PUC

Series: Monografias em Ciência da Computação

Nº 1/83

PROBLEM SOLVING VIA DIVIDE-AND-CONQUER AND ABSTRACT DATA TYPES

by

Paulo A. S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453

RIO DE JANEIRO – BRASIL

Series Monografias em Ciência da Computação

Nº 1/83

Series Editor: A. L. Furtado January 1983

PROBLEM SOLVING VIA DIVIDE-AND-CONQUER AND ABSTRACT DATA TYPES

by

Paulo A. S. Veloso

Departamento de Informática

ABSTRACT

The divide-and-conquer approach to problem-solving consists of decomposing a problem into smaller problems. Here this powerful approach is formalized by means of an incompletely specified abstract data type and a general program, proven to be correct, on it. Thus, solving a problem via divide-and-conquer amounts to interpreting the abstract data type on the problem domain. Examples are given to illustrate the applicability of this viewpoint to problem-solving.

Keywords

Problem-solving, problem decomposition, abstract data types, program construction, incomplete specification, divide-and-conquer.

RESUMO

O método de divisão-e-conquista para a resolver um problema consiste em decompô-lo em subproblemas menores. Aqui se formaliza este método poderoso através de um tipo abstrato de dados incompletamente especificado e um programa geral, este verificado correto sobre aquele. Assim, resolver um problema por divisão-e-conquista consiste em interpretar esse tipo abstrato de dados no domínio do problema.

A aplicabilidade deste enfoque à resolução de problemas é ilustrada por meio de exemplos.

Palavras chaves

Resolução de problemas, decomposição de problemas, tipos abstratos de dados, construção de programas, especificação incompleta, divisão-e-conquista.

INTRODUCTION

Consider the following problems
1. Flying from San Francisco to London
2. The monkey-and-bananas problem
3. Assemblying an airplane
4. Proving a mathematical theorem
5. Computing the lenght of a list
6. Sorting a sequence of numbers
7. Merging two ordered files.

Apparently they form a quite diversified collection of problems, coming from various walks of life.
One can argue that these are instances of a same abstract problem and can be solved by the same general method. This can be done by formalizing the process of decomposing a problem into subproblems. The tools used are abstract programs manipulating abstract data types, mainly the so-called "incompletely specified data types".

Now, what do the above problems have in common? A possible solution for problem 1 is obtained by first flying from San Francisco to New York, and then from New York to London. The second problem, frequently quoted in artificial-intelligence literature (e.g. [Nilsson 71, p. 35]), can be regarded as a variation of the first one. For problem 3, one might assembly in parallel some parts wich will then be combined into bigger parts until one has the entire plane. In problem 4, auxiliary lemmas play the role of subparts.

This approach is quite common in artificial intelligence and useful in programming. In fact [Wirth 73, p. 124] calls this decomposition principle one of the most general programming strategies and [Aho, Hopcroft, Ullman 75, p. 60] state

"A common approach to solving a problem is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole. This approach, especially when used recursively, often yields efficient solutions to problems in which the subproblems are smaller versions of the original problem".

We shall formalize this "divide-and-conquer" approach using abstract data types and justify why it works so often.

THE PROGRAM

In order to regard problem-solving from an abstract viewpoint let us consider a data type consisting of two sorts: sort P (of problems) and sort S (of solutions): this data type has a binary predicate, call it solves?, between sorts S and P, the intuitive intention being that solves?(s,p) indicates whether or not s is a satisfactory solution to problem p. We want a function procedure, call it solution, verifying the output assertion solves?(solution(p),p) for all p in P.

To simplify the notation we shall assume that each problem will be splitted into two subproblems, the solutions of which will then be combined. Thus, we suppose the existence of 2 unary operations split1! and split2! on sort P and a binary operation combine! on sort S. This process of problem decomposition is not necessary when the problem is simple enough to have a direct solution. We shall model this by means of a unary predicate simple? on P and a unary function direct! from P to S. The ADJ-like diagram [Goguen et al 77] in Fig. 1 displays the syntactical specification of these operations.

Now, the divide-and-conquer approach suggests the following (recursive) definition.

$$solution(p) = \begin{cases} direct!(p) & \text{if } simple?(p) \\ combine!(solution[split1!(p)], solution[split2!(p)]) & \text{otherwise} \end{cases}$$

The above definition involves only the primitive operations and predicates of the data type. So, it can be described as an abstract function procedure in a PASCAL-like notation as follows

```
function solution(p:P): S ;
solution:=if simple?(p)then direct!(p)
          else combine!(solution(split1!(p)),solution(split2!(p)))
```

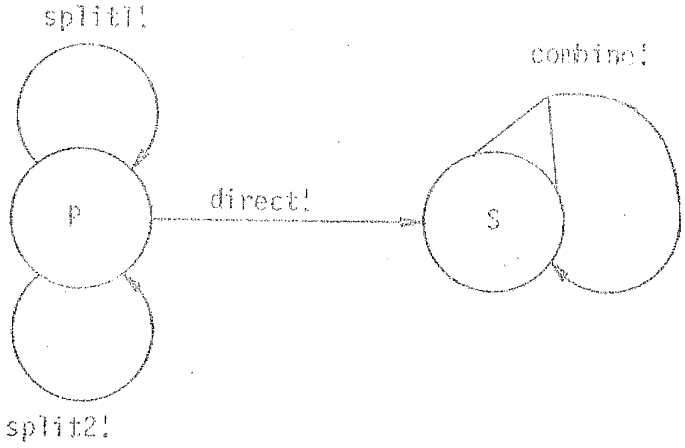

Fig. 1 : Operations of P-S

CORRECTNESS

According to our intuitive interpretation of the primitives, the procedure appears to be correct. In order to verify it we need a precise semantical specification of the data type, which was not given so far. But this is precisely the idea of the incompletely-specified-data-type methodology [Lucena, Pequeno 77]: first write an abstract program, and then specify the data type so as to make the program correct. The very nature of the program text suggests the following two axioms

(B) $simple?(p) \rightarrow solves?(direct!(p),p)$

(R) $\neg simple?(p) \rightarrow [solves?(s,split1!(p)) \wedge solves?(s',split2!(p)) \rightarrow solves?(combine!(s,s'),p)]$

which are well in accordance with the intuitive ideas of divide-and-conquer. Using them one concludes that if solution(p) is defined (i.e. the recursive calls terminate) then solves?(solution(p),p) holds, so the procedure is partially correct.

A basic idea underlying divide-and-conquer is that the subproblems are less difficult, in a sense, then the original problem, so that the process of problem decomposition will eventually lead to easy subproblems with direct solution.

Our data type captures this idea into a binary predicate smaller? on P satisfying

(I) $\neg smaller?(p,p)$

(T) $smaller?(p,q) \wedge smaller?(q,r) \rightarrow smaller?(p,r)$

(D) $\neg simple?(p) \rightarrow smaller?(split1!(p),p) \wedge smaller?(split2!(p),p)$

(W) smaller? is well-founded, i.e. it has no chain going down forever.

Now, each call to solution(p) may generate two further calls, thereby giving rise to a binary tree. The effect of (I), (T) and (D) is to ensure that a path in this tree corresponds to a smaller? chain, which (W) forces to be finite. Thus, solution(p) must eventually terminate.

The 6 axioms above can also be used to prove the total correctness of the recursive procedure solution by, say, Burstall's structural induction (cf. [Manna 74, p. 408]).

Now it is easy to extend the preceding argument to problem decomposition into n subproblems, by considering n unary operations split1,...,splitn on P and correspondingly an n-ary operation combine on S.

We present some examples that fit nicely into this framework: problems on lists, such as computing the lenght and concatenation, and sorting by the familiar methods mergesort and straight selection sort.

LISTS

As a simple example, consider the problem of computing the length of a list. In this case we shall interpret P as lists and S as naturals, and consider solves?(s,p) to mean lenght(p)=s.

We assume the type lists to be equipped with the usual operations head, tail and construct together with the unary predicate null. The naturals shall have the constant 0 and the operation of adding one.

Here it will be convenient to take n=1, considering the following definitions

simple?(p) = null(p); direct!(p) = 0;

split1!(p) = tℓ(p); combine!(s) = s+1.

Then the unary versions of axioms (B) and (R) become

null(p) → length(p) = 0

¬null(p) → [length(tℓ(p)) = s → length(p)=s+1]

which clearly hold, as for any list p

$$length(p) = \begin{cases} 0 & \text{if } null(p) \\ 1+length[t\ell(p)] & \text{otherwise} \end{cases}$$

Now, consider smaller? defined by

```
function smaller?(p: list, q:list): Boolean;
         smaller?:=if null(q) then false
                   else if p=tℓ(q)then true
                        else smaller?(p,tℓ(q))
```

So, smaller?(p,q) holds exactly when p can be obtained from q by successive applications of tℓ. Then axioms (1), (T) and (D) so interpreted become simple properties of lists.Finally (W) has to hold because we are dealing with finite lists.

Hence, the above definitions make lists-naturals into a realization of the data type P-S. Therefore the procedure solution with the corresponding definitions plugged in, namely.

```
function solution (p: list): natural;
solution : = if null(p) then 0
                        else 1 + solution (tℓ(p))
```

does compute the length of a list.

A more interesting example is concatenation of two lists. Here we interpret P as consisting of pairs <p,q> of lists, S as lists, and solves?(r,<p,q>) to mean r = append(p,q).

We now consider n=2 and the definitions

simple?(<p,q>) = null(p); direct!(<p,q>) = q;

split1!(<p,q>)=<nil,$\overline{hd}$(p)>; split2!(<p,q>)=<tℓ(p),q>;

combine!(p,q) = cons [hd(p),q].

Here nil is the empty list and $\overline{hd}$(p) = cons[hd(p),nil] is the list consisting solely of hd(p).

Our axioms (B) and (R) then become

null(p) → q = append(p,q)

¬null(p) → [s=append(nil,$\overline{hd}$(p)) ∧ t=append(tℓ(p),q) →
→ cons(hd(s),t) = append(p,q)]

which hold because cons[hd(cons[hd(p),nil],t]=cons(hd(p),t) and

$$\text{append}(p,q)=\begin{cases} q & \text{if null}(p) \\ \text{cons}(\text{hd}(p),\text{append}[t\ell(p),q]) & \text{otherwise} \end{cases}$$

With smaller?(<p,q>, <p',q'>) defined as length(p) < <length(p') axioms (I) and (I) clearly hold, axiom (D) becomes

¬null(p)→length(nil)<length(p) ∧ length(tℓ(p))< length(p)

which also holds and (W) follows from the finiteness of the lists.

Hence these definitions make the data type pair of lists - lists into a realization of P-S and the procedure solution so interpreted satisfies the output assertion
solution(<p,q>) = append(p,q).

SORTING

The problem of sorting can be obtained from the data type P-S by interpreting both P and S as sequences of numbers and solves?(s,p) as "s is the sequence p rearranged in increasing order of its components".

Now consider the definitions for n=2

simple?(p) ↔ length(p) ≤ 1; direct!(p) = p;

split1!(p) = the first half of p;

split2!(p) = the second half of p;

combine!(p,q) = the merge of p with q.

Then our axioms are satisfied and we have a realization of P-S, where solution becomes the familiar mergesort algorithm.

On the other hand if we define split1!(p) = the sequence consisting of the minimum element in p;

split2!(p) = the result of removing this minimum element from p;

combine!(p,q) = append(p,q)

and keep the remaining definitions then we get another realization of P-S, where solution interprets into straight selection.

Apparently other sorting methods could be so obtained by means of convenient interpretations.

GENERALIZATION

Most programmers and mathematicians have encountered a problem, a generalization of which is more amenable to decomposition into subproblems than the original one [Polya 71, p. 121]. A similar phenomenon occurs with inductive proofs and constructions.

In order to capture this idea we modify our problem-solution data type. We shall have two new sorts: sort O, of original problems and sort T of final solutions related by the binary predicate solvorg? with solvorg?(t,r) intended to mean that t is a satisfactory solution for the original problem r. Sorts P and S remain as before, except for two new unary functions generalize!: O → P and retrieve!: S → T. The diagram in Fig. 2 summarizes the syntactical specification of this new data type, with predicates represented by branches without arrows.

We now want realsol from O to P to satisfy solvorg?(realsol(r),r) and the above ideas suggest the following definition

realsol(r) = retrieve!(gensol[generalize!(r)])

where

$$\text{gensol}(p) = \begin{cases} \text{direct!}(p) & \text{if simple?}(p) \\ \text{combine!}(\text{gensol}[\text{split!}(p)]) & \text{otherwise} \end{cases}$$

Some natural axioms to ensure correctness are the unary versions of the previous ones (B, R, I, T, D, W), for then we get for all p in P solves?(gensol(p),p).
Now the following axiom is natural to guarantee the correctness of the generalization

(G)solves?(s,generalize!(r)) → solvorg?(retrieve!(s),r)

Indeed, with it we immediately obtain

solvorg?(retrieve!(gensol[generalize!(r)],r).

We shall now examine how iterative constructions and the treesort method can be regarded from this viewpoint.

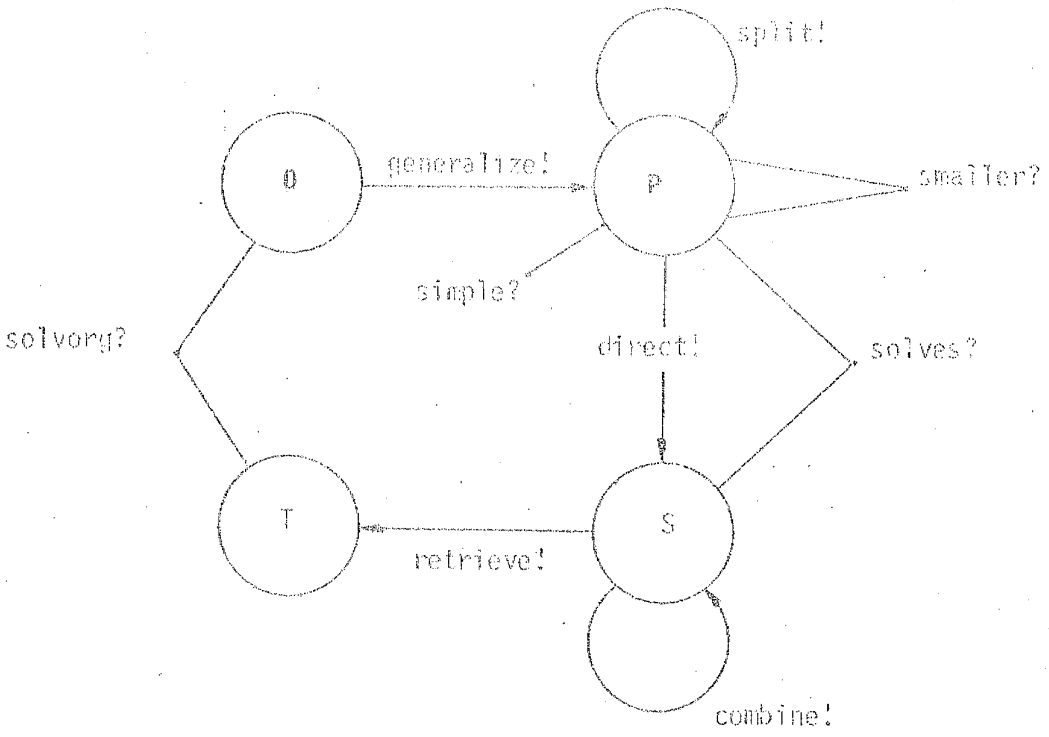

Fig. 2 : Syntax of O-P-S-T

ITERATION

It frequently happens that the solution to a problem is obtained by iterating simpler constructions. But the choice of the construction at each stage may depend on the stage, say via an auxiliary index. Such cases also fit into our generalization scheme.

For a simple example, consider the following PASCAL-like iterative program

```
var i,n : integer; var r : 0; var t : T;
begin r := init(r);
      for i:=1 to n do r: = transf(r,i);
      t : = final(r)
end
```

In our generalized data type interpret P and S as consisting of pairs <r,i> with r in 0 and i an integer and consider the definitions

generalize!(r)=<init(r),0>; retrieve!(<r,i>)=final(r);

simple?(<r,i>) ↔ i > n; direct!(<r,i>) = <r,i>;

split!(<r,i>) = <r,i+1>; combine!(<r,i>) =transf(r,i)

Then our function realsol is equivalent to the above iterative program.

TREESORT

Let us call a binary tree of numbers ordered if the contents of the nodes decrease as one moves to the left and increase as one moves to the right. Treesort is based on constructing an ordered tree an then traversing it in inorder.

Since we have a sorting problem we interpret, as before, O and T as sequences of numbers and solverg?(t,r) to mean "t is the result of rearranging r in increasing order". But now we interpret S as binary ordered trees and P as consisting of pairs <p,q> with p in O and q in S. Then consider the definitions

generalize!(r) = <r,Λ> (Λ is the null tree);
retrieve!(<p,q>) = the contents of the nodes of q in inorder;

simple?(<p,q>) ↔ null(p); direct!(<p,q>) = q;

split!(<p,q>) = <tℓ(p), put[hd(p),q]>(where put(a,q) inserts a into q so as to give an ordered tree);

combine!(q) = q;

smaller?(<p,q>, <p',q'>) ↔ length(p) < length(p')

solves?(s,<p,q>) ↔ s is the ordered tree made up from the elements of p and of q.

Then our axioms are satisfied and we have a realization of the data type, where gensol interprets into an algorithm to insert a sequence into an ordered tree and realsol into tree sort.

NONUNIFORM BRANCHING

Our data types model the process of repreatedly decomposing a problem into a fixed number n of subproblems until they have direct solutions. But for some problems, it may be more convenient to allow the number of splits to vary at each stage depending on the problem. We shall now show why this is already included in the generalization scheme.

Indeed, we may view this process as follows

for each subproblem q of p (starting with p itself)

do if easy?(q) then halt
else decompose q into subproblems

This constructs a problem-tree with p at the root and easy problems as leaves. The latter are solved, by immediate!, and these solutions climb up the solution-tree towards a solution for p.

Let O be the sort of original problems and T that of final solutions. Also, suppose that decompose!(r) gives the list of subproblems into which r is decomposed and climbup!($<t_1,\ldots,t_n>$) obtains a solution for r from those $t_1,\ldots,t_n$ of its component problems.

Now interpret P and S as consisting of trees of elements from O and T, respectively. Also, take generalize!(r) as the tree consisting only of r and retrieve!(s)=root(s). Let simple?(p) mean that all the leaves of p satisfy easy? and similarly extend the other predicates and operations to trees to obtain an O-P-S-T data type. This will satisfy our axioms if the problem-decomposition process we started with does work.

Alternatively, we may view P as sequences of elements from O and similarly for S.

Notice that this viewpoint indicates that our O-P-S-T data type with one split! and a unary combine! is already general enough.

CONCLUSION

The process of decomposing a problem into subproblems down to the level where they can be easily solved and their solutions combined has been formally described as follows. We regarded "divide-and-conquer" as an (incompletely specified) abstract data type. A general problem-decomposition procedure manipulating these data was presented and enough semantical specification of the data type was then given so as to guarantee the correctness of the procedure.

The solution to a problem on a particular data type thus would consist in defining on it special realizations of the primitive abstract operations so that the interpretations of the axioms now hold. To put it succintly: problem-solving(by means of decomposition) is thus equivalent to obtaining a realization of the divide-and-conquer data type on the problem domain.

Since the abstract data type is just enough specified, it can have many diverse realizations. This is indicated by the various examples, given to illustrate the wide applicability of such framework.

This applicability is enhanced by its feature of being compatible with stepwise refinement. For instance, the mergesort algorithm uses the operation merge. Now, merging can be viewed as a problem to be solved by a convenient interpretation of our divide-and-conquer data type. A similar example is the case of treesort with the operation of insertion into an ordered tree.

ACKNOWLEDGEMENTS

Many helpful conversations with T.H.C. Pequeno, C.J. Lucena, W. Hesse, A. Pereda, B. and M.A. Lopes on the subject of program development are gratefully acknowledged.

This report is a revised version of a paper presnted at the VII Conferencia Latinoamericana de Informática, Caracas, Venezuela, January 1980.

For more information on further development, see (Veloso 82) and references therein.

REFERENCES

A. V. Aho, J.E. Hopcroft, J.D. Ullman - The design and analysis of computer algorithms. Addison-Wesley, Reading, Mass, 1975.

J.A. Goguen, J.W. Thatcher, E.G. Wagner - An initial algebra approach to the specification, correctness and implementation of abstract data types in R. T. Yeh(ed.) Current trends in programming methodology, vol. 4, Prentice-Hall, Englewood Cliffs, N.J. 1978.

M.A. Lopes, P.A.S. Veloso - Operations on problems and their solution spaces. 5th European Meeting on Cybernetics and Systems Research, Vienna, Apr. 1982.

C. J. Lucena, J. M. C. Pequeno - A view of the program derivation process based on incompletely specified data types. Res. Rept. MCC 25/77, Depto. Informática, PUC/RJ, 1977;
revised version: Program Derivation using data types: a case study. IEEE Trans. on Software Engineering, vol. SE-5 (n96), p. 586-592, Nov. 1979

Z. Manna - The mathematical theory of computation. McGraw-Hill, New York, 1974

N. J. Nilsson - Problem-solving methods in artificial intelligence. McGraw-Hill, New York, 1971.

G. Polya - How to solve it: a new aspect of the mathematical method. Princeton Univ. Press. Princeton, NJ, 1971

P. A. S. Veloso, S.R.M. Veloso - Problem decomposition: applicability, soundess, completeness, 5th European Meeting on Cybernetics and Systems Research Vienna. Apr. 1980.

P.A.S. Veloso - Outlines of a mathematical theory of problems. Res. Rept. MCC 14/82, Depto. de Informática, PUC/RJ, Dec. 1982

N. Wirth - Systematic programming an introduction. Prentice-Hall Englewood Cliffs, NJ, 1973.