1985

A program structuring principle

R. Geoff Dromey
University of Wollongong

A PROGRAM STRUCTURING PRINCIPLE

R. Geoff Dromey

Department of Computing Science,
University of Wollongong,
P.O. Box 1144, Wollongong, N.S.W. 2500,
Australia.

ABSTRACT

A principle for program structuring is introduced. The principle follows from adopting the idea that structure in a program is realized by changing or setting particular subsets of a program's variables under certain defined conditions. Individual structural components may then be defined by changing minimum subsets of variables that permit progress. This method of program structuring is widely applicable. Examples are presented demonstrating how the structuring principle can be applied to advantage in the implementation of several well-known algorithms.

May 9, 1985

A PROGRAM STRUCTURING PRINCIPLE

R. Geoff Dromey

Department of Computing Science,
University of Wollongong,
P.O. Box 1144, Wollongong, N.S.W. 2500,
Australia.

1. INTRODUCTION

The structure of a program determines its quality. The guidelines available for structuring grams are few. A commonly held view is that program structure is simply problem-dependent. While of course this is in part true, what we hope to demonstrate is that there is a simple higher-level problem-independent, structuring principle, that can often be very useful in making structuring choices when designing programs. We find that with recursively defined data structures most program designers readily accept the principle that there should be a match between program structure and data structure. Few for example would attempt to implement a linearly recursive algorithm to completely traverse a binary tree - a binary recursive mechanism obviously provides the match. Similarly when dealing with data processing problems those familiar with Jackson's work [1,2] readily accept that the structure of the program should mirror the structure of the file. The advantages of making these matches between program structure and data structure are well understood [1]. However when problems must be dealt with in other domains program designers are generally far less conscious of program structure. This comes about because there frequently does not appear to be any structure in the data or the problem with which the program structure can be matched. A program structuring principle that can deal with a wide range of problems in this category and which also encompasses the problems where data structure may determine program structure is therefore highly desirable.

Our goal is to define and apply a program structuring principle that will achieve a similar kind of effect to what is achieved by applying the normalization rules in Codd's theory of relational data-base organization [3]. A major goal in data base organization is to factor data into a set of relations (structural components) that minimize redundancy and maximize independence for the relations so that update operations are in general kept simple and the risk of inconsistencies caused by updates is removed. This leads to relations with strongly localized intra-relational bonds and weak inter-relational interactions. What we are seeking in our attempt at program structuring is also to remove redundancy but the redundancy we are seeking to remove is the unnecessary assignment or change of values for variables and the unnecessary testing of conditions. Removal of these forms of computational redundancy is usually achieved when there is a match between the structure of the data or problem and the structure of the program.

2. PROGRAM STRUCTURING

The program structuring principle that we will present is a natural extension of adopting the following view of how structure is realized in a program. We choose to assume that **structure in a program is realized by changing or setting particular subsets of the program's variables under certain defined conditions.** From this definition it follows that when choosing the program structure for a particular problem we can view the process as one of deciding which variables to change or set and deciding under what conditions these variables should be changed or set.

This view of the program structuring process needs further qualification to make it useful. There are a number of ways in which this view of the program structuring process can be qualified. We will choose to adopt a way that is simple, easy to apply, and widely applicable. Two other principles are needed before we can introduce our qualified structuring principle. They are the principle of separation of concerns [4] and the principle of least action [5]. One interpretation of separation of concerns as it is applied to program design is that well-defined tasks or sub-tasks should be treated separately.

An interpretation of the principle of least action relevant to program design is that progress towards a solution should be made at all times by the simplest and least costly means possible.

Our chosen program structuring principle which embodies both the principle of separation of concerns and the principle of least action requires that **individual structural components in a program should be defined by changing or setting the least number of variables under the simplest possible condition consistent with making progress towards the postcondition (which usually involves changing the variant function) for the precondition associated with the given structural component.**

A corollary to this is that **where possible each structural component should be a component that is capable of establishing the postcondition at least for some restricted precondition.** A method of top-down stepwise refinement based on this idea is discussed in detail elsewhere [6].

3. APPLICATIONS

To illustrate these ideas we will now discuss in detail several examples that have previously been examined in the literature.

3.1. Maximum Sum-section

With this problem, given an array of numbers (they can be positive or negative) it is required to find the contiguous subsection a[ℓ.u] of the array a[1..N] which has the maximum sum m for its set of elements. More formally we can write the specification (Q,R) with precondition Q and postcondition R as:

$$Q: N \geq 1$$

$$R: 1 \leq \ell \leq u \leq N \wedge m = \Sigma(j: \ell \leq j \leq u: a_j) \wedge \mathbf{A}(p,q: 1 \leq p \leq q \leq N: \Sigma(k: p \leq k \leq q: a_k) \leq m$$

Bentley [7] in his discussion of this problem gives the following example:

31	-41	59	26	-53	58	97	-93	-23	84

ℓ ↑ (under 59) u ↑ (under 97)

for which the sum m of the maximum subsection $a[3..7] = 187$. He then provides the very elegant linear algorithm given below for computing the maximum sum-section m.

Implementation (1):

```
m := 0; c := 0; i := 0;
do i≠N →
        i,c := i+1, max(0, c+a_{i+1});
        m := max(c,m)
od
```

where *max* is a function that returns the maximum of its two arguments The implementation is given here using Dijkstra's guarded commands [4].

Taking a more abstract view of things, in basic strategy, there is a strong similarity between this problem and the simpler problem of finding the maximum in an array of elements. The difference arises because we must deal with **super-elements** (which are sums of adjacent elements) rather than single elements when deciding upon the maximum. Apart from this the two problems are very similar in that we may have a variable *m* that defines the current maximum super-element and another variable *c* that defines the **next** super-element to be compared with the current maximum super-element.

As simple and as elegant as the above implementation is, it can end up doing a lot of unnecessary comparisons and assignments that may be avoided by a better structured program. In the implementation above, with each array element examined, checks are made whether both the maximum sum-section and the current sum-section need to be updated. This will usually be more than is required if we adopt our more abstract perception of the problem.

Our interest is to put together an implementation built from structural components that change the least number of variables to make progress. In trying to identify such structural components to make progress the smallest subset of free variables that needs to be changed will at least involve changing an array index variable *i* as in Bentley's implementation. In fact, if all the array elements happened to be negative, we could essentially get away with changing **only** the variable *i* because, in this instance, the maximum sum-section would be the empty subsection which has a sum of zero. Similarly, if all the array elements were positive, the task could be completed by changing just *c* and *i* in a loop and then setting *m* the maximum sum-section at the end. Studying the problem more carefully in this way we find that there are in fact three phases in which subsets of variables may be changed under different conditions. Figure 2 illustrates the basic cycle and identifies these subsets of variables and the conditions under which they may be changed.

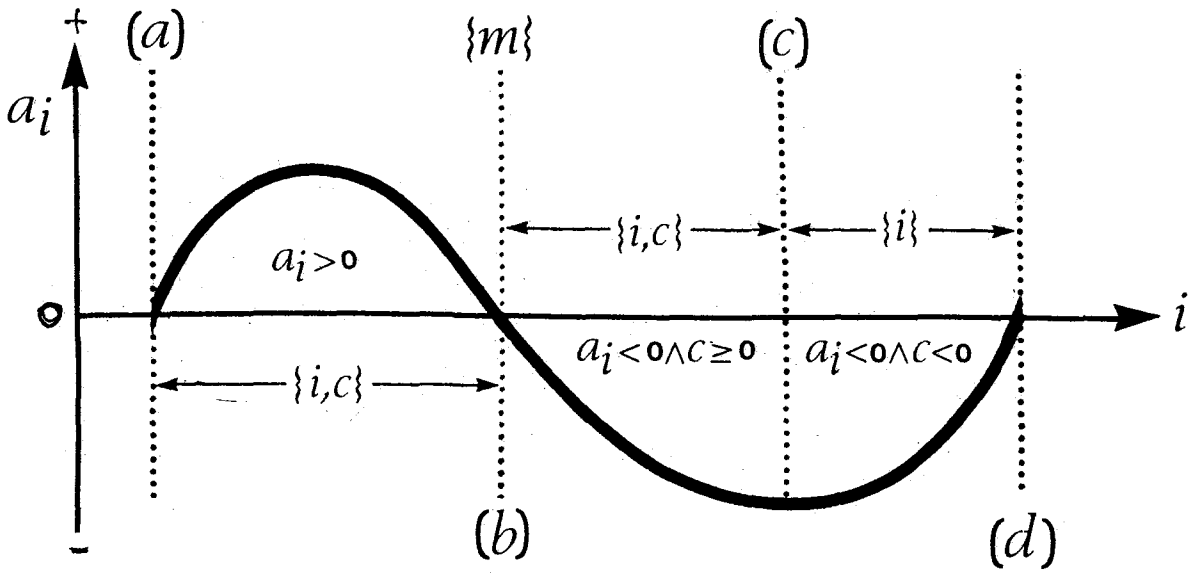

Figure 2 needs some explanation. There are three regions (any of which may be empty) corresponding to an iteration of the main loop of the program. Accordingly, there are three loops within the main loop, one to handle each region. The vertical coordinate on the figure is only intended to distinguish between positive and negative values of a_i. It does not imply that the values in the array exhibit any sinusoidal behaviour whatsoever.

An implementation that follows from this interpretation of the problem and which includes the two structural components mentioned earlier, and one other is:

Implementation (2):

```
i := 0; m := 0; c := 0;
repeat
    do i ≠ N and c ≥ 0 cand a_{i+1} < 0 → i,c := i+1, c+a_{i+1} od;
    if c<0 ↠ do i≠N cand a_{i+1} < 0 ↠ i := i+1 od; c := 0 fi;
    do i ≠ N cand a_{i+1}>0 → i, c := i+1, c+ a_{i+1} od;
    if c ≥ m ↠ m := c fi †
until i=N
```

Any runs of either positive or negative values in the array will be handled considerably more efficiently by this implementation than by the earlier implementation. Of course in the special case where the array consists of alternating positive and negative values there will be no gain from the structural components we have used in this alternative implementation. However such a data configuration would be unlikely in applications of this algorithm.

In comparing the two implementations the following observations may be made:

(i) In the second implementation the current maximum super-element is only *tested* for, and if necessary updated, after a complete cycle consisting of negative and positive runs. In contrast m is tested for update and updated after each element is examined in implementation (1). For random data we may expect m to be updated something less than logN times (see Knuth [8]) in implementation (2).

(ii) When c is negative and a run of negative elements is encountered with the first implementation c is unnecessarily and repeatedly set to zero. This is avoided in the second implementation. In fact no variables are unnecessarily set or updated in the second implementation.

(iii) The structure of the second implementation, we would suggest matches the abstract view of the problem in that it is consistent with comparing a *current* maximum super-element with the *next* complete super-element that could be the new maximum super-element seen so far. This structural match is absent from the first implementation.

(iv) In comparing the two implementations we say that the second implementation is **strongly structured** because each of its structural components changes the minimum sufficient subset of variables under the simplest sufficient conditions. The first implementation which does not conform to these structuring principles is said to be only a **weakly structured** implementation.

An examination of the basic cycle suggests there are two conditions (i.e. $a_i > 0 \wedge c \geq 0$ and $a_i < 0 \wedge c \geq 0$) under which the **same** subset of variables i, and c are changed. As an alternative to implementation (2) we could choose not to distinguish between positive and negative array element values. This leads to the following implementation for the body of the algorithm.

Implementation (3):

```
i:=0; m:=0;
repeat
    do i≠N cand a_{i+1} < 0 → i := i+1 od; c:=0;
    do i≠N and c ≥ 0 →
        i,c := i+1, c+a_{i+1};
        if c>m ↠ m := c fi
    od
until i=N
```

We have used "↠" to indicate that **if B ↠ S fi** is shorthand for if B → S [] ¬ B → skip fi to avoid the risk of abortion when ¬ B is encountered.

This implementation is resolved or normalized in terms of its variable subsets but not in terms of the conditions associated with change of the variable subsets (i.e. it does not distinguish between $a_i > 0$ and $a_i < 0$). The consequence of this is that with each iteration of the main inner loop it is forced to check if m needs to be updated. Only complete **variable subset** and **conditional** resolution removes this redundant checking and so for most data configurations implementation (2) will involve less testing.

3.2. Prime Factorization

The problem of finding the prime factors of a natural number is well-known. Alagic and Arbib [9] give an implementation with the following structure. †

Implementation (1):

```
i := 0; k := 0; n := N;
q := n div p[0];
r := n mod p[0];
do r = 0 ∨ q > p[k] →
      if r=0 → i,f[i+1] := i+1,p[k]; n := q
      [] r ≠0 → k := k+1
      fi;
      q := n div p[k];
      r := n mod p[k]
od;
if n ≠ 1 →» i,f[i+1] := i+1, n fi
```

In this implementation the array p is assumed to contain an adequate list of primes necessary to complete the factorization of N, and the array f is used to store all the prime factors. In providing an alternative implementation we will make the same assumptions and attempt to produce a similar factor array f. In seeking to provide an alternative implementation for this problem we are interested in identifying structural components that involve the change of the least number of variables and still allow progress. It would for example be possible to completely factor a number without changing k the index that allows the prime number variable to change. This would occur when N consisted of a single integer raised to a given power (e.g. $N = 2^{23}$). The major structural component may therefore be one that handles a single prime raised to a power. When this mechanism is placed in a loop we have the following prime factorization algorithm.

Implementation (2):

```
k := 0; n := N; i := 0;
repeat
      k := k+1;
      q := n div p[k];
      r := n mod p[k];
      do r=0 →
            i,f[i+1] := i+1, p[k];
            n := q;
            q := n div p[k];
            r := n mod p[k]
      od
until q ≤ p[k]
if n > 1 →» i,f[i+1] := i+1, n fi
```

† In their implementation they use Pascal. We have instead chosen to use what is essentially Dijkstra's guarded commands [4].

In comparing the two implementations the following observations may be made:

(i) The inner loop in the second implementation unlike the main loop in the first implementation avoids changing k and hence the prime number variable.

(ii) In the first implementation the guard "r=0" must be tested twice with each iteration of the loop. This is not the case with the second implementation

(iii) Furthermore with the second implementation while the power-multiplicity for a given prime is being determined it is not necessary to check "q ≤ p[k]" each time. Consequently there will usually be a gain in efficiency for the second implementation over the first implementation.

(iv) The second implementation could also be improved further by removing the variables r, and q as they result in unnecessary assignments. When this is done we get the following implementation:

Implementation (3):

```
k := 0; n := N; i := 0;
repeat
      k := k+1;
      do n mod p[k] = 0 → i,n,f[i+1] := i+1, n div p[k], p[k] od;
until (n div p[k]) ≤ p[k];
if n > 1 →» i,f[i+1] := i+1, n fi
```

3.3. Heapsort

Heapsort provides another interesting example of the effects of appropriate structuring. The *sift* operation as it is usually constructed in conventional heapsort implementations [10] is seemingly the most appropriate for dealing with heaps. In the sift operation for the sort, the heap is readjusted so that the element at the root of the heap is inserted into the heap to restore the heap property and consequently force the smallest element in the heap into the root position. This "smallest element" is the next available "sorted" element. The body of Wirth's implementation (in Pascal) [10] for the *sift* operation takes the following form where the heap with root at a[l] is stored in the array segment a[l..r].

Implementation (1):

```
i := l; j := 2*i; x := a[i];
while j ≤ r do begin
      if j < r then if a[j] > a[j+1] then j := j+1;
      if x ≤ a[j] then goto 13;
      a[i] := a[j]; i := j; j := 2 * i
      end;
13: a[i] := x
```

This implementation treats the tasks of finding the next "sorted" element and inserting x, the element at the root of the heap, as a composite task. If these two tasks are treated separately a significantly more efficient implementation given below is obtained. This strategy for implementing heapsort has also been suggested by Floyd [11].

Implementation (2):

```
i := l; j := 2*i; x := a[i];
while j ≤ r do begin
      if j < r then if a[j] > a[j+1] then j := j+1;
      a[i] := a[j]; i := j; j := 2*i
      end;
j := i div 2;
while a[j] > x do begin
      a[i] := a[j]; i := j; j := j div 2
      end;
a[i] := x
```

When this second sifting implementation is used in a heapsort it will on average use approximately the same number of array-element comparisons as quicksort [12] for random data. In contrast with this when a conventional sift is used heapsort on average uses nearly twice the number of comparisons used by quicksort [10]. It is a pity that this faster version of heapsort is not more widely known, as it makes it much more competitive with quicksort in average case behaviour. There is another improvement that we can make to the sift operation that avoids applying the guard "j < r" directly after the guard "j ≤ r". Instead we may use the following loop:

```
while j < r do begin
      if a[j] > a[j+1] then j := j+1;
      a[i] := a[j]; i := j; j := 2*i
      end
```

It is then necessary to include a finalization mechanism *after* this loop to handle the very infrequent case where j=r occurs.

4. CONCLUSIONS

The program structuring principle that we have demonstrated can be used in several ways. It can be used to examine and if necessary convert existing algorithms to a strongly structured form that may give a better match between the problem structure and program structure and hopefully better efficiency. In our demonstration of the principle this is the course we have taken. Differences in implementations for a problem are often very easily understood and explained in terms of this structuring principle. The structuring principle can also be used to assist in the design of new algorithms as it can give guidance on problem decomposition when used in conjunction with a stepwise refinement method [6]. Perhaps the most important contribution that this structuring principle can make is to provide a benchmark or reference for choosing the structure of programs. A worthy goal is to raise the status of program structuring in program design to a similar level to the status of data normalization in database design. The present proposals are intended as a contribution in this direction.

REFERENCES

1. M. Jackson, "Principles of Program Design", Academic Press (1976).
2. M. Jackson, "Systems Development", Prentice-Hall, London (1983).
3. E.F. Codd, "Further Normalization of the Data Base Relational Model, Vol. 6, Courant Computer Science Symposia Series, Prentice-Hall, Englewood Cliffs, N.J. (1972).
4. E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J. (1976).
5. A. Mayer, "Geschichte des Prinzips der kleinston Action", Leipzig (1877).
6. R.G. Dromey, "Systematic Program Development", (submitted for publication).
7. J.L. Bentley, "Algorithm Design Techniques", Comm. ACM, 27(9), 865-869, (1984).
8. D.E. Knuth, "Art of Computer Programming", Vol. 1, Fundamental Algorithms, (Section 1.2.10), Addison Wesley, Reading, Mass. (1969).
9. S. Alagic and M. Arbib, "The Design of Well-Structured and Correct Programs", Springer-Verlag, New York (1978).
10. N. Wirth, "Algorithms + Data Structures = Programs", Prentice-Hall, Englewood Cliffs, New Jersey, (1976).
11. D.E. Knuth, "Art of Computer Programming", Vol. 3, Sorting and Searching (Section 5.2.3, problem [18]), Addison-Wesley, Reading, Mass. (1973).
12. R.G. Dromey, (unpublished results).