Reprinted from the COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY
Volume 4, Number 1, January 1961
Made in U.S.A.

Thunks

A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations*

P. Z. Ingerman

University of Pennsylvania, Philadelphia, Pa.

Introduction

This paper presents a technique for the implementation of procedure statements, with some comments on the implementation of procedure declarations. It was felt that a solution which had both elegance and mechanizability was more desirable than a brute-force solution. It is to be explicitly understood that this solution is *one* acceptable solution to a problem soluble in many ways.

*This work was supported by the University of Pennsylvania, Office of Computer Research and Education, and the U. S. Air Force under Contract AF-49(638)-951.

Origin of Thunk

The basic problem involved in the compilation of procedure statements and declarations is one of transmission of information. If a procedure declaration is invoked several times by several different procedure statements,

the actual parameters which are substituted for the formal parameters may differ. Even if the several invocations are from the same procedure statement, the value of the actual parameters may change from call to call.

There are three basic types of information that need to be transmitted: first, the value of a parameter; second, the place where a value is to be stored; and third, the location to which a transfer is to be made.

In each of the three cases above, the requirements can be met by providing an address: first, the address in which the desired value is located; second, the address into which a value is to be stored; and third, the address to which a transfer is to be made. (This is somewhat simplified; more details are considered below.)

A *thunk* is a piece of coding which provides an address. When executed, it leaves in some standard location (memory, accumulator, or index register, for example) the address of the variable with which it is associated. There is precisely one thunk associated with each actual parameter in each specific procedure statement. (The handling of arrays requires a slightly extended definition —see below.) If an actual parameter is an expression, the associated thunk (each time it is used) evaluates the expression, stores the value in some temporary location, and delivers the address of the temporary location. If an actual parameter is a subscripted variable, the thunk (each time it is used) delivers the address of the specified element of the array. If an actual parameter is a conditional expression, the thunk selects from the alternatives and delivers the appropriate address.

In the most general case, the address transmitted by the thunk may be desired information, the address of the desired information, or the address where information is stored enabling the calculation of the desired information. The translator knows what kind of thunk to create by considering the syntax of the formation of the actual parameter and the previously scanned declarations. On the other hand, when a procedure declaration is being compiled, the translator, again by observing syntax, knows what kind of address to expect from a thunk.

The Simple Case

The simplest case, for explanatory purposes, involves a procedure whose formal parameters are all either labels or simple (non-subscripted) variables called by name. To compile such a procedure, one must consider both the procedure statement and the procedure declarations.

The procedure statement or function designator in its ALGOL-60 form looks like:

glub (a, b, ··· , m, n) (3.2.1, 4.7.1)

where "glub" is the procedure identifier. For simplicity's sake, the comma has been used exclusively, rather than the optional parameter delimiter ") ⟨letter string⟩ :(", as mentioned in the above-cited paragraphs.

When this procedure statement is compiled, it produces coding of the following description:

return-jump to glub
thunk a
thunk b
⋮
thunk m
thunk n

The procedure declaration heading corresponding to the above procedure statement contains in part:

procedure glub (p, q, ··· , y, z)

in which formal parameter p corresponds to actual parameter a, etc.

In the simple case under consideration, there are three types of parameters; those on the right side of a :=, and those on the left side of a :=, and those embedded in **go to** statements. Also, a formal parameter in the procedure body is identifiable because of its appearance in the procedure heading.

When a formal parameter appears on the right side of a :=, the generated coding may be described thus:
(1) Store any necessary registers
(2) Return-jump to the appropriate thunk
(3) Remove the desired quantity from the address provided by the thunk, and
(4) Restore the status quondam

When the formal parameter appears on the left side of a :=, what is needed at run time is not a quantity, but merely a location. In this case, step (3) above is replaced by
(3a) Store the calculated quantity at the address provided by the thunk

When the formal parameter is embedded in a **go to,** step (3) is replaced by
(3b) Transfer control to the address provided by the thunk

and step (4) is vacuous. Note that in each of the three cases, the thunk provided only an address; the interpretation of that address is a function of point-of-call on the thunk. This means that procedure declarations and procedure statements can be interpreted without cross-reference at compile time.

MORE DETAIL:

A simple, non-subscripted, variable is a special case of an expression; in addition to expressions, strings, array identifiers, switch identifiers, and procedure identifiers are also valid actual parameters (3.2.1, 4.7.1). In this section, the thunks for expressions in general and for strings will be discussed.

If an actual parameter is a string (2.6.1), the thunk delivers the address of the string. Note that in some machine implementations this might involve delivering the address of the first character of the string, with the expectation that the end of the string is marked by some suitable delimiter.

If an actual parameter is a simple variable (3.1.1), the associated thunk—each time it is used—delivers the address of the quantity named by the simple variable.

If the actual parameter is a label (3.5.1), the associated thunk (each time it is used) delivers the address to which transfer is to be made. This may require some additional mechanisms, as described in the paper on Recursive Procedures and Blocks (*see page* 65).

If an actual parameter is a Boolean or arithmetic expression (3.3.1, 3.4.1), the associated thunk (each time it is used) evaluates the expression, stores the value in some temporary storage location, and delivers the address of the temporary location. For example, if an actual parameter is a subscripted variable, the thunk delivers the address of the specified element in the array.

If an actual parameter is a conditional expression (3.3.1, 3.4.1), the associated thunk (each time it is used) evaluates the conditions and if necessary, the expression selected by them, and delivers the address as described in the preceding paragraph.

If an actual parameter is a function designator (3.2.1), the associated thunk (each time it is used) delivers the address to which transfer should be made. Note that this may involve several steps, if the desired element of the switch declaration is itself a switch designator, etc. (3.5.3).

If the actual parameter is a conditional designational expression (3.5.1), the associated thunk (each time it is used) evaluates the conditions to select the alternative and delivers the address to which transfer is to be made, using one of the two techniques described above.

Array Identifiers

When an actual parameter is an array identifier, its interpretation becomes somewhat more involved. At the time the procedure declaration heading is being scanned, there may be no way of telling that a formal parameter is an array identifier. However, when the body of the declaration is scanned, the sequence

⟨formal parameter⟩[

identifies the parameter as an array identifier. This sequence can occur on either side of a :=. In this case the return-jump to the thunk provides not the specific address, but rather the address of a table in which the necessary information for the subscription operation will be found. (The necessary information is a residue from the corresponding array declaration, which may have been evaluated dynamically when the block was entered.)

When the procedure statement is being compiled and the array declaration is available, it is obvious that the actual parameter is an array identifier, and the thunk which is compiled will provide the address of the table.

Switch Identifiers

A switch declaration may be construed as a vector whose elements are unconditional transfer instructions rather than data. With this interpretation, a switch identifier as a formal parameter may be interpreted in precisely the same manner as the array identifier discussed supra. In other words, a switch declaration may be treated in the same manner as a static, non-**own**, one-dimensional array.

Procedure Identifiers

If one makes the (not severely) restrictive assumption that the address (in the final coding) of the exit line of a procedure declaration can be determined if the address (in the final coding) of the entrance line is known, procedure identifiers cause no difficulty when used as formal parameters. In this case, the thunk provides the address of the entrance line of the procedure. The coding which calls on the thunk calculates the address of the exit line and places the two addresses into a return-jump. (Note again that this is a philosophical description; on, e.g., the IBM 704 the technique is actually simpler.)

However, there are some additional complications when using procedure identifiers as actual parameters. The technique described above works in all cases except when the formal parameter part is empty and the procedure is a function designator. In this case, the associated thunk (each time it is used) return-jumps to the procedure and delivers the address of the location in which the value of the procedure had been placed.

Procedure Declarations

When the formal parameter part of a procedure heading is scanned by the compiler, a list is made of the formal parameters. This list is used to distinguish those identifiers for which return-jumps to thunks must be generated. With the exception of the two cases to be discussed below, this is the only non-conventional coding generated. (By conventional is here meant the coding that would be turned out if the same statements were to be written outside a procedure declaration.)

Calls by Value

Normally, the procedure declaration introduces no coding as such. If the heading contains a value part, the corresponding formal parameters are understood to be replaced with generated local identifiers which must have values assigned to them each time the body of the procedure is entered. This is equivalent to inserting a number of ALGOL-60 statements of the form:

⟨generated local identifier⟩ := ⟨formal parameter⟩

immediately in front of the coding provided by the programmer, and then interpreting these statements as though they had been there all along. If a formal parameter was an array identifier, and the array was called by value, the generator ALGOL-60 statements must have the effect of (nested) **for** loops to move the array. The generated local identifiers replace the corresponding formal parameters wherever they appear in the body of the procedure declaration.

MORE DETAILS.

If a formal parameter is called by value, it must have its value bound before entering the body of the procedure,

and must be considered as a local identifier inside the procedure body.

If a formal parameter called by value is not used as an array identifier, coding of the following general description is turned out:

Return-jump to appropriate thunk
Fetch quantity from address provided by the thunk and assign it as the value of the associated generated local parameter.

It is worth noting that if a formal parameter called by value is used as a procedure identifier in the procedure body, the identifier identifies a procedure which has an empty formal parameter part and defines the value of a function designator (4.7.5.4). Thus the same general coding scheme applies to this case.

If a formal parameter called by value is used as an array identifier, the generated coding is of the form:

Return-jump to the appropriate thunk to get the address of the dope vector for the array.
Activate the FUSBUDGET mechanism and move the array to the newly assigned storage.

The FUSBUDGET mechanism is described in the paper by Sattley on Allocation of Storage (*see page* 60).

Nested Procedures

A formal parameter in a procedure declaration heading might appear only as an actual parameter of a procedure statement contained in the procedure declaration body. This is illustrated:

```
procedure glub (a, b, ··· , m, n)
        ⋮
george (a, x, y)
```

The same technique applies here. The procedure statement is replaced by

```
return-jump to george
thunk a
thunk x
thunk y
```

However, at the time the procedure statement for george is being scanned, a has already been noted as a formal parameter in the procedure declaration. Hence, the thunk that is compiled for a under the return-jump to george is in itself a return-jump to the thunk for the formal parameter a associated with the return-jump to glub.

Thus, if a *formal parameter called by name* in a procedure declaration appears as an actual parameter of some contained procedure statement, the thunk for that parameter (generated when the procedure statement is read) jumps to the thunk for the formal parameter. The thunk for the formal parameter may again jump (the chain may continue indefinitely) or may store the necessary information in the standard place.

In other words, if a formal parameter in a procedure delcaration heading appears only as an actual parameter in some contained procedure statement, the thunk generated by the procedure statement must go up one level, and must pass down unchanged the information there acquired. Hence it will call on the thunk corresponding to the formal parameter involved. The chain so set up can continue to any depth; it is only at the two ends of the chain that anything need be known about the type of address that is being transmitted.

Further Comments

This paper, admittedly, gives no attention to the question of the correct matching of arithmetic types within the procedure body when one operand is a formal parameter. Hence, this paper assumes that a programmer composing a call on a procedure must observe all the explicit and implicit assumptions of the procedure. (In addition, it assumes a doctrine like, for example, the following: "The value of a formal parameter appearing in an arithmetic expression in the body of the procedure will be assumed to be of type **real**, unless the formal parameter has been *specified* to be of type **integer** in the procedure heading").

An alternative means of handling this question might be: Have the thunk for the formal parameter return not only the required address, but also a type code for the value of the actual parameter. The correct data-matching would then be done dynamically during the execution of the procedure.

ACKNOWLEDGMENTS

The Author wishes to acknowledge the valuable contributions made to this paper by K. Sattley and W. Feurzeig of the University of Chicago, Laboratory for Applied Sciences; and E. T. Irons, Princeton University and Institute for Defense Analyses. Others who contributed were R. Floyd, Armour Research Foundation; and Miss M. L. Lind, and H. Kanner, University of Chicago, Institute of Computer Research.

Dynamic Declarations*

P. Z. Ingerman

University of Pennsylvania, Philadelphia, Pa.

A routine is described in this paper for mapping one array into another. This situation arises in the consideration of **own array**'s which are declared dynamically. In this case, the subscript bounds may change each time the declaration is invoked.

Table 1 gives the values of subscripted variables in the old and new arrays.

TABLE 1

Old Array	New Array	
	Subscript undefined	Subscript defined
Subscript undefined	Value undefined	Value undefined
Subscript defined	Value lost	Value preserved

There are three constraints on the mapping function First, if a set of subscripts is defined in both the old and new arrays, the value of the corresponding subscripted variable must remain unchanged by the mapping. Second, if a set of subscripts defined in the old array is undefined in the new array, the value of the corresponding subscripted variable is lost. Third, if a set of subscripts which was undefined in the old array becomes defined in the new array, the value of the corresponding subscripted variable is undefined, although space must be left in the new array so that a value may be assigned. These three conditions are an expansion of ALGOL Report Paragraph 5.2.5.

All non-**own array**'s stored in memory consist of two parts. The first part of the physical storage is a dope vector of the form:

0 number of subscripts declared for the array
1 (upper bound minus lower bound plus 1) of first subscript. This is the number of allowable values that the first subscript may take.
2 lower bound of first subscript
3 (upper bound minus lower bound plus 1) of second subscript
4 lower bound of second subscript
⋮
etc. for remaining subscripts.

After this dope vector, the elements of the array are listed—in numerical order by subscript. This implies, for example, that matrices are listed in order by rows.

With **own array**'s, an additional two-word "packet" follows the last element of the array. The first word of the packet contains the address of word zero of the dope vector. The contents of the second word of the packet is the address of the word in the body of the program which keeps track of the location of the **own array** when it is active.

Static **own array**'s are handled as described in the paper by Sattley. When the dynamic **array** declaration for an **own array** is invoked at run time, it causes the generation of the dope vector at an appropriate place in the memory.

If an array is stored as stated above, it can be represented by an equivalent vector. The mapping of the elements of the array into the elements of the vector is given by the relation:

$$A[i, j, \cdots, m] = V[(\cdots((i - i_0)J + j - j_0)\cdots)M + m - m_0]$$

where i_0, j_0, etc. are the lower bounds of the respective subscripts, and J, M, etc. are the number of allowable values that the respective subscripts may take (as defined above).

When considered in this manner, the mapping of the "old" vector-representing-an-array into a "new" vector is a fairly simple problem. There are three cases to be considered, corresponding to the three constraints described above. First, all elements that are defined for both the old and new versions of the array, in the sense that the subscripts are within bounds in both declarations, must be moved to their appropriate places in the new version of the array. Second, space must be left in the new version of the array for elements that are defined for the new version but undefined for the old version. Third, elements in the old version of the array that are undefined in the new version must be ignored, in the sense that they must be specifically excluded from transmission to the new version of the array.

The procedure ARSHIFT described below performs this mapping when necessary. It is called upon by FUSBUDGET when a previously declared **own array** is re-declared. If necessary, ARSHIFT moves the array to a location specified by FUSBUDGET, mapping the old version into the new version.

ARSHIFT is a Boolean function designator. On exiting from the procedure, ARSHIFT will have the value **false** if the new version of the array is identical in size and shape to the old version of the array. If ARSHIFT exits with the value **true**, the old version of the array has been mapped into the new version as described above.

ARSHIFT is a function designator which may, in a sense, change the values of global parameters. That is, when ARSHIFT exists with the value **true**, the memory has been rearranged. ARSHIFT contains within its declaration the recursive procedure MOVE which actually performs the mapping.

* This work was supported by the University of Pennsylvania, Office of Computer Research and Education, and the U. S. Air Force under Contract AF-49(638)-951.

B is the address of word zero of the dope vector of the old version of the array, and T is the address of word zero of the dope vector of the new version of the array. After the dope vector at the head of the new version has been stored by FUSBUDGET according to the data within the array declaration, FUSBUDGET calls on ARSHIFT (B, T).

EXIMUS is a global procedure. It is executed when FUSBUDGET requests that an array be mapped into an array of a different dimensionality; it and the statement in which it appears may be superfluous, if it is known that this sort of error will never occur.

It will be noted that ARSHIFT, as do several other procedures, uses a pseudo-vector referred to as "Memory". This vector is assumed to be declared in some high-level block by a declaration of the form:

array Memory [0 : < maximum memory address>]

The values of the elements of the vector are the contents of the cells whose addresses are the subscipts.

This pseudo-vector is a device for making the contents of memory cells available to a routine when the address of the cell is known. Because of the lack of synonym facilities in ALGOL 60 (but see ALGOL Maintenance Committee Proposal 9, Argonne National Laboratories), the actual declaration of such a vector would leave no room in the memory for program or storage. It is, however, a useful concept.

The Author wishes to acknowledge the valuable contributions made to this paper by K. Sattley and W. Feurzeig of the University of Chicago, Laboratory for Applied Sciences; and E. T. Irons, Princeton University and Institute for Defense Analyses. Others who contributed were R. Floyd, Armour Research Foundation; and Miss M. L. Lind, and H. Kanner, University of Chicago, Institute of Computer Research.

```
Boolean procedure ARSHIFT (B, T) ; value B, T ;
integer B, T ; begin integer m, k, p, a, y ;
    procedure MOVE ;
    begin integer S, Q, BU, TU ;
    m := m + 1 ; Q := Memory [B + 2 * m] ;
        S := Memory [T + 2 * m] ;
    BU := Memory [B + 2 * m - 1] + Q - 1 ;
    TU := Memory [T + 2 * m - 1] + S - 1 ;
    xyz: if Q = S then
        begin if k ≠ m then MOVE
            else begin comment move values ;
                Memory [T + n] := Memory [B + p] ;
                n := n + 1 ; p := p + 1 end ;
            Q := Q + if Q < BU then 1 else 0 ;
            S := S + if S < TU then 1 else 0 end
    else if (Q > S) ∨ (S > BU) then
        begin comment make space in the "new" array ;
            a := 1 ;
            if k ≠ m then for y := m + 1 step 1 until k do
                a := a * Memory [T + 2 * y - 1] ;
            n := n + a ; S := S + if S < TU then 1 else 0 end
    else begin comment skip elements because Q < S < BU ;
            a := 1 ;
            if k ≠ m then for y := m + 1 step 1 until k do
                a := a * Memory [B + 2 * y - 1] ;
        p := p + a ; Q := Q + if Q < BU then 1 else 0 end ;
    if (S ≠ TU) ∨ (Q ≠ BU) then go to xyz ;
    m := m - 1 ; end of MOVE ;
k := Memory [B] ; if k  Memory [T] then EXIMUS ;
ARSHIFT := false ;
for m := 1 step 1 until 2 * k do if Memory [B + m] ≠ Memory
    [T + m] then
    begin p := n := 2 * k + 1 ; m := 0 ; MOVE ;
        Memory [T + n + 2] := Memory [B + p + 2] ;
        Memory [B + p + 2] := 0 ;
        Memory [Memory [T + n + 2]] := Memory [T + n + 1]
        := T ;
        ARSHIFT := true ; m := 3 * k end
end of ARSHIFT ;
```

FIG. 1.