数理解析研究所講究録
第381巻 1980年 100-115

A Derivation of Cook's Simulation Algorithm by Program Transformation

Osamu Watanabe

Department of Information Science
Tokyo Institute of Technology

Abstract We derive Cook's O(n) time simulation algorithm for 2DPDA from obvious but inefficient one by program transformation. The basic idea of transformation strategy is that of Bird's and uses two transformation techniques, that is, (1) stack elimination by recursion introduction (a genelarization of Bird's one) and (2) recursion elimination by tabulation.

1. Introduction

Recently several authors have proposed methods for transforming algorithms mechanically to improve their efficiency ([2], [5], ...). As one of these methods, Bird proposed a recursion introduction technique and demonstrated its usefulness by deriving Knuth-Morris-Pratt's algorithm (an algorithm for searching an occurence of a given word in a given text) from an obvious and inefficient algorithm ([3]). His method consists of the following steps.

(Step 1) Construct an obvious algorithm that uses a stack.

(Step 2) Transform it into an algorithm that has recursive calls but does not use a stack (stack elimination).

(Step 3) Using the tabulation method, eliminate recursive calls

from the algorithm.

It is well-known that Knuth-Morris-Pratt's algorithm is a special case of Cook's algorithm for simulating 2-way deterministic pushdown automata (2DPDA, for short) in O(n) time ([1]). In the present paper we show that Bird's method works also for Cook's algorithm. Thus we can derive Cook's algorithm by mechanical transformations starting with an obvious and inefficient algorithm.

2. Recursion Introduction

The algorithm with which we start has more complicated use of stack than the one with which Bird started. Hence Bird's technique for stack elimination does not work directly for our case. We use the stack elimination method proposed by Brown, Gries and Szymanski ([4]).

The basic idea of their method is to use a recursive function f which is almost the same as the whole program, has a call by value parameter and in which all variables other than the parameter are global. Instead of executing 'push u to stack S' or 'pop v from stack S', we just call f(u) or perform 'v := p and return'. The only problem is the next statement to be executed after the call or the return. But using an another parameter x (in the case of call), and using the value returned by f (in the case of return), we can specify the next statement to be executed.

The algorithm with which we start has the following form.

<S-1>

```
     begin A;
           S ⇐ u;
       L1: B;
           v ⇐ S;
       L2: C;
     L3: end
```

Here A, B and C do not contain any stack instruction (i.e., we consider a stack algorithm that has only one push and one pop), and stack S is initially empty.

The method of Brown, Gries and Szymanski transforms this form into the following form.

<S-2>

```
     begin function f(p);
           begin    goto L1;
                    A;
                    if f(u) = 0 then goto L3
                                else goto L2;
                L1: B;
                    v := p; return 1;
                L2: C;
                L3: return 0;
           end;
           A;
           if f(u) = 0 then goto L3
                       else goto L2;
       L1: B;
       L2: C;
     L3: end
```

The parameter x mentioned above is not used because push instruction appears only once in S-1, and consequently execution always begins

at L_1 whenever f is called. Intuitively, a call of f(u) in S-2 performs what S-1 would perform when S-1 starts at L_1 with u on the top of the stack, until either (1) S-1 arrives at L_2 with the value u popped up, or (2) S-1 arrives at L_3 without popping u up. The value of f(u) is 1 or 0 according as the case (1) or the case (2) occures.

3. 2DPDA

We give a formal definition of 2DPDA and introduce some notations. These are almost the same as those used in [1].

A 2DPDA is a 7-tuple $P = (S, I, T, \delta, s_0, Z_0, s_f)$, where

(1) S is the set of states of the finite control.

(2) I is the input alphabet (excluding # and $).

(3) T is the pushdown list alphabet (excluding Z_0).

(4) δ is the next move function defined on a subset of $(S - \{s_f\}) \times (T \cup \{Z_0\})$ and the value of $\delta(s, a, A)$, if defined, is of one of the forms (s', d, push B), (s', d, pop), and (s', d) where $s' \in S$, $d \in \{-1, 0, 1\}$, and $B \in T$. The symbols # and $ are used as the left and the right endmarkers of the input tape respectively. So the second component of $\delta(s, \#, A)$ is not -1 and the second component of $\delta(s, \$, A)$ is not 1 for any s and A. Similarly Z_0 is used as the bottom marker of the pushdown list, and hence $\delta(s, a, Z_0)$ is not of the form (s', d, pop).

(5) s_0 is the initial state of the finite control.

(6) Z_0 is the bottom marker of the pushdown list.

(7) s_f is the final state.

An instantaneous description (ID) of P on an input $w = a_1$

$a_2 \ldots a_n$ is a triple (s, i, α), where

(1) s is a state in S.

(2) i is an integer, $0 \leq i \leq n + 1$, indicating the position of the input head.

(3) α is a word representing the contents of the pushdown list with the leftmost symbol of α on top.

A move of P is a change from an ID to another defined as follows.

(1) A change from (s, i, Aα) to (s', i + d, BAα), if $\delta(s, a_i, A)$ = (s', d, push B).

(2) A change from (s, i, Aα) to (s', i + d, α), if $\delta(s, a_i, A)$ = (s', d, pop).

(3) A change from (s, i, Aα) to (s', i + d, Aα), if $\delta(s, a_i, A)$ = (s', d).

The initial ID of P is $(s_0, 1, Z_0)$. A terminal ID of P is an ID on which δ is not defined (the word 'terminal' is used in a different way in [1]). We may assume that a terminal ID is always of the form (s, i, Z_0) for some s and i, that is, P always terminates with the pushdown list containing only the bottom marker Z_0. The execution of P is a sequence of moves starting with the initial ID and ending with a terminal ID. The 2DPDA P is said to accept the input $w = a_1 a_2 \ldots a_n$ if this terminal ID is of the form (s_f, i, Z_0) for some i.

4. Derivation of Cook's algorithm

4.1 An obvious simulation algorithm

First, we construct an obvious and inefficient algorithm to simulate a 2DPDA. This algorithm uses a stack.

To simulate a 2DPDA P = (S, I, T, δ, s_0, Z_0, s_f) given an input w = $a_1a_2\cdots a_n$, the following variables are used.

tape : A one dimentional array of elements from I $\cup$ {#, \$}. The subscript ranges from 0 to n + 1. This array represents the content of the input tape of P. Its initial value is tape[0] = #, tape[1] = a_1, ... , tape[n] = a_n, tape[n + 1] = \$.

c : A pair of the form (s, i) with s $\in$ S and $0 \leqq i \leqq n + 1$. This pair is the first two components of ID of P.

z : An element of T $\cup$ {Z_0}. This is the top symbol of the pushdown list.

Pd : A stack of elements from T $\cup$ {Z_0}. This stack represents the content of the pushdown list of P except the top symbol. Initially it is empty.

The simulation algorithm is as follows.

<A1-1>

```
begin     c := (s0, 1);  z := Z0;
      L: if push then begin c := (s', c.i + d);
                            Pd ⇐ z;
                         L1: z := B;  goto L
                      end
    else if npp  then begin c := (s', c.i + d);
                            goto L
                      end
    else if pop  then begin c := (s', c.i + d);
                            z ⇐ Pd;
                         L2: goto L
                      end;
         if c.s = sf then 'YES' else 'NO';
L3: end
```

In this algorithm, push is a Boolean function whose value is true if and only if $\delta(c, z)$ (or more precisely, $\delta(s, i, z)$ where s, i are values such that c = (s, i)) is of the form (s', d, push B). Similarly, npp (abbreviation for 'neither push nor pop') and pop are Boolean functions corresponding to the cases $\delta(c, z) = (s', d)$ and $\delta(c, z) = (s', d, pop)$ respectively. Three letters s', d, B are functions whose values satisfy $\delta(c, z) = (s', d, push\ B)$, $\delta(c, z) = (s', d)$, or $\delta(c, z) = (s', d, pop)$ according as push is true, npp is true, or pop is true. Note that values of push, npp, pop, s', d, B depend on the values of c and z. The algorithm essentially uses the fact that P terminates if and only if none of push, npp, pop holds true.

The expressions c.s and c.i in the algorithm denote the first component (the 'state' component) and the second component (the 'position' component) of the pair c respectively.

4.2 Stack elimination

We can apply the transformation rule S-1 ⇒ S-2 described in Section 2 to A1-1. The algorithm corresponding to S-2 is the following.

```
<A1-2>
    begin function f(p);
          begin    goto L1;
                   c := (s0, 1);  z := Z0;
               L:  if push then begin c := (s', c.i + d);
                                      if f(z) = 0 then goto L3
                                                  else goto L2;
                                L1: z := B;  goto L
                                end
```

```
        else if npp  then begin c := (s', c.i + d);
                                 goto L
                          end
        else if pop  then begin c := (s', c.i + d);
                                 z := p;  return 1;
                           L2: goto L
                          end;
         (*) if c.s = sf then 'YES' else 'NO';
         L3: return 0
    end;
    c := (s0, 1);  z := Z0;
 L: if push then begin c := (s', c.i + d);
                       if f(z) = 0 then goto L3
                                   else goto L2;
                  L1: z := B;  goto L
                end
 else if npp  then begin c := (s', c.i + d);
                       goto L
                end
(**) else if pop  then begin c := (s', c.i + d);
                  L2: goto L
                end;
    if c.s = sf then 'YES' else 'NO';
L3: end
```

In this algorithm the statement (*) corresponds to the case where P terminates with its pushdown list containing at least one symbol besides the bottom marker Z_0. But this does not occur. Hence the statement (*) and the following return statement are never executed. This also implies that the value of f(p), if defined, is always 1. Hence we can replace the function f(p) by a procedure R(p). The statement (**) corresponds to the case where P tries to pop up the bottom marker. This does not occur. Hence we may delete the statement 'goto L' in the statement (**).

These consideration and other obvious simplification lead to the following algorithm.

<A1-3>

```
begin procedure R(p);
      begin     z := B;
                L: if push then begin c := (s', c.i + d);
                                      R(z);  goto L
                                end
          (*)  else if npp  then begin c := (s', c.i + d);
                                      goto L
                                end
               else if pop  then c := (s', c.i + d);
                                 z := p
      end;
      c := (s0, 1); z := Z0;
    L: if push then begin c := (s', c.i + d);
                           R(z);  goto L
                    end
  else if npp  then begin c := (s', c.i + d);
                           goto L
                    end
  else if pop  then c := (s', c.i + d);
      if c.s = sf then 'YES' else 'NO'
end
```

If we define a procedure T corresponding to (*), the definition of R becomes

```
procedure R(p);
begin   z := B;  T;  z := p   end
```

Replacing procedure calls R(z) in A1-3 by the definition above, we have an algorithm in which the procedure R does not appear.

<A1-4>

```
begin procedure T;
      begin  L: if push then begin c := (s', c.i + d);
                                   p := z;
                                   z := B;  T;  z := p;
                                   goto L
                             end
           else if npp  then begin c := (s', c.i + d);
                                   goto L
                             end
           else if pop  then c := (s', c.i + d)
      end;
      c := (s0, 1);  z := Z0;
  L: if push then begin c := (s', c.i + d);
                        p := z;
                        z := B;  T;  z := p;
(*)                     goto L
                  end
else if npp  then begin c := (s', c.i + d);
                        goto L
                  end
else if pop  then c := (s', c.i + d);
      if c.s = sf then 'YES' else 'NO'
end
```

(where p is a local variable)

We can go one stage further if we discover that (*) is the same as a call of T and use the fact that the last jump in the procedure body to the first instruction of the procedure body can be replaced by a recursive call of the procedure (this is an inverse use of the principle of recursion elimination showed in [6]). A simpler version of the algorithm is thus obtained.

<A1-5>

```
begin procedure T;
      begin     if pop  then c := (s', c.i + d)
           else if npp  then begin c := (s', c.i + d);
                                   T
                             end
           else if push then begin c := (s', c.i + d);
                                   p := z;
                                   z := B;  T;  z := p;
                                   T
                             end
      end;
      c := (s0, 1);  z := Z0;  T;
      if c.s = sf then 'YES' else 'NO'
end
```

Note that the effect of a call of T is simply to change the values of c, z and that the new values of c, z are completely determined by the values of c, z at the time of the call. Hence we may replace each call of T by an assignment statement $(c, z) := r(c, z)$ for an appropriate function r.

From now on it is convenient to use variables that have values of the form (s, i, z) with $s \in S$, $0 \leqq i \leqq n + 1$ and $z \in T \cup \{Z_0\}$. We call a value of this form a value of type CO (abbreviation for configuration), and call a variable that has values of type CO a variable of type CO. If x is a variable of type CO, then by x.s, x.i, x.z we denote the first, the second, and the third components of the value of x respectively.

The pair of c and z is essentially a variable of type CO. Hence we may regard the function r as a function r(x) whose parameter x is a variable of type CO and whose value r(x) is also of type CO. Replacing the procedure T of A1-5 by this function r(x), we obtain the following algorithm.

<A1-6>

```
begin function r(x);
      begin if pop  then return (s', x.i + d, x.z);
            if npp  then return r((s', x.i + d, x.z));
            if push then begin co := r((s', x.i + d, B));
                               return r((co.s, co.i, x.z))
                         end;
            return x
      end;
      co := (s0, 1, Z0);  co := r(co);
      if co.s = sf then 'YES' else 'NO'
end
```

In this algorithm co is a local variable of type CO. In the previous algorithms, functions push, npp, pop, s', d, B were functions of c, z. In A1-6, these functions are functions of x.

4.3 Tabulation

In this section we apply the tabulation method to the recursive function r. The basic idea of this method is to compute the value of r(x) just once for each x and store the value in the position F[x] of a table F.

If we apply this method directly to the recursive definition of r(x), we have to ask whether F[x] is defined or not for each recursive call of r(x). Instead of this top-down approach, we adopt the bottom-up approach, that is, we repeatedly search for x such that the value of F[x] is immediately determined from already defined values of F.

The recursive definition of r(x) gives the following computation rules for this bottom-up approach.

(Rule 1) If pop holds true, then F[x] = (s', x.i + d, x.z).

(Rule 2) If ¬ pop ∧ ¬ npp ∧ ¬ push holds true, then F[x] = x.

(Rule 3) Suppose that F[y] is defined and that x becomes y in one move of type 'npp' or 'push'.

(Rule 3.1) If npp holds true for x (hence y = (s', x.i + d, x.z)), then F[x] = F[y].

(Rule 3.2) If push holds true for x (hence y = (s', x.i + d, x.z)), then we may set F[x] = F[z] as soon as F[z] is defined, where z = (F[y].s, F[y].i, x.z).

Once we have the table F, we answer 'YES' if $F[(s_0, 1, Z_0)]$ is defined and $F[(s_0, 1, Z_0)].s = s_f$, and 'NO' otherwise.

The algorithm uses the following variables.

NEW : A set of values of type CO. This is the set of y such that F[y] has been defined but Rule 3 has not been applied to this F[y]. Initially NEW = ∅.

PRED : A one-dimensional array of sets of values of type CO. The subscript ranges over all values of type CO. The element PRED[z] is the set of x such that we may set F[x] = F[z] as soon as F[z] is defined (Rule 3.2). Initially F[z] = ∅ for all z.

F : A one-dimentional array of values of type CO or an indicator 'undefined'. The subscript ranges over all values of type CO. Initially F[x] = 'undefined' for all x.

TEMP : A set of values of type CO. This is a work variable in procedure UPDATE.

The algorithm is

<A1-7>

```
begin for all x of type CO do  (Rule 1, Rule 2)
        if pop then begin F[x] := (s', x.i + d, x.z);
                          NEW := NEW ∪ {x}
                    end
        else if ¬ npp  ∧ ¬ push then
                    begin F[x] := x;
                          NEW := NEW ∪ {x}
                    end;
      while NEW ≠ ∅ do
        begin select y from NEW;
              NEW := NEW - {y};
              for all x which becomes y in one move do
                if npp then begin F[x] := F[y];
                                   UPDATE(x)
                            end  (Rule 3.1)
                else if push then
                begin z := (F[y].s, F[y].i, x.z);
                      if F[z] = 'undef' then
                         PRED[z] := PRED[z] ∪ {x}
                      else
                         begin F[x] := F[z];
                               UPDATE(x)
                         end
                end
         end;
      if F[(s0, 1, Z0)] ≠ 'undef' ∧ F[(s0, 1, Z0)].s = sf
      then 'YES'
      else 'NO'
end

(where,

procedure UPDATE(x);
begin NEW := NEW ∪ {x};
      TEMP := PRED[x];
      while TEMP ≠ ∅ do
        begin select z from TEMP;
              TEMP := TEMP - {z};
```

```
                    TEMP := TEMP - {z};
                    F[z] := F[x];
                    NEW := NEW ∪ {z};
                    TEMP := TEMP ∪ PRED[z]
                end
        end;)
```

5. Conclusion

We derived Cook's simulation algorithm for 2DPDA by program manipulation without taking properties of 2DPDA into consideration. Analyzing the manipulation a little, we note that at A1-7 for the first time we could improve efficiency. What was done in the transformations from A1-1 to A1-6 ? At A1-1 we hardly had an idea of the final A1-7. But using recursion introduction and other manipulation, we could derive the algorithm A1-6 from which the idea of A1-7 occured to us naturally. It is very similar to the derivation of mathematical formulas, and the author thinks that this type of manipulation is important not only in improving efficiency but also in the theoretical study of programs.

Acknowledgements

I would like to express deep appreciation to Prof. Kojiro Kobayashi for his careful reading of the first drafts and to Mr. Takehiro Tokuda for suggesting the problem.

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The design and analysis of computer algorithms, Addison Wesley, Reading

Mass. (1974).

[2] J. J. Arsac, Syntactic source to source transforms and program manipulation, Comm. ACM 22, 1 (Jan. 1979), 43-54.

[3] R. S. Bird, Improving programs by the introduction of recursion, Comm. ACM 20, 11 (Nov. 1977), 856-863.

[4] S. Brown, D. Gries, T. Szymanski, Program scheme with pushdown stores, SIAM J. Comput. 1, 3 (Sep. 1972), 242-268.

[5] R. M. Burstall, J. Darlington, A transformation system for developing recursive programs, J. ACM 24, 1 (Jan. 1977), 44-67.

[6] D. E. Knuth, Structured programming with goto statements, Computing Surveys 6, 4 (Dec. 1974), 261-302.