Recursive Procedure Analysis

AKIHIRO NOZAKI*

1. *Introduction*

In a previous paper [1], the author proposed a new method (the predecessor method) of storage allocation for local dynamic variables in Algol programs. The discussions in that paper was, however, mainly limited to the treatment of the variables declared in a main program.

In this paper, we consider a treatment of the local variables declared in procedure bodies which involves, if we allow recursive callings of procedures, highly complex problems. As a result, the go-to-administration proposed by J. Jensen et al [2] is shown to be a complete but too tedious solution to be simplified to a considerable extent.

Now suppose that we can use correct addresses of storage registers allocated to current variables. Then we can utilize the predecessor method to decide the corresponding address of a new variable which is now going to be declared (cf. [1]).

The absolute address of an element of a dynamic array is by the way computed with the values of suffices and a number of coefficients. We assume that the set of these coefficients is stored in a control word, i.e. a static variable. Then the correct addresses of dynamic variables can be known from the correct addresses of static variables. Therefore we shall consider only the question how to know the corresponding address of a current local-static variable declared in a procedure body.

Remark When a procedure A is called recursively, some variables declared in A will again be declared. Then each of these variables has two (or possibly more) memory locations. So the computer has to select in the executing phase the proper address among the addresses of these memory locations according to cases.

Hereafter we shall call the local static variable declared in a procedure body a *semi-static variable.*

2. *Preliminaries*

For rapid access to semi-static variables, it is preferable to utilize index registers. The points of the technique will be illustrated as following:

1) Semi-static variables are allocated in a storage area reserved for local dynamic variables. In this connection, "semi-static" implies "dynamic".

2) To each semi-static variable we associate a corresponding address in the same

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 6, No. 1 (1965), pp. 12–20.

* College of General Education, University of Tokyo.

way as to an ordinary local static variable (cf. [1], Section 2). This address is, however, considered as a relative address to a base which is determined in the execution phase.

3) Every instruction referring a semi-static variable contains the relative address corresponding to the variable in its address part and is modified by an index register containing a relevant base.

4) At every entrance to a procedure A by a procedure statement, the current value L of the first address of the usable storage cells for local dynamic variables is assigned to a relevant index register. Note that the current value L can be known by the control word L_B being attached to the block B which contains the procedure statement calling A at the moment (cf. [1], Section 3).

For the correct use of addresses, the values of index registers must be kept properly. This is the principal point of our problem.

Let us consider an arbitrary fixed program P written in Algol. We call the main program of P *a procedure of degree* 0. *A procedure of degree n* is defined to be a sub-procedure declared in the procedure body of a degree $n-1$.

We denote the degree of a procedure A by $d(A)$.

We shall call a variable declared in a procedure of degree n *a variable of degree n*.

In the following we shall assume that many index registers are available and that for each $n>0$, a separate index register $M[n]$ is utilized for modifying semi-static variables of degree n.

Remark An instruction referring a semi-static variable of degree n is always modified by the register $M[n]$, even if it appears in a procedure of a degree not equal to n. When a semi-static variable is used as an actual parameter called by name, the software should evaluate the absolute address of the variable taking account of the relevant index register. This absolute address is used as an actual entity of the parameter.

3. *A Simple Case*

Let us start with a discussion about a simple case under the following restriction on the program P.

Restriction (*) Any procedure identifier, designational expression and label do not be used as actual parameters of procedure statements or functions.

This restriction (*) eliminates the possibility of the control transfer to a location labeled by an identifier from the outside of its scope. So we can utilize an algorithm of index register administration proposed by J. M. Watt [3].

The principle of his algorithm would be summarized as follows.

Consider a procedure call from a procedures P of degree m to a procedure Q of degree n.

(1) If m is not smaller than n, then the values of index registers

$$M[n], \ldots\ldots\ldots, M[m]$$

are reserved in a working storage before the execution of Q.

(2) On the normal exit from Q to P, these values are all restored to
$M[n]$.........., $M[m]$, if and only if $m \geqq n$.

No other operation is executed even on an abnormal exit.

In the following we shall examine rigorously the validity of the algorithm.

A program P written in Algol can be considered as a finite string, or rather a finite ordered set of basic symbols taking the same symbols (strings) in different positions as different elements (subsets, respectively). Any procedures in P are naturally regarded as subsets of P.

Let A be procedure of a degree not equal to zero. Then A is a subprocedure declared in procedure B of degree $n-1$. We denote the procedure B by $p(A)$. ($p(P)$ is not defined.)

Let us denote an entry from a procedure A to a procedure B with $A+B$, a normal link from B to A with $B-A$ and an abnormal exit by a go to statement from B to C with $B \sim C$. Then we can represent a sequence of control transfers by a diagram e.g.

$$P+A+A+B \sim A. \tag{3.1}$$

We call such a diagram representing a sequence of control transfers as (3. 1) *a control sequence.*

Under the restriction (*), every control sequence of the form

$$A_0+A_1+\ldots\ldots+A_n \Delta A \tag{3.2}$$

satisfies the following axioms.

Axiom 1. If $\Delta=+$, then $A \neq P$ and $A_n \subset p(A)$.
In general, $A_{i-1} \subset p(A_i)$ for $i>0$.

Note that $p(A)$ is just the scope of the procedure identifier of the procedure A.

Axiom 2. If $\Delta=-$, then $n>0$ and $A=A_{n-1}$.
Axiom 3. If $\Delta=\sim$, then $A_n \subset A$.

Theorem 1. Suppose that A_0 in the sequence (3. 2) equals to P and $\Delta=\sim$. Then there is an integer i such that $A_i=A$.

Theorem 2. Suppose that, in the sequence (3. 2), $\Delta=\sim$, $A_i=A$ and $A_j \neq A$ for all $j>i$. Then

$$d(A)<d(A_j)$$

for all $j>i$.

Remark The exit $\sim$ in this case is interpreted as an exit to A_i. After the exit $\sim$ in (3. 1), consequently, the value of index register $M[1]$ must be the the same with its value at the moment when the control was advanced to the procedure A for the second time:

$$P+A+A. \tag{3.3}$$

(3. 1) is equivalent to (3. 3) in the sense that the value of the related index register $M[1]$ is the same.

Let us denote by M[n, t] the value of $M[n]$ at the moment when the control was transferred to A_t.

Theorem 3. Consider a control sequence of the form

$$P+A_1+\ldots\ldots+A_{n-1}\Delta A_n. \tag{3.4}$$

If $\Delta=\sim$, $A_i=A_n$ and $A_j\neq A_n$ for $i<j<n$, then

$$\left.\begin{array}{l} M[l,\ i]=M[l,\ n], \\ \vdots \\ \vdots \\ M[k,\ i]=M[k,\ n] \end{array}\right\} \tag{3,5}$$

where $k=d(A_n)$.

If $\Delta=-$, then the equations (3.5) hold for $i=n-2$.

The proof is obvious by theorem 2 and the operations (1)-(2).

This theorem shows the validity of the algorithm.

4. *General Case*

4.1. *Formal parameters in go to statements*

Consider again a control sequence of the form

$$P+A+A+B\sim A. \tag{4.1}$$

If the exit $\sim$ is caused by a go to statement designated explicitly by a label, then (4.1) is equivalent to the sequence

$$P+A+A \tag{4.2}$$

(cf. Section 3, **Remark**). But if the go to statement is designated by a formal parameter X of B, and if the actual parameter for X is again a formal parameter Y of A to which a label q in A has been assigned as an actual parameter, then the control should be transferred back to the first A. So the sequence

$$P+A \tag{4.3}$$

instead of (4.2) should be equivalent to (4.1).

In this connection, some additional operations are required if we remove the restriction (*).

$$\begin{tikzcd}
P \arrow[r, "A(\cdots)"] & A \arrow[r, "A(q)"] & A \arrow[r, "B(Y)"] & B \arrow[r, "\text{go to } X"] & {} \\
 & \Cup & \Cup & \Cup & \\
 & (q \arrow[r] & Y \arrow[r] & X) &
\end{tikzcd}$$

Fig. 1.

Let

$$P+A_1+\ldots\ldots+A_n\sim A \qquad (4.4)$$

be a control sequence. We call the suffix i of each term A_i *the dynamic level* of the procedure A_i in (4. 4), after Watt. We assume that every procedure A has a semi-static variable $L(A)$ containing a dynamic level of A. On every entrance to the procedure A_i, the value of $L(A_i)$ is set to be $L(A_{i-1})+1$.

Suppose that the procedure statement in A_k which called the procedure A_{k+1} contains a label X (or a designational expression α) as an actual parameter for a formal parameter Y. Then the dynamic level k of A_k should have been transmitted to the procedure A_{k+1} together with the label X (or the necessary information to evalute α). If Y is again used as an actual parameter in a procedure statement calling A_{k+2}, then the pair of the label X and the dynamic lebel k should be transmitted to A_{k+2} as an actual entity of Y.

Whenever the control is advanced to a go to statement designated by a formal parameter (or a designational expression whose current value is a formal parameter), both the exact address of a label q and the dynamic level k of the actual parameter are evaluated. The control is first "turned back" to a procedure of dynamic level k and then is transferred to the location q.

In the first step of these transfers, the following turning-back-operation is required to keep the proper values of index registers.

(1) Let n be the dynamic level of the procedure containing the go to statement (see (4. 4)). Then the operation (2) in Section 3 is applied repetitively, $n-k$ times, to recover the values of index registers at the moment when the control was transferred to the procedure of dynamic level k.

No additional operation is required for the second step.

Roughly speaking, the sequence (4. 4) is first reduced to

$$P+A_1+\ldots\ldots+A_k\sim A \qquad (4.5)$$

where $\sim$ is interpreted as a control transfer specified by an explicit go to a statement "*go to q*". Note that Axiom 3 holds for the sequence (4. 5).

4. 2. *Formal parameters as procedure statements or functions*

The dynamic level k of a procedure A_k should also be utilized when a procedure identifier B (or an identifier of a function designator) is used as an actual parameter in a procedure statement in A_k. We shall assume that the dynamic level k is also transmitted to the new procedure together with the necessary information to call B. If the procedure B is delivered to the procedure A_n through formal parameters and is used as a procedure statement or a function in A_n, then the following operations are executed before the execution of B.

(2) The values of index registers

$$M[1],\ M[2],\ldots\ldots,\ M[j] \qquad (j=d(A_n))$$

are reserved in a working storage.

(3) The values of index registers are so changed as if the control were turned back to the procedure A_k. (of (1)).

(4) On the normal exit from B, the reserved values are restored to $M[1]$, $M[2]$, ..., $M[j]$.

Note that the dynamic level of the procedure B in this case is defined to be $n+1$. Variables in B are allocated to the area next to the area for the variables in A_n.

If an abnormal exit out of B occurs, then the same principle is applied to the index-register-administration as what is described in Section 4.1. In any case the procedures A_{k+1},......, A_n are all canceled together with B.

5. *Precise Algorithm*

Now we shall describe precisely some system procedures to show the feasibility of the operations (1)-(2) in Section 3 and (1)-(4) in Section 4.

In the following Memory $[i]$ denotes a memory cell having address i.

procedure Entry (L, m, n);
value L, m, n;
comment This system procedure is called before every control transfer from a procedure A of degree m to a procedure B of degree n. The compiler inserts automatically a statement calling this system procedure in front of every procedure statement written by a programmer. L is the first address of usable storage cells

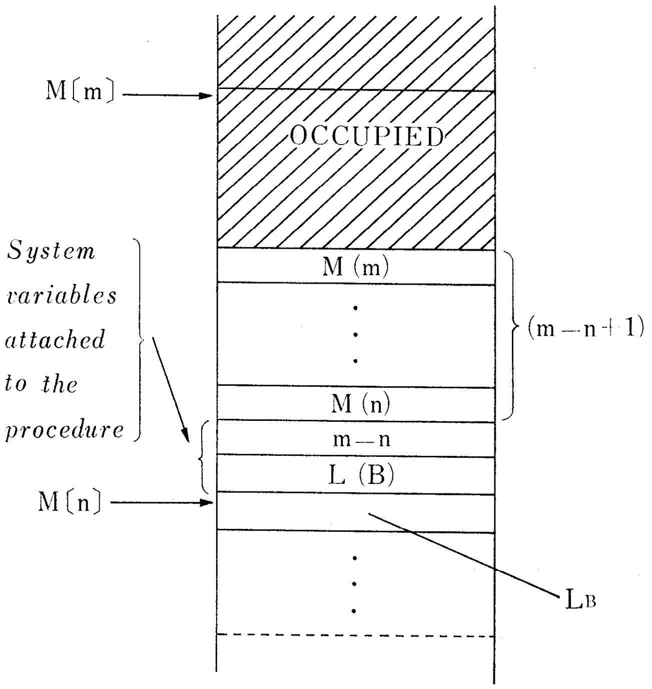

Fig. 2.

which can be known by a control word being attached to a block containing the procedure statment (cf. [1]).

We assume that the semi-static variable $L(A)$ is allocated to the register Memory $[M[n]-1]$ and that the control word L_A containing the first address of usable storage cells is allocated to Memory $[M[n]]$.;

```
begin integer i;
        if $m<n$ then go to procedure initialization;
        for $i:=0$ step 1 until $m-n$ do Memory $[L+i]:=M[m-i]$:
        $L:=L+(m-n+1)$;
procedure initialization: $i:=$if $m=0$ then 1 else Memory $[M[m]-1]+1$;
        $M[n]:=L+2$;
        Memory $[M[n]-2]:=m-n$;
        Memory $[M[n]-1]:=i$;
        Memory $[M[n]]:=M[n]+1$ end
procedure  Exit $(n)$;
value  $n$;
```

comment This system procedue is called on every normal exit from a procedure B of degree n to any procedure A. The compiler inserts automatically a statement calling this system procedure at the end of every procedure body, just before the instructions for the linkage.;

```
begin integer $i$;
        if Memory $[M[n]-2]<0$ then go to end;
        for $i:=$Memory $[M[n]-2]$ step $-1$ until 0 do $M[n+i]:=$Memory $[M[n]$
        $-i-2]$;
end:  end
```

Note that this procedure does not contain instructions to transfer the control from B to A.

The operations (1) and (2) in Section 3 are efficiently executed by these procedures. The operation (1) in Section 4 is easily executed by a formal-parameter-administration routine which contains the following statement.

```
        begin integer $i$;  for $i:=n$ step $-1$ until $k$ do Exit($i$) end
```

The operations (2) and (3) are properly executed by inserting the following statements automatically in front of a formal parameter X used as a procedure statement.

```
        Entry $(L, j, 1)$;
        begin integer $i$;  for $i:=n$ step $-1$ until $k$ do Exit($i$) end
```

Note that the relative address of L and the degree j are uniquely determined in the compiling phase by the position of the statement specified by X. The value of k depends on the actual parameter for X. So the above statements may be preceded by some instructions for transmitting the relevant dynamic level from X to k.

The operation (4) in Section 4 is automatically executed as a special case of the operation (2) in Section 3.

References

[1] NOZAKI, A., Problems of Storage Allocation. *Information Processing in Japan, 3* (1963), 44–47, Information Processing Society of Japan.

[2] JENSEN, J. et al., A Storage Allocation Scheme for Algol 60. *Communications of the Association for Computing Machinery, 4,* 10 (1961), 441–445.

[3] WATT, J. M., The Realization of ALGOL Procedures and Designational Expressions. *Computer Journal, 5,* 4 (1963).