Andrzej Blikle

SPECIFIED PROGRAMMING

333

Warsaw 1978

Mailing address: prof. dr hab. Andrzej Blikle
Institute of Computer Science
Polish Academy of Sciences
P.O. Box 22
00-901 Warsaw, PKiN

P r i n t e d   a s   a   m a n u s c r i p t

N a   p r a w a c h   r ę k o p i s u

Nakład 700 egz. Ark. wyd. 1,80; ark. druk. 1,75.
Papier offset. kl. III, 70 g, 70 x 100. Oddano do druku w sierpniu 1978 r. W.D.N.Zam.535/0/78

Abstract . Содержание . Streszczenie

The paper presents a method of mathematically supported correct programming. Totally correct programs are developed be means of transformation rules. These programs are considered and transformed together with their specifications. The specifications are of two types: global (pre- and post- conditions) and local (redundant tests). The transformations always preserve the total correctness of programs and are rather flexible; e.g. one may add or remove variables in the program or switch from one data type to another.

Программирование со спецификацией

В работе излагается математический метод вывода полностью корректных программ. Метод основан на правилах преобразования, которые применяются к программам со спецификациями. Правила таковы, что их применение сохраняет свойство полной корректности преобразуемой программы со спецификацией. Метод допускает два типа спецификаций задаваемых вместе с программами: глобальные (pre- и post - условия) и локальные (редундантные тесты). Правила дают разнообразные возможности преобразования программ, например введение новых переменных в программу, устранение переменных из программы, переход от одной структуры данных к другой и др.

Programowanie ze specyfikacją

W pracy przedstawiono matematyczną metodę wyprowadzania programów poprawnych. Metoda operuje regułami transformacji stosowanymi do programów wraz ze specyfikacją i zachowującymi całkowitą poprawność programów. Stosowane są dwa typy specyfikacji programów: globalne (pre- i post- warunki) oraz lokalne (redundantne testy). Reguły transformacji pozwalają na dokonywanie różnorodnych przekształceń programów, np. dodawanie lub usuwanie zmiennych czy przechodzenia od jednej struktury danych do drugiej.

This paper will appear in the Proceedings of the International Conference on Mathematical Studies of Information Processing, Kyoto, August 24-26, 1978

1. INTRODUCTION

This paper concerns the technique of mathematically supported correct programming (correct program derivation). We are dealing here with programs extended by input-output specifications. Such programs are called <u>specified programs</u> and are of the syntactical form

$$\underline{\text{pre}}\ c_1;\ \text{IN}\ \underline{\text{post}}\ c_2$$

where IN is the operational part (the instruction) and c_1, c_2 are conditions called respectively the <u>precondition</u> and the <u>postcondition</u>. A specified program is called <u>correct</u> if IN is totally correct with respect to c_1 and c_2.

The method which is sketched in this paper provides a mathematical technique of the derivation of correct specified programs. Starting from some initial correct specified program (whose correctness must be proved) we apply transformation rules which produce only correct programs. In this way the correctness of each successive refinement of the initial program is guaranted by the method and needs not to be proved in each step separately. In general, our transformations change not only the operational part IN of the program, but also the specification c_1, c_2. This gives the necessary flexibility of transformation rules. For instance we may easily add and remove variables in the program or switch from one data type to another. The latter option is especially useful in programming with abstract data types (Liskov and Zilles 1975 and Meertens 1976).

The described method is neither a formalized axiomatic system of derivation rules nor it offers a list of magic heuristic techniques. Instead, it is supposed to provide mathematical tools which may be used by a programmer as the support - but not the alternative - of his intuitive experience. The core of the method consists of a pseudo--programming language and of a set of transformation rules. The semantics of the language is given a denotational description and each transformation rule is supplied with a soundness theorem.

The description of the method given in this paper should be regarded as preliminary and very incomplete. We are trying to explain the main general ideas and to show the examples of some concrete transformations. It is understood that any practical application of the method requires a more technical extension. For more comments see Sec.9. Due to space limitation the proofs of theorems were omitted.

The concept of programming by refinement rules is a particular realization of the idea of structured programming (Dijkstra 1968, 72). For several years structured programming has been understood as philosophy of programming providing lucid and easily provable programs. Recently it started to evoluate towards a systematic method of programming by transformations. Some authors, such as Bär (1977),Burstal and Darlington (1977), Darlington (1975, 76), Dijkstra (1975), Sinzoff (1977), Spitzen, Levitt and Lawrence (1976), Wegbreit (1976) describe the transformations as heuristic or only syntactical operations. In this case the correctness of each program's refinement must be proved separately. Other authors, like Dershowitz and Manna (1975), van Emden (1975, 76), Irlik (1976,78), Blikle (1977 A,B) suggest that program transformations formally guarantee the correctness of all successive versions of the derived program. Regarding the idea of dealing with specified programs it was described explicitly by Dershowitz and Manna (1975) and Bär (1977) and implicitly by van Emden (1975,76). In contrast to the present approach those methods guarantee that the derived programs are only partially correct. Also the techniques of program development and refinement are different from ours. The present paper is a continuation of Blikle (1977 A,B).

2. ABSTRACT PROGRAMMING LANGUAGE

The general description of our method should not be restricted to any fixed programming language. On the other hand, we have to deal with some language since otherwise we cannot talk about program transformations. As a compromise we introduce below the concept of an abstract programming language which represents the class of programming languages to which our method (in its present form) may be applied. This concept does not pretend to universality or completeness. It hass been chosen rather ad hoc, just for the sake of this paper.

Formally, the abstract programming language (abbreviated apl which must not be confused with APL) is, of course, a 273-tuple consisting of several syntactical and semantical objects. We shall describe these objects successively leaving the tupling operation to our more rigorous readers.

The basic component in the definition of apl is the abstract data type. This is an abstract algebra $DT = (D,f_1,\dots,f_n,q_1,\dots,q_m)$ where D is a nonempty set (the carier) and $f_i:D^{a_i} \longrightarrow D$, $q_j:D^{b_j} \longrightarrow \{true, false\}$ are partial functions. a_i and b_j are integers - the arities of f_i and q_j respectively. Since f_i and q_j are partial, D may be regarded as the union of different data types such as integers, reals, lists, records, sets etc. In this way DT represents a class of data types. We shall assume that one of q_j's represents the identity relation in D. This relation will be denoted by $=$.

Important remark. In the applications DT is a concrete algebra but it still may contain abstract (i.e. not implemented and not intended for implementation) data types (Liskov and Zilles 1975). The same concerns the syntax of the language which is defined below. We assume that each concrete representation of apl will contain only a small implementable subset. Beside this subset we have in apl data types and programming constructions chosen entirely for the sake of compact and lucid description of algorithms. Programs written initially in these non-implementable terms are next systematically transformed into implementable programs. □

Given DT we establish the syntactical components of apl. First we assume that with each f_i and q_j there are associated symbols F_i and Q_j respectively. For simplicity, "=" will denote both, the identity in D and the corresponding predicate symbol. Next we assume to have in apl an infinite set IDE of symbols called *identifiers*. Having this we define the set EXP of *expressions* and CON of *conditions* over DT:

EXP is the set of terms (in the usual sense) over the set of functional symbols $F_1,\dots,F_n$ and the set of variables IDE. For instance, $x_1,x_2,\dots,\ F_1(x_1,\dots,x_{a_i}),\ F_1(F_2(x_1,\dots,x_{a_2}),\dots)$, etc. are expressions.

CON is the set of first order formulas (in the usual sense) over the set of predicate symbols $Q_1,\dots,Q_m$ and the set of expressions EXP. For instance, if $E_1,E_2,\dots$ are expressions, then $Q_j(E_1,\dots,E_{b_j})$, $\forall x_1 \exists x_2 Q_j(E_1,\dots,E_{b_j})\ \&\ Q_k(E_1',\dots,E_{b_k}')$, etc. are conditions.

Remark. In the applications - hence also in our examples - we shall identify the symbols F_i with f_i and Q_j with q_j and allow the infix notation. Typical expressions are therefore $x+\sqrt{y}$, $(x+y)\ast z$, $\max\{k \mid k<2^n\}$, etc. and typical conditions are $x=y$, $x<2^y$, $(\exists y)(x=2^y)$, etc. $\square$

The main syntactical class in apl is the class INS of *instructions*. We assume that INS is the least set of words (a formal language) which satisfies the following axioms:

(1) - *abort* and *skip* are instructions,

- if $x_1,\dots,x_n$ are mutually different identifiers, $E_1,\dots,E_n$ are arbitrary expressions and c is a condition, then

if c *fi* , $x_1 := E_1$, *si* $x_1 := E_1\ \&\dots\&\ x_n := E_n$ *is*

are instructions.

(2) if c is a condition and IN_1, IN_2 are instructions, then

$IN_1;IN_2$ *while* c *do* IN_1 *od*
if c *then* IN_1 *fi* *inv* c; IN *vni* c
if c *then* IN_1 *else* IN_2 *fi*

are instructions.

The instructions of the form *if* c *fi* are called *tests*. The instructions *inv* c; IN *vni* c are called *invariant-guarded instructions* and were introduced by Blikle (1977 A,B) in the syntactical form *begin* c; IN *end* c. This earlier syntax was confusing since the use of the words *begin* and *end* may suggest a block-like structure (in the ALGOL sense) which does not apply in our case. The pair of words *inv* c and *vni* c is called the *declaration of the invariant c*.

The last syntactical class in apl is the class of specified programs of the form

$$\underline{\text{pre}}\ c_1;\ \text{IN}\ \underline{\text{post}}\ c_2 \qquad (2.1)$$

where c_1 and c_2 are conditions called respectively the precondition and the postcondition of (2.1) and IN is the instruction of (2.1). It should be emphasised that specified programs are in fact statements about programs. In (2.1) only IN is executable, whereas the pair c_1, c_2 constitutes the input-output specification of IN.

In order to define the semantics of apl we first introduce a few basic concepts. By a state we shall mean any total function $s : \text{IDE} \longrightarrow \text{D}$. The fact that states are total means that the identifiers represent global variables (see Sec.9 for comments). By S we shall denote the set of all states, hence $S = [\text{IDE} \longrightarrow \text{D}]$. For an arbitrary set $V \subseteq \text{IDE}$ of identifiers and arbitrary states $s_1, s_2 \in S$ we shall say that s_1 and s_2 are equal out of V, in symbols $s_1 = s_2\ \underline{\text{outof}}\ V$, if $s_1(x) = s_2(x)$ for $x \notin V$.

By Int we denote the function of interpretation defined in EXP $\cup$ CON in the way usual for mathematical logic. Therefore

$$\text{Int} : \text{EXP} \longrightarrow [S \longrightarrow D]$$
$$\text{Int} : \text{CON} \longrightarrow [S \longrightarrow \{\text{true, false}\}].$$

For instance, $\text{Int}(x)(s) = s(x)$, $\text{Int}(F_i(x_1,\dots,x_{a_i}))(s) = f_i(s(x_1),\dots,s(x_{a_i}))$, ect. In the sequel we shall write $\text{Int}(E)(s) = !$ as a shorthand of $(\exists d \in D)(\text{Int}(E)(s)=d)$, which means that the value of Int(E) in s is defined, and we shall write $\text{Int}(E)(s) = ?$ if the value of Int(E) in s is undefined.

By Rel(S) we denote the set of binary relations in S. If $R_1, R_2 \in \text{Rel}(S)$, then R_1R_2 denotes the usual composition of R_1 and R_2, $R_1 \cup R_2$ denotes the set-theoretical union and $R^* = R^0 \cup R^1 \cup R^2 \cup \dots$ denotes the iteration of R. Moreover, $\emptyset$ will denote the empty relation (and the empty set as well) and $I = \{(s,s) \mid s \in S\}$ the identity relation in S. For more details see Blikle (1977 B,C,D).

The main semantical concept in apl is the function of semantics which maps the set INS$\cup$CON into Rel(S). We denote this function by square brackets [ ], thus for any $c \in \text{CON}$ and $\text{IN} \in \text{INS}$ we have $[c], [\text{IN}] \in \text{Rel}(S)$. Since in the present version of apl all instructions are deterministic, all [IN] are functions. However, we define the semantics of apl in a more general - relational - framework since the definition of semantics and the program verification methods become much simpler if described for this general case. The definition of [ ] is the following:

(1) For any condition c we set $[c] = \{(s,s) \mid Int(c)(s)=true\}$. Consequently $[c] \subseteq I$.

(2) In the set INS the function of semantics is defined recursivele wrt the syntactical definition of this set:

$$[\underline{abort}] = \emptyset\ ,\ [\underline{skip}] = I\ ,\ [\underline{if}\ c\ \underline{fi}] = [c]$$

$$[\underline{si}\ x_1:=E_1\&\ldots\&x_n:=E_n\underline{is}] =$$

$$= \{(s_1,s_2) \mid (\forall i\leq n)(Int(E_i)(s_1)=!\ \&\ s_2(x_i)=Int(E_i)(s_1))\&$$

$$\&\ s_2=s_1\underline{outof}\{x_1,\ldots,x_n\}.$$

$$[x_1:=E] = [\underline{si}\ x_1:=E\ \underline{is}]$$

$$[IN_1;IN_2] = [IN_1][IN_2]$$

$$[\underline{if}\ c\ \underline{then}\ IN\ \underline{fi}]=[c][IN]\cup[\sim c]$$

$$[\underline{if}\ c\ \underline{then}\ IN_1\ \underline{else}\ IN_2\ \underline{fi}] = [c][IN_1]\cup[\sim c][IN_2]$$

$$[\underline{while}\ c\ \underline{do}\ IN\ \underline{od}] = ([c][IN])^*[\sim c]$$

The case of inv c; IN vni c is more complicated. Intuitively speaking to execute this instruction means to execute IN in checking simultaneously whether the successive states, including the initial and the terminal, satisfy c. If this is the case, then the execution continues. Otherwise the execution aborts. The condition c is therefore called the guarding invariant (cf. Dijkstra's (1975) guarded commands). Similar constructions appear also, although in a restricted version, in other languages. Forinstance, type specifications like integer x, array z, etc. are such quarding invariants in ALGOL blocks. In apl we may use more sophisticated invariants, e.g. $y=n-x^2\ \&\ p=xz$ (see Sec.7). The formal semantics of the instruction inv c; IN vni c is recursive:

$$[\underline{inv}\ c;\ \underline{abort}\ \underline{vni}\ c] = \emptyset$$

$$[\underline{inv}\ c;\ \underline{skip}\ \underline{vni}\ c] = [c]$$

$$[\underline{inv}\ c;\ \underline{si}\ x_1:=E_1\ \&\ldots\&\ x_n:=E_n\underline{is}\ \underline{vni}\ c] =$$

$$=[c][\underline{si}\ x_1:=E_1\ \&\ldots\&\ x_n:=E_n\ \underline{is}][c]$$

$$[\underline{inv}\ c;\ \underline{if}\ c_1\ \underline{fi}\ \underline{vni}\ c] = [c][c_1]$$

$$[\underline{inv}\ c;\ IN_1;IN_2\ \underline{vni}\ c] = [\underline{inv}\ c;\ IN_1\ \underline{vni}\ c][\underline{inv}\ c;\ IN_2\ \underline{vni}\ c]$$

$$[\underline{inv}\ c;\ \underline{if}\ c_1\ \underline{then}\ IN\ \underline{fi}\ \underline{vni}\ c] = [\underline{if}\ c_1\ \underline{then}\ \underline{inv}\ c;IN\ \underline{vni}\ c\ \underline{fi}]$$

$$[\underline{inv}\ c;\ \underline{if}\ c_1\ \underline{then}\ IN_1\ \underline{else}\ IN_2\ \underline{fi}\ \underline{vni}\ c] =$$

$$= [\underline{if}\ c_1\ \underline{then}\ \underline{inv}\ c;\ IN_1\ \underline{vni}\ c\ \underline{else}\ \underline{inv}\ c;\ IN_2\ \underline{vni}\ c\ \underline{fi}]$$

$$[\underline{\text{inv}}\ c;\ \underline{\text{while}}\ c_1\ \underline{\text{do}}\ IN\ \underline{\text{od}}\ \underline{\text{vni}}\ c] = [\underline{\text{while}}\ c_1\ \underline{\text{do}}\ \underline{\text{inv}}\ c;\ IN\ \underline{\text{vni}}\ c\ \underline{\text{od}}]$$
$$[\underline{\text{inv}}\ c;\ \underline{\text{inv}}\ c_1;\ IN\ \underline{\text{vni}}\ c_1\ \underline{\text{vni}}\ c] = [\underline{\text{inv}}\ c\&c_1;\ IN\ \underline{\text{vni}}\ c\&c_1]$$

In the sequel, for any instruction IN the relation [IN] will be called the resulting relation of IN. Two instructions IN_1 and IN_2 will be called equivalent, if $[IN_1] = [IN_2]$.

Since the specified programs are statements about instructions, their semantical meaning reduces to the truth values. This is formalized in the next section.

3. CORRECTNESS AND REDUNDANCY OF SPECIFIED PROGRAMS

We shall need in this section a few farther technical concepts. First we extend the composition in Rel(S) to the case where one of the arguments is a set. Let $R \in Rel(S)$ and $B \subseteq S$:

$$BR = \{s_2 \mid (\exists s_1)(s_1 \in B\ \&\ s_1 R s_2)\},\quad RB = \{s_1 \mid (\exists s_2)(s_1 R s_2\ \&\ s_2 \in B)\}.$$

If R = [IN], for some IN, then B[IN] is the set of all outputs generated by IN from the inputs of B and [IN]B is the set of these inputs which generate outputs in B. For more details see Blikle (1977 B,D).

With every condition c we associate the set of states denoted by {c} and defined as follows:

$$\{c\} = \{s \mid Int(c)(s) = true\}.$$

For instance $\{x_1<x_2\} = \{s \mid s(x_1) < s(x_2)\}$. Since the predicates $q_1,\ldots,q_m$ in DT are partial (Sec.2) all conditions are partial as well. This means that in general the set $\{c\} \cup \{\sim c\}$ is a proper subset of S. For instance, if " < " denotes the ordering in the set REAL of reals, the $\{x_1<x_2\} \cup \{x_2 \leq x_1\} = \{s \mid s(x_1), s(x_2) \in REAL\}$.

The specified program pre c_1; IN post c_2 is called correct if

$$\{c_1\} \subseteq [IN]\{c_2\}.$$

This definition coincides exactly with the Floyd-Hoare total correctness (see Blikle 1977 D).

Each transformation rule is restricted to a class of programs which satisfy certain properties. Some of these properties are global, i.e. follow from the specification c_1 and c_2 but some others are local. A typical local property says that a given condition c is satisfied at a given cut-point. In order to express such properties we introduce the concept of a redundant test and of a redundant invariant's declaration. Two technical concepts are required in the definition.

Let W,X,Z,Y be words over an arbitrary alphabet V. We say that W occurs in Z in the context (X,Y) if Z = XWY. The ordered triple (X,W,Y) will be called the occurence of W in Z.

Now, consider arbitrary conditions c and c_1 and arbitrary instructions IN and IN_1. Let if c fi occur in IN in the context (X,Y), i.e. let IN = X if c fi Y. We say that this occurence of if c fi in IN is redundant under the precondition c_1 if

$$[c_1][IN] = [c_1][X \text{ skip } Y]$$

Intuitively this means that if we precede IN by the test (precondition) if c_1 fi, then we may remove the test if c fi from IN and the new instruction will have the same resulting relation as the former. In other words, all the executions of IN which satisfy c_1 at the beginning and which terminate must satisfy the occurence of if c fi in the contect (X,Y).

Let inv c; IN_1 vni c occur in IN in the context (X,Y), i.e. let IN = = X inv c; IN_1 vni c Y. We say that the invariant's declaration in this context is redundant under the precondition c_1 if

$$[c_1][IN] = [c_1][X\ IN_1\ Y].$$

The interpretation is the same as above.

LEMMA 3.1 Let pre c_1; IN post c_2 be arbitrary specified program and let IN' denote the instruction which results in from IN by the removal of any number of occurences of test and/or declarations of invariants which are redundant under the precondition c_1. The specified program pre c_1; IN' post c_2 is correct iff the specified program pre c_1; IN post c_2 is correct. □

Intuitively this lemma says that redundant tests and redundant invariant's declarations are operationally useless in the program. This, in turn, means that they may be regarded as the specifications of local program's properties. The specified program pre c_1; IN post c_2 is called redundant if all the occurences of tests and invariant's declarations in IN are redundant under the precondition c_1.

4. HOARE-TYPE TRANSFORMATIONS OF SPECIFIED PROGRAMS

Hoare's rules for proving total correctness of programs may be regarded as correctness preserving transformations of specified programs. Below we give the examples of two such transformations which we shall use in the sequel of this paper.

LEMMA 4.1 If pre c_1; IN post c_2 is correct and if $c_3 \Longrightarrow c_1$ and $c_2 \Longrightarrow c_4$, then pre c_3; IN post c_4 is also correct. Moreover, if the former program is redundant, then the latter is redundant as well. $\square$

LEMMA 4.2 If the programs pre c_1; IN_1 post c_2 and pre c_3; IN_2 post c_4 are correct and $c_2 \Longrightarrow c_3$, then

pre c_1; IN_1; if c_2 fi ; IN_2 post c_4

and pre c_1; IN_1; if c_3 fi ; IN_2 post c_4

are also correct. Moreover, if the initial programs are redundant, then the resulting programs are redundant as well. $\square$

5. THE INSERTION OF NEW VARIABLES INTO PROGRAMS

The transformations of the type described in Sec.4 do not change the set of variables of the transformed program. In this section we define the transformation - described earlier by Blikle (1977 A,B) in a slightly different way - which allows the addition (and the removal) of variables into (from) programs. This transformation concerns a rather particular case where the value of the new variable y is related to the value of the old variables $x_1,\dots,x_n$ by the equation of the form y=E where E is an expression which does not contain y. As was shown by Blikle (1977 A,B) , see also Sec.7, this transformation is useful in program optimisation. In Sec.8 we show that it also provides a very natural instrument for the transformation of programs from one data-type into another.

In order to describe the syntax of our transformation we define the syntactical function INSERT(y=E, IN) which, given an instruction IN an expression E and an identifier y as arguments, yields a new instruction IN_1. The function INSERT will be defined by cases wrt the syntaxt of IN. We start by the case, where IN is the simultaneous assignment si x_1:=E_1&...& x_n:=E_n is. Three subcases are to be considered:

(1) if $y \in \{x_1,\dots,x_n\}$, then INSERT(y=E,IN) is undefined,

(2) if $y \notin \{x_1,\dots,x_n\}$ and no x_i occurs in E, then INSERT(y=E, IN) = IN,

(3) if $y \notin \{x_1,\dots,x_n\}$ and at least one x_i occurs in E, then INSERT(y=E, IN) is of the form

si x_1:=E_1 &...& x_n:=E_n & y:=E($x_1/E_1,\dots,x_n/E_n$) is

where E($x_1/E_1,\dots,x_n/E_n$) denotes the effect of the simultaneous substitution of E_i for each occurence of x_i in E for i=1,...,n. In the applications, where we allow infix notation, substitution may require the addition of parentheses. We shall add these parentheses whenever required. E.g. the substitution of x+y for z in zx results in (x+y)x.

The case where IN is of the form x_1:=E_1 is analogous to the former since it may be considered as the case of <u>si</u> x_1:=E_1 <u>is</u>. In all the remaining cases INSERT(y=E, IN) is the effect of the insertion of y=E into all simultaneous and simple assignment statements in IN. For the formal definition see Blikle (1977 B).

Now, consider an arbitrary instruction IN and let the instruction

<u>if</u> c <u>fi</u> ; <u>inv</u> c' ; IN' <u>vni</u> c'   (5.1)

occur in IN. Let for y∈IDE and E∈EXP the instruction IN_1 results in from IN by the substitution of the instruction

```
if c fi;
y:=E;
inv c' & y=E;
    INSERT(y=E, IN')
vni c' & y=E
```

for some chosen occurence of (5.1) in IN. The step from IN to IN_1 describes the transformation which adds new variable y to IN. The value of this variable is kept equal to the value of E during the whole execution of IN'. In order to describe the soundness of this transformation one more concept is needed.

Let E be an expression. By the <u>domain</u> of E, in symbols Dom E, we mean the set of states in which E may be evaluated (has a value). Formally Dom E = {s | Int(E)(s) =!}. If c is a condition, then the inclusion {c} ⊆ Dom E means that for any state which satisfies c, the value of E is defined.

<u>THEOREM</u> 5.1 Let c_1 and c_2 be arbitrary conditions. If y does not occur in c_2 and IN and if {c} ∪ {c'} ⊆ Dom E, then the program

<u>pre</u> c_1; IN <u>post</u> c_2 (5.2)

is correct <u>iff</u> the program

<u>pre</u> c_1; IN_1 <u>post</u> c_2 & y=E   (5.3)

is correct. Moreover (5.3) is redundant <u>iff</u> (5.2) is redundant. □

The occurence of the test <u>if</u> c <u>fi</u> and the declaration <u>inv</u> c' , <u>vni</u> c' in (5.1) is a technical trick which allows the description of the fact that y:=E may be executed before entering IN' ($\{c\} \subseteq$ Dom E) and that it may be executed in each step of the execution of IN ($\{c'\} \subseteq$ Dom E).

6. EXHAUSTING PROGRAMS

The process of program derivation may be described by the sequence $P_1, P_2, \ldots, P_n$ of successive refinements of some initially given program (or program specification) P_1. So far we have been dealing with the refinement transformations allowing the steps $P_i \longrightarrow P_{i+1}$ for $i \geq 1$. Here we shall concentrate on the problem of establishing the initial program P_1. Of course, the way in which we establish P_1 cannot be formalized since in this step we describe our intuitive understanding of P_1. However, one may suggest to have P_1 in some particular form. For instance, Burstal and Darlington (1977) suggest that P_1 be given as a recursive procedure. In this section we shall discussed another solution. Anticipating usual arguments it should be stressed that this solution is not considered by the author as unique, universal or better than any other. This is just another solution which may deserve the attention of the reader. We explain it first on the example.

Consider two correct specified programs:

```
P : pre integer n & n ≥ 1;            P' : pre integer n,m & n,m ≥ 1;
    x:=0;                                  x:=0;
    while (x+1)²≤n do x:=x+1 od;           while (x+1)m≤n do x:=x+1 od;
    post x=intsqr(n)                       post x=n÷m
```

where integer n is a condition which is satisfied if the value of n is an integer, intsqr(n) denotes the integer square root of n and n÷m denotes the integer quotient of n and m. These programs compute different values but they are searching for these values in exactly the same way: starting from 0 and proceeding through successive positive integers. This suggests that both P and P' may be derived from the same program which describes that way of searching. We shall show that this is the case. Consider the specified program

```
P1 : pre integer k & k ≥ 1
     x:=0;
     while x+1≤k do x:=x+1 od;
     post x=k
```

which is, of course, correct. Both P and P' may be derived from P_1. We shall show this for P. The other case is analogous. Some steps in the derivation below are described informally but they can easily be formalized by everybody familiar with program verification techniques.

First observe that the condition integer n & n ≥ 1 & n=intsqr(k) implies the condition integer k'& k≥1. Therefore, on the strength of Lemma 4.1 we may derive from P_1 the following program

```
P2 : pre integer n & n≥1 & k=intsqr(n);
     x:=0;
     while x+1≤k do x:=x+1 od
     post x=k
```

which is also correct. Since k and n are constant in this program and since k=intsqr(n) occurs in the precondition, we may replace k in the instruction of the program and in the postcondition by intsqr(n):

```
P3 : pre integer n & n≥1 & k=intsqr(n)
     x:=0;
     while x+1≤intsqr(n) do x:=x+1 od
     post x=intsqr(n)
```

Now we use the arithmetical fact that for n≥1, the condition x+1≤ ≤ intsqr(n) is equivalent to $(x+1)^2 \leq n$. We replace the former condition in P_3 by the latter and we remove the condition k=intsqr(n) from the precondition since k does not appear neither in the instruction nor in the postcondition. In this way we get the required program P.

Observe, that our programs P and P' are rather slow. This, of course, is the consequence of the fact that the initial program P_1 is slow. If we replace P_1 by faster program (sec.7), then we get faster programs computing intsqr(n) and n÷m respectively. On the other hand, we cannot expect to speed up P and P' too much otherwise. This example shows that in the derivation of at least some programs it may be advisable to make a careful choice of the basic "P_1-like" program. In making this choice we may forget, for a moment, about the function which our target program is supposed to compute, concentrating on the pure searching method in the set where the values of this function belong.

Now we can formalize and generalize the concept of "P_1-like" program. Suppose that we are going to derive a program computing a certain function $h:D \longrightarrow A$, where $A \subseteq D$ (see Sec.2 for D). According to our earlier remarks we begin by establishing a program of the form

pre a∈A; IN post x=a

where a does not occur in the assignments of IN (it may occur in conditions). The total correctness of this program means that its instruction may "reconstruct" or "retriev" each element of A. Totally correct programs of this form will be called exhausting programs for A.

7. AN EXAMPLE OF PROGRAM DERIVATION AND REFINEMENT

This section is devoted to the systematic derivation of an efficient and rather tricky program (known to the author from J.O.Dahl) computing the integer square root of a positive integer. In Sec.8 we transform this program into an analogous one over the data type of binary strings. The same program was already investigated by Blikle (1977 B). The present version corresponds to the new setting of the method.

First we shall assume that our data type contains everything which we may need in the sequel: integer arithmetics, set theory, binary strings etc. We do not need to care about the size and complexity of this data type since we are not going to implement all its subsets (cf. the remarks of Sec.2). We shall start our programming by establishing a fast program exhausting the set of positive integers.

Let INT denote the set of positive integers, let integer k be the condition defined as in Sec.6 and let $B = \{2^m \mid m=0,1,\dots\}$. We begin by a few general mathematical observations about integers.

(1) For every positive integer k there exists a unique integer $y \in B$ such that $y \le k < 2y$. This integer will be called the magnitude of k and will be denoted by mag(k).

(2) If y=mag(k), then there exist a unique string $a_0, a_1, \dots, a_{\log(y)}$ of 0's and 1's, called the binary representation of k such that

$$k = \sum_{i=0}^{\log(y)} (y \div 2^i) a_i .$$

(3) The string defined in (2) has the following property: $a_0=1$ and for any $i = 1,\dots,\log(y)$

$$a_i = 1 \quad \underline{\text{iff}} \quad \sum_{j=0}^{i-1} (y \div 2^j) a_j + (y \div 2^i) \le k.$$

Starting from these observations we can easily construct the following programs and prove them correct and redundant (the proofs may be caried out by any of the well known methods and are left to the reader):

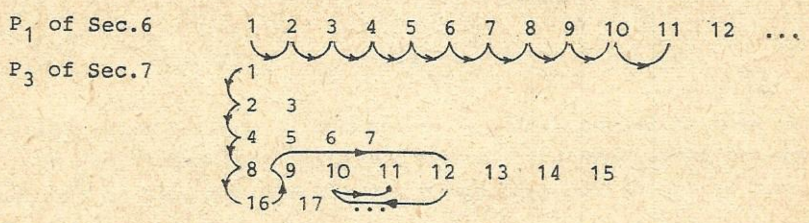
```
$P_1$: pre integer k & k ≥ 1;
    z:=1;
    inv z∈B;
      while z≤mag(k) do z:=2z od;
    vni z∈B;
    post integer k & k ≥ 1 & z=2mag(k)

$P_2$: pre integer k & k ≥ 1 & z=2mag(k);
    x:=0;
    while z>1 do z:=z÷2; if x+z ≤ k then x:=x+z fi od
    post x=k & z=1
```

Remark. The reader may wonder why have we chosen P_1 in such a way that it computes 2mag(k) rather than simply mag(k). The reason is purelly technical - in this case we get z=1 at the output of P_2 which allows later on (in P_6 to P_7) a nice optimization of our program. □

Within the scope of the invariant $z \in B$ the condition $z \le mag(k)$ is equivalent to $z \le k$. Therefore, it may be replaced by that condition in P_1 (for the formal description of such transformations see Sec.6 of Blikle 1977 B). Now, we combine P_1 with P_2 by Lemma 4.2 and we omit the redundant test by Lemma 3.1 We get,

```
$P_3$: pre integer k & k ≥ 1;
    z:=1;
    while z≤k do z:=2z od;
    x:=0;
    while z>1 do z:=z÷2; if x+z≤k then x:=x+z fi od
    post x=k & z=1
```

This is the exhausting program for the set of positive integers which we wanted to construct. It is of course much faster than the program of Sec.6. While the latter computes k in polynomial time, the former uses only a logarithmic amount of time. The explanation of both algorithms is shown below.



Now, similarly as in Sec.6 we may transform P_3 into a program computing the function intsqr (n). Replacing in one step $z \leq$ intsqr(n) by $z^2 \leq n$ and $x+z \leq$ intsqr(n) by $(n+z)^2 \leq n$ we get

```
P4: pre integer n & n ≥ 1;
     z:=1;
     while $z^2 \leq n$ do z:=2z od;
     x:=0;
     while z>1 do z:=z÷2 ; if $(x+z)^2 \leq n$ then x:=x+z fi od
     post x=intsqr(n) & z=1
```

In the subsequent steps we shall optimize this program using the transformation described in Theorem 5.1 as the main technique. First observe that P_4 computes the value of z^2 in each execution of both loops. This is certainly nonoptimal. We may improve the program introducing new variable q with guarding invariant $q=z^2$. For the formal application of Theorem 5.1 we need that in P_4 the statement z:=1 be followed by if integer z fi and the remaining part be closed between parentheses inv integer z and vni integer z. Since this test and these invariant declarations are obviously redundant we shall omit (or we shall not insert) them for the benefith of the readability of our programs. In the same step we perform the appropriate arithmetical transformations.

```
P5: pre integer n & n ≥ 1;
     z:=1;
     q:=1;
     inv $q=z^2$;
        while q≤n do si z:=2z & q:=4q is od;
        x:=0;
        while z>1 do si z:=z÷2 & q:=q÷4 is;
                     if $x^2+2xz+q \leq n$ then x:=x+z fi od
     vni $y=z^2$
     post x=intsqr(n) & z=1 & $q=z^2$
```

Since within the scope of the invariant $q=z^2$ the condition z>1 is equivalent to q>1, we may replace the former by the latter in P_5 (this transformation is described formally in Sec.6 of Blikle 1977 B). We also introduce the new identifiers y and p with the invariants $y=n-x^2$ and p=xz (this tricky choice of invariants only proves that the discipline of programming must be supported by the art of it).

```
$P_6$: pre integer n & $n \geq 1$
    z:=1;
    q:=1;
    inv $q=z^2$;
      while $q \leq n$ do si z:=2z & q:=4q is od;
      x:=0;
      y:=n;
      p:=0;
      inv $y=n-x^2$ & p=xz;
        while q>1 do si $z:=z \div 2$ & $q:=q \div 4$ & $p:=p \div 2$ is;
             if $2p+q \leq y$ then si x:=x+z & p:=p+q & y:=y-2p-q is fi od
      vni $y=n-x^2$ & p=xz
    vni $q=z^2$
    post x=intsqr(n) & z=1 & $q=z^2$ & $y=n-x^2$ & p=xz
```

In the subsequent step we shall remove z from our program. In the present step we prepare the program for this transformation. Anticipating, this transformation consists of the "backward" application of Theorem 5.1. Therefore we first transform P_6 into the form which may be regarded as the result of the insertion of the invariant $z=\sqrt{q}$ into some program P where z does not appear. This program P may be regarded, in turn, as the result of the removal of z from P_6. First, we replace in P_6 the condition $q=z^2$ by the equivalent condition $z=\sqrt{q}$. Next, z is replaced by $\sqrt{q}$ everywhere between inv $z=\sqrt{q}$ and vni $z=\sqrt{q}$ except for the left sides of assignments. Finally the postcondition is simplified by obvious substitutions and z:=1; q:=1 is replaced by q:=1; z:=1.

```
$P_7$: pre integer n & $n \geq 1$;
    q:=1;
    z:=1;
    inv $z=\sqrt{q}$ ;
      while $q \leq n$ do si $z:=2\sqrt{q}$ & q:=4q is od;
      x:=0;
      y:=n;
      p:=0;
      inv $y=n-x^2$ & $p=x\sqrt{q}$;
        while q>1 do si $z:=\sqrt{q} \div 2$ & $q:=q \div 4$& $p:=p \div 2$ is;
             if $2p+q \leq y$ then si $x:=x+\sqrt{q}$ & p:=p+q & y:=y-2p-q is fi od
      vni $y=n-x^2$ & $p=x\sqrt{q}$
    vni $z=\sqrt{q}$
    post x=intsqr(n) & z=1 & q=1 & $y=n-x^2$ & p=x
```

In this program z is an independent variable in the sense that it does not appear neither in conditions nor in those assignments which modify the remaining variables. Since we are not interested in the final value of z we remove it from P_7. As was mentioned earlier we do it by the backward application of Theorem 5.1. Indeed, P_7 may be regarded as the result of the insertion of z with $z=\sqrt{q}$ into the following program:

```
P_8: pre integer n & n≥1;
     q:=1;
     while q≤n do q:=4q od;
     x:=0;
     y:=n;
     p:=0;
     inv y=n-x^2 & p=x√q ;
       while q>1 do si q:=q÷4 & p:=p÷2 is;
            if 2p+q≤n then si x:=x+√q & p:=p+q & y:=y-2p-q is fi od
     vni y=n-x^2 & p=x√q
     post x=intsqr(n) & q=1 & y=n-x^2 & p=x
```

Observe that in our postcondition the condition x=intsqr(n) & p=x may be replaced by p=intsqr(n) & x=p. Now, it turns out that x may be removed from P_8 in the same way as we have removed z from P_7. We get

```
P_9: pre integer n & n ≥ 1
     q:=1;
     while q≤n do q:=4q od;
     y:=n;
     p:=0;
     inv y=n-(p^2 ÷ q);
       while q>1 do si q:=q÷4 & p:=p÷2 is;
            if 2p+q≤y then si p:=p+q & y:=y-2p-q is fi od
     vni y=n-(p^2 ÷ q)
     post p=intsqr(n) & q=1 & y=n-(p^2 ÷ q)
```

In the last step we shall make the following transformations: First we remowe the redundant declarations of $y=n-(p^2 \div q)$. Second, we replace the instruction si q:=q÷4 & p:=p÷2 is by the equivalent instruction q:=q÷4; p:=p÷2. Third, we replace the instruction

```
    p:=p÷2 ; if 2p+q≤y then si p:=p+q & y:=y-2p-q is fi
```

by the equivalent instruction

```
    if p+q≤y then si p:=(p÷2)+q & y:=y-p-q  is else p:=p÷2 fi
```

This equivalence may be proved easily using the calculus shown in Blikle (1977 C). Firth, we replace si p:=(p÷2)+q & y:=y-p-q is by y:=y-p-q ; p:=(p÷2)+q. We get in this way

```
P10: pre integer n & n≥1
     q:=1;
     while q≤n do q:=4q od;
     y:=n;
     p:=0;
     while q>1 do q:=q÷4;
               if p+q≤y then y:=y-p-q; p:=(p÷2)+q else p:=p÷2 fi od
     post p=intsqr(n) & q=1 & y=n-(p^2÷q)
```

This is the final version of our (Dahl's) program. On the strength of theorems justifying the transformations which conducted us to P_{10}, this program is correct (i.e. totally correct).

8. AN EXAMPLE OF THE TRANSFORMATION OF A PROGRAM FROM ONE DATA TYPE INTO ANOTHER

In this section we shall transform the program P_{10} from Sec.7 into an analogous program dealing with binary representations of integers.

Let BR denote the set of all binary strings of the form 0 or 1X where $X \in \{0,1\}^*$. We shall use the following functions and relations in BR. Let X,Y,Z,... denote variables ranging over BR.

(1) shift left, $SL : BR \longrightarrow BR$
- $SL(0) = 0$
- $SL(X) = X0$ for $X \neq 0$

(2) shift right, $SR : BR \longrightarrow BR$
- $SR(0) = 0$
- $SR(X0) = SR(X1) = X$ for $x \neq 0$

(3) arithmetical operations + and -, for simplicity we shall use the same symbols as for operations on integers.

(4) lexicographical ordering $\sqsubset$ and the corresponding "less or equal" $\sqsubseteq$.

(5) birep X iff $X \in BR$

We shall also need conversion functions int : $BR \longrightarrow INT$ and br : $INT \longrightarrow BR$ defined in the usual way. The following equations are true for $X,Y \in BR$ and $x,y \geq 0$:

(6) $\underline{int}(\underline{br}(x)) = x$, $\underline{br}(\underline{int}(X)) = X$

(7) $\underline{br}(2x) = SL(\underline{br}(x))$

(8) $\underline{br}(x\div 2) = SR(\underline{br}(x))$ (10) $x < y$ iff $\underline{br}(x) \sqsubset \underline{br}(y)$

(9) $\underline{br}(x\pm y) = \underline{br}(x)\pm\underline{br}(y)$ (11) $x \le y$ iff $\underline{br}(x) \sqsubseteq \underline{br}(y)$

Now, consider program P_{10} of Sec.7. We shall replace its precondition by integer n & $n \ge 1$ & $N=\underline{br}(n)$ and introduce new variables Q, Y and P with the invariants $Q=\underline{br}(q)$, $Y=\underline{br}(y)$ and $P=\underline{br}(p)$. We also omit $y=n-(p^2\div q)$ in the postcondition.

```
P11: pre integer n & n≥1 & N=br(n)
     q:=1; Q:='1';
     inv Q=br(q);
       while q≤n do si q:=4q & Q:=SL(SL(Q)) is od;
       y:=n; p:=0;
       Y:=N; P:='0';
       inv Y=br(y) & P=br(p);
         while q>1 do si q:=q÷4 & Q:=(SR(SR(Q)) is;
                        if p+q≤y then si y:=y-p+q & Y:=Y-P+Q is;
                                               si p:=(p÷2)+q & P:=SR(P)+Q is
                                      else si p:=p÷2 & P:=SR(P) is
                        fi
                     od
       vni Y=br(y) & P=br(p)
     vni Q=br(q)
     post p=intsqr(n) & q=1 & q=br(q) & P=br(p) & Y=br(y)
```

Now, we perform local transformations preparing our program for the removal of q, y and p. First we replace the precondition by the equivalent one: birep N & '1' $\sqsubseteq$ N & $n=\underline{int}(N)$. Next, we replace the conditions in integers by equivalent conditions in binary strings. Finally, we transform the declarations of invariants into respectively $q=\underline{int}(Q)$, $y=\underline{int}(Y)$ and $p=\underline{int}(P)$ and perform the obvious substitutions in the postcondition. Now, we remove q, y and p by Theorem 5.1. We also remove the unnecessary condition $n=\underline{int}(N)$ from the precondition. In this way we get

```
P12: pre birep N & '1'⊑N;
     Q:='1';
     while Q⊑N do Q:=SL(SL(Q)) od;
     Y:=N; P:='0';
     while '1'⊏Q do Q:=SR(SR(Q));
                       if P+Q⊑Y then Y:=Y-P+Q; P:=SR(P)+Q
                                  else P:=SR(P) fi od
     post P=br(intsqr(int(N))) & Q = '1'
```

9. FINAL REMARKS

As was already mentioned in the Introduction the present method has been only sketched in this paper. First of all the given list of transformations, even if extended by the transformations of Blikle (1977 B), is rather limited. Secondly, our programs contain only global variables which is an essential limitation at least in the case where one wants to extend apl by recursive procedures.

The option of having global and local variables may be introduced if we slightly change the concept of the state in Sec.2. Namely, instead of defining states as functions $s: IDE \longrightarrow D$ we extend them to functions $s:IDE \longrightarrow D^*$, where D^* denotes the set of all finite strings over D including the empty string ε. The elements of D may be interpreted as staks of values having the current (available) value on the top. The function $Int:EXP \longrightarrow [S \longrightarrow D]$ must be redefined in the following way: for any $x \in IDE$: $Int(x)(s) = TOP(s(x))$. The remaining part of the definition is analogous. The fact that $Int(x)(s) = \varepsilon$ means that the value of x in s is undefined. The work on the extension of apl in this direction is in progress.

ACKNOWLEDGEMENT

I wish to express my thanks to P.Dembinski, W.Kwasowiec, J.Leszczylowski, J.Maluszynski, A.Mazurkiewicz, J.Winkowski with whome the discussions on the subject of the present paper were so stimulating and instructive for me.

Received August 25, 1978

REFERENCES

Bär, D.(1977) A methodology for simultaneously developing and verifying PASCAL programs, manuscript.

Blikle, A.(1977A) A mathematical approach to the derivation of correct programs, In: Semantics of Programming Languages (Proc. International Workshop, Bad Honnef, FRG, March 1977), Abteilung Informatik, Universitat Dortmund, Bericht Nr 41 (1977), 25-29

Blikle, A.(1977B) Toward mathematical structured programming, In: Formal Description of Programming Concepts (Proc. IFIP Working Conf. St.Andrews, N.B., Canada, August 1-5, 1977, E. J.Neuhold ed.), 183-202, North Holland, Amsterdam 1978

Blikle, A.(1977C) An analytic approach to the verification of iterative programs, In: Information Processing (Proc. IFIP Congress 1977, B.Gilchrist ed.) North Holland 1977, 285-290

Blikle, A.(1977D) A comparative review of some program verification methods, In: Mathematical Foundations of Computer Science (Proc. 6th Symposium, Tatranska Lomnica, September 1977, J.Gruska ed.) 17-33, Lecture Notes in Computer Science, Springer Verlag, Heidelberg 1977

Burstal, R.M. and Darlington, J.(1977) A transformation system for developing recursive programs, Journal of ACM 24 (1977),44-67

Darlington, J.(1975) Applications of program transformation to program synthesis, Proc. Symp. on Proving and Improving Programs, Arc-et-Senans 1975, 133-144

Darlington, J.(1976) Transforming specifications into efficient programs, In: New Directions in Algorithmic Languages 1976 (S.A. Schuman ed.) IRIA Rocquencourt 1976

Dershowitz, N. and Manna, Z.(1975) On automating structured programming, Proc. Symp. on Proving and Improving Programs, Arc-et-Senans 1975

Dijkstra, E.W.(1968) A constructive approach to the problem of program correctness, BIT 8 (1968), 174-186

Dijkstra, E.W.(1972) Notes on structured programming, In: Structured Programming, by O.J.Dahl, E.W.Dijkstra, C.A.R. Hoare, Academic Press, London 1977

Dijkstra, E.W.(1975) Guarded commands, non-determinancy and a calculus for the derivation of programs, Proc. 1975 Int. Conf. Reliable Software 1975, pp.2.0-2.13, also in Comm. ACM, 18 (1975) 453-457

Emden van, M.H. (1975) Verification conditions as representations for programs, manuscript, Waterloo, Ontario, 1975

Emden van, M.H.(1976) Unstructured systenatic programming, Dept.of CS, University of Waterloo, CS-76-09 (1976)

Irlik, J.(1976) Constructing iterative version of a system of recursive procedures, In: Mathematical Foundations of Computer Science (Proc. 5th Symposium, Gdansk,September 1976, A.Mazurkiewicz ed.), LNCS No 45, Springer Verlag, Heidelberg 1976

Irlik, J.(1978) A system of recursive programming, In: Mathematical Foundations of Computer Science 1978 (Proc. 7th Symposium, Zakopane, September 1978, J.Winkowski ed.) LNCS, Springer Verlag, Heidelberg 1978

Liskov, B.H. and Zilles, S.N.(1975) Specification techniques for data abstraction, IEEE Trans. on SE. Se-1 No 1 (1975), 7-19

Meertens, L.(1976) From abstract variable to concrete representation, In: New Directions in Algorithmic Languages 1976 (S.A.Schuman ed.) IRIA, Rocquencourt 1976

Sinzoff, M.(1977) Inventing program construction rules, manuscript

Spitzen, J.M., Levitt, K.N. and Lawrence, R.(1976) An example of a hierarchial design and proof, In: New Directions in Algorithmic Languages 1976 (S.A.Schuman ed.) IRIA, Rocquencourt 1976

Wegbreit, B.(1976) Goal-directed program transformations, IEEE Trans. SE, Vol.SE-2 No 2 (1976), 69-79