Information Technology

JCIT3

PREPRINTS

Edited by
J. Moneta

North-Holland

Information Technology

Proceedings of the 3rd Jerusalem Conference
on Information Technology (JCIT3)
Jerusalem, August 6 - 9, 1978

PREPRINTS

Edited by
JOSEF MONETA

1978

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM · NEW YORK · OXFORD

Information Technology, J. Moneta (editor)

SOME THOUGHTS ON THE CONSTRUCTION OF PROGRAMS - A DATA-DIRECTED APPROACH

D.D. Cowan
Computer Science Department
University of Waterloo
Waterloo Ontario Canada.

Carlos J.P. Lucena
Departmento de Informatica
Pontificia Universidade Catolica
Rio de Janeiro - Brazil

ABSTRACT

The present paper discusses a method of program construction based on the specification of the data types. The input and output data types and the mapping between them are specified at a high level of abstraction and this non-procedural specification is used to develop a program schema. The data type and mapping specifications are modified to include a concrete representation of the data and these are used to expand the program schema into a program. The method is illustrated by developing a program for the bubblesort.

KEY WORDS: Program construction, program derivation, program specification, program schema, data types.

1. INTRODUCTION

There are large collections of programs which have a long lifetime but which are frequently being modified. Since modifications are so important, it is reasonable to develop a disciplined approach to deriving a program from its non-procedural specification such that future modifications are as easy as possible. Various attempts at such methods are exemplified by Dijkstra's Structured Programming [1] and predicate transformers [6,7], Manna's automatic synthesis techniques [8], Wirth's Systematic Programming [2], Jackson's method [3] and recent work reported in [9].

It has been observed that many changes in typical data processing programs are caused by changes in the structure of the data to be processed. Hence, if a program's structure can be made to resemble the data structure it processes, then modifications to the data structures could easily be reflected in the program. This paper describes a disciplined approach to program construction which leads to programs that can be systematically maintained since their structure closely follows the structure of the data they process.

In developing this approach we take two specific views of the programming process. First we adopt the approach which has been described by Hoare [1,4] namely that the structuring of data should be handled by the following three mechanisms: direct or Cartesian product, discriminated union, and sequence. Second we use the approach of Jackson [3] that a program is a mapping which transforms the input data as described by a data-type specification into the output data also described by a data-type specification. Both Jackson [3] and Hoare [4] observed that the control structures which are required to handle data structures of types record, sequence, and discriminated union are the simple sequencing of statements, a looping construct such as the while-do, and the case or if-then-else statement, respectively. This observation will also be used in the development of our programs.

In order to present our ideas in a clear manner we shall treat an example. This example, a sort program, is used to illustrate the notation and also to introduce the concept of data levels of abstraction into the method.

2. A SORTING PROGRAM: - CONSTRUCTION OF A PROGRAM SCHEMA

This section discusses the derivation of a program for an internal sort. We illustrate our techniques by deriving a program schema which operates on an abstract definition of the input and output data types. In the next section one concrete representation for the data will be used to refine the sort to produce a bubblesort and to illustrate how one might move through levels of abstraction and develop an operational program.

2.1 Data Type Specification

A sort can be described as a procedure which takes an unordered set and gradually converts it into an ordered set. In effect, such a procedure takes a sequence of partially ordered sets and transforms this sequence into a sequence of partially ordered sets with the last member of the sequence being completely ordered. We call these partially ordered sets of numbers, arrays. Here the word array is not used in the sense implied by most programming languages although later in the paper we shall use it as a representation for the set we wish to sort.

At a high level of abstraction these arrays will be the lowest level entities manipulated by our program, and hence will be used as the terminal types in our input data specification. In a corresponding fashion the lowest level entities produced by our abstract program are partially-ordered-arrays. These partially-ordered-arrays will be used as terminal data-types in the output data specification. The input specification and output specifications shown in Figures 1 and 2 are in a form similar to Hoare's [1] specification of data types.

type set = sequence of array

Figure 1.

type partially-ordered-set = sequence of partially-ordered-array

Figure 2.

set → partially-ordered-set
array → partially-ordered-array

Figure 3.

2.2 Construction of a Program Schema

Many programs can be viewed directly as transformations of the input data-types to the output data-types. In the current example the abstract program which sorts the set transforms the type set into partially-ordered-set and the type array into partially-ordered-array. This transformation is described by the set of mappings shown in Figure 3. The mappings are mappings from the domain of one type to the domain of the other type. We now describe the steps of an informal method for constructing a program schema from the specifications of Figures 1, 2 and 3.

(i) To construct a program we note that there must exist a function which converts an array into a partially-ordered-array, since there is a direct mapping between them. This operation we call change (array) and it replaces the mapping

array → partially-ordered-array.

The reader should note that the name "array" has been used for three different but related concepts in this paper; "Array" has been used to designate a data type, the domain of the data type array in the mapping, and to represent a variable of type array.

(ii) Both set and partially-ordered-set are represented by sequences and there is a direct correspondence between them. Hoare [4] and Jackson [3] observed that structures defined in this way are controlled in a program by a while-do construct. Hence if we use sort (set) to represent the mapping

set → partially-ordered-set,

then we can attempt to construct the next step in our program. This step becomes

```
sort (array)
initialize
while not ordered (array)
do
   change (array)
od
end
```

The predicate for the while statement in this program should contain a mechanism for transforming the set into a sequence of arrays and then test when there are no more arrays to be sorted. However, we observed earlier that this set of arrays is somewhat artificial, and that really there is only one array which is gradually transformed into a sorted array. Hence the argument of sort is "array" and the only predicate we need is one that tests if the array is sorted. We shall use the predicate ordered (array) to test if the array is sorted. The statement "initialize" indicates that some variables may have to have values before the predicate can be tested.

3. INTRODUCTION OF A DATA REPRESENTATION

In this section we transform the abstract program schema into a procedure by introducing the usual representation of an array; this transformation occurs in two stages. First the array is divided into two parts, an unsorted-part and a sorted-part. Because the unsorted-part is going to be ordered, it is described in more detail as a sequence of overlapping pairs (o-pair) and the overlapping pair is expanded as a discriminated union of good overlapping pair (g-o-pair) or bad overlapping pair (b-o-pair). The extended type specifications and mappings for the first stage are presented in Figures 4, 5 and 6 where the type array is shown as a record whose components are separated by a semi-colon (;) and an o-pair is a discriminated union whose parts are separated by a comma (,). In the second stage the type array will be defined explicitly as an array of integers. The next few steps illustrate the method of constructing the program for the first stage.

```
type set   =  sequence of array
type array  =  (unsorted-part; sorted-part)
type unsorted-part = sequence of o-pair
type o-pair = (g-o-pair, b-o-pair)
```

Figure 4.

```
type partially-ordered-set = sequence of
                partially-ordered-array
type partially-ordered-array = (unsorted-
                part; sorted-part)
```

Figure 5.

```
set → partially-ordered-set
array → partially-ordered-array
unsorted-part → unsorted-part
sorted-part → sorted-part
```

Figure 6.

(i) There is a mapping between each type in the output specification and a corresponding input specification. There is no correspondence at the level of overlapping pairs since although sorted-parts are mapped into sorted-parts and unsorted-parts into unsorted-parts, each pass of the process over the set may make the unsorted-part smaller and the sorted-part larger.

(ii) We now construct procedures for mapping of the two parts of array. These are decrease (unsorted-part) and increase (sorted-part). With these new procedures the program becomes

```
sort (array)
initialize
while not ordered (array)
do
   decrease (unsorted-part)
   increase (sorted-part)
od
end.
```

(iii) The unsorted-part is composed of a sequence of overlapping pairs and must be a program under control of a while-do construct. The predicate must check whether the end of the unsorted-part has been reached. Hence the code for decrease (unsorted-part) is

```
decrease (unsorted-part)
initialize
while not end (unsorted-part)
do
   process-o-pair
od.
```

(iv) The type o-pair consists of a discriminated union of two types and is processed using the if-then-else control structure [3,4]. Process-o-pair becomes

```
if bad (o-pair)
then
   process-b-o-pair
else
   process-g-o-pair
fi.
```

The program we have constructed now has the following form:

```
sort (array)
initialize
while not ordered (array)
do
   initialize
   while not end (unsorted-part)
   do
     if bad (o-pair)
     then
        process b-o-pair
     else
        process g-o-pair
     fi
   od
   increase (sorted-part)
od
end.
```

At this point decisions must be made about the actual form of the procedures and predicates thus forcing us into a final choice of sorting method, namely the bubblesort.

To make sure the next level of program is equivalent to the higher level presented previously we need to re-state the concepts used in the higher level in terms of the new notation. This has been done in Figure 7 where type array is effectively defined as being structured as an array. Note that the name "array" is used as the name of a type and also as the structuring mechanism.

```
type set = sequence of array
type array = (unsorted-part; sorted part)
type sorted-part = array 1..j-1 of integer
type unsorted-part = array j..n-1 of o-pair
type o-pair = (g-o-pair, b-o-pair)
type g-o-pair = (integer; integer)
type b-o-pair = (integer; integer)
```

Figure 7.

We now decide that if a is of type array then we have a bad overlapping pair (b-o-pair) if

$$a_i > a_{i+1}$$

and the values of a_i and a_{i+1} will be interchanged. Process-b-o-pair will be implemented by a procedure swap (x,y). If an o-pair is of type g-o-pair then no processing needs to occur.

Hence process-o-pair becomes

```
if a_i > a_i+1
then
   swap (a_i,a_i+1)
fi
```

Since the representation of the array has been specified, it is now possible to construct the predicate end (unsorted-part) and its initialization. The bubblesort starts with an index i = n-1 (the index of the last overlapping pair in the unsorted-part) and terminates when i < j since the end of the unsorted-part would have been reached.
Hence while not end (unsorted-part) can now be replaced with

```
i:= n
while (i:=i-1)≥j
```

which is equivalent to a for-loop

```
for i:=n-1 downto j.
```

The final step in the construction of our bubblesort program is the construction of the predicate ordered (array).

This predicate will be false only when j > n-1 because the sorted-part of the array will be the full array. The statement

```
while not ordered (array)
```

can be replaced by

```
j:= 1
while j≤n-1
```

This is not quite enough since there must be a method of incrementing j, for the procedure to terminate. This is the function played by the procedure increase (sorted-part). It is simply replaced by j:=j+1. Since j is incremented each time through the while loop the while can be replaced by

```
for j:= 1 to n-1.
```

The entire program (without declarations and with variable name "a" substituted for array) can now be written as:

```
sort (a)
for j:= 1 to n-1
do
   for i:= n-1 downto j
   do
     if a_i > a_i+1
     then
        swap (a_i,a_i+1)
     fi
   od
do
end.
```

4. CONCLUSIONS

We have discussed the construction of an Algol-like program by considering it to be a mapping between input and output data types. Specifically we have followed a procedure consisting of a number of well-defined steps.

First we specified abstract input and output data-types and the mapping between them. We then used this combination to derive a program schema. As a second step the abstract data types are expanded by choosing concrete representations which can be implemented in most Algol-like languages. Of course the mapping between the data types is expanded to include the concrete representations Finally we expand the program by using the concrete representations and the mapping between them.

It should be observed that the program and the data structures it processes are closely related and in fact there is a correspondence between parts of the program and the data structures. Such a relationship implies that changes in data structures, a common occurrence in program maintenance, can be easily reflected in corresponding changes in the program.

The close correspondence between the data structures and the program structure as in the relationship between sequences and loops, and discriminated unions and selection means that a basic program framework can be systematically derived from the syntax of the data structure. This systematic derivation allows the programmer to concentrate on other aspects of the program such as the construction of the correct predicates and the development of the statements within each of the structures.

This formal model of the types and mapping can also be represented graphically. This graphical representation has been found to be quite convenient for expressing ideas about programs, and as a tool to aid in program development, and has been discussed in [9].

The technique described in this paper has been applied to a large number of programs both large and small and has in our limited experience been quite successful. The reader is referred to [9] for a more complete discussion of the ideas which have been by necessity presented rather briefly here.

ACKNOWLEDGEMENTS

The authors wish to acknowledge their indebtedness to J.W. Graham and J.W. Welch of the University of Waterloo and W. Turski of the University of Warsaw; the point of view expressed in this paper resulted from many fruitful discussions with these three individuals.

BIBLIOGRAPHY

[1] O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (1972) Structured Programming pp 1-174 Academic Press.

[2] N. Wirth (1973) Systematic Programming: An Introduction Prentice-Hall.

[3] M.A. Jackson (1975) Principles of Program Design Academic Press.

[4] C.A.R. Hoare (1975) Data Reliability pp 528-533 Proceedings International Conference on Reliable Software April 1975.

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

[6] Dijkstra E.W. (1975) Guarded Commands, Non-determinacy and a Calculus for the Derivation of Programs. Proceedings of the International Conference on Reliable Software April 1975.

[7] Dijkstra E.W. (1976)A Discipline of Programming Prentice-Hall.

[8] Manna Z. and Waldinger R. (1975) Knowledge and Reasoning in Program Synthesis. Artificial Intelligence Journal Vol. 6.

[9] Cowan D.D. and Lucena C.J.P. (1978) A Data-Directed Approach to Program Construction University of Waterloo Computer Science Department CS 78-02