SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(4), 377–386 (APRIL 1994)

Using Procedural Parameters and Continuations in Combinatorial Searches

WEN-PING HWANG
Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C. (email: wphwang@cis.nctu.edu.tw)

AND

CHING-LIN WANG
Department of Computer and Information Engineering, Tamkang University, Tamsui, Taipei, Taiwan, R.O.C. (email: tkut101@twnmoe10)

SUMMARY

We use procedural parameters as a means to cut off unwanted branches in a search tree. The technique may be used to effect non-blind backtracking. A recursive algorithm for generating all strings of *n* pairs of balanced parentheses is chosen as an illustrative example, since it cannot be formulated by conventional recursive backtracking.

KEY WORDS Imperative programming Procedural parameters Continuations Pascal

INTRODUCTION

Procedural parameters in the imperative paradigm are hardly exploited in computing literature. Among the rare examples, Allison[1,2] used procedural parameters to represent continuations, and van Eijk[3] employed procedural parameters in generating all paths from the root of a binary tree to each of its leaves.

To fertilize the field of procedural parameters, this note presents a use of procedural parameters to cut away unwanted branches in a search tree. The technique is based on that of van Eijk, and may be illustrated by a couple of examples.

GENERATING BALANCED PARENTHESES

Consider the problem of generating all strings of n pairs of balanced parentheses described by the following context-free grammar:

$$S \rightarrow (S)S \mid \epsilon$$

This problem has a natural recursive solution based on the grammar:

For each i, $0 \leq i \leq n-1$, $n > 0$, generate all strings of the form

$$\text{'('} + s_i + \text{')'} + s_{n-i-1}$$

CCC 0038–0644/94/040377–10

Received 12 September 1991
Revised 5 November 1993

subject to $s_0 = \epsilon$, where s_k denotes a string of k pairs of balanced parentheses.

A straightforward formulation of this algorithm, as one would usually give in the functional paradigm, is to generate a list of all s_is and a list of all s_{n-i-1}s in the first place and then concatenate all possible pairs of elements of these two lists, with each element of the first list being surrounded by one pair of parentheses. Clearly, this formulation is time- and space-consuming, since there are

$$\binom{2k}{k}/(k+1)$$

s_ks. An alternative is to use backtracking. This is easily done in a logic language such as Prolog with built-in backtracking and non-determinism mechanisms. But doing so in the imperative and functional paradigms turns out to be a little harder.

Er[4] claimed that this algorithm cannot be formulated by conventional recursive backtracking. This is true only for 'conventional' recursive backtracking. By employing a technique known as continuation-passing style in the functional paradigm, we come up with a variation of conventional recursive backtracking, called continuation-passing backtracking, for formulating this algorithm.

In passing, the first part of this paper is primarily concerned with implementing the aforesaid algorithm efficiently in the imperative paradigm using procedural parameters and makes no attempt to solve the problem of generating all balanced parentheses of n pairs efficiently. For efficient algorithms for solving this problem, consult Reference 4 or adopt well-established tree-generating algorithms, since there is a one-to-one correspondence between the set of balanced parentheses of n pairs and the set of binary trees of n nodes. Incidentally, the formulation can be easily rephrased in any functional language.

CONTINUATION-PASSING BACKTRACKING

In conventional recursive backtracking,[5–8] it is assumed that each time the recursion reaches its end, a leaf of the search tree is encountered. But the aforementioned algorithm does not meet this condition—each time the recursion directed by the first non-terminal symbol S reaches its end, the computations demanded by the second non-terminal symbol S are still pending and must be activated to extend the path being searched to a leaf.

Thus, conventional recursive backtracking cannot be directly applied to formulate this algorithm. It could, however, be slightly modified to cope with the situation just described by adding an extra parameter to record the information required to continue the pending computations. The recursive nature of the algorithm suggests immediately that the information be held in a stack. Initially, the stack is empty, because there is no pending computation at the outset. As the recursion winds up, the stack grows up to reflect the pending computations accumulated so far. When the recursion reaches its end, the stack is checked for emptiness. If it is not empty, the information recorded in the top of the stack is retrieved and the associated computation is activated. This causes the recursion to wind up again, and the above process repeats. Eventually, the stack will become empty, and in that case a string of n pairs of balanced parentheses is generated. The fact that the stack will eventually

become empty can be easily seen by observing that the number of pairs of balanced parentheses remaining to be generated decreases continuously as the recursion winds up.

Figure 1 shows the procedure S, which should be initially called with

S(n, 1,con)

where con is an empty stack and n is the desired number of pairs. Observe that the size of the global character array a is $2n$; and the maximum size of the stack is n, because there is one stack element for each unmatched '(' in a[1..p−1] and at any time there are at most n such '('s. Observe also that the parameter con is call-by-value, so that the changes to it within the procedure body have no effect on the argument.

The pending computations as described above are known as continuations in the theory of denotational semantics[9] and the functional or applicative paradigm as exemplified by Scheme.[10] In these areas, continuations are represented as functions, but in this context they may more naturally be represented as procedures. The essential idea is simple: the information required to continue the pending computations already exists in the run-time stack and may be gathered by appropriate procedures. The detailed formulation is given in Figure 2.

The procedure S of Figure 2 should be initially called with

S(n,1,con)

where the initial continuation procedure con is as given. It can be verified that this procedure version is equivalent to the stack version of Figure 1 by showing that the stack parameter con effectively represents the procedural parameter con.[11] Less formally, the equivalence may be justified in terms of stack operations. First of all,

```
{Write every string of the form
   a[1..p-1] + s_n + ∑_{i=top}^{bottom} [')' + s_con[i]]
 where a is assumed to be a global character array}
procedure S(n:integer;p:integer;con:stack);
  var i,m:integer;
begin
  if n = 0 then
    if emptystack(con) then begin
      for i := 1 to p-1 do write(a[i]); writeln
    end else begin
      a[p] := ')'; pop(n,con); S(n,p+1,con)
    end
  else begin
    a[p] := '(';
    for i := 0 to n-1 do begin
      push(n-i-1,con); S(i,p+1,con); pop(m,con)
    end
  end
end;
```

Figure 1. Continuations as stacks

```
{Write every string of the form a[1..p-1] + sn + x, assuming
 that con(q) writes every string of the form a[1..q-1] + x,
 where a[1..q-1] = a[1..p-1] + sn}
procedure S(n:integer;p:integer;procedure con(q:integer));
  var i:integer;
  {Write every string of the form a[1..r-1] + ')' + sn-i-1 + x,
   assuming that a[1..q-1] = a[1..r-1] + ')' + sn-i-1.
   N.B.  ⊢ a[1..r-1] ⊣
         a[1..p-1] ( si ) sn-i-1 x
         ⊢——  a[1..q-1]  ——⊣      }
  procedure con0(r:integer);
  begin a[r] := ')'; S(n-i-1,r+1,con) end;
begin
  if n = 0 then con(p)
  else begin a[p] := '(';
             for i := 0 to n-1 do S(i,p+1,con0)
       end
end;

{Write a[1..p-1] + ε}
procedure con(p:integer);
  var i:integer;
begin for i := 1 to p-1 do write(a[i]); writeln end;
```

Figure 2. Continuations as procedures

observe that the procedure con0 carries the necessary information for the pending computations in its non-local references to n, i and con. Passing con0 around in the call S(i,p+1,con0) corresponds to a push, and returning from the call corresponds to a pop. The procedural parameter con is bound to either the initial continuation or an instance of con0. The call con(p) automatically distinguishes these two cases, and so the test for stack emptiness is not needed. If con is bound to an instance of con0, the references to n, i and con in con0 correspond to a pop.

The procedure version has several advantages over the stack version. For one thing, there is no need to manipulate the stack explicitly; for another thing, it is more efficient in both time and space, because of the avoidance of duplicating information. Table I compares these two versions running on an IBM RS/6000. Both sequential-array and linked-list representations for stacks are considered. To speed up the stack version, the operations on stacks are coded in line and the operation pop(m,con) is simplified to adjusting the stack top. Also, the entire loop

Table I. Performance summary

n	Procedure	Stack	
		Array	List
12	4·75	8·70	8·23
13	17·17	32·79	30·07
14	62·71	125·03	109·57

for writing the array is removed. In each case, the time in seconds is the average of 50 runs, and the standard deviation is less than 3 per cent.

The use of procedural parameters is controversial, due partly to expensive procedure calls and partly to obscure programs that may result. Used sparingly, however, it can improve the efficiency of programs without compromising their clarity.

TRAVERSING SEARCH PATHS

In terms of search trees, the procedural parameter con in effect records a chain of nodes of the path being searched. As the search progresses, this chain either extends when con is passed con0, shortens when con is invoked, or vanishes finally when a leaf is reached. In a like manner, one may arrange a procedural parameter for recording a chain that extends successively as the search progresses. By means of such a procedural parameter, a chain of nodes of the path being searched will be available by the time a leaf is reached. This chain can then be traversed, performing whatever operations are needed on each node of the chain. The formulation of Figure 3 illustrates this technique, which is based on Reference 3.

The procedure S of Figure 3 should be initially called with

S(*n*,path,con)

```
{Write every string of the form x + sn + y, assuming that
 path writes x and con(path) writes every string of the form z + y,
 where z = x + sn is written by con's parameter path}
procedure S(n:integer;procedure path;procedure con(procedure path));
  var i:integer;
  {Write x + '('}
  procedure path0; begin path; write('(') end;
  {Write every string of the form w + ')' + sn-i-1 + y, assuming
   that path writes w and z = w + ')' + sn-i-1.
   N.B.  ⊢ w ⊣
         x ( si ) sn-i-1 y
         ⊢——  z  ——⊣      }
  procedure con0(procedure path);
    {Write w + ')'}
    procedure path1; begin path; write(')') end;
  begin S(n-i-1,path1,con) end;
begin
  if n = 0 then con(path)
  else for i := 0 to n-1 do S(i,path0,con0)
end;

{Write ε}
procedure path; begin end;

{Write z + ε, assuming that path writes z}
procedure con(procedure path); begin path; writeln end;
```

Figure 3. Traversing search paths

where path and con are as given. Observe that the chain recorded in the procedural parameter path extends each time path is passed path0 or path1. The call con(path) ensures that the chain currently recorded in path will not be broken. On encountering a leaf, the chain recorded in the procedure path passed to the initial continuation is traversed from the root to the leaf while outputting a '(' or ')' accordingly on each node of the chain.

It is interesting to note that within this formulation there is no need to have a character array as in the earlier ones. This is not surprising because the character array behaves just like a stack. The theme analogous to the use of continuation procedures is: instead of holding a string of parentheses in a separate stack, we may hold it in the run-time stack by appropriate procedures. The procedural parameter path essentially does this. This may be seen by imagining the characters '(' and ')' as values of local variables of S and con0, respectively.

It is also possible, though unnecessary, to use a character array within this formulation. We leave it to interested readers.

CUTTING UNWANTED BRANCHES

We now come to the main result of this paper: by modifying the data contained in the nodes of the chain recorded in a procedural parameter, we may even alter the search tree. To give a case in point, suppose that instead of generating all strings of n pairs of balanced parentheses at once, it is desired to generate the next string, if any, only on request.

There are a number of ways of doing so. The most efficient way would be to have a goto out of the initial continuation procedure. A more structured way requires a test on a global flag in the procedure S to decide whether the computation should go further, but this leads to cluttered code. An alternative structured way is to raise an exception in the initial continuation procedure, which is then handled in the place where S was initially invoked. But this is not possible in Pascal for lack of exception handling mechanisms.

Using procedural parameters to alter the search tree allows one to attain advantages similar to those of a typical exception handling mechanism. This is shown in Figure 4. Now, the procedure S should be initially called with

```
S(n,path,cut,con)
```

where path, cut and con are as given. Observe that the procedure S has one more procedural parameter, the procedure cut, which is responsible for cutting away all the unexplored branches along the path that starts at the root and ends in the node where S is called. Just like the procedure path, the procedure cut must also be passed to the continuation procedure in due time so as to keep the recorded chain growing; thus, the continuation procedure now has cut as its parameter. In response to the request for abortion, the procedure cut passed to the initial continuation is then invoked to prune off all of the remaining branches along the path related to the string just generated; hence subsequent answers, if any, will not be produced.

A few notes are in order. First, the order of the two statements in the procedure cut0 is immaterial, whereas that of the two statements in the procedure path0 cannot be altered.

```
{cut cuts off all nodes with n > 0 related to x; path writes x}
procedure S(n:integer;procedure path;procedure cut;
            procedure con(procedure path;procedure cut));
  var i:integer;
  procedure path0; begin path; write('(') end;
  {cut0 cuts off all nodes with n > 0 related to x + '('}
  procedure cut0; begin i := n-1; cut end;
  {cut cuts off all nodes with n > 0 related to w; path writes w}
  procedure con0(procedure path;procedure cut);
    procedure path1; begin path; write(')') end;
  begin S(n-i-1,path1,cut,con) end;
begin
  if n = 0 then con(path,cut)
  else begin i := 0;
             while i <= n-1 do begin
               S(i,path0,cut0,con0); i := i+1
             end
       end
end;

procedure path; begin end;

procedure cut; begin end;

{cut cuts off all nodes with n > 0 related to z; path writes z}
procedure con(procedure path;procedure cut);
begin path; writeln; if no more answers are desired then cut end;
```

Figure 4. Cutting unwanted branches

Secondly, to cut off unwanted branches of a node, we may either assign to i a value greater than n−2 or assign to n a value less than i+2. Obviously, both should be coded with a WHILE, rather than FOR, loop. Although the former would illegally jeopardize the FOR loop's control variable, several popular Pascal compilers either fail to detect it (e.g. Sun Pascal 2.0, Vax Pascal 4.1) or do not take it seriously (e.g. IBM AIX Pascal 1.1). Perhaps this negligence is because threatening a FOR loop's control variable from outside the loop seldom occurs in practice, but then our example suggests that it be taken seriously.

Lastly, observe that the procedure cut links together those nodes or backtrack points that correspond to calls to S with n > 0. It is easily seen that, whichever *n*-pair string is generated, there are in total *n* backtrack points linked up in the corresponding procedure cut, because each call to S with n > 0, being a choice point as to how to split the remaining pairs into two parts, must generate one pair of parentheses in the first place.

It is possible to diminish the number of backtrack points by noting that there are no more alternatives for a node with i = n−1. Thus, it suffices to pass recursively the procedure cut, instead of cut0, on condition that i = n−1. This is shown in Figure 5(a). It might seem at first glance that the test of the condition in Figure 5(a) is redundant, but it is needed to ensure correct cutting. In spite of its strange look, this code is better than that of Figure 5(b), since the latter makes too many redundant

```
while i <= n-2 do begin
  S(i,path0,cut0,con0); i := i+1
end;
if i = n-1 then S(i,path0,cut,con0)
```

(a) Without redundant tests

```
while i <= n-1 do begin
  if i = n-1 then S(i,path0,cut,con0)
  else S(i,path0,cut0,con0);
  i := i+1
end
```

(b) With redundant tests

Figure 5. Saving backtrack points

tests of the condition. However it is coded, the number of backtrack points reduced is stated below.

Lemma 1

At the moment an n-pair string is generated, the number of backtrack points linked up in the corresponding procedure cut ranges from 0 to $n-1$. Furthermore, only the very first and last strings generated reach the upper and lower bounds, respectively.

Proof

Let d_n be the maximum number of times S is called with n > 0 and not followed by the call S(n−1,path0,cut,con0) in the course of generating an n-pair string. Then,

$$d_n = \max\left(d_{n-1}, 1 + \max_{0 \le i \le n-2} (d_i + d_{n-i-1})\right), \quad n \ge 2$$

subject to $d_0 = d_1 = 0$. We show by induction that $d_n = n - 1$, for all $n \ge 1$. Assume for the induction step that $d_k = k - 1$, for all $1 \le k < n$. Then,

$$\begin{aligned} d_n &= 1 + \max_{0 \le i \le n-2} (d_i + d_{n-i-1}) \\ &= \max\left(1 + d_0 + d_{n-1}, 1 + \max_{1 \le i \le n-2} (d_i + d_{n-i-1})\right) \\ &= \max(n-1, n-2) \\ &= n - 1 \end{aligned}$$

The maximum is reached only when $i = 0$. The other part may be shown either by a similar argument with max replaced by min or by observing that only the last string has no further alternatives. □

When seen in the large, the saving of up to n backtrack points is not really worth

while because of the extra effort made to examine the condition i = n − 1 before cutting. To see this, observe that the number of times the condition in Figure 5(a) is examined for each string generated before cutting is bounded above by $n - n_b$, where n_b is the number of backtrack points linked up in the corresponding procedure cut. This is easily seen by Lemma 1 and the fact that some tests of the condition are shared by strings whose search paths collide at the beginning. Furthermore, it is bounded below by 1, because the condition must at least be examined when generating the last pair of parentheses. It follows that the total number of times the condition is examined before cutting is bounded below by k and above by kn, where k is the number of strings generated before cutting.

NON-BLIND BACKTRACKING

The technique may also be adapted for cutting off the unexplored branches of some, instead of all, backtrack points. This allows the expression of non-blind backtracking, such as backing up several levels at once when meeting a dead-end condition.[12] For example, we modify the balanced parentheses problem to allow specifying the number of levels, or the level, to back up before generating the next string.

What is needed is an integer variable, which could be declared globally or as a parameter of the procedure cut, to hold the number of levels, or the level, to back up. In the former case, the number of levels to back up is decremented by one for each node pruned; and when it becomes zero the cutting process stops. Figure 6(a) shows the procedure cut0 written along these lines. In order for the latter case to work, the procedure S needs an extra integer parameter to record the level of a backtrack point. The cutting process may then go on until the level of a backtrack point equals the level to back up. Figure 6(b) shows the procedure cut0 written along these lines. In either case, the detailed formulation should be easily obtained,

```
{level0 = the number of levels to back up}
procedure cut0(level0:integer);
begin
  if level0 > 0 then begin
    i := n-1; cut(level0-1)
  end
end;
```

(a) Backing up a specified number of levels

```
{level0 = the level to back up
 level = the level of a backtrack point}
procedure cut0(level0:integer);
begin
  if level0 <> level then begin
    i := n-1; cut(level0)
  end
end;
```

(b) Backing up to a specified level

Figure 6. Non-blind backtracking

not to mention the fact that in order to record the levels of backtrack points correctly the continuation procedure also needs an extra integer parameter.

It is worth while comparing this technique with other ways of doing the same thing. First of all, gotos alone cannot deal with this situation, but gotos plus the procedure cut can. To do so, simply modify the procedure cut0 by removing i := n−1 and adding i := i+1; goto *label* as an alternative branch of the if statement, where *label* is the label of the while statement. Since jumping into branches of a conditional statement is disallowed, the if statement of the procedure S must also be rewritten in a non-structured way. On the other hand, the variable level0, when declared globally, may be tested accordingly to obtain the same effect. But this would clutter the code. Finally, exception handling mechanisms are competitive in clarity, except that the variable level0 must be global unless exception handlers may have parameters.

CONCLUSIONS

Although the code is given in Pascal, the technique is applicable in other imperative languages such as Modula-2, and impurely functional languages such as ML and Scheme that allow side-effects. However, it is not applicable in C, since functions may not be defined within other functions, and Ada, since recursive instantiations of generic subprograms are disallowed.

The technique is applicable to many combinatorial problems in its own right; it need not necessarily be used in combination with continuations.

ACKNOWLEDGEMENT

We would like to thank the referees for valuable comments on this paper.

REFERENCES

1. L. Allison, 'Some applications of continuations', *The Computer Journal,* **31**, (1), 9–11 (1988).
2. L. Allison, 'Continuations implement generators and streams', *The Computer Journal,* **33**, (5), 460–465 (1990).
3. P. van Eijk, 'A useful application of formal procedure parameters', *SIGPLAN Notices,* **21**, (9), 77–78 (1986).
4. M. C. Er, 'A note on generating well-formed parenthesis strings lexicographically', *The Computer Journal,* **25**, (3), 205–207 (1983).
5. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, MD, 1978.
6. G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
7. E. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
8. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
9. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.
10. IEEE Computer Society, *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990, New York, 1991.
11. M. Wand, 'Continuation-based program transformation strategies', *JACM,* **27**, (1), 164–180 (1980).
12. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.