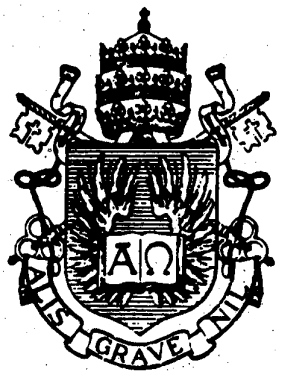

PUC

Series: Monografias em Ciência da Computação
Nº 25/77

A VIEW OF THE PROGRAM DERIVATION PROCESS BASED ON INCOMPLETELY DEFINED DATA TYPES A CASE STUDY

by

Carlos J. Lucena
Tarcísio H. C. Pequeno

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 — ZC-19
Rio de Janeiro — Brasil

Series: Monografias em Ciência da Computação
Nº 25/77

Series Editor: Michael F. Challis December 1977

A VIEW OF THE PROGRAM DERIVATION PROCESS BASED ON INCOMPLETELY DEFINED DATA TYPES: A CASE STUDY*

by

Carlos J. Lucena
Tarcísio H.C. Pequeno

* This work was partially sponsored by FINEP.

ABSTRACT:

The present paper discusses some issues in program synthesis by relating the idea of systematic program derivation with the concepts of data type and correctness of data representation. The notion of an incomplete definition of a data type at a high level of abstraction is introduced. The ideas are illustrated through an example previously discussed in the literature by D. Gries.

KEY WORDS: Program synthesis, program derivation, program specification, program schema, data types, correctness of data representations.

RESUMO:

O presente trabalho discute algumas sugestões em síntese de programas relacionando a idéia de derivação sistemática de programas com os conceitos de tipo de dados e correção de representação de dados. A noção de uma definição incompleta de um tipo de dados a um alto nível de abstração é introduzida. As idéias são ilustradas através de um exemplo previamente discutido na literatura por D. Gries.

PALAVRAS CHAVE: Síntese de programas, derivação de programas, especificação de programas, esquema de programa, tipos de dados, correção de representação de dados.

CONTENTS

1. INTRODUCTION

The state of the art in the area of program verification, started in the late 60's by Floyd [1], is now reaching a stage in which many of the existing results about the analysis of programs are starting to be transferred to practice through research on software engineering. Efforts have now turned to the goal of providing a methodology for the systematic (possibly semi-automatic) synthesis of programs. In fact, many people are presently working on the problem of deriving a program from a given program specification. Our goal in the present work is to contribute to a better understanding of this process.

Depending on the system of notation which is used to express the program specification, the above problem can be seen either as deriving a practical program from an inefficient one (vis à vis the current computer architectures) or from a less operational program statement. It is possible to include in the first category the works by Burstall and Darlington [2], Bauer [3] and Arsac [4]. These authors have chosen to specity program through recursion equations. The recursive form being well adapted to manipulations (transformations) allows for the establishment of a set of rules to transform programs (specifications), written for maximal clarity, into practical or adequately efficient programs. Most of these efforts are being accompanied by the development of software systems for program development.

The basic approach taken for the synthesis of the usual Algol-style form of a program, requires the use of non-procedural program specifications. One handle to the problem solution can be in this case the use of Dijkstra's ideas on constructive programming together with Hoare's rules for the verification of Algol-like programs. Dijkstra has recently been exploring the idea of predicate transformers [5,6] and proposes a methodology for the derivation of programs from their post-conditions and pre-conditions (specifications). Manna, together with several collaborators e.g. [7], developed work on a spectrum ranging from the automatic synthesis of simple programs specified by their input and output assertions to the design of an interactive system in which the computer takes the more straightforward steps on its own, while the human guides the machine in the more creative ones.

Since programmers will not be inventing completely new programs all the time, efforts are being made to provide program-writing systems with the capability to learn from old programs. Gerhart [8,9] has been working on the compilation of a handbook of program schemas which can be abstracted from most current programming applications and that can be used for the synthesis of more complex programs.

In the above, we attempted to make a very brief summary of some of the current ideas on program synthesis. While these efforts are taking place, an overwhelming majority of programmers spend their time writing programs in Algol-like languages with very little understanding about the objects they are producing. A very important task to be undertaken at the present, consists of explaining to these programmers some of the more established ideas on the nature of the program synthesis process.

Gries [10] contributed to this purpose by illustrating some of Dijkstra's ideas while applying them to a reasonably typical programming example. Our attention was called to the fact that the current practice of separating the algorithm and the data aspects of a program in the program development process (program = algorithm + data structures [11]) was not taken into consideration in Gries example. In fact, the synthesis counterpart of Hoare's ideas on the correctness of data representations [12] can be found, in a formalized way, only in conjunction with Burstall's work on recursive programs [2]. Works by Liskov and Zilles [13], Dennis [14] and Wulf [15], have dealt with some of the advantages (mostly from the point of view of programming practice) of program development through the use of data abstractions. These works are very important for the discussion of the synthesis of Algol-like programs.

In what follows we discuss an alternative solution to the example proposed by Gries in [10]. For explaining the synthesis ideas we use the language of first order predicate calculus which is adequate for a critical approach and discussion of the problem. We leave out all considerations about efficiency to concentrate on the discussion of the viability of integrating the notion of a data type specification with Hoare's or Dijkistra's style of proof rules. Our intent is not to propose a methodology for program derivation but rather to try to contribute to the understanding of the program derivation process.

2. THE LINE-JUSTIFIER EXAMPLE: SYNTHESIS OF THE PROGRAM SCHEMA

As stated above, we will present our view about the understanding of the program synthesis process as we discuss the derivation of a program for an example suggested by Gries. The emphasis will be on the interaction between the synthesis of a program schema and the synthesis of a data representation model for the program.

2.1. The Problem Statement

A line-justifier is the part of a text editor that inserts blanks between the words in a line in a way which avoids the existence of blanks after the last word or before the first word in the line. We want to construct a line-justifier program according to the following specifications:

(i) It accepts as input a numbered left justified line having more than one word in which there will be just one blank between words and possibly several blanks after the last word.

(ii) It will produce as output a justified line, that is, a line in which the extra blanks to the right of the last word will have been distributed in the spaces between the words on the line. The difference between the number of blanks in two arbitrary intervals will be at most one. When there is a difference, the number of blanks between words will be the same up to a given word in the line; and after this word the number of blanks between words will again be uniform, but there will be either one more or one les than the previous number of blanks. For aesthetic reasons the even lines will have more blanks at the beginning of the line and the odd lines more blanks towards the ends.

2.2. The Type Line

As indicated in the problem statement, the line-justifier program will manipulate objects called lines. For that reason we shall define a data type line formed by the set of these abstract objects (ambiguously also called lines).

The informal problem statement refers to restrictions associated with lines that are to be accepted by the program. It also refers to the characteristics of the output lines and distinguishes between the treatment to be given to odd and even lines. These facts suggest the definition of the following functions and predicates:

(i) A unary predicate is-initial (x) that determines if a line x satisfies the input restrictions.

(ii) A binary predicate is-just (x,y) that determines if a line y is the result of the justification of a line x.

(iii) A pair of unary operations for line justification. The first will perform the justification by inserting a larger number of extra blanks to the left of the line and the second by inserting a larger number of extra blanks to the right. They will be called just-left (x) and just-right (x) respectively.

(iv) A unary predicate even (x) that determines if the number associated with a line x is even.

The above operations define a first order language L = <is-initial, is-just, even, just-left, just-right> that we will use to talk about lines. The informal program specification given in 2.1 requires that the above described operations satisfy the following self-explanatory axioms:

(1) is-initial (x) $\wedge$ even (x) $\rightarrow$ is-just (x, just-left (x))

(2) is-initial (x) $\wedge \neg$ even (x) $\rightarrow$ is-just (x, just-right (x))

It is interesting to note that the above axioms define a class of data types. In fact, to be able to define the type completely, we would have to state some further properties about the operations and predicates which would capture the details of the justification method contained in the informal problem definition. We will see that in the present case we will be able to do that when we associate a particular representation to the type (an interpretation of the specification). The properties expressed through the given axioms are sufficient for the derivation of a program schema which will, as a first approximation, solve a class of line-justification problems.

2.3. Program Schema Derivation

Using the L-language defined above, the program specification can now be re-stated in the following manner:

{is-initial (x)} P(x,y) {is-just (x,y)}

In the input-assertion (pre-condition) {is-initial (x)}, the variable x is an input variable. In the output-assertion (post-condition) {is-just (x,y)} the variable y is both a program variable and an output variable. Our goal, at this point, is to derive the program P(x,y).

For an x, such that, is-initial (x) is true, the two axioms given above can be re-stated as:

$$[\text{even}(x) \rightarrow \text{is-just}(x, \text{just-left}(x))]$$
$$[\neg \text{even}(x) \rightarrow \text{is-just}(x, \text{just-right}(x))]$$

The axioms so expressed suggest the use of the ifthenelse, with even (x) as the predicate. Thus we get the following program:

{is-initial (x)}
P(x,y) ≡ if even (x)
then $S_1(x,y)$
else $S_2(x,y)$
fi
{is-just (x,y)}

The ifthenelse verification rule reads as follows [16]:

$$\frac{\{Q \wedge t\}\ S_1\ \{R\},\ \{Q \wedge \neg t\}\ S_2\ \{R\}}{\{Q\}\ \text{if } t \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{R\}}$$

In this particular case, we have

Q = is-initial (x);
t = even (x);
R = is-just (x,y).

In program analysis, verification rules are applied by checking the pre and post conditions of the antecedent of the rule to allow the statement of the expression used as its consequent. When deriving a program we must invert this process. The application of the rule consists now of using the expression proposed as the consequent to derive the pre and post conditions of the program segments structured by the control mechanism defined by the rule. Using the above rule for synthesis purposes, we can state;

(i) $\{\text{is-initial}(x) \wedge \text{even}(x)\}\ S_1(x,y)\ \{\text{is-just}(x,y)\}$ and

(ii) $\{\text{is-initial}(x) \wedge \neg \text{even}(x)\}\ S_2(x,y)\ \{\text{is-just}(x,y)\}$

By "modus ponens" of axiom 1 with the pre-condition of (i) above, we have:

$\{\text{is-just}(x, \text{just-left}(x))\}$

Analogously, for (ii) we can write:

$\{\text{is-just}(x, \text{just-right}(x))\}$

We are now ready to apply the assignment rule [16], which reads

$\{Q(x, f(x))\}\ y := f(x)\ \{Q(x,y)\}$

Its application will produce

$S_1(x,y) \equiv y := \text{just-left}(x)$ and
$S_2(x,y) \equiv y := \text{just-right}(x)$

which implies the following program:

```
        {is-initial (x)}
P(x,y) ≡ if even (x)
            then y:= just-left (x)
            else y:= just-right(x)
         fi
         {is-just(x,y)}
```

This program schema can be encoded in the following CLU-like [13] notation:

```
line-justifier = procedure (x:line) returns (line)
  y:line;
    ¢  is-initial(x) ¢
  if line$even(x)
       then y:= line$just-left(x)
       else y:= line$just-right(x)
  fi
  return (y);
  ¢ is-just(x,y) ¢
  end line-justifier
```

The reason for encoding the program schema in CLU is that we shall later make use of CLU's cluster mechanism for expressing data types. We must, however, call the reader's attention to the fact that we are not bound to any particular programming language. A programmer, in the context of our work, can choose to use any control or data structure, providing he is able to state its axioms formally.

3. THE LINE-JUSTIFIER EXAMPLE: SYNTHESIS OF THE DATA REPRESENTATION

So far we have abstracted some properties of any representation of the object line. We are now going to associate a specific model (representation + operations) with the theory defined by the axioms in the language L.

By the problem definition, our program receives as input a line expressed in a given representation and produces as output a line expressed in the same representation. Therefore the line representation is an integral part of the problem definition. We are going to solve the proposed problem through the use of two different representations. The first one is a simple representation that exactly matches the specific problem, and the second is a representation similar to that adopted in Gries' solution [10].

We are initially going to think of a line as a 6-tuple of natural numbers, having the following components:

p - number of blanks in the leftmost intervals of the line.
q - number of blanks in the rightmost intervals of the line.
t - index of the word after which the number of blanks changes.
n - number of words on a given line.
s - number of extra blanks at the end of the line.
z - line number.

Note that the program being developed does not handle the text itself. We, as Gries did, suppose that the text was pre-processed to produce a representation that contains only the aspects directly related to the problem. In fact, Gries includes some extra information in his representation, and we will do approximately the same in our second choice of representation.

The domain of the type line, which we will also call line, will be the following subset of $\mathbb{N}^6$:

$$\underline{\text{line}} = \{\langle p,q,t,n,s,z\rangle \in \mathbb{N}^6 \mid t \leq n \wedge n>1 \wedge |p-q| \leq 1 \wedge p \geq 1 \wedge q \geq 1\}$$

We will now define the operations on the type line in our model. For this purpose we will use the variables

$x = \langle p, q, t, n, s, z\rangle$ and
$x' = \langle p', q', t', n', s', z'\rangle$

The proposed definitions are the following:

(i) is-initial (x) $\stackrel{df}{=}$ $p = q = 1 \wedge t = n$;

(ii) even (x) $\stackrel{df}{=}$ z is even;

(iii) just-left (x) $\stackrel{df}{=} \langle p + s/(n-1) + 1,$
$q + s/(n-1)\ ,$
$\text{mod}\ (s,(n-1)) + 1,$
$n, 0, z\rangle$;

(iv) just-right (x) $\stackrel{df}{=}$ <p + s/(n-1),
q + s/(n-1) + 1,
n - mod(s,(n-1)),
n, o, z>;

(v) is-just (x,x') $\stackrel{df}{=}$ s' = 0 $\wedge$ s + p(t-1) + q(n-t) = p'(t'-1) + q'(n'-t') $\wedge$ n=n' $\wedge$ z=z'

3.1. Verification of the Representation

To verify the proposed data representation we need to prove first that its functions and relations are well defined in the specified domain. The relations are obviously well defined since they are expressed in terms of the operations and predicates defined over the naturals. We must then start by proving the closure of the functions which alter objects of type line.

The next step will be to prove that we have defined a model leder of the theory (types) proposed in section 2. In other words we need to verify if the model satisfies axioms 1 and 2.

Although just-right and just-left are applied only once in the present example, the closure of these operations will be checked to guarantee that the output predicate is applicable.

3.2. Proof of the Closure Property

just-left (x) = x' = <p', q', t', n', s', z'>

(i) p' = p + s/(n-1) + 1

Since n>1, n-1 is a natural different from 0, therefore s/(n-1) (integer division) is a natural. Since p$\in$ $\mathbb{N}$ and $\mathbb{N}$ is closed under addition, p' $\in$ $\mathbb{N}$ and p' $\geq$ 1. $\square$

(ii) We can analogously conclude that $q' \in \mathbb{N}$ and since $q' = q + s(n-1)$ we have $p' = q' + 1$ or $p' - q' = 1$ and therefore $|p' - q'| \leq 1$. We also have $q' \geq 1$. □

(iii) $t' = \text{mod}\ (s,(n-1)) + 1$
Since $\text{mod}\ (s,(n-1)) < n-1$ we have that $t'-1 < n-1$ or $t' < n$.
Since $n' = n$, then $t' < n'$. □

(iv) Since $n' = n$, $s' = 0$, $z' = z$ we trivially have that $n', s', z' \in \mathbb{N}$ and $n' > 1$. □

The closure of just-right can be verified through the same procedure.

3.3. Proof of the Satisfiability Property

We will now prove that we do have a model that satisfies the given theory. We will show that the model satisfies axiom 1 of section 2 and will leave the proof of satisfiability of axiom 2 to the reader.

Since even (x) $\wedge$ is-initial (x), we can write:

z is even, p = 1, q = 1 and t = n.

Now, let x' = just-left (x), that is,

$$p' = p + s/(n-1) + 1 = 1 + s/(n-1) + 1$$
$$q' = q + s/(n-1) = 1 + s/(n-1)$$
$$t' = \text{mod}\ (s,(n-1)) + 1$$
$$n' = n,\ s' = 0,\ z' = z$$

We must show that is-just (x,x') is true. Since we have that $s'=0$ we need only to verify that

$$s + p(t-1) + q(n-t) = p'(t'-1) + q'(n'-t')$$

Let us now replace in the right hand side of the above equality the values of p', t', q', n':

$$(2 + s/(n-1))(\mathrm{mod}(s,(n-1)) + 1 - 1) + (1 + s/(n-1))(n-\mathrm{mod}(s,(n-1))-1) =$$

$$2\ \mathrm{mod}(s,(n-1)) + \frac{s}{n-1}\ \mathrm{mod}(s,(n-1)) + (n-1) + (n-1)\ \frac{s}{n-1} - \mathrm{mod}(s,(n-1)) -$$

$$\frac{s}{n-1}\ \mathrm{mod}(s,(n-1)) = \mathrm{mod}(s,(n-1)) + (n-1) + \frac{s}{n-1}\ (n-1) =$$

$$s - \frac{s}{n-1}\ (n-1) + (n-1) + \frac{s}{n-1}\ (n-1) = s + n - 1$$

Replacing the values of p, q and t in the left side of the equality above, we get

$$s + 1\ (n-1) + 1\ (n-n) = s + n - 1$$

It is trivial to show through an analogous procedure that axiom 2 is satisfied.

In section 2.2. we formalized through axioms (i) and (ii) some aspects of the informal problem definition. These aspects were chosen to be those informally considered necessary for the derivation of a program schema that captures the general idea suggested by the informal problem definition. Therefore, axioms (i) and (ii) were not meant to define the type line completely. As we moved to the representation level our model was intentionally required to satisfy some more axioms (not explicitly stated) which refer to the additional program requirements contained in the problem definition.

Following our approach the complete type specification comprises both the so-called abstract and representation levels [17,18] and each cannot be used independently to derive a program to solve the problem completely. We are sacrificing the modularity principle looked for in [17,18], where one expects to be able to use each different level of data abstraction as a base machine in which the problem can be completely solved. Instead, our goal is to try to enhance the visibility of the problem solving aspects involved in the program construction process.

3.4. Derivation of the Cluster for the Data Representation

Having shown that we have a legitimate model, we need now to produce a programmed version of the model. For that purpose we are going to use the cluster mechanism, as proposed in [13]. The cluster will contain a representation (global to the procedures in the cluster) whose invariant is to be a line (defined above). The invariant has been verified in 3.3 and will remain valid if the operations in the cluster follow their respective definitions. For the various procedures implementing the operations within the cluster, the post-condition is the definition of the operation and the pre-condition is the invariant line.

The derivation of the programs implementing the various operations is a very simple exercise for the present example. All that needs to be done is the successive application of the rules of assignment and concatenation to the procedures' post-conditions. Since we have already derived in section 2 a program segment by using the assignment rule, we are going to omit this straightforward discussion from the text.

The cluster program has, for the present case, the following form:

```
line = cluster is even, just-left, just-right;
rep  = record (p:integer; q:integer; t:integer; n:integer; s:integer;
               z:integer);

create
   ℓ:rep;
even = oper (x:cvt) returns(boolean);
        return(EVEN(z));
        end even;
just-left = oper (x:cvt) returns (cvt);
                  ℓ.q:= x.q + x.s/(x.n-1),
                  ℓ.p:= ℓ.q + 1;
                  ℓ.t:= mod(x.s, (x.n-1)) + 1
                  ℓ.s:= 0; ℓ.n:= x.n; ℓ.z:=x.z;
                  return (ℓ);
                  end just-left;
```

```
just-right = oper (x:cvt) returns (cvt);
                ℓ.p:= x.p + x.s/(x.n-1);
                ℓ.q:=  .p + 1;
                ℓ.t:= x.n-mod(x.s,(x.n-1));
                ℓ.s:= 0; ℓ.n:= x.n; ℓ.x:= x.z;
                return (ℓ);
                end just-right;
            end cluster;
```

We have then derived a program for the given specification, that is, a program that caputures what was stated by the original problem definition.

4. CHANGE OF THE DATA REPRESENTATION

The derivation of the programmed data representation, as it was proposed before, was extremely simple because we adopted a minimal (in some sense) configuration for the representation data space. It contained just the necessary elements for the satisfaction of the problem specification. In Gries' example [10] some extra features were added to the representation. In fact, he uses an array of indices that indicates where each word in the text begins. The reader who wants to compare the two solutions must pay attention to the fact that the meanings of the naturals p and q in Gries' example are sliglity different from ours, since in his example they stand for the number of blanks to be inserted between the words. It is interesting to show how our program can be modified so that we can use a similar model.

Let A be the set of all arrays of naturals and b a variable ranging over its domain. We need now to re-state the domain line.

Let us now call line, the following set

$$\text{line} = \{\langle p,q,t,n,s,z,b\rangle \in \mathbb{N}^6 \times A \mid t \leq n \wedge n > 1 \wedge |p-q| \leq 1 \wedge |b| = n \wedge b_1 =$$

$$1 \wedge (1 \leq i < t) \rightarrow b_{i+1} > b_i + p \wedge (t \leq i < n) \rightarrow b_{i+1} > b_i + q\}$$

We shall now define the operations of the new model. The operations is-initial and even will have the same definitions as before. The operations just-left and just-right will now be defined as follows:

just-left $(\langle \bar{x}, b \rangle) = \langle \bar{x}', b' \rangle$ and

just-right $(\langle \bar{y}, b \rangle) = \langle \bar{y}', b' \rangle$

where $\bar{x}'$ and $\bar{y}'$ are to be computed as in the previous definitions of just-left and just-right and b', for both definitions, will have the following meaning:

$$b' = \begin{cases} \text{for } 1 \leq i \leq t' \quad , \quad b'_i = b_i + (p'-p)\,(i-1) \\ \text{for } t' < i \leq n \quad , \quad b'_i = b_i + (p'-p)\,(t'-1) + (q'-q)\,(i-t') \end{cases}$$

Finally, given $x = \langle p,q,t,n,s,z,b \rangle$ and $x' = \langle p',q',t',n',s',z',b' \rangle$ we can define the operation is-just

$$\text{is-just}(x,x') \overset{df}{\equiv} s' = 0 \wedge s+p\,(t-1) + q(n-t) = p'(t'-1) + q'(n'-t') \wedge n=n' \wedge z=z' \wedge$$
$$(1 \leq i \leq t') \rightarrow b'_i = b_i + (p'-1)\,(i-1) \wedge$$
$$(t' \leq i \leq n) \rightarrow b'_i = b_i + (p'-1)\,(t'-1) + (q'-1)\,(i-t')$$

4.1. Proof of the Closure Property

Having defined the new model, it is necessary to verify the closure of the above operations and the fact that line, as defined above, is a model of the theory used for the program definition. We will restrict ourselves to showing the closure of one of the operations (just-left) with respect to the part of the predicate which was added to the first definition (section 3). The verification of the other aspects are trivial and we leave them to the reader.

(i) $|b'| = |b|$ and therefore, since $n' = n$ and $n = |b|$, we have that $n' = |b'|$. □

(ii) $b'_1 = b_1 + (p'-1)\,(1-1) = b_1 = 1$. □

(iii) For $1 \leq i < t'$, $b'_{i+1} = b_{i+1} + (p'-1)\,(i+1-1) = b_{i+1} + (p'-1)i$ since

$b_{i+1} > b_i + p$, then $b'_{i+1} > b_i + p + (p'-1)i$ (a)

We have that

$$b'_i = b_i + (p'-1)(i-1) = b_i + (p'-1)i - (p'-1)$$

$$b'_i + p' = b_i + (p'-1)i - (p'-1) + p', \quad \text{then}$$

$$b'_i + p' = b_i + (p'-1)i + 1 \qquad (b)$$

Since $p \geq 1$ we have $b_i + p + (p'-1)i \geq b_i + (p'-1)i+1$, thus from (a)

$$b'_{i+1} > b'_i + p'. \square$$

(iv) For $i = t'$, $b'_{t'+1} = b_{t'+1} + (p'-1)(t'+1-1) + (q'-1)(t'+1-t')$

$$= b_{t'+1} + (p'-1)t' + (q'-1)$$

$$b_{t'+1} > b_{t'+q}, \quad b'_{t'+1} > b_{t'} + q + (p'-1)t' + (q'-1)$$

$$= b_{t'} + q + (p'-1)t' + q' - 1 \qquad (c)$$

$$b'_{t'} = b_{t'} + (p'-1)t' - (p'-1)$$

$$b'_{t'} + q' = b_{t'} + (p'-1)t' - (p'-1) + q'$$

$$= b_{t'} + (p'-1)t' - p' + 1 + q' \qquad (d)$$

Since $q, p' \geq 1$ and $q-1 > 1-p'$, then (c)>(d) and therefore $b'_{t'+1} > b'_{t'} + q'$. $\square$

(v) For $t' < i < n$,

$$b'_{i+1} = b_{i+1} + (p'-1)(t'-1) + (q'-1)(i+1-t')$$

$$= b_{i+1} + (p'-1)(t'-1) + (q'-1)(i-t') + (q'-1)$$

$$b'_{i+1} > b_i + q + (p'-1)(t'-1) + (q'-1)(i-t') + q' - 1 \qquad (e)$$

$$b'_i = b_i + (p'-1)(t'-1) + (q'-1)(i-t')$$

$$b'_i + q' = b_i + (p'-1)(t'-1) + (q'-1)(i-t') + q' \qquad (f)$$

We will call (g) on (h) the right hand sides of (e) and (f), respectively.

Since $q \geq 1$, (g) $\geq$ (h) and

$b'_{i+1} > b'_i + q'$. □

4.2. Derivation of the Cluster for the New Representation

We have now to encode the new representation as a cluster of procedures. As mentioned before, the procedures which implement the operations can be derived from the definitions of the operations. We shall illustrate this procedure by deriving the operation just-left. This way we have the opportunity of using for the first time in this paper the proof rule for an iteration (even though a simple one).

Input variables: $\langle p,q,t,n,s,z,b\rangle = \bar{x}$
Program variables: $\langle p',q',t',n',s',z',b'\rangle = \bar{y}$
Pre-condition: T
Post-condition: (a) $p' = p+s/(n-1) + 1 \wedge q' = q+s/(n-1) \wedge$

$t' = \mathrm{mod}(s,(n-1))+1,\ n'=n,\ s'=0,\ z'=z \wedge$

(b) For $1 \leq i \leq t'$, $b'_i = b_i + (p'-1)(i-1) \wedge$

(c) For $t' < i \leq n$, $b'_i = b_i + (p'-1)(t'-1) + (q'-1)(i-t')$

We shall call Q the predicate that expresses the post-condition and will split it into three components, such that

$$Q(\bar{x},\bar{y}) = (a) \wedge (b) \wedge (c)$$

To the predicates (a), (b) and (c) correspond three program segments, which can be expressed in the following form:

$$\{T\}\ s_1(\bar{x},\bar{y})\ \{(a)\}\ s_2(\bar{x},\bar{y})\ \{(a)\wedge(b)\}\ \ s_3(\bar{x},\bar{y})\ \ \{(a)\wedge(b)\wedge(c)\}$$

We shall derive the program segment s_2:

$$\{(a)\}\ s_2(\bar{x},\bar{y})\ \{(a)\wedge \text{for } 1\leq i\leq t',\ b_i' = b_i + (p'-1)(i-1)\}$$

The predicate (b) suggests the utilization of an iterative control structure with the following form

$$\{(a)\}\ \underline{\text{for}}\ \ k:=1\ \ \underline{\text{to}}\ t'\ \underline{\text{do}}\ S\ \{(a)\wedge(b)\}$$

The proof rule for the for statement can be expressed as follows [16]:

$$\frac{\{(a\leq k\leq b)\wedge P\ \ ([a..k-1])\}\ \ S\ \{P([a..k])\}}{\{T\}\ \text{for } k:=a\ \underline{\text{to}}\ b\ \underline{\text{do}}\ S\ \{P([a..b])\}}$$

In the present case $P \equiv (b)$, hence through the application of the rule we get

$$\{(a)\wedge(1\leq k\leq t')\wedge(b)\ [a..k-1]\ \ S\ \{(b)\ \ [a..k]\}$$

If we now apply the assignment rule, we can state

$$S \equiv b_k = b_k + (p'+1)(i-1)$$

The program segment S_2 will then have the following form

$$\{(a)\}\ \underline{\text{for}}\ k:=1\ \underline{\text{to}}\ t'\ \underline{\text{do}}$$
$$b_k = b_k + (p'+1)(i-1)$$
$$\{(a)\wedge(b)\}$$

The segment S_3 can be obtained in a similar manner. The derivation of S_1 will be the result of the successive application of the assignment rule, leading to a result which is identical to the one produced for the first data representation (tuple of naturals). The expression of the cluster follows directly from what was said and is left as an exercise to the reader.

5. CONCLUSIONS

We have discussed the synthesis process of an Algol-like program by dealing separately with the algorithm and data aspects of the program. For the establishment of this separation we have used the concept of a cluster which is instrumental in providing a programming mechanism for the encoding of the data representation. The same effect could be obtained by mechanisms such as classes [12] and forms [20].

In deriving a program statement we have proceeded through three distinct phases: derivation of a program schema from a formalized version of the problem definition; derivation of the problem data type ultimately in terms of formally well known and more primitive types (naturals and arrays in the given example); derivation of a programmed version of the data type definition (synthesis of the cluster). We not only defined the problem data type but also checked its correctness. The checking procedure differs slightly from Hoare's [12] since we do not start from a completely defined type and a completely defined representation and try to define a mapping function connecting them. Instead, we express the model constructively in terms of the representation and then verify if it is in fact a model of the theory (an incompletely defined type). Some authors have been working on algebraic approaches to the correctness of data representations (e.g. [21]). Pequeno and Veloso [22] also are presently working on a formalization of some of the ideas presented in this paper.

We are presently trying to expand and formalize the concepts presented above through a simple example in search of a better understanding of the program derivation process. We are also investigating the idea of dealing with the problem of program transformations (in Gerhart's sense [9]) viewing these transformations along the two axes dealt with in this paper: algorithm and data.

ACKNOWLEDGEMENT: The authors are grateful to T. Kowaltowski, P. Veloso, F. Tompa, P. Lauer, J. Arsac and D. Berry for their many comments and criticisms on this work.

REFERENCES

[1] Floyd, R.W., "Assigning Meanings to Programs". In "Mathematical Aspects of Computer Science", vol. XIX, v.t. Schwartz (ed.), American Mathema - tical Soci·ty, Providence, Rhode Island, 1967.

[2] Burstall, R.M., and Darlington, J. "A Transformation System for Devel - oping Recursive Programs", J. ACM 24.1, Jan. 1977.

[3] Bauer, F.L., "Programming as an Evolutionary Process". Berisht Nr.7617. Institut F'r Informatik, Technische Universität München, 1976.

[4] Arsac, J. Nouvelles Lessons de Programmacion.Dunod, Paris, to appear.

[5] Dijkstra, E., "Guarded Commands, Non-determinacy and a Calculus for the Derivation of Programs", Proceedings of the International Conference on Reliable Software, 1975.

[6] Dijkstra, E., "A Discipline of Programming", Englewood Cliffs, N.J., Prentice Hall, 1976.

[7] Manna, Z., and Waldinger, R. "Knowledge and Reasoning in Program Synthesis. Artif. Intel. J. 6.2, 1975.

[8] Gerhart, S.L., "Knowledge About Programs: A Model and a Case Study". Proceeding of the International Conference on Reliable Software, Los Angeles, 1975.

[9] Gerhart, S.L., "Proof Theory of Partial Correctness Verification Systems." SIAM J. Comput. Vol. 5, No.3, 1976.

[10] Gries, D., "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs", IEEE Transactions on Software Engineering, Vol. SE-2, Nº 4, 1976.

[11] Wirth, N., "Algorithms + Data Structures = Programs", Prentice Hall, 1976.

[12] Hoare, C.A.R., "Proof of Correctness of Data Representations". Acta Informatica 1, 1972.

[13] Liskov, B. and Zilles, S., "Programming with Abstract Data Types". Proceedings of ACM SIGPLAN Conference on Very High level Languages, 1974.

[14] Dennis, J. "An Example of Programming with Abstract Data Types". Sigplan Notices v. 10, n.7, 1975.

[15] Wulf, W.,London R.L. and Shaw M. "An Introduction to the Construction an Verification of Alphard Programs". IEEE Transactions on Software Engineering Vol. SE-2, N.4, 1976.

[16] Hoare, C.A.R. and Wirth, N. "An Axiomatic Definition of the Programming Language Pascal". Acta Informatica 2, 1973.

[17] Liskov, B., Zilles, S. "Specification Techniques for Data Abstractions". IEEE Transactions on Software Engineering, Vol. SE-1, No.1, 1975.

[18] Guttag, J. "Abstract Data-Types and the Development of Data Structures. CACM, Vol. 20, No. 6, 1977.

[19] Dahl, O. and Hoare C.A.R., "Hierarchical Program Structures." It Structured Programming, Academic Press, 1972.

[20] Wulf, W.A. "Alphard: Toward a Language to Support Structured Programs. Technical Report, Carnegie-Mellon University, 1974.

[21] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. "Abstract Data Types as Initial Algebras and the Correctness of Data Representations". Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure, May, 1975.

[22] Pequeno, T., Veloso, P. - Private Communication.