MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 131

An Example of Programming With Abstract Data Types

by

Jack B. Dennis

(A paper published in SIGPLAN Notices, Special Issue on Programming Language Design, July 1975)

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

September 1975

An Example of Programming With Abstract Data Types

Jack B. Dennis
Massachusetts Institute of Technology
Cambridge, Massachusetts

Concepts of good program structure can and should be reflected in the design of programming languages. Thus we have two criteria for use by language designers: First, work in the area of formal proof of program correctness has shown that program proofs are greatly simplified if certain features are absent from the language in which the programs are expressed -- for example, no goto's, or the absence of side effects in the execution of procedures. Second: the closeness of the language to the problem domain affects the ease with which the programmer may express a problem solution -- in particular, the data types of the language should provide a convenient match to the abstractions of the problem. A programming language should provide tools for building abstractions that are natural for the problem at hand.

These two design criteria are being applied to the concept of data type in the design of the programming language CLU being developed by Professor Liskov and colleagues at MIT[1].

I would like to show how the concept of abstract data type -- as realized in CLU -- can simplify the construction of provably correct programs. To this end we will consider the development of a program by "stepwise refinement" in PASCAL and in CLU. The problem to be solved is the sequence problem studied by Niklaus Wirth[2]:

> On the set containing the integers 1, 2 and 3, construct a sequence of length N containing no adjacent equal subsequences.

The strategy of solution is to generate a series of candidate sequences such that:

- Every sequence that could be a Good sequence is generated.
- It is easy to check that a candidate is a Good sequence.

We start with an initial candidate -- the empty sequence -- and form new candidates (Figure 1) by means of two operations: Extend and Change.

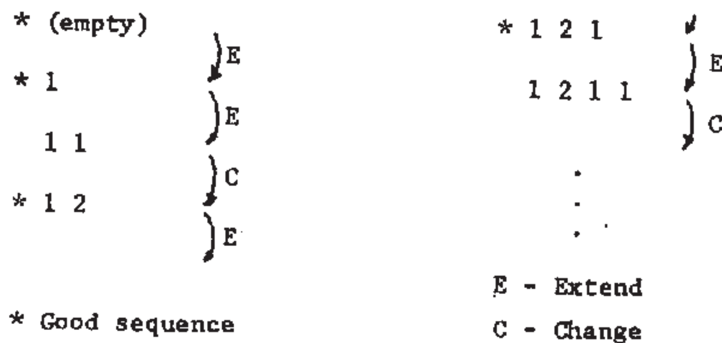

Figure 1. Construction of candidates.

- "Extend" appends a '1' to a candidate
- "Change" increments the last element

  (If the last element is '3', it is deleted and "Change" is applied to the truncated sequence.)

Since a Good candidate cannot be formed by extending a not Good candidate, "Extend" is only applied to Good candidates -- those marked by an asterisk. Testing that a new candidate is Good only requires testing for adjacent equal subsequences that include the last element.

The first version of a solution expressed as a Pascal function is:

```
function generate(N: integer);

var L: 0...N; good: boolean;
var X: "Candidate";

begin L := 0; good := true;
     repeat if good then "Extend X"
                    else "Change X";
            good := "X Checks";
     until good and (L = N);
     generate := X
end
```

The integer "L" keeps track of the length of the candidate "X"; the boolean "good" indicates whether "X" denotes a Good sequence. In this version, the representation for a "Candidate" is not specified and the operations Extend, Change, and Check remain to be elaborated.

The second version reflects the decision to represent the current candidate by an array of integers.

```
function generate(N: integer);

var L: 0..N;  good: boolean;
var X: array [1..N] of integer;

          [procedure declarations]

begin L := 0; good := true;
     repeat if good then extend else change;
          check
     until good and (L = N);
     generate := X;
end
```

Correspondingly, the abstract operations are implemented as Pascal procedures.

This methodology of refinement has some drawbacks:

- In the process of refinement important information about structure is lost.

- The relevance of the variables "L" and "good" to the operations on candidates is not evident from the program text.

In CLU, stepwise refinement starts with a top-level program written in terms of operations on an abstract data type "candidate".

```
generate = procedure(N: integer) returns(candidate);
    X: candidate := candidate$initial();
    repeat if candidate$check(X)
              then candidate$extend(X)
              else candidate$change(X)
    until (candidate$length(X) = N)  &  candidate$check(X);
    return (X);
    end generate;
```

The operations are:

Initial	create the initial candidate
Extend, Change	generate all acceptable candidates
Check	check if a Good candidate
Length	yields the number of elements of a candidate

The next step is to implement the "candidate" data type by writing a separate program module called a <u>cluster</u>. The <u>procedure</u> module "generate" is not changed. The header of the cluster (Figure 2) indicates that "candidate" is the name of the abstract data type implemented by the cluster, and that operations initial, extend, change, check and length are defined on objects of the abstract type. The next line states that the data type "seq" (sequence) has been chosen as the representation for candidates. The operations of the candidate cluster are written in terms of the operations on objects of type sequence:

- Initial returns the empty sequence.

- Extend appends a "1".

The special symbol <u>cvt</u> (convert) means the object passed is of abstract type outside, and of the representing type inside the operation.

```
candidate = cluster is initial, extend, change, check, length;

rep = seq;

initial = oper() returns(cvt);

     return(seq$empty());
     end initial;

extend = oper(x: cvt);

     seq$append(x, 1);
     return;
     end extend;

change = oper(x: cvt);

    repeat begin
                 n := seq$last(x);
                 seq$shrink(x)
           end
           until  ~(n=3);
    seq$append(x,n+1);
    return;
    end change;

check = oper(x: cvt) returns(boolean);

     good: boolean := true;
     L: integer := seq$length(x);
     i: integer := 1;
     while good & (2*i ≤ L) do
     begin good :=  ~ seq$equal(
                  seq$sub(x, L - 2*i + 1, L - i),
                  seq$sub(x, L - i + 1, L));
          i := i + 1
     end;
     return good;
     end check;

length = oper(x: cvt) returns(integer);

     return(seq$length(x));
     end length;

end candidate;
```

Figure 2. The candidate cluster.

The representing type, sequences, is another abstract data type. Since it is not a basic data type in CLU, it must be implemented by a second separately written cluster.

```
seq = cluster is empty, last, append, shrink,
                 sub, length, equal;
rep = array of integer;

     empty = oper() returns(cvt);

     append = oper(s: cvt; i: integer)

     last = oper(s: cvt) returns(integer)

                  . . .
```

As shown, this cluster might use an array of integers to represent a sequence, and would then contain the definition of the operations empty, append, etc. as manipulations of arrays.

Thus, we have solved the sequence problem using a hierarchy of data types: candidates; sequences; arrays. The correctness of each module of the program may be established independently:

- procedure generate -- This is proved on the basis of its text and the behavioral properties of candidates.
- The properties of candidates are proved from the text of the candidate cluster using the properties (or axioms) of sequences.
- Similarly, the properties of sequences are derived from the properties of arrays using the text of the seq cluster.

Some examples of assertions about candidates for use in proving generate are given below. We have abbreviated the operations on candidates using I for Initial, E for Extend, C for Change and T for Check. Good is the property that a candidate contains no adjacent equal subsequences. Reach is the property that a candidate is generated by a series of E's and C's.

1. Good(I); Reach(I)

2. Reach(x) implies $\begin{cases} \text{Good(E(x)) implies Reach(E(x))} \\ \text{Reach(C(x))} \end{cases}$

3. Reach(x) implies $\begin{cases} \text{Good(E(x)) iff T(E(x))} \\ \text{Good(C(x)) iff T(C(x))} \end{cases}$

4. Good(x) implies Reach(x)

Assertion (1) states that the initial candidate is both reachable and good. Assertion (2) characterizes those candidates that are reachable. Assertion (3) states the sufficiency of Check-ing to establish Goodness of candidates formed using Extend and Change, and (4) states that no Good candidate will be overlooked.

Thus a language that supports the concept of abstract data type improves the clarity of the completed program. It also provides a basis for hierarchical proof of correctness.

Features to support abstract data types are a strong candidate for inclusion in future programming languages.

References

[1] Liskov, Barbara, "A Note on CLU," Computation Structures Group Memo 112, MIT Project MAC, Cambridge, Mass., November, 1974.

[2] Wirth, Niklaus. Systematic Programming: an Introduction. Prentice-Hall, 1973.