MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 84

Translation of a Block Structured Language With Non-Local Go To Statements and Label Variables to the Base Language

by

Nimal Amerasinghe

This work was submitted for credit in Subject 6.534, "Semantic Theory for Computer Systems," Spring 1973.

June 1973

TRANSLATION OF A BLOCK STRUCTURED LANGUAGE WITH NON-LOCAL GO TO STATEMENTS AND LABEL VARIABLES TO THE BASE LANGUAGE

by Nimal Amerasinghe

1. Introduction

Dennis (2) first put forward a scheme of translation of block structured languages to the common Base Language. Amerasinghe (1) defined a block structured language called BLKSTRUC which had extensive facilities for handling procedure variables and detailed a scheme of translation to a base language defined in terms of a chosen set of interpreter primitives. The block structured languages used by Dennis (2) and Amerasinghe (1) did not permit label variables and non-local go to statements. In the present paper a block structured language BLKSTRUC II is defined which incorporates label variables, non-local go to statements and begin--end blocks distinct from procedure blocks.

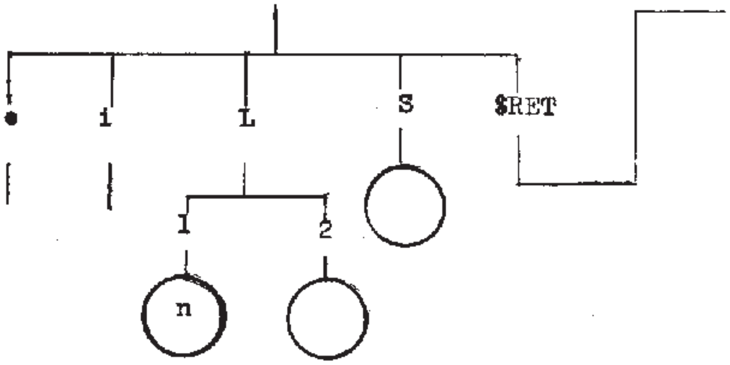
An interpreter is defined which enables correct execution of translated BLKSTRUC II programs. Garbage collection is done by cooperation between code introduced by the translator and the interpreter. The garbage collection scheme attempts to implement retention as defined by the contour model. A translation scheme is introduced from BLKSTRUC II programs to the base language defined.

2. A Base Language Interpreter

The interpreter defined in this Section differs from that introduced by Amerasinghe (1) in that the control structure of the interpreter state is modified, some interpreter primitives are redefined, and some new interpreter primitives are introduced.

The primitives add, mult, subtr, div, exp, selct, link, *const*, *delete*, *move*, *create*, *if* - *then go to*, *assign*, are defined in the same manner as in (1). The primitives *go to*, *apply* and *return* are redefined. In addition new primitives, *cmove*, *collect test* and *siteno*, are introduced.

The control structure consists of a series of 'sites of activity' selected by integer selectors. A typical site of activity is of the form:



The e component is linked to the corresponding component local structure, the i component is linked to the next instruction to be executed, the s component contains 1 if the site of activity is active and 0 if the site of activity is dormant and the $RET component is linked to the site of activity of the calling procedure activation. The L component plays a key part in the implementation of label variables and the implementation of the garbage collection scheme. The L.1 component contains the number of the site of activity and L.2 component contains a reference count for garbage collection. The reference count may be updated either by the interpreter or the executable code of a procedure structure. Whenever a site of activity is dormant and its reference count (L.2 component) becomes zero the site of activity is made active.

The following primitives are redefined:

apply primitive:

The apply instruction takes the form apply f, $ARG. When the apply primitive is executed a new site of activity and a new component local structure are created. The new component local structure is linked by a $PAR link to the argument structure selected by $ARG in the calling procedure activation. The new site of activity is linked to the calling site of activity by a $RET link, and the reference count in the L.2 component of the calling site of activity is incremented by one. The i component of the calling site of activity is linked to the next instruction in the calling procedure and the calling site of activity is made dormant (s component to 0). The new site of activity is made active (s component to 1).

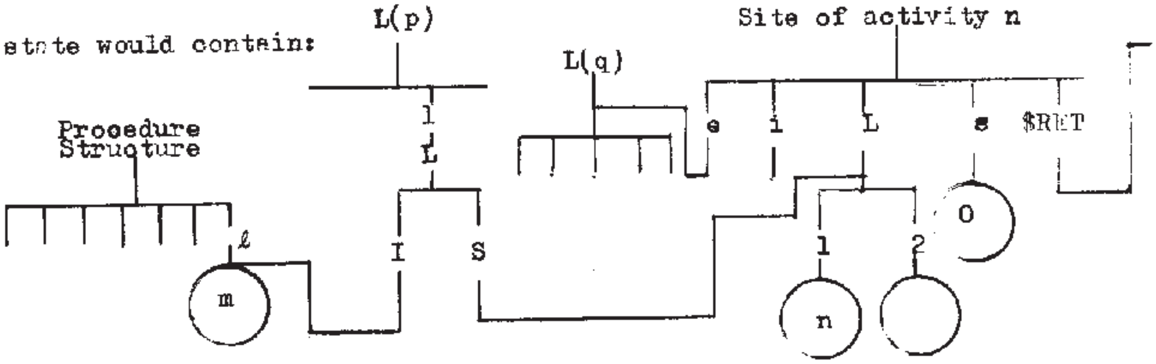
return primitive:

The i component of the current site of activity is linked to the next instruction in the procedure structure. The current site of activity is made dormant (s component to 0) and the site of activity selected by the $RET component is made active. Note that the garbage collection function which was combined with the return primitive in (1) is separated from the present return primitive. This is because in the presence of label variables component local structures could no longer be deleted on the execution of the return primitive.

go to primitive:

Whenever the form of the instruction is 'go to n' where

where n is a number the primitive is assumed to be a local go to and is defined as in (1). If the instruction is of the form 'go to l' where l is a label variable a different mechanism comes into play. At the time a non-local go to is executed in procedure p the interpreter state would contain:



The interpreter links the i component of the current site of activity to the next instruction and makes the current site of activity dormant. Then it selects the site of activity 'n' linked to by the S component of l and links its i component to instruction 'm' where m is contained by the I component of l. Finally the site of activity n is made active.

The new primitives *cmove*, *test*, *collect* and *site no* are defined below.

cmove primitive:

The instruction takes the form *cmove* L, l. Execution of the primitive makes the L component of the current site of activity the l component of the corresponding component local structute.

test primitive:

The instruction takes the form *test* p, q, t. If a selector named q emanates from the node selected by p in the current component local structure t is set to 'true'; otherwise t is set to false.

collect primitive:

The 'collect' primitive causes the current component local structure to be deleted. The L.2 component of the site of activity selected by the $RET component of the current site of activity is decremented by one, and the current site of activity is deleted. The collect primitive performs the garbage collection function which was usurped from the return primitive in (1).

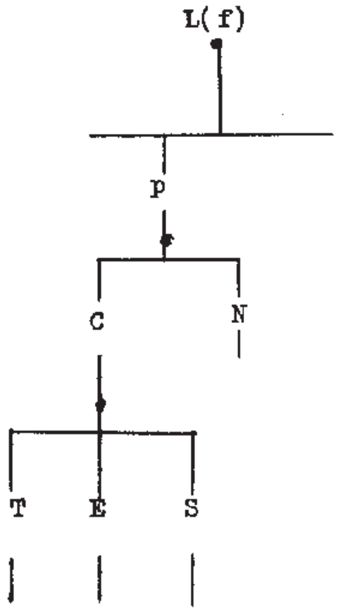
site no primitive:

When an instruction of the form 'site no t' is executed the number of the current site of activity number is stored in t.

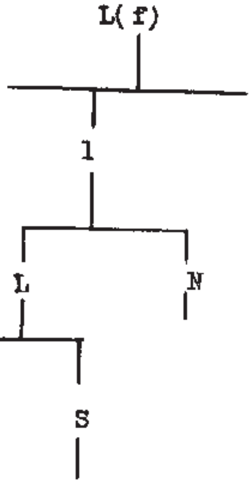
The philosophy behind the garbage collection scheme supported by the translator is to mark each procedure variable and label variable at the time of its declaration by its own site of activity number. Whenever a procedure assignment or a label assignment is made a check is made whether the environment of declaration of the C structure or the L structure which is being assigned is the same as the environment of declaration of the variable to which assignment is being made. If the environments are different the reference count in the site of activity corresponding to the declaration environment of the C structure or L structure is incremented by one. At the same time any C structures or L structures which were previously linked to the variable being assigned to, have their 'reference counts' decremented by one if environments are different from that of the variable. Whenever a label variable or a procedure variable is passed as an argument in a procedure activation the reference count linked to by the corresponding C structure or

L structure is incremented by one. When the above control of reference counts is combined with interpreter control of the reference counts, the scheme simulates a scheme where in terms of the contour model a count is being kept of all the external references to each contour. Deletion of a 'contour' occurs whenever this reference count to a given 'contour' is zero.

Every procedure variable p has a N component which contains the site of activity number of the environment of declaration. A procedure value is a C structure containing a T component linked to the procedure structure, E component linked to the externals and an S component linked to the L component of the site of activity corresponding to the declaration environment. A typical procedure variable which is assigned to a C structure is represented in the interpreter state as,



Similarly a label variabal l assigned to a L structure is represented as,



3. An Informal Introduction to BLKSTRUC II

A BLKSTRUC II program is a set of nested procedure blocks and begin-end blocks nested within one another in tree structured fashion. A typical procedure block is of the form,

```
p = procedure (x1, x2, ---------xn)
    integer --------proced --------label -- declaration statements
    begin
      :
      :
      :   executable BLKSTRUC II statements
      :
      :
    end
```

A typical begin-end block is of the form

```
    begin
    integer----------- proced ------- label ------
      :
      :
      :
    end
```

In the procedure declaration shown p is a procedure variable, x_1, x_2——x_n are formal parameters of the type integer or proced (procedure) or label. All local variables and arguments in a procedure block and local variables in a begin–end block are defined as type integer, proced, or label. Declaration statements are only permitted in procedure blocks in the lines following the procedure declaration but before the line containing the begin statement for the block. The begin and end statements in a procedure block are delimiters enclosing a sequence of executable BLKSTRUC II statements. Declaration statements in begin–end blocks follow immediately after the begin statement of the block before the first executable statement of the block.

Executable statements in a BLKSTRUC IIprogram are one of the following types:

(a) An arithmetic assignment statement
(b) A procedure assignment statement
(c) A label assignment statement
(d) An application statement of a non value returning procedure.
(e) An application statement of a value returning procedure.
(f) A local go to statement
(g) A non–local go to statement
(h) A conditional statement
(i) An iteration statement
(j) A return statement for a non–value returning procedure
(k) A return statement for a value returning procedure
(l) A stop statement

Any executable statement may be labelled by an identifier ; e.g. 1 : S_1

An identifier labelling a statement is not explicitly declared at the head of the procedure block or begin-end block. Such an identifier is called an implicitly declared label variable.

The different types of executable statements are described in detail below:

(a) Arithmetic assignment statement

An arithmetic assignment statement takes the form,

x = <expression >

A typical example is

$x = a + (b * c) / a + c \uparrow d - b$

a, b, c, d, x are integer type variables

The permissible operators are +, -, *, /, ↑ .

Nested parentheses impose a precedence relationship for expression evaluation. In the absence of parentheses the precedence ordering of operators for evaluation is, ↑ , * or /, + or -.

(b) Procedure assignment:

A procedure assignment takes the form p = q where p and q are procedure variables. For the statement to be meaningful q must be assigned to a closure before the statement is executed.

(c) A label assignment takes the form $l_1 = l_2$ where l_1 is an explicitly declared label variable and l_2 is either an implicits declared label variable or an explicitly declared label variable which has been previously assigned to a label value.

(d) An application statement of a non value returning procedure:

A statement takes the form,

$$\text{apply } f\ (x_1,\ x_2,\ \text{———},\ x_n)$$

where f is the procedure variable being applied and x_1, x_2, ——— x_n are arguments of the type integer or proced or label.

(e) An application statement of a value returning procedure:

A statement takes the form,

$$z = \text{apply } f\ (x_1,\ x_2,\ \text{———} x_n)$$

where f is the procedure variable being applied and x_1, x_2, ——— x_n are arguments of the type integer, proced or label. If f returns an integer value z is of the type integer; if f returns a procedure value z is of the type proced.

(f) A Local go to statement:

A local go to statement takes the form ‘go to l’ where l is an implicitly declared label variable declared within the same procedure block or begin-end block.

(g) A non-local go to statement

A non-local go to statement takes the form go to l where either l is an implicitly declared label variable in another block or procedure block or is an explicitly declared label variable.

(h) A conditional statement

A conditional statement takes the forms,

if p(x) then S_1

if p(x) then S_1 else S_2

p denotes an unspecified predicate and S_1 and S_2 are either single executable statements or a sequence of executable statements delimited by begin and end statements.

(i) An iteration statement

An iteration statement takes the form

$$\underline{\text{while}}\ p(x)\ \underline{\text{do}}\ S_1$$

where p is an unspecified predicate and S_1 is either a single executable statement or a sequence of executable statements delimited by begin and end statements.

(j) Return statement for a non-value returning procedure

A statement takes the form, 'return'. Its effect is to transfer control to the next statement in the calling procedure following the apply statement.

(k) Return Statement for a value returning procedure

A statement takes the form,'return z' z may be of the type **integer**, **proced** or **label**.

(l) Stop statement

A statement takes the form, 'stop' and terminates execution of the program.

Transformation of program containing procedure blocks and begin-end blocks to program containing only procedure blocks

Following Amerasinghe (1) the translation scheme to be specified is for BLKSTRUC II programs containing no begin-end blocks. Accordingly BLKSTRUC II programs containing begin-end blocks have to be transformed into equivalent BLKSTRUC II programs not containing begin-end blocks before translation to the base language by the procedure to be outlined. The transformation consists of converting each begin-end block to a parameterless procedure and assigning the text of the parameter less procedure to a new procedure variable declared in the block containing the begin-end block being transformed. The transformation is completed by introducing a non-value returning apply statement 'apply p()' where p is the procedure variable to which the parameterless procedure text is assigned, immediately after the text of the parameterless procedure.

e.g. consider the BLKSTRUC II program,

```
p = procedure ()
    integer x, y, z
    begin
    x = 1
    y = 2
    z = x + y
            begin
            integer a
               a = z + y
               x = a + x
               end
    z = x + y
    stop
    end
```

Eliminating the begin-end block by the transformation we have,

```
p = procedure ()
    integer x, y, z proced q
```

```
     begin
     x = 1
     y = 2
     z = x + y
         q = procedure ()
             integer a
             begin
               a  =  z + y
               x  =  a + x
               return
               end
     apply q ()
     z = x + y
     stop
     end
```

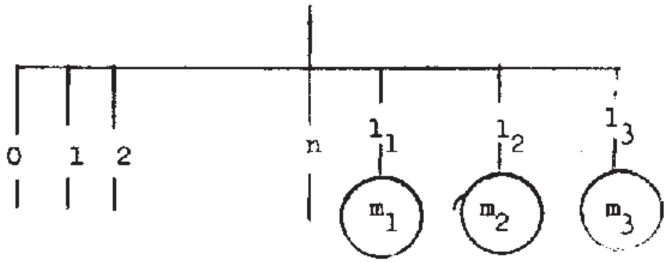
The new procedure variable q is declared within the procedure block p containing the begin-end block being transformed. Conversion to a parameterless procedure involves the introduction of a procedure declaration statement and a return statement. 'apply q()' is introduced immediately after the text of q to ensure proper activation of the transformed begin-end block.

In the translation procedure to be described in the next section all refeferences to BLKSTRUC II programs would refer to programs in which begin-end blocks have been eliminated by the transformation.

4. Translation of BLKSTRUC II programs to the base language

Amerasinghe (1) showed how 'housekeeping instructions' have to be introduced to the translated BLKSTRUC programs to simulate the non-local environments associated with procedure variables. In the procedure described in this section additional housekeeping instructions are introduced which enable environments associated with label variables to be correctly simulated and garbage collection to be performed.

Each procedure block in the BLKSTRUC II program is translated into a procedure structure of the form



The selectors 0 through n select the instructions comprising the procedure structure. l_1, l_2, l_3 are implicit label variables which are declared within the corresponding procedure block. The leaf node selected by l_1 contains the selector corresponding to the base language instruction which is effectively labelled by l_1. The complete procedure structure is a tree structured nesting of procedure structures of the type shown in the figure; the nesting of component procedure structures within one another is the same as the nesting of the corresponding procedure blocks within one another.

To specify the translation procedure some notation is necessary. Suppose T is the text of a procedure declaration. B(T) denotes the set of identifiers declared within T (local to T). The set X(T) of external identifiers associated with text T is defined as follows: $T' < T$ denotes that text T' is nested within text T; i.e. there is a sequence of texts $T_0, T_1, T_2,$ -------T_k such that $T = T_0$, $T' = T_k$ and T_i encloses T_{i+1} for $i = 0, 1,$ -----$,k-1$. Then X(T) contains each identifier x that has a non-local appearance in some text T', $T' \leq T$ and is not local to any text T'', $T' < T'' \leq T$.

Using the above procedure it is possible to determine the sets B(T) and X(T) corresponding to each procedure declaration.

(a) Arithmetic assignment statement

Consider the arithmetic assignment statement,

$$x = (a + b) * c + d/e - f$$

when translated to the base language we get,

```
add  a, b, temp 1
mult templ, c, temp 1
div d, e, temp 2
add templ, temp2, templ
subtr templ, f, x
```

templ and temp2 are temporary variables introduced by the translation. In a similar way assignment to any arithmetic expression may be translated.

Assignment to a constant viz x = 5
may be translated as, const 5, x

Assignment to a single variable viz x = y
may be translated as, assign y, x

(b) A Procedure declaration Statement:

A statement takes the form p = procedure (   ).

$X(p) = y_1, y_2, \ldots\ldots\ldots y_k$.

The base language code is as follows:

```
    test  p, C, templ
    if  (templ) then go to m
    go to n
 m: if (p. N = p. C. S. 1) then go to n
    subtr  p. C. S. 2, 1, p. C. S. 2
 n: delete p, C
    move  p, p. C. T.
    cmove L, p. C. S
    link  p. C. E, $y_1$, $y_1$
    link  p. C. E, $y_2$, $y_2$
     .
     .
     .
     .
    link  p. C. E, $y_k$, $y_k$
    if  (p. C. S. 1 = p. N) then  go to 1
    add  p. C. S. 2, 1, p. C. S. 2
    1: ------------------------------------
```

The code checks whether p is already assigned to a C structure and if so the corresponding reference count is decremented. Then it sets up a C structure corresponding to the procedure declaration linking the T component to the text, the S component to the L component of the site of activity and the E component to the externals. If necessary the reference count of the C structure being assigned is incremented.

(c) A Label declaration statement:

A label is assumed to be declared in the innermost procedure block in which it appears. If a label l is implicitly declared in a BLKSTRUC II procedure, a L structure is set up in the local structure. The I component of the L structure is linked to the l component of the procedure structure and the S component is linked to the L component of the corresponding site of activity. The base language code is as follows:

```
move  l, l. L. I
cmove L, l. L. S.
```

(d) A procedure assignment statement:

A statement takes the form p = q. The code checks whether p is already attached to a C structure and if so decrements the corresponding reference count. Then it deletes the C structure and links to the structure selected by the q. C component. The p. C. S. 1 component is compared with the p. N. component and if different the reference count of the C structure is incremented by one. The base language code is as follows:

```
   test   p, C, templ
   if ( temp 1) go to m
   go to n
m: if (p. N = p. C. S. 1) then go to n
   subtr   p. C. S. 2, 1, p. C. S. 2
n: delete  p, C
```

```
     link p, C, q. C
     if (p. C. S. 1 = p. N) then go to 1
     add p. C. S. 2, 1, p. C. S. 2
1:   ------------------------------------
```

(e) Label Assignment Statement:

The statement takes the form $l_1 = l_2$. If l_1 is already allocated to a label structure the code checks whether l_1. N component is the same as the l_1. L. S. 1 component and if different decrements the reference count by one. The old label structure attached to l_1 is deleted and the l_2. L component is linked to l_1. A check is made whether the l_1. N component is the same as the l_1. L. S. 1 component and if not the same increments the reference count of the L structure by one. The base language code is as follows:

```
   test $l_1$, L, templ
   if (templ) then go to m
   go to n
m: if ($l_1$. N = $l_1$. L. S. 1) then go to n
   subtr $l_1$. L. S. 2, 1, $l_1$. L. S. 2
n: delete $l_1$, L
   link $l_1$, L, $l_2$. L
   if ($l_1$. N = $l_1$. L. S. 1) then go to 1
   add $l_1$. L. S. 2, 1, $l_1$. L. S. 2
1: ------------------------------------
```

(f) An application statement of a non-value returning procedure:

A typical statement takes the form

apply f (x_1, x_2, p_1, p_2, l_1, l_2) where x_1, x_2 are *integer* arguments, p_1 and p_2 are procedure arguments and l_1 and l_2 are label arguments. The reference counts of the C structures attached to p_1 and p_2 have to be incremented by one and the reference counts of the L structures attached to l_1 and l_2 have to be incremented by one. The base language code is as follows:

```
add     $p_1$. C. S. 2, 1, $p_1$. C. S. 2
add     $p_2$. C. S. 2, 1, $p_2$. C. S. 2
add     $l_1$. L. S. 2, 1, $l_1$. L. S. 2
add     $l_2$. L. S. 2, 1, $l_2$. L. S. 2
delete  $ARG
create  $ARG
link    $ARG, 1, $x_1$
link    $ARG, 2, $x_2$
link    $ARG, 3, $p_1$
link    $ARG, 4, $p_2$
link    $ARG, 5, $l_1$
link    $ARG, 6, $l_2$
link    $ARG, E, f.C.E
apply   f. T, $ARG
delete  $ARG
```

(g) Application of a value-returning procedure:

A typical statement takes the form

$$z = \text{apply } f\,(x_1, x_2, p_1, p_2, l_1, l_2)$$

where x_1, x_2 are integer type arguments, p_1 and p_2 are procedure type arguments and l_1, l_2 are label type arguments. If z is an integer type variable an arithmetic assignment is made to z in the statement and the base language code is as follows:

```
add     p_1. C. S. 2, 1, p_1. C. S. 2
add     p_2. C. S. 2, 1, p_2. C. S. 2
add     l_1. L. S. 2, 1, l_1. L. S. 2
add     l_2. L. S. 2, 1, l_2. L. S. 2
delete $ARG
create    $ARG
link      $ARG, 2, x_2
link      $ARG, 3, p_1
link      $ARG, 4, p_2
link      $ARG, 5, l_1
link      $ARG, 6, l_2
link      $ARG, E, f. C. E
select    $ARG, $RET, z
delete    $ARG
```

If z is a proced type variable then f returns a procedure value. On encountering the return statement in f the reference count of the C structure being returned is incremented by one in anticipation of the assignment to z. After linking the returned C structure to the code has to check whether the z. F component is the same as the z. C. S. 1 component. If the same the increment of the reference count in anticipation, in the text of f, has been made in error and is compensated for by decrementing the reference count by one. The code is as follows:

```
    add     $p_1$. C. S. 2, 1, $p_1$. C. S. 2
    add     $p_2$. C. S. 2, 1, $p_2$. C. S. 2
    add     $l_1$. L. S. 2, 1, $l_1$. L. S. 2
    add     $l_2$. L. S. 2, 1, $l_2$. L. S. 2
    delete  $ARG
    create  $ARG
    link    $ARG, 1, $x_1$
    link    $ARG, 2, $x_2$
    link    $ARG, 3, $p_1$
    link    $ARG, 4, $p_2$
    link    $ARG, 5, $l_1$
    link    $ARG, 6, $l_2$
    link    $ARG, E, f. C. E.
    apply   f. T, $ARG
    test    z, C, templ
    if      (templ) then go to m
    go to n
m:  if (z. N = z. C. S. 1) then go to n
    subtr  z. C. S. 2, 1, z. C. S. 2
n:  delete z, C
    select  $ARG, $RET, z
    delete $ARG
    if  (z. N = z. C. S. 1) then go to k
        go to l
    k: subtr z. C. S. 2, 1, z. C. S. 2
    l: ----------------------------------------
```

If z is a <u>label</u> type variable then f returns a label value. On encountering the return statement in f the reference count of the L structure being returned is incremented by one in anticipation of the assignment to z. After linking the returned structure to z the code has to check whether the z. N component is the same as the z. L. S. 1 component. If the same the increment of the reference count in anticipation in the text of f has been made in error and is compensated for by decrementing the reference count by one.

The code may be obtained in a manner exactly analogous to the case where z is a procedure variable.

(h) A local go to statement:

A statement of the form 'go to l' is translated to 'go to n' where n is the selector of the first instruction of the instruction block corresponding to labelled statement l.

(i) A non-local go to statement:

A statement takes the form 'go to l'. The base language code contains go to l followed by a block of housekeeping instructions which participate in garbage collection. After the procedure activation is exit on the execution of the non-local go to instruction, the corresponding site of activity is made dormant. However, at any time the reference count may reach zero and the site of activity may be made active again. At this stage the block of garbage collection instructions added after the non-local go to is obeyed resulting in the decrement of reference counts of L structures and C structures attached to locally declred label and procedure variables. This decrementing operation is followed by the execution of the primitive <u>collect</u> which results in garbage collection of the component local structure and the site of activity.

Suppose the locally declared procedure variables are p_1, p_2, ...p_n and the locally declared label variables are l_1, l_2, ...l_m. The base language code is as follows:

```
      go to l
```

Block of garbage collection instructions.

```
      test $p_1$, C, temp
      if (temp) then go to $m_1$
      go to $n_1$
$m_1$: subtr $p_1$. C. S. 2, 1, $p_1$. C. S. 2
$n_1$: test $p_2$, C, temp
      if (temp) then go to $m_2$
      go to $n_2$
$m_2$: subtr $p_2$. C. S. 2, 1, $p_2$. C. S. 2
$n_2$: ---------------
      ---------------
      test $p_n$, C, temp
      if (temp) then go to $m_n$
      go to $n_n$
$m_n$: subtr $p_n$. C. S. 2, 1, $p_n$. C. S. 2
$n_n$: test $l_1$, L, temp
      if (temp) then go to $k_1$
      go to $l_1$
$k_1$: subtr $l_1$. L. S. 2, 1, $l_1$. L. S. 2
$l_1$: test $l_2$, L, temp
```

```
     if (temp) then go to $k_2$
     go to $l_2$
$k_2$: subtr $l_2$. L. S. 2, 1, $l_2$. L. S. 2
$l_2$: --------------------
     --------------------
     --------------------
     test $l_n$, L, temp
     if (temp) then go to $k_m$
     go to $l_m$
$k_m$: subtr $l_n$. L. S. 2, 1, $l_n$. L. S. 2
$l_m$: collect
```

(j) A conditional statement:

A conditional statement is translated as in (1)

(k) An Iteration statement:

An iteration statement is translated as in (1).

(1) A return statement for non-value returning procedure:

The statement which takes the form 'return' is translated to the return primitive followed by a block of garbage collection instructions.

When the return primitive is executed the corresponding site of activity is made dormant and control passes to the site of activity attached to the $RET link. However, subsequently when the reference count reaches zero the site of activity is again made active and executes the block of garbage collection instructions following the return primitive.

If $p_1, p_2, \quad, p_n$ is the set of locally declared procedure variables and $l_1, l_2, \ldots\ldots l_m$ is the set of locally declared label variables the corresponding base language code is:

```
return
( Block of garbage collection      )
( instructions identical to        )
( that for the non local go to     )
( instruction                      )
```

(m) **Return state for a value returning procedure:**

The translation is dependent on whether the value being returned is an integer value, procedure value or label value. The garbage collection block of instructions is identical to that for a non-value returning 'return' statement.

Suppose the statement is of the form 'return z' where z is an <u>integer</u> variable. The base language code is,

```
link      $PAR, $RET, z
return
( block of garbage collection   )
( instructions identical to     )
( non-value returning 'return'  )
```

Suppose the statement is of the form, 'return z' where z is a <u>proced</u> type variable. The reference count of the returned C structure is incremented in anticipation of the procedure assignment in the called procedure. The base language code is,

```
add  z. C. S. 2, 1, z. C. S. 2
link  $PAR. $RET, C, z. C
return
```

```
block of garbage collection instructions
identical as for non-value returning
'return'
```

Suppose the statement is of the form 'return z' where z is a label variable. The reference count of the returned L structure is incremented by one in anticipation of the label assignment in the calling procedure. The base language instructions are as follows:

```
add z. L. S. 2, 1, z. L. S. 2
link $PAR. $RET, L, z. L
return
block of garbage collection instructions
identical as for non-value returning
'return'
```

(n) Initiation of Procedure structure:

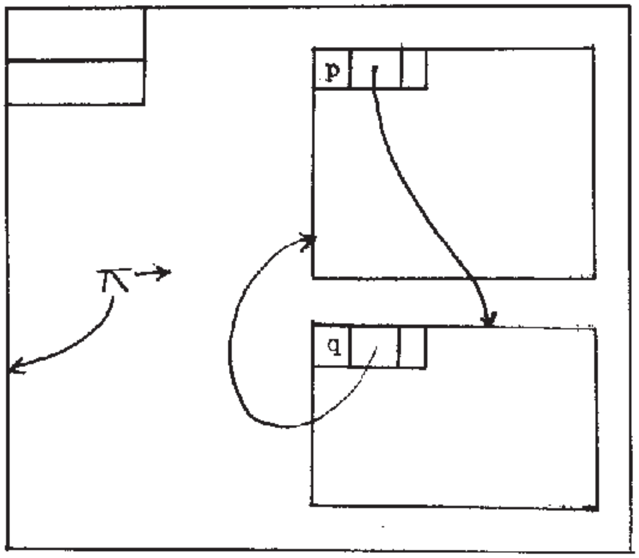
At the head of each procedure structure housekeeping instructions are introduced to provide direct access to arguments and externals and to mark the locally declared procedure variables and label variables with the site of activity number.

Suppose the formal parameters are x_1, x_2, p_1, p_2, l_1, l_2 where x_1, x_2 are <u>integer</u> type, p_1, p_2 are <u>proced</u> type and l_1, l_2 are <u>label</u> type. Let p_1', p_2' be the locally declared procedure variables, l_1', l_2' the locally declared label variables and x_1', x_2' the locally declared integer variables. Let y_1, y_2 be the externals. The base language code for initialization is as follows:

```
select    $PAR, 1, $x_1$
select    $PAR, 2, $x_2$
select    $PAR, 3, $p_1$
select    $PAR, 4, $p_2$
select    $PAR, 5, $l_1$
select    $PAR, 6, $l_2$
select    $PAR. E, $y_1$, $y_1$
select    $PAR. E, $y_2$, $y_2$
siteno    temp
assign    temp, $p_1$. N
assign    temp, $p_2$. N
assign    temp, $l_1$. N
assign    temp, $l_2$. N
assign    temp, $p_1'$. N
assign    temp, $p_2'$. N
assign    temp, $l_1'$. N
assign    temp, $l_2'$. N
create    $x_1'$
create    $x_2'$
```

5. Conclusion

The scheme of translation presented incorporates a garbage collection scheme which attempts to mimic 'retention' in the contour model. However, the scheme being essentially a reference count scheme works only when the contours do not form isolated cycles such as:



In the case shown above the enclosed contours could be de-allocated together asthey are inaccessible to the processor. However, in our reference count scheme we would have a reference count of one for each contour and hence would not de-allocate the contours. It is clear from a study of contour structure that such cases only arise relatively infrequently in practical programs. As such we stipulate that our garbage collection scheme is of some value. It should be noted that cycles in the contours of the types stipulated above <u>do</u> <u>not</u> correspond to cycles in the interpreter states.

It should be noted that the reference count we keep in the garbage collection scheme is effectively the total number of external references to the 'equivalent' contour. These references include return pointers, environment pointers of procedure variables and environment pointers of label variables.

The scheme of implementing label variables does not introduce directed cycles to interpreter states.

Since concurrency has not been handled in the interpreter it is important that only one site of activity actually executes during a computation. Hence although several sites of activity may become active in the garbage collection mode it is important that they are executed in sequence to avoid indeterminacy arising from several sites of activity attempting to update the same reference count simultaneously. The interpreter could adopt some simple stragety to ensure that the sites of activity are executed in sequence.

The scheme outlined involved repeating a block of garbage collection instructions immediately after each return and non-local go to statement. This repetition could be avoided if we have multiple entry points to each procedure among which is a special garbage collection entry point to be used by the interpreter when a site of activity becomes spontaneously active in the garbage collection mode.

REFERENCES

(1) Amerasinghe, S. N. "The Handling of Procedure Variables in a Base Language," S.M. Thesis, Dept. of Electrical Engineering, September 1972, MIT, Cambridge, Mass.

(2) Dennis, J. B. "On the Design and Specification of a Common Base Language," Computation Structures Group Memo 60 PROJECT MAC, MIT, Cambridge.

(3) Johnston, J. B. "The Contour Model of Block Structured Processes," Proceedings of a Symposium on Data Structures and Programming Languages, SIGPLAN Notices Vol. 6 No. 2, ACM February 1971.