n a total function mapping:

$$D^n \rightarrow D$$

To each predicate constant p_i^n a total predicate mapping:

$$D_n \rightarrow (F,T)$$

Note that the individual constant f_i^0 or a_i is assigned some fixed element of D, and the propositional constant p_i^0 is assigned the value T or F.

The pair P= $\langle S,I \rangle$, where S is a 1-recursive schema and I is an interpretation of S is called a 1-recursive program. For given values of $\underset{\sim}{x}$, a 1-recursive program P= $\langle s,I \rangle$ can be executed by constructing the sequence of s-terms $s_0,s_1,\ldots$ as follows:

$s_0=G(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x}))$ after all possible simplifications (see below)

s_{i+1} is obtained from s_i by replacing in s_i the leftmost-innermost occurrence of $G(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x}))$ with

$$\text{IF } p(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x})) \text{ THEN } w(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x})) \text{ ELSE } G(\underset{\sim}{x},\underset{\sim}{t}'(\underset{\sim}{t}(\underset{\sim}{x})))$$

and then applying all possible simplifications.

The simplification rules are

a) Replace each occurence of $f_i(t_1,\ldots t_n)$ and $p_i(t_1,\ldots t_n)$, where t_i is an element of the domain D, by its value.

b) Replace

$$(\text{IF } T \text{ THEN } t_1 \text{ ELSE } t_2) \quad \text{by } t_1$$

and

$$(\text{IF } F \text{ THEN } t_1 \text{ ELSE } t_2) \quad \text{by } t_2$$

If the computation sequence $s_0,s_1,\ldots$ is finite, and the last term s_k is not $\perp$, we say val $\langle P,\underset{\sim}{x} \rangle = s_k$; otherwise val $\langle P,\underset{\sim}{x} \rangle$ is undefined.

Example 1 The following is a 1-recursive schema with one input variable x and two program variables y_1,y_2:

$$S: \quad z=G(x,x,a) \text{ where}$$
$$G(x,y_1,y_2) \Leftarrow \text{IF } p(y_1) \text{ THEN } y_2$$
$$\text{ELSE } G(x_1,f(y_1),g(y_1,y_2))$$

Choose the interpretation I as follows: D={0,1,2,...}, 1 for a, y_1=0 for $p(y_1)$, $f(y_1)=y_1-1$ [define f(0)=0 to make f total], $g(y_1,y_2)=y_1*y_2$. Then we obtain the following 1-recursive program:

P=(S,I): z=G(x,x,1) where

$G(x,y_1,y_2) \Leftarrow$ IF $y_1=0$ THEN 1
ELSE $G(x,y_1-1,y_1*y_2)$.

For the initial value x-3, the computation sequence is as follows: $s_0=G(3,3,1)$, $s_1=G(3,2,3)$, $s_2=G(3,1,6)$, $s_4=G(3,0,6)$, $s_5=6$. Thus, val <P,3> = 6. In general, val <P,x> = x!

2.2 Program characterization by arithmetical predicates

An arithmetical interpretation of a recursive schema S is an interpretation such that

1. The domain D is the set of non-negative integers.
2. The functions and predicates assigned to the function and predicate variables of S are all arithmetical.

An arithmetical 1-recursive program is a 1-recursive program which is obtained from a 1-recursive schema under an arithmetical interpretation. Thus, I and (S,I) used in the example above are an arithmetical interpretation and an arithmetical 1-recursive program, respectively. Now we can state the main result of this section.

Theorem 1 Let P be an arithmetical 1-recursive program. Then val $<P,\underset{\sim}{x}>$ = u is an arithmetical predicate.

Before proving the theorem, we need the following:

Lemma 1 Under an arithmetical interpretation, s-terms and conditional terms become arithmetical functions.

Proof First suppose t is an s-term. We prove by induction that t is arithmetical.

Basis t is some individual variable x. Then t=u is equivalent to x=u which is an arithmetical predicate.

Induction step t is $f(t_1,\ldots,t_n)$ and all $t_i=u_i$ are arithmetical predicates. Then t=u is equivalent to

$$\exists y_1 \ldots \exists y_n (y_1=t_1 \& \ldots \& y_n=t_n \ \& \ f(y_1,\ldots,y_n)=u)$$

which is an arithmetical predicate. Suppose t is IF $p(t_1)$ THEN t_2 ELSE t_3 and we assume that t_1, t_2, and t_3 are arithmetical. Then t-u is equivalent to

$$\exists y:(y=t_1 \& (p(y) \& t_2=u \vee \sim p(y) \ \& \ t_3=u))$$

which is an arithmetical predicate. Next, suppose w is a conditional term. Again we can prove by induction that w is arithmetical.

Basis If w is $\perp$ then w=u is equivalent to

$$\perp = u$$

which is F (since the domain of variable is 0,1,....); therefore it is an arithmetical predicate.

Induction step Same as for s-terms.

Proof of Theorem 1 Let the program P=(S,I) be

(1) $z=G(\underset{\sim}{x},\underset{\sim}{t}(\underset{\sim}{x}))$ where

$G(\underset{\sim}{x},\underset{\sim}{y}) \Leftarrow$ IF $p(\underset{\sim}{x},\underset{\sim}{y})$ THEN $w(\underset{\sim}{x},\underset{\sim}{y})$
ELSE $G(\underset{\sim}{x},\underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{y}))$

Let n be the number of program variables in P (i.e. the number of elements in $\underset{\sim}{y}$). Then n is a fixed number for P. With the ith term in the computation sequence for val $\langle P,\underset{\sim}{x}\rangle$, we can associate an n-tuple $\underset{\sim}{a}^i$ such that the jth component of $\underset{\sim}{a}^i$ gives the value of the program variable $\underset{\sim}{y}_j$ at the ith step of execution in $\langle P,\underset{\sim}{x}\rangle$. Then for val $\langle P,\underset{\sim}{x}\rangle$ = u to be true, there must exist an integer k such that

$$\underset{\sim}{a}^0 = \underset{\sim}{t}(\underset{\sim}{x})$$

$$\underset{\sim}{a}^1 = \underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{a}^0) \ \& \ \sim p(\underset{\sim}{x},\underset{\sim}{a}^0)$$

$$\underset{\sim}{a}^2 = \underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{a}^1) \ \& \ \sim p(\underset{\sim}{x},\underset{\sim}{a}^1)$$

$$\cdots$$

$$\underset{\sim}{a}^{(k-1)} = \underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{a}^{(k-2)}) \ \& \ \sim p(\underset{\sim}{x},\underset{\sim}{a}^{(k-2)})$$

$$u = w(\underset{\sim}{x},\underset{\sim}{a}^{(k-1)}) \ \& \ p(\underset{\sim}{x},\underset{\sim}{a}^{(k-1)})$$

Gödel (see [8]) has introduced the arithmetical function $\beta(u,v,i)$ with the properties:

(i) The predicate $\beta(u,v,i)=w$ is arithmetical,

(ii) For any finite sequence of natural numbers $n_0,n_1,\ldots,n_k$, one can find two integers c,d such that $\beta(c,d,i)=n_i$ for $i=0,1,\ldots,k$.

Note that a suitable definition of is

$$\beta(u,v,i) = u \bmod (i+1)*v+1$$

To use β to encode arbitrarily long, finite sequences of n-tuples, we need 2n constants $c_1,c_2,\ldots,c_n,d_1,d_2,\ldots,d_n$. Let us denote the vectors $\langle c_1,\ldots,c_n\rangle$, $\langle d_1,\ldots,d_n\rangle$, and $\langle\beta(c_1,d_1,i),\ldots,\ \beta(c_n,d_n,i)\rangle$ by $\underset{\sim}{c},\underset{\sim}{d},\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},i)$, respectively, and denote

$$\exists c_1 \ldots \exists c_n,\ \exists d_1 \ldots \exists d_n \text{ by } \exists \underset{\sim}{c}, \exists \underset{\sim}{d} .$$

Then we may write (2) as

$$\begin{aligned}(3)\quad & \exists k \exists \underset{\sim}{c} \exists \underset{\sim}{d} [\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},0)=\underset{\sim}{t}(\underset{\sim}{x}) \\ & \quad \& \forall i [(0<i<k) \supset [\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},i) = \underset{\sim}{t}'(\underset{\sim}{x},\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},i-1)) \\ & \qquad \& \neg p(\underset{\sim}{x},\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},i-1))]] \\ & \quad \& u = w(\underset{\sim}{x},\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},k)) \ \& \ p(\underset{\sim}{x},\underset{\sim}{\beta}(\underset{\sim}{c},\underset{\sim}{d},k))] \ .\end{aligned}$$

Being equivalent to (3), val $\langle P,\underset{\sim}{x}\rangle = u$ is an arithmetical predicate.

We remark that it is only for simplicity that 1-recursive schemas have been defined to have a single output variable. Their definition, as well as Theorem 1, can be extended to the case of any finite number of output variables.

Example 2 As seen previously, the following is an arithmetic 1-recursive program.

P: $z=G(x,x,1)$ where

$G(x,y_1,y_2) \Leftarrow$ IF $y_1=0$ THEN y_2

ELSE $G(x,yl-1,y_1*y_2)$

Here G is arithmetical, and the predicate val(P,x)=u is equivalent to

$$\begin{aligned}\exists k \exists c_1 \exists c_2 \exists d_1 \exists d_2 [\beta(c_1,d_1,0)=x \ & \& \ \beta(c_2,d_2,0)=1 \\ & \& \ \forall i [0<i<k \supset [\beta(c_1,d_1,i)= \beta(c_1,d_1,i-1)-1 \\ & \qquad \& \ \beta(c_2,d_2,i)= \beta(c_1,d_1,i-1)*\beta(c_2,d_2,i-1) \\ & \qquad \& \sim \beta(c_1,d_1,i-1)=0]] \\ & \& \ u = \beta(c_2,d_2,k) \ \& \ \beta(c_1,d_1,k) = 0]\end{aligned}$$

2.3 Arithmetical predicates for flowchart programs

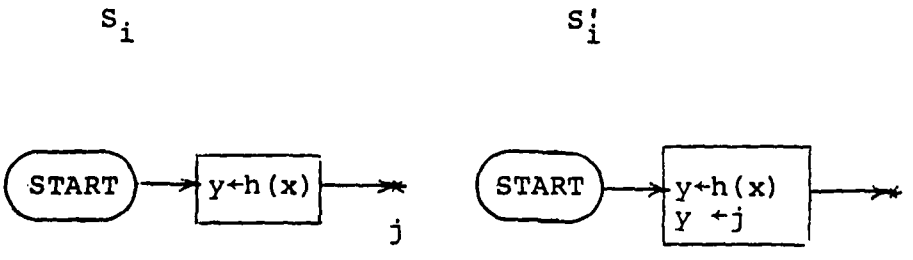
The formalization in terms of arithmetical predicates obtained above for 1-recursive programs will now be extended to flowchart programs in general. The reader is referred to [9] for the definition of flowchart schemas, flowchart programs, and the computation sequences associated with flowchart programs. We define an arithmetical flowchart program to be a flowchart program obtained from a flowchart schema under an arithmetical interpretation.

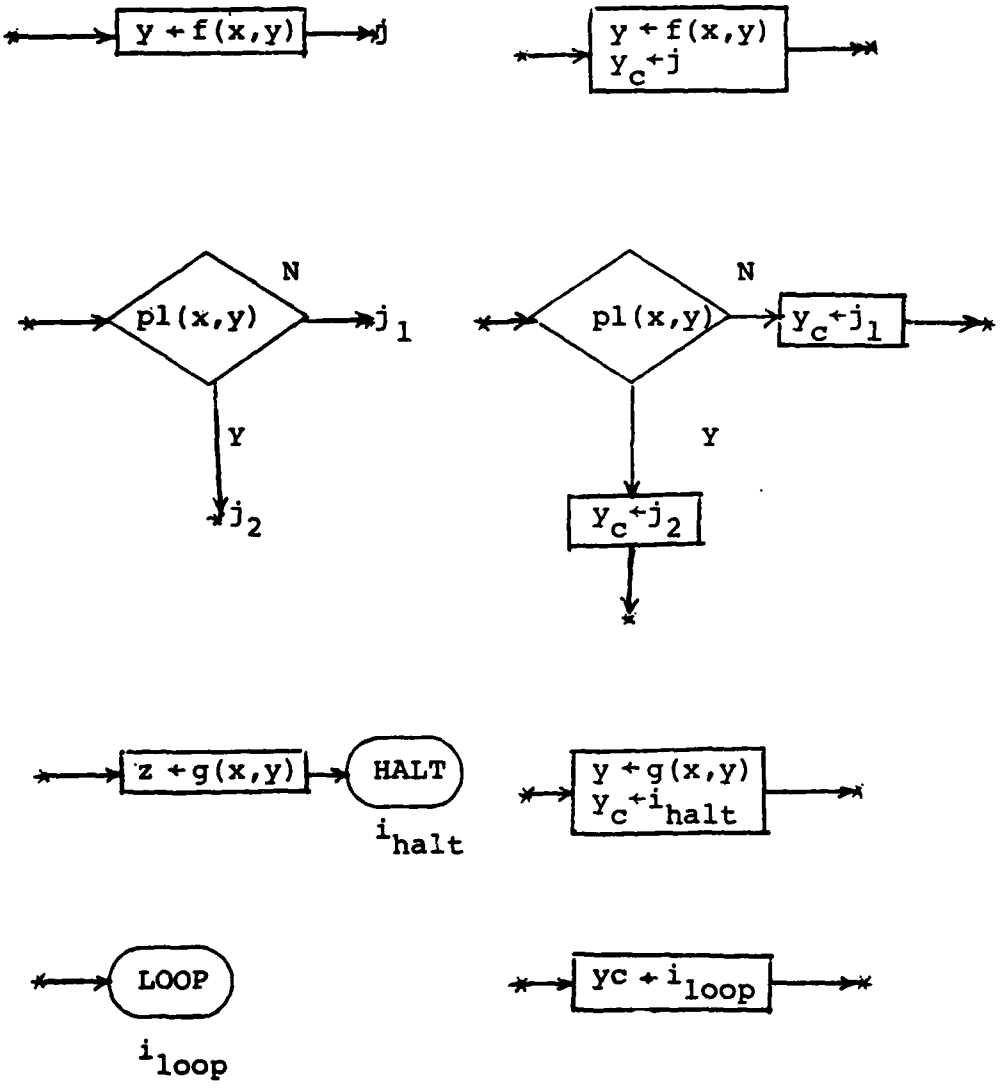
Lemma 2 Every flowchart program can be translated into an equivalent flowchart program with at most one loop.
This is a well known part of computer science folklore but its proof is not commonly available in print. Similar results include: the need of only one mu-operator in recursive function theory (see (see [8], or of only one backward GOTO in PL[11].

Proof We give a transformation in three steps as follows:

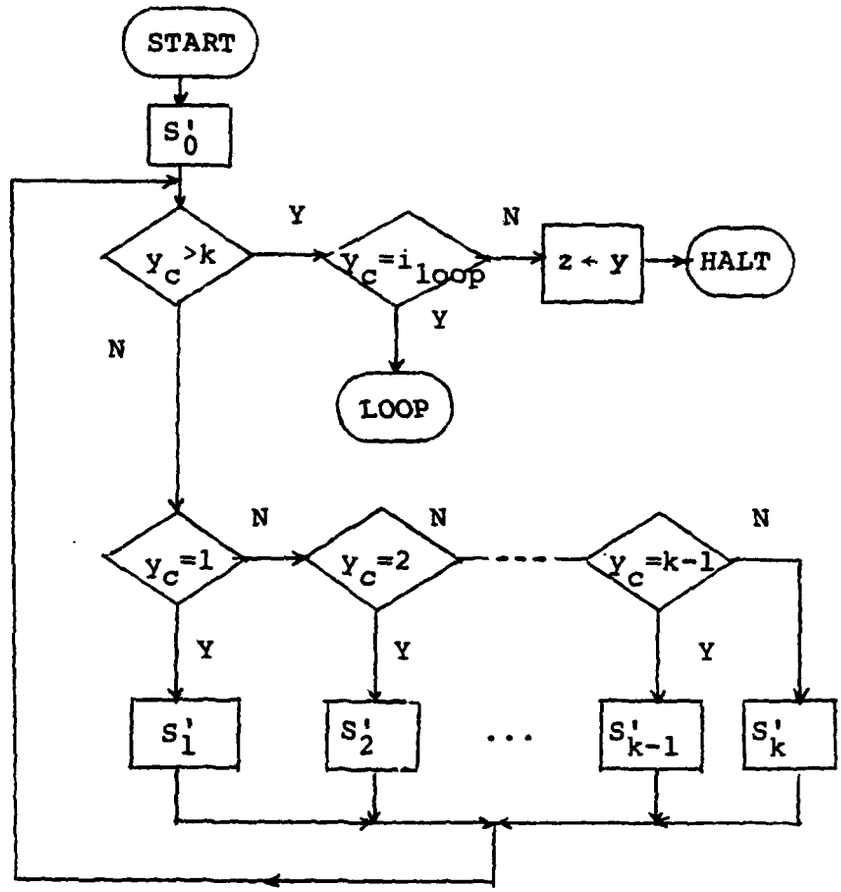
1) Enumerate all statements, viz. associate and integer i (i=1,2,...k) with every assignment statement and every test. To each HALT statement assign a distinct integer, i_{halt}, such that $i_{halt}>k$, and to each LOOP statement an integer i_{loop} such that $i_{loop}>i_{halt}$ for all i_{halt}.

2) From each statement S_i, i=1,2,...,k, obtain the statement S_i' as follows:
(The number j associated with a star denotes that in the original flowchart C the statement following this point was numbered j.)





3) The flowchart C', equivalent to the original flowchart C, is then:



The equivalence of flowcharts C and C' can be proven by induction on the length of their computational sequences.

Lemma 3 Every flowchart program can be translated into an equivalent 1-recursive program.

Proof Using the method of the previous lemma, it is possible to translate a flowchart into a one-loop flowchart. Moreover, the body of this loop does not contain any LOOP statement, and can therefore be translated into an expression of the form

$$\begin{aligned}&\text{IF } y_c = 1 \text{ THEN } S_1' \\ &\quad\text{ELSE IF } y_c = 2 \text{ THEN } S_2' \\ &\qquad\text{ELSE} \\ &\qquad\quad\vdots \\ &\qquad\quad\text{IF } y_c = k-1 \text{ THEN } S_{k-1}' \\ &\qquad\qquad\text{ELSE } S_k'\end{aligned}$$

which is an s-term. The segment of the program following the termination condition ($y_c > k$) can be translated as

$$\text{IF } y_c = i_{loop} \text{ THEN } \perp \text{ ELSE } y$$

which is a conditional term. Now, the flowchart C' can be translated into a 1-recursive program by the algorithm of McCarthy [10] which in fact is not restricted to one loop.

Theorem 2 Let C be an arithmetical flowchart program with input variables $\underset{\sim}{x}$. Then the predicate $\text{val}\langle C,x\rangle = u$ is arithmetical.

Proof The arithmetical predicate equivalent to $\text{val}\langle C,x\rangle = u$ can be constructed in the following way.

1) Translate the flowchart C into flowchart C' containing a single loop, using the method of Lemma 2
2) Translate the flowchart C' into a 1-recursive program P using McCarthy's algorithm.
3) Translate the program P into an arithmetical predicate using the method of Theorem 1.

Let us denote the above predicate by $R_p(\underset{\sim}{x},u)$. This predicate completely characterizes the computation of P on input

$\underset{\sim}{x}$. It can be used to prove various properties of P. Examples:

(i) P halts on input $\underset{\sim}{x}$ if $\exists u R_p(x,u)$ holds.

(ii) Given an "output predicate" $A(\underset{\sim}{x},z)$, P is partially correct with respect to A if $\exists u[R_p(\underset{\sim}{x},u)\ \&\ A(\underset{\sim}{x},u)]$ holds.

We point out that in the work of Cooper and Manna (see [9]), a second-order formula is associated with a flowchart and the proofs of certain properties of programs in their formalization may require second-order methods. In contrast, $R_p(\underset{\sim}{x},u)$ is a first-order predicate.

3. SYNTHESIS OF INVARIANT ASSERTIONS

To prove the partial correctness of a flowchart program, one often needs invariant assertions associated with various points in the program. These are predicates involving the variables in the program such that whenever the control passes through the associated point, the predicate at that points holds. Among the possible assertions at a point in the flowchart program, there exists a strongest assertion defined as follows (see [9]):

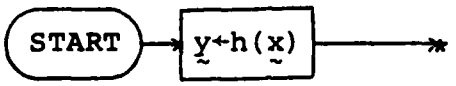
Definition 4 A predicate $q_i(\underset{\sim}{x},\underset{\sim}{y})$ over D -- the domain of interpretation of a program -- is a minimal predicate of cutpoint i in the program, if $q_i(\underset{\sim}{x},\underset{\sim}{y}^*)$ is true for every y^* from D such that during the execution of $<P,x>$, we reach the cutpoint i with $\underset{\sim}{y}=\underset{\sim}{y}^*$, and for no other $\underset{\sim}{y}^{*'}$ is $q(\underset{\sim}{x},\underset{\sim}{y}^{*'})$ true.

Theorem 3 The minimal predicate $q_i(\underset{\sim}{x},\underset{\sim}{y})$ of a cutpoint A in an arithmetic flowchart P is arithmetical.

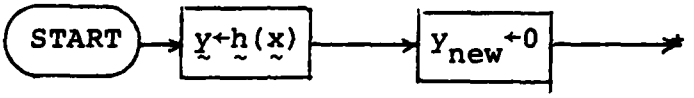
Proof We give a constructive proof as follows. First, from the program P we obtain another program P' with the following property: P' uses a new variable y_{new} whose value at the cutpoint i equals the number of times the cutpoint i is passed during the execution of $<P',\underset{\sim}{x}>$. For any integer n, if during the execution of P, a HALT or LOOP statement is reached before the cutpoint A is passed, then val $<P',<\underset{\sim}{x},n>>$ is undefined. Otherwise, val $<P',<\underset{\sim}{x},n>>$ gives the values of the program variables y when A is reached n times. Next, we use Theorem 2 to obtain the arithmetical predicate, call it $p_i(\underset{\sim}{x},\underset{\sim}{y},\underset{\sim}{n})$, equivalent to

val $\langle P', \langle \underset{\sim}{x}, n \rangle \rangle = \underset{\sim}{y}$. Finally, we show that the required minimal predicate is $\exists n p_i(\underset{\sim}{x}, \underset{\sim}{y}, n)$.

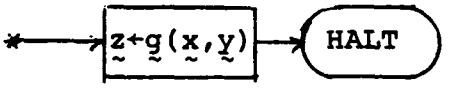
1) From the program P construct a new program P' as follows: Let y_{new}, $\underset{\sim}{z}'$ be new variables distinct from all variables in P, the number of new output variables $\underset{\sim}{z}'$ being equal to the number of program variables $\underset{\sim}{y}$ in P. Then replace



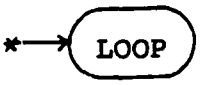
with



and replace all HALT statements

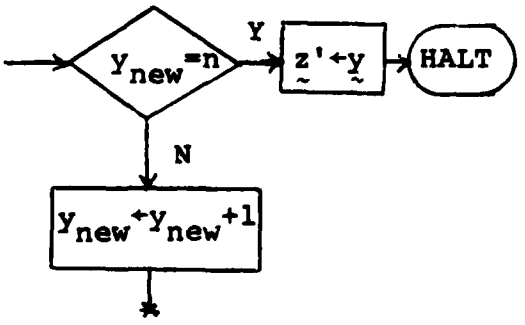


with



Further, replace the cutpoint i

with



2) Using the method of Theorem 2, derive an arithmetical predicate $P_i(\underset{\sim}{x},\underset{\sim}{y},n)$ equivalent to $val\langle P',\langle \underset{\sim}{x},n\rangle\rangle = \underset{\sim}{y}$.

3) It is now needed to show that the predicate $\exists n\ P_i(\underset{\sim}{x},\underset{\sim}{y},n)$ is equivalent to $q_i(\underset{\sim}{x},\underset{\sim}{y})$.

Part 1: $q_i(\underset{\sim}{x},\underset{\sim}{y}^*) \supset \exists n\ P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n)$

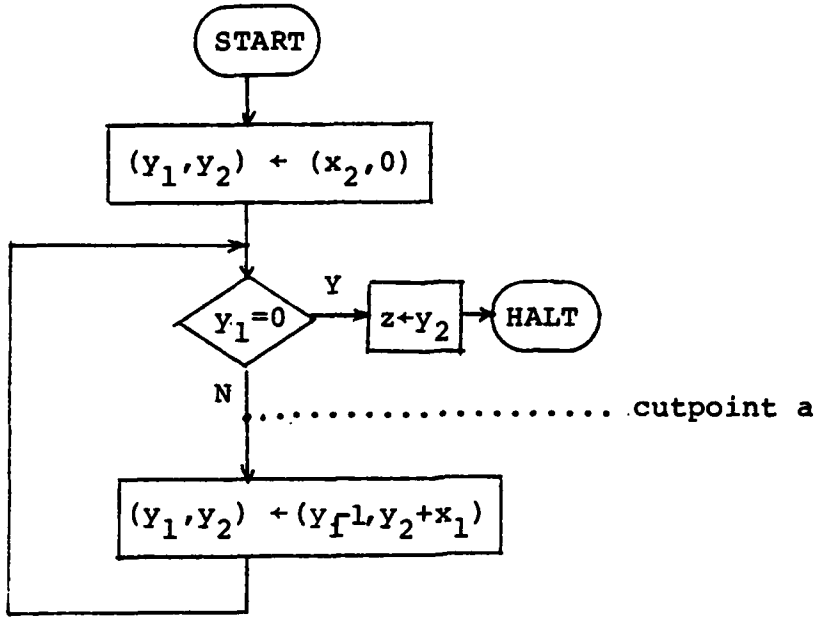
If $q_i(\underset{\sim}{x},\underset{\sim}{y}^*)$ is true, then from Definition 4 it follows that during the execution of $\langle P,\underset{\sim}{x}\rangle$ the cutpoint i was reached, say n* times, and the value of y was y*. But then $\exists n\ P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n)$ is true because there exists an n, namely n*, such that $P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n^*)$ is true.

Part 2: $\exists n\ P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n) \supset q_i(\underset{\sim}{x},\underset{\sim}{y}^*)$

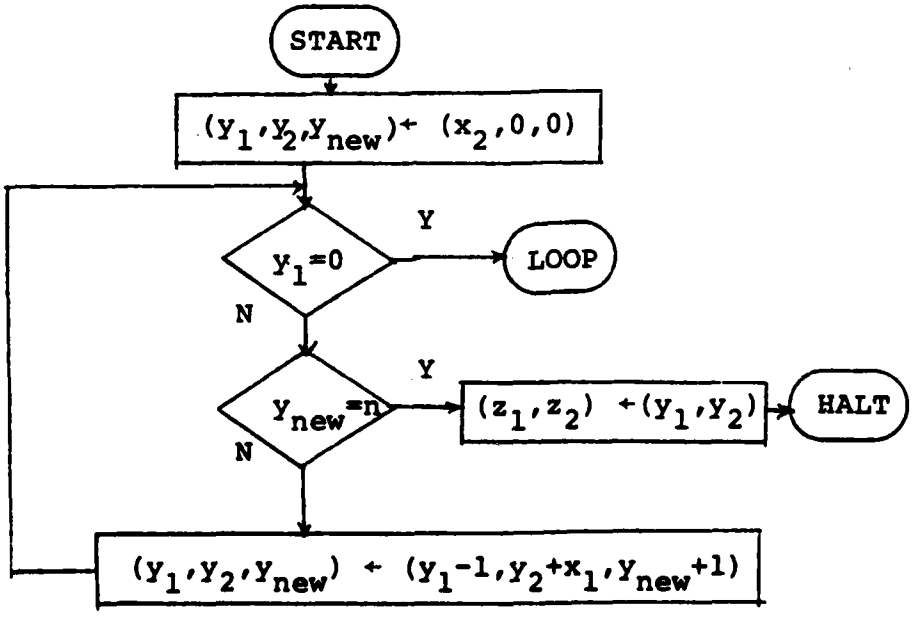
Assume $\exists n\ P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n)$ is true and let n* be the minimum n such that $P_i(\underset{\sim}{x},\underset{\sim}{y}^*,n)$ is true. Then $q_i(\underset{\sim}{x},\underset{\sim}{y})$ is true because there is a point in the execution of $\langle P,\underset{\sim}{x}\rangle$ when i is reached and $\underset{\sim}{y}$ is equal to $\underset{\sim}{y}^*$, namely when we reach the cutpoint i n*-time.

As indicated by the above theorem, first-order predicates suffice for expressing invariant assertions for arithmetic programs, and these assertions can actually be generated mechanically. Also note that the Cooper-Manna second-order formulas [9] associated with programs contain existential quantifiers over predicate variables, having a prefix of the form $\exists Q_1 \exists Q_2 \exists Q_3 \ldots$ The appropriate Q_i to satisfy such formulas are indeed the minimal predicates. By substituting explicit solutions obtained above, we can eliminate the second-order prefixes from Cooper-Manna formulas. Of course, such a reduction to first-order formulas is quite involved. It is also unnecessary, since Theorem 1 furnishes equivalent first-order formulas directly.

Example 3 We will construct the minimal predicate for the cut-point a in the following flowchart program.

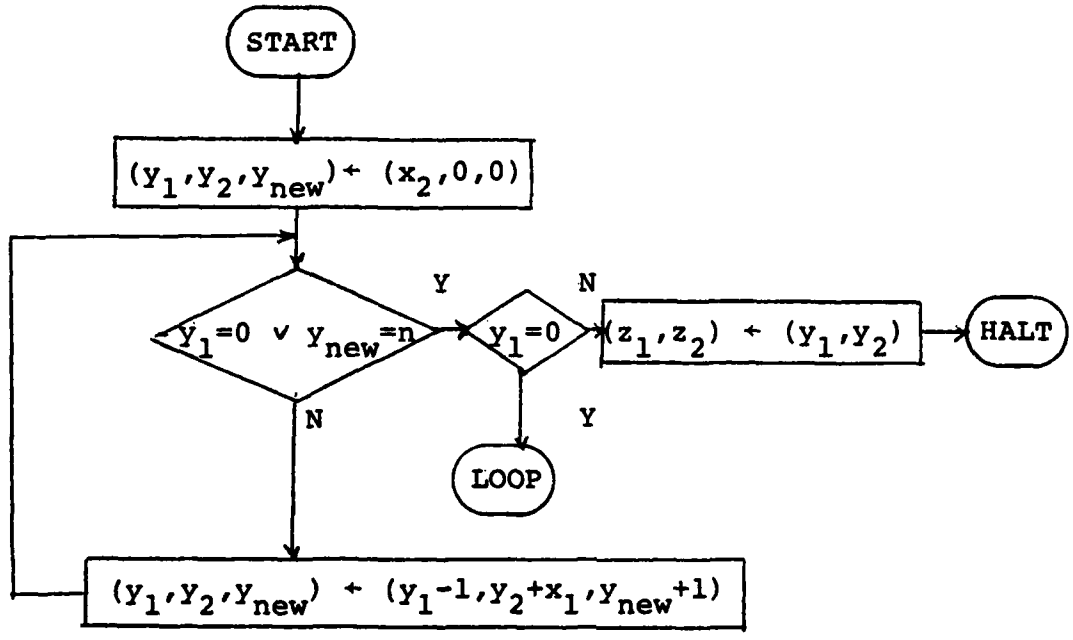


Applying the transformations described in the proof of the Theorem 3, we obtain the following program:



As the next step, we can use the method of Theorem 2 to obtain

a one-loop flowchart with no LOOP statement in (or attached to) the loop body. But this method is unnessary for our simple example. The flowchart already has only one loop, and all we need is to separate the LOOP statement from the body of the loop. So using an intuitively correct transformation, we transform the above flowchart into:



As the next step, using McCarthy's algorithm, the equivalent 1-recursive program is obtained:

$$\langle z_1,z_2\rangle = F(x_1,x_2,x_2,0,0)$$

$$F(x_1,x_2,y_1,y_2,y_{new}) \Leftarrow \text{IF } (y_1=0 \vee y_{new}=n)$$

$$\text{THEN IF } y_1=0 \text{ THEN } \perp \text{ ELSE } \langle y_1,y_2\rangle$$

$$\text{ELSE } F(x_1,x_2,y_1-1,y_2+x_1,y_{new}+1)$$

The minimal predicate at cutpoint a is equivalent to

$$\exists n \; val\langle C,\langle x,x_2,n,x_2,0,0\rangle\rangle = \langle y_1,y_2\rangle$$

and is given by:

$$\exists n \exists k \exists c_1 \exists c_2 \exists c_3 \exists d_1 \exists d_2 \exists d_3$$
$$[\beta(c_1,d_1,0)=x_2 \;\&\; \beta(c_2,d_2,0)=0 \;\&\; \beta(c_3,d_3,0)=0$$
$$\&\; \forall i[(0<i<k) \supset [\beta(c_1,d_1,i) = \beta(c_1,d_1,i-1)-1$$
$$\&\; \beta(c_2,d_2,i) = \beta(c_2,d_2,i-1) + x_1$$
$$\&\; \beta(c_3,d_3,i) = \beta(c_3,d_3,i) + 1$$
$$\&\; \sim [\beta(c_1,d,i) = 0 \quad \beta(c_3,d_3,i) = n]]]$$
$$\&\; y_1=\beta(c_1,d_1,k) \;\&\; y_2=\beta(c_2,d_2,k) \;\&\; \beta(c_1,d_1,k)\neq 0)]$$

This predicate is the minimal predicate for the cutpoint a, because it describes all the possible values of the variables y_1 and y_2 at a, and is true of no other y_1 and y_2. Using the axioms of the first-order predicate calculus extended with Peano axioms, we can simplify this predicate to

$$y_2 = x_1*(x_2-y_1) \;\&\; 0\leq y_1 < x_2$$

which is clearer and shorter description of the relation between the variables.

4. EQUIVALENCE OF MINIMAL AND OPTIMAL PREDICATES

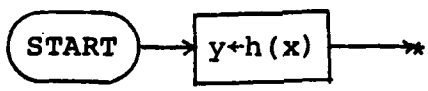
The problem of defining and discovering inductive assertions has been explored by Cousot and Cousot [3] using the fixed-point theory of programs. They describe how a system of logical equations can be associated with a flowchart such that the least solution of this system of equations constitutes optimal invariant assertions. Using Tarski's theorem [14], they show that such assertions always exist. But because of their non-constructive proof, it is not clear how to obtain the solution by a finite process. We prove that minimal predicates are the least solution to the system of equations defining optimal invariant assertions, thus showing how to synthesize optimal invariant assertions mechanically.

Following [3], we define the deductive semantics of a programming language by the rules associating with each statement of a programming language an equation which has first-order

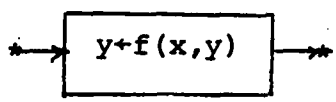
predicates as indeterminates. The deductive semantics of a program is defined by the set of these equations.

The basic elements of a flowchart are:

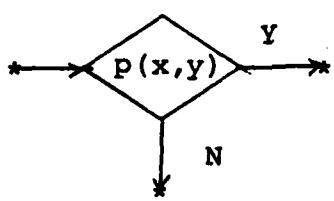
Start statement:



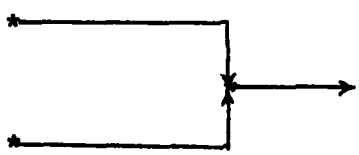
Assignment statement:



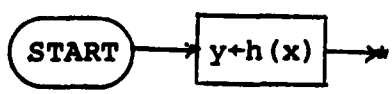
Test statement:



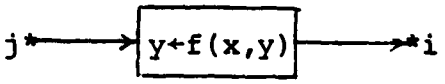
Merge statement:



Given a flowchart we assign cutpoints to this flowchart so that there are only basic elements between cutpoints. Then we associate a predicate $P_i(x,y)$ with each cutpoint i. Finally with each basic element we associate one or two equations as follows:

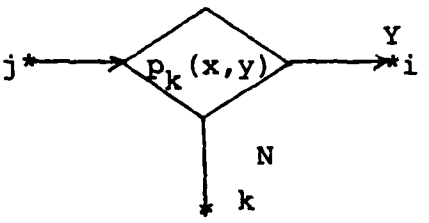
Start statement: 

$$P_0(x,y) = (y=h(x))$$

Assignment statement: 

$$P_i(x,y) = \exists v(P_j(x,v) \ \& \ (y=f(x,v)))$$

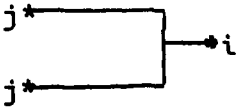
where P_j, respectively P_i, are predicates associated with cutpoints j, respectively i.

Test statement: 

$$P_i(x,y) = p_k(x,y) \ \& \ P_j(x,y)$$

$$P_k(x,y) = \sim p_k(x,y) \ \& \ P_j(x,y)$$

where P_i, P_j, P_k are the predicates associated respectively with cutpoints i,j, and k.

Merge statement: 

$$P_i(x,y) = P_j(x,y) \vee P_k(x,y)$$

where P_i, P_j and P_k are predicates associated with cutpoints i,j, and k.

By applying the rules of deductive semantics to a flowchart, a system of equations of the following form is obtained:

$$\begin{aligned} P_0 &= (y=h(x)) \\ P_1 &= G_1(P_0,P_1,....P_n) \\ P_2 &= G_2(P_0,P_1,....P_n) \\ &... \\ P_n &= G_n(P_0,P_1,....P_n) \end{aligned} \qquad (5)$$

In the above $P_i(x,y)$ has been abbreviated to P_i for convenience.

It is possible to prove that the set of first-order pre-

dicates form a lattice as follows:

$\supset$	as	the ordering relation
F	as	the least element
T	as	the largest element
$\vee$ (respectively $\exists$)	as	join for finite (respectively infinite) operations.
& (respectively $\forall$)	as	meet for finite (respectively infinite) operations.

It is also possible to prove that G_i are continuous functions. It, therefore, follows from Tarski's theorem [14] that the above system (4) has a least solution P_{opt}.

Theorem 4 Let P be a flowchart program with input variable x. Let G be the set of the equations obtained by the method of deductive semantics as described above. Then the minimal predicates $q_1(x,y)$, $q_2(x,y),\ldots,q_n(x,y)$ are the least solution of (4).

Proof

Part 1: $q(x,y) = \langle q_1(x,y), q_2(x,y), \ldots, q_n(x,y)\rangle$ is a fixed-point of G.

By case analysis .

Let the ith equation of the set G of the equation be:

Case 1: $P_1(x,y) = (y=h(x))$ (5)

This equation must have been associated with the START statement. At the cutpoint 1, which follows START and assignment y←h(x), the only possible value of y is h(x). Therefore the minimal predicate $q_1(x,y)$, describing all (and the only) possible values of y at cutpoint 1, is y=h(x). When we substitute this $q_1(x,y)$ in the equation (5) we obtain

$$(y=h(x)) = (y=h(x))$$

Thus the equation is satisfied.

Case 2: $P_i(x,y) = \exists v P_j(x,v)\ \&\ y = f(x,v)$ (6)

This equation must have been associated with the assignment

statement y←f(x,y) which follows cutpoint j. Suppose the minimal predicate at cutpoint j is $q_j(x,y)$. Then, after the execution of the assignment y←f(x,y), all (and the only) possible values of y at cutpoint i are those values of y equal to f(x,v), where v is a possible value of y at j. In other words, if $q_j(x,y)$ is the minimal predicate at cutpoint j, the minimal predicate at cutpoint i must be

$$\exists v(q_j(x,v) \ \& \ y=f(x,v)).$$

If we substitute these q_i and q_j into the equation (6), we obtain

$$\exists v(q_j(x,y) \ \& \ y=f(x,v))) = \exists v(q_j(x,v) \ \& \ y=f(x,v)))$$

Thus the equation (6) will be satisfied.

Case 3: $P_i(x,y) = p_1(x,y) \ \& \ P_j(x,y)$ (7)

This equation must have been associated with the test statement that follows cutpoint j. Suppose all (and the only) possible values of y at cutpoint j are described by $q_j(x,y)$, then in order to arrive at cutpoint i we had to pass test $p_1(x,y)$. Thus all (and the only) possible values at cutpoint i are described by the predicate

$$q_j(x,y) \ \& \ p_1(x,y)$$

Case 4: $P_i(x,y) = P_j(x,y) \vee P_k(x,y)$ (8)

This equation was associated with the merge statement. Let $q_i(x,y)$ and $q_j(x,y)$ describe all (and the only) possible values at cutpoints j and l. To arrive at cutpoint i we had to take a path either coming from cutpoint j or from cutpoint k. Therefore the possible values of variable i at cutpoint i are either those at cutpoint j or k. In other words

$$q_i(x,y) = q_j(x,y) \vee q_k(x,y) \quad .$$

The predicates q_i,q_j and q_k satisfy the equation (8).

Part 2: $q(x,y)$ is the least fixed point of G

The proof is by contradiction. Suppose

$$\sim(q(x,y) \supset Q(x,y))$$

where $Q(x,y)$ is a fixed-point of G. It means there must exist x^*,y^* and cutpoint i such that $q_i(x^*,y^*)$ is true and $Q_i(x^*,y^*)$ is false. We recall from Section 2.3 that if $q_i(x^*,y^*)$ is true, then there exists a statement sequence

$$s_1, s_2, \ldots, s_k$$

such that cutpoint i is reached and the value of y at this point is y^*. Let $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ be the statements executed, and y^*_j be the value of y at the step j. We first note that

$$Q_{i_1}(x^*,y^*_1) \supset Q_{i_2}(x^*,y^*_2) \supset Q_{i_3}(x^*,y^*_3) \supset \ldots$$

This is shown in the following lemma:

Lemma 4 If $j>1$ then

$$Q_{i_{j-1}}(x^*,y^*_{j-1}) \supset Q_{i_j}(x^*,y^*_j)$$

Proof: By case analysis.

Case 1: Suppose $S_{i_{j-1}}$ is an assignment statement. Then $Q_{i_j}(x,y) = \exists v(Q_{i_{j-1}}(x,v) \;\&\; y=f(x,v))$. If $Q_{i_j}(x^*,y^*_j)$ is false, then $Q_{i_{j-1}}(x^*,y^*_{j-1})$ must be false. But we know that there is a v such that $y^*_j = f(x^*,v)$, specifically $v=y^*_{j-1}$. Thus if $Q_{i_j}(x^*,y^*_j)$ is false, $Q_{i_{j-1}}(x^*,y^*_{j-1})$ must also be false.

Case 2: Supose $S_{i_{j-1}}$ is a test statement. Then $Q_{i_j}(x,y) = Q_{i_{j-1}}(x,y) \;\&\; p_k(x,y)$. To reach cutpoint

i, $p_{k_j}(x^*,y^*_j)$ must have been true so if $Q_{i_j}(x^*,y^*_j)$ is false, $Q_{i_{j-1}}(x^*,y^*_j)$ must also be false.

Case 3: Suppose $S_{i_{j-1}}$ is a merge statement. Then $Q_{i_j}(x,y) = Q_{i_{j-1}}(x,y) \vee Q_{i_{j-2}}(x,y)$. Hence, if $Q_{i_j}(x^*,y^*_j)$ is false, then both $Q_{i_{j-1}}(x^*,y^*_{j-1})$ and $Q_{i_{j-2}}(x^*,y^*_{j-2})$ must be false.

Proof of Theorem 4 (continued)

If we assume that there is a cutpoint i_k such that $Q_{i_k}(x^*,y^*_k)$ is false, then from Lemma 4 it follows that $Q_1(x^*,y^*_1)$ is also false. (S_1 must be the first term of the computational sequence.) On the other hand we know that $q_{i_j}(x^*,y^*_j)$ is true for all j. This follows from the definition of q. But $q_1(x,y)$ is y=h(x) and if Q is a fixed-point of F, then we must have

$$Q(x,y) = (y=h(x)) .$$

So on the one hand

$$q1(x^*,y^*_1) = (y^*_1 = h(x^*))$$

is true but on the other hand

$$Q1(x^*,y^*_1) = (y^*_1 = h(x^*))$$

is false so that we have a contradiction. Thus we conclude that our assumption

$$\sim(q(x,y) \supset Q(x,y)) \text{ is false .}$$

5. CONCLUDING REMARKS

This paper has dealt with arithmetical programs in which the data type consists of nonnegative integers and the operations are those of recursive arithmetic. We have characterized the computations of an arithmetical program by an arithmetical predicate. This is a first-order formalization of programs, compared to the well-known second-order formalization (e.g. given in [9]). We have also shown that arithmetical predicates are powerful enough to express invariant assertions, and we have given an algorithm to generate invariant assertions for arithmetical programs. This result should be seen in contrast to Misra [12] where it is argued that the "general problem of generating loop invariants from input specification is impossible." Although it has not been proven formally, the time complexity of our algorithm is a polynominal function of program length. Thus arithmetical programs furnish a clear exception from the implication in Wegbreit[15] that in the general case the synthesis of invariant assertion may require exponential time.

Unfortunately, the invariant assertions generated by our algorithm are quite complex. They do not provide any insight about the computations done by a program, but essentially re-express the computations in terms of arithmetical predicates. Although it is conceivable that these predicates can be simplified by the rules of logic and formal arithmetic, their usefulness in applications such as program verification is doubtful. In any case, our concern in this paper is theoretical -- in showing the solvability of the assertion synthesis problem for arithmetical programs. We would like to emphasize one serious problem, however. Invariant assertions have been defined in the literature purely semantically. For example, minimal predicates (Manna[9]) and optimal predicates (Cousot & Cousot [3]) have been defind in terms of the relations they ought to satisfy. But their intended "practical usefulness" is not reflected in their definition. Although the literature abounds with examples of elegant, concise assertions, there are no clear criteria to specify "nice" assertions. The fact that our synthesized assertions satisfy the letter of the definition, but not the spirit, underscores the inadequacy of the definition.

REFERENCES

1. Basu, S. K. 1980: "A note on synthesis of inductive assertions:, IEEE Trans. Software Engineering, SE-6, 1 (January). pp. 32-39.

2. Cooper, D. C. 1971: "Programs for mechanical program verification", Machine Intelligence 6 (ed. Meltzer, B. and Michie, D.), Edinburgh Univ. Press, Edinburgh. pp. 43-59.

3. Cousot, P. and Cousot, R. 1977: "Automatic synthesis of optimal invariant assertions:, Proc. Symp. Art. Intel. Prog. Lang., SIGPLAN Notices 12, 8. pp. 1-12.

4. Elspas, B. 1974: "The semiautomatic generation of inductive assertions for proving program correctness", Interim Report, SRI International, Menlo Park, CA.

5. Floyd, R. W. 1967: "Assigning meaning to programs:, Proc. Symp. Appl. Math. 19 (ed. Schwartz, J. I.), AMS, Providence, RI. pp. 19-32

6. German, S. M. and Wegbreit, B. 1975: "A synthesizer of inductive assertions", IEEE Tran. Software Engg. SE-1, 1. pp. 68-75.

7. Katz, S. M. and Manna, Z. 1976: "Logical analysis of programs", CACM 19, 4 (April). pp. 188-206.

8. Kleene, S. C. 1952: Introduction to Metamathematics, North-Holland, Amsterdam.

9. Manna, Z. 1974: Mathematical Theory of Computation, McGraw-Hill, New York.

10. McCarthy, J. 1962: "Towards a mathematical basis of computation", Information Processing: Proc. IFIP Congress 1962. pp. 21-28.

11. Meyer, A. R. and Ritchie, D. M. 1969: "The complexity of loop programs", Proc. ACM Nat'l Meeting. pp. 465-469.

12. Misra, J. 1977: "Prospects and limitations of automatic assertion generation for loop programs", SIAM J. Computing 6,4 (December). pp. 718-729.

13. Tamir, M. 1980: "ADI - Automatic derivation of invariants", IEEE Trans. Software Engg., SE-6, 1 (January), pp. 40-48

14. Tarski, A. 1955: "A lattice-theoretical fixed-point theorem and its applications", Pacific J. Math. 5. pp. 285-309.

15. Wegbreit, B. 1974: "The synthesis of loop predicates", CACM 17, 2 (February). pp. 102-112

16. Wegbreit, B. 1977: "Complexity of synthesizing inductive assertions", JACM 24, 3 (July). pp. 504-512.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CS-8101	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYNTHESIS OF INVARIANT ASSERTIONS FOR ARITHMETICAL PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) S. Kamal Abdali Jan Vytopil		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-75-C-1026
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematical Sciences Department Rensselaer Polytechnic Institute Troy, NY 12181		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Residence Representative 715 Broadway-5th Floor, N.Y., N.Y. 10003		12. REPORT DATE June 1981 13. NUMBER OF PAGES 29
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Invariant assertion; loop invarient,
arithmetical predicate
program verification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A program is called arithmetical if it is obtained from a flowchart schema under an interpretation in which the domain consists of non-negative intergers and the assigned functions and predicates are arithmetical in the sense of Gödel. The computation done by arithmetical programs is characterized

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

in terms of arithmetical predicates. This is a first-order formalization, compared to the second-order, Cooper-Manna formalization.

It is shown that for the class of arithmetical programs, the problem of synthesis of invariant assertions is solvable, and a simple algorithm is given to construct arithmetic predicates which can serve as invariant assertions. Two alternative definitions of invariant assertions found in the literature -- minimal predicates and optimal predicates -- are shown to be equivalent. It is proved that the arithmetical predicates generated by our algorithm satisfy the definition of both minimal and optimal predicates.