1982

Program development by data abstraction

Reinhold Friedrich Hille
University of Wollongong

PROGRAM DEVELOPMENT BY DATA ABSTRACTION

by

R.F. HILLE

Preprint No. 82-18

PROGRAM DEVELOPMENT BY DATA ABSTRACTION

R.F. Hille

Department of Computing Science
The University of Wollongong
P.O. Box 1144
Wollongong, NSW 2500
AUSTRALIA

ABSTRACT

This paper explains the vital role of data abstraction in the development of computer programs. Abstract data types provide the mechanism to formulate a solution to a computing problem. They transform functions into algorithms and unltimately into programs. They also provide a mechanism for the development of a hierarchy of levels of abstraction, specific to the problem at hand. Examples are presented to explain these concepts. Consequences of this approach for the proof of program correctness are discussed.

Keywords and phrases:
algorithm development, data abstraction, programming methodology.

C.R. Categgories: 1.5, 4.2, 5.7

1. INTRODUCTION

The power and complexity of computing machinery has increased rapidly in recent times, but there has not been a corresponding development of a methodology of programming. Some steps have been made in the right direction, such as "structured programming" [Dahl, Dijkstra, and Hoare, 1972]. It can be argued that applicative languages, whose forerunner LISP, hold much promise for the development of a much more powerful programming methodology than has been achieved through the medium of conventional programming languages, which reflect far too closely the structure of the underlying machinery, and who for that reason contain many features that are inhibitive rather than helpful to program development. Assignment statements and loop constructs are manifestations of that fact. There is absolutely no necessity that the mode of operation of "one word at a time" should be reflected in high level languages. Backus [1978] points out that conventional programming languages are large, complex, and inflexible. He introduces a functional style of programming which facilitates abstraction.

In this paper we give a tutorial style introduction to a functional style of program development which allows arbitrary abstraction.

The ideas presented here are based mainly on the work of Guttag [1975], Guttag, Horowitz, and Musser [1978a,b]. An introduction to Guttag's data abstraction can also be found in the book by Horowitz and Sahni [1976]. Soem related discussions can be found in papers by Liskov and Zilles [1974,1975].

The main benefit is derived from the omission of irrelevant language

detail from an algebraic formulation of programs. Thereby the intrinsic properties of the program are emphasised. It becomes possible to recognise fundamental similarities between apparently different programs.

Finally, the functional hierarchy introduced by this approach allows hierarchial organisation of coorectness proofs.

We try to demonstrate that data abstraction is the driving mechanism behind functional abstraction.

For a long time there has been widespread failure to recognise the fact that data types consist of data objects together with primitive operations. Mathematically speaking, the structure of a set consists of the operations and relations defined for the members of the set.

It cannot be pointed out often enough that data are the sole reason for the existenc of computer programs. Hence the great importance of data abstraction in program development.

2. ABSTRACT PROGRAMMING

The expression of the need for a particular computer program can be regarded to be abstraction in a functional sense. One specifies what data are available and what is to be done to them, i.e. one specifies a function. Functions can be described by giving a procedure for calculating the point in the range wich belongs to a given point in the domain. It is clear that a particular function may be represented by many different procedures or algorithms. The problem of computer programming is to specify such procedures in a language acceptable to the

machine. Before one can do that, one must have a complete description of the effect of such a function. At tis point the argument seems to become circular, but we have not yet made any use of the specification of the data for the program. Abstract data types, designed specifically for the problem at hand provide a set of primitive operations wich will form the basis for the formulation of the desired program. Therefore, data abstraction is the driving mechanism behind the top down development of programs. (Here we deliberately keep the words "program" and "algorithm" interchangeable.)

The abstract data types used in the description of the original program may be regarded as composite, and their operations may be expressed in terms of the primitives of the component data types. In this way a hierarchy of levels of abstraction is introduced until a set of data types has been specified, which is easy to implement in the chosen programming language.

It is very important to use a simple set of rules for the construction of the symbolic expressions that define the effect of functions. A number of different forms of data abstraction have been proposed [R.T. Yeh (ed.), 1978], but some of them suffer from the defect that their syntax is too far removed from natural language and they fail to emphasise sufficiently strongly that data abstraction is the driving mechanism behind program development.

Since it has been accepted generally that the use of global variables and side effects are bad programming practice, it is clear that any rules for data abstraction should reflect that.

It often suffices to specify the primitives of an abstract data type in natural language. However, a formal specification is required for automatic program verification and, of course, for automatic program transformation. Guttag [1975] introduced a form of data abstraction that appears best suited for a programming methodolgy. Operations are specified as functions, whose effect is described by symbolic expressions. Side effects are forbidden, at least in the design stages. At the implementation stage it may be advantageous to make limitted use of them.

The rules can be stated very simply:

1. All variables are free.

2. If-then-else expressions are permitted.

3. Boolean expressions are permitted.

4. Recursion is permitted.

In the next section we use two examples to demonstrate functional programming with abstract data types.

3. EXAMPLES

We use the first example as an elementary introduction to the theme. The first problem is to find the largest one in a given collection of elements. We require a function which accepts as its argument the collection of elements and returns the largest element.

```
LARGEST(Set) -> Element
```

is the syntax of the required function. Its transformation into an algorithm is achieved by introducing the required abstract data types.

We must define a data type to represent the collection of elements. They are ususally given to us in the form of a list, which can be defined as follows:

Definition:

> A list is either empty or it consists of an element at the head which is followed by another list.

From this definition and the recognition of the fact that we must look at every element if we want ot find the largest one we can define the operations that we require for lists. We also require an operation which compares two elements of a list.

Our function LARGEST(List) is defined only for lists with at least one element in them, since it makes no sense to ask for the largest element in an empty list. We must be able to obtain the head of a list, find the length of a list, delete the head element to obtain the tail of the list. The operation GREATER is defined on elements rather than lists.

This is the semantic definition of LARGEST(List):

```
LARGEST(List) ::= if LEN(List)=1 then HEAD(List)
                  else GREATER(HEAD(List),LARGEST(TAIL(List)))
```

It is now a trivial matter to show that our function will terminate, because it uses a sequence of lists decreasing in length, where the length is bounded below by one.

The element returned cannot be anything but the largest, if our operation GREATER is implemented properly. This illustrates that program correctness is easy to prove in a well defined hirarchy of data types.

Recursion is used in the definition of the functions, but it is a matter for the programmer to decide, whether he wants to use it in his implementation. However, we contend that recursive programs are easy to design for recursively defined data types. The corresponding iterative ones are often much more difficult to write.

To complete our informal design, we must obtain a representation of the list in a particular programming language and program the primitives. We assume that we are given a set of integers stored contiguously in an array, from index "first" to index "last", and that the programming language is Pascal. The primitives can then be described as:

```
LEN(List) ::= last - first + 1
HEAD(List) ::= Array[first]
DELETE(List) ::= first <- first + 1
```

This translates into Pascal as the following package:

```
const MAXLEN = xxxx;
type  intlist = array[1..MAXLEN] of integer;
var List : intlist;
    first,
    last : integer;

function LEN(var list : intlist) : integer;
begin
    LEN := last - first + 1
end;

function HEAD(var list : intlist) : integer;
begin
    HEAD := list[first]
end;

function GREATER(elm1, elm2 : integer) : integer;
begin
    if elm1 > elm2 then GREATER := elm1
    else GREATER := elm2
end;
```

Te problem with this implementation of the list as an array is that it is not clear how to access the list, either by specifying the array or by specifying the index "first". Therefore, DELETE will be implemented as a procedure rather than a function. Hence we must break up our one-liner LARGEST. Note also that access to the list is only through its primitives. Consequently, the indices "first" and "last" are not visible to the outside world.

```
procedure DELETE(var list : intlist);
begin
    if first < last then first := first + 1
end;
```

The form of our Pascal program is now completely determined.

```
function LARGEST(var list : intlist) : integer;
var temp : integer;
begin
    if LEN(list) = 1 ten
        LARGEST := HEAD(list)
    else begin
        temp := HEAD(list);
        DELETE(list);
        LARGEST := GREATER(temp, LARGEST(list))
    end
end;
```

An alternative implementation, which reflects more closely the logic imposed by the functional approach results from the implementation of the list as an ascii file. This is because it avoids the main shortcoming of the previous implementation, namely the necessity to specify dummy arguments for the functions.

The effects of the two functions HEAD and DELETE are combined in the Pascal procedure "read" or "readln". Again, this forces us to break up our one-line program because we cannot use "read" without also introducing an assignment statement. The function LEN as we specified it cannot be implemented with files. Instead of testing for LEN(list)=1, we read and then test for end of file.

With that we obtain the following Pascal implementation:

```
var List : text;

function LARGEST(var list : text) : integer;
var temp : integer;

begin
    readln(temp);
    if eof(list) then
        LARGEST := temp
    else
        LARGEST := GREATER(temp, LARGEST(list))
end;
```

In the previous example we specified the abstract data type "List" in an informal way. Clearly, it was organised like a queue, except that we did not make any additions at the other end. In the following example we make use of the abstract data type "Queue", for which we use a specification gievn by Guttag et al. [1978b]. This also serves as an illustration of the functional style of specifying abstract data types.

```
type Queue[Item]

     declare

          NEW() -> Queue
          ADD(Queue,Item) -> Queue
          DEL(Queue) -> Queue
          FRONT(Queue) -> Item or UNDEFINED
          ISNEW(Queue) -> Boolean
          APPEND(Queue,Queue) -> Queue

     for all q,r in Queue, i in item let

          ISNEW(NEW) = true
          ISNEW(ADD(q,i)) = false
          DEL(NEW) = NEW
          DEL(ADD(q,i)) =
               if ISNEW(q) then NEW
               else ADD(DEL(q),i)
          FRONT(NEW) = UNDEFINED
          FRONT(ADD(q,i)) =
               if ISNEW(q) then i
               else FRONT(q)
          APPEND(q,NEW) = q
          APPEND(r,ADD(q,i)) = ADD(APPEND(r,q),i)

end Queue
```

We shall use the abstract data type "Queue" to develop aprogram that translates postfix notation into fully parenthesised infix notation. To avoid unnecessary overhead, operand names are single alphabetic characters, and only the four arithmetic operators are permitted.

Definition:

A postfix expression either consists of a single operand or it is composed of two postfix expressions followed by an operator.

An infix expression consists either of a single operand or it is composed of two infix expressions separated by an operator.

The syntactic part of the definition of our function is:

POST_IN(Postfix) -> Infix

To get on with the job, we must specify abstract data types "Postfix" and "Infix". We can then use the suitably designed primitives to carry out the parsing of the postfix expression, in order to re-assemble it as an infix expression. The definition given above points out how to go about this. After removing the operator from the end of the postfix expression, we are left with two expressions of the same kind appended to each other. Reading right to left, we can recognise the beginning of the second expression as the point at which the operand count exceeds by one the opertor count.

The abstract data type Postfix has the primitives:

LAST, to return the last token,

EXPR1 and EXPR2 to return the two subexpressions.

If we imagine the postfix expression to be stored in a stack, then we can implement the data type "Potfix" as a stack of characters, whose set of primitives has been augmented by the three mentioned above. They can be expressed in terms of the primitives of the stack example taken from Guttag et al. [1978b]:

```
type Stack[Item]

    declare

        NEW() -> Stack
        PUSH(Stack, Item) -> Stack
        TOP(Stack) -> Item
        POP(Stack) -> Stack
        ISNEW(Stack) -> Boolean

    for all s in Stack, i in Item let

        ISNEW(NEW) = true
        ISNEW(PUSH(s,i)) = false
        POP(NEW) = NEW
        POP(PUSH(s,i)) = s
        TOP(NEW) = UNDEFINED
        TOP(PUSH(s,i)) = i

end
```

The function LAST is identical to TOP. EXPR2 is the same as POP, because removal of the last operator lets us look at the end of the second sub- expression. EXPR1 requires the removal of expression2, or setting a pointer to the position one character below expression2.

Our translation program POST_IN must now add to the queue obtained from the translation of the first subexpression, the last operator, and then append to the resulting queue the one obtained from the translation of the second subexpression:

```
POST_IN(Postfix) -> Infix

POST_IN(Postfix) =
    if LEN(Postfix)=1 then LAST(Postfix)
    else APPEND(ADD(POST_IN(EXPR1(Postfix)),LAST(Postfix)),
        POST_IN(EXPR2(Postfix)))
```

To get fully parenthesised infix expressions, we must replace the calls to POST_IN on EXPR1 and EXPR2 by:

APPEND(ADD(NEW),'(',POST_IN(EXPR1))

and

ADD(POST_IN(EXPR2),')').

In the programming language "C" we implemented "Postfix" as a character array and "Infix" as a linked list. The primitives become functions which are very simple to implement correctly. The function POST_IN written in "C" looks very similar to its formal definition given above.

```
struct node*POST_IN(postfix)
/* converts a given potfix expression to parenthesised infix form */
char *postfix;
{
        struct node     *tmp,
                        *NEW(),
                        *ADD(),
                        *DEL(),
                        *APPEND();

        char    *EX1(),
                *EX2(),
                LAST();

        if (OPND(postfix))
                tmp = ADD(NEW(),LAST(postfix));
        else
                tmp = APPEND(ADD(APPEND(ADD(NEW(),'('),
                POST_IN(EX1(postfix))),LAST(postfix)),
                ADD(POST_IN(EX2(postfix)),')'));
        return(tmp);
```

4. CONSEQUENCES OF FUNCTIONAL PROGRAMMING

The abstract (and implemented) program POST_IN looks very much like a traversal of a tree when written in terms of the following data type

Binarytree together with the type Queue.

```
type BINARYTREE [Item]

     declare

          EMPTYTREE() -> Binarytree
          MAKE(Binarytree,Item,Binarytree) -> Binarytree
          ISEMPTY(Binarytree) -> Boolean
          LEFT(Binarytree) -> Binarytree
          DATA(Binarytree) -> Item or UNDEFINED
          RIGHT(Binarytree) -> Binarytree
          ISIN(Binarytree) -> Boolean

     for all l,r in Binarytree, d,e in Item let

          ISEMPTY(EMPTYTREE) = true
          ISEMPTY(MAKE(l,d,r)) = false
          LEFT(EMPTYTREE) = EMPTYTREE
          LEFT(MAKE(l,d,r)) = l
          DATA(EMPTYTREE) = UNDEFINED
          DATA(MAKE(l,d,r)) = d
          RIGHT(EMPTYTREE) = EMPTYTREE
          RIGHT(MAKE(l,d,r)) = r
          ISIN(EMPTYTREE,e) = false
          ISIN(MAKE(l,d,r),e) =
               if d=e then true
               else ISIN(l,e) or ISIN(r,e)

end Binarytree.
```

The three usual modes of traversal can be stated very succinctly by using another data structure, namely the Queue, and taking advantage of the recursive definition of binary trees.

For example:

```
INORD(Binarytree) -> Queue

INORD(EMPTYTREE) = NEWQ
INORD(MAKE(l,d,r)) =
     APPENDQ(ADDQ(INORD(l),d),INORD(r)),
```

where l and r stand for "left subtree" and "right subtree", respectively, and d stands for "data" to be stored at the root.

Comparison with the program POST_IN reveals the same structure in both cases. This suggests that we are really doing the same thing in both cases. It is of course well known that one can build the parse tree for a postfix expression and then traverse it in-order to abtain the corresponding infix expression. The nice thing about our style of functional programming is that it helps us to recognise when two apparently different actions really are the same. This recognition of the same pattern in different actions is possible because of the level of abstraction achievable with this programming approach.

The consequences for proving program corectness are obvious. One only has to prove the correctness of the implementation of any data type in terms of its cocnstituents. Therefore the members of e team working on a large project do not have to wait until the lowest level is implemented. They can prove their level correct based on the correctness of the next lower level.

5. CONCLUSION

We have introduced a functional style of top down development of computer programs, which is based on the recognition of the importance of data abstraction in programming.

The transformation of a function into an algorithm is achieved by the introduction of the primitive operations of the data types used.

The formal specifcation of abstract data types as introduced by Guttag [1975] has far reaching implications for correctness proofs of programs and automatic program transformation systems.

REFERENCES

Backus, J., "Can programming be liberated from the von Neumann style?" CACM 21, 8 (1978) 613-641.

Dahl, A.-J., Dijkstra, E.W., and Hoare, C.A.R., "Structured Programming" Academic Press, London (1972).

Guttag, J.V. "Specification and applicaton to programming of abstract data types", PhD thesis, University of Toronto (1975).

Guttag, J.V., Horowitz, E., and Musser, D.R. "Abstract data types and software validation" Comm ACM 21 (1978a)1048-1064.

Guttag, J.V., Horowitz, E., and Musser, D.R. "The design of data type specifications" in "Current trends in programming methodology", vol. 4: Data structuring, R.T. Yeh (ed.), Prentice-Hall, Englewood Cliffs, N.J. (1978b).

Horowitz, E. and Sahni, S. "Fundamentals of data structures" Pittman Publishing Ltd., London (1976).

Liskov, B. and Zilles, S. "Programming with abstract data types" Proc. ACM SIGPLAN Conf. Very High Level Languages, SIGPLAN Notices (ACM) 9, 4 (April 1974), 50-59.

Liskov, B. and Zilles, S. "Specification techniques for data abstraction" IEEE Trans. Softw. Eng. SE-1 (1975)7-18.

Yeh, R.T. (ed.), "Current Trends in Programming Methodology", vol. 4, Data Structuring, Prentice-Hall, Englewood Cliffs, N.J. (1978).