AN ALGORITHM FOR THE SELECTION OF OVERLOADED FUNCTIONS IN ADA

G. V. Cormack
Department of Computer Science
University of Manitoba
Winnipeg, Canada R3T 2N2

1. INTRODUCTION

In ADA, many distinct functions may have the same name. Many languages such as FORTRAN, PL/I and ALGOL 68 support this concept, although FORTRAN does so only for built-in operators. In these languages the types of the parameters are the only criteria used in selection of the appropriate function. ADA allows the type of the result to be considered during selection.

In conventional languages, the appropriate functions may be selected by a single bottom-up traversal of the parse tree. The ADA rationale [ICH 79] states an algorithm which selects functions using a number of top-down then bottom-up traversals. Dausmann et al. [DAU 79] show that this algorithm requires at most three traversals: top-down, then bottom-up, then top-down. They then show that the selection can be done with two traversals: bottom-up then top-down.

Both algorithms are complex and difficult to state precisely, and require the manipulation of sets whose size is unknown. The processing time required depends on the total number of set elements processed, rather than the number of traversals. Furthermore, storage is more likely to be a constraint than execution time. Thus it is not obvious that the [DAU 79] algorithm is superior to the [ICH 79] algorithm in terms of either memory usage or execution time.

This paper presents a succinct recursive algorithm which selects functions in ADA. Only stack (local) storage is required, and the stack depth will never exceed the depth of the parse tree being processed. In our examples, processing time has been found to be negligible. Nevertheless, three optimizations are presented. The first two require no extra storage but can save a great deal of time. The third requires some storage but reduces processing time to less than [ICH 79], and probably less than [DAU 79].

2. AN ALGORITHM TO COUNT THE NUMBER OF INTERPRETATIONS OF AN EXPRESSION

The following algorithm is expressed in the form of a recursive procedure, SELECT, which accepts as parameters a parse tree for an expression and the desired type of the expression. It returns the number of combinations of functions which may be selected such that the expression has a result of the desired type.

```
SELECT(DESIRED_TYPE,PARSE_TREE) returns NUMBER_SOLUTIONS;
   if PARSE_TREE is a leaf then
      if type of PARSE_TREE is DESIRED_TYPE then
         NUMBER_SOLUTIONS := 1
      else NUMBER_SOLUTIONS := 0
   else
      NUMBER_SOLUTIONS := 0
      let Q = name of proc at root of parse tree
      let R = list of actual parameters to Q (as parse trees)
      for every proc QQ with name Q which yields DESIRED_TYPE do
         let S = list of formal parameters to QQ
         if R matches S in terms of length, parameter names, and
            default values
         then
            PARM_COMBOS := 1
            for I from 1 to length of R do
               PARM_COMBOS := PARM_COMBOS * SELECT(type of S[I],R[I])
            NUMBER_SOLUTIONS := NUMBER_SOLUTIONS + PARM_COMBOS
END SELECT
```

If NUMBER_SOLUTIONS is 1, the expression is correct; if it is greater than 1, the expression is ambiguous; if it is 0, there is no consistent way of selecting the functions in the expression.

Note that all variables (NUMBER_SOLUTIONS, PARM_COMBOS, QQ, and I) are of fixed size and that the number of levels of recursion is equal to the depth of the parse tree. Furthermore, no functions are investigated unless their result could potentially be used in the expression. This statement also holds for [ICH 79] but is not true for [DAU 79]. This fact raises doubts as to whether or not [DAU 79] is faster than [ICH 79].

3. OPTIMIZATIONS TO THE ALGORITHM

3.1 OPTIMIZATIONS WHICH REQUIRE NO EXTRA STORAGE

Clearly, the loop beginning "for I" can be aborted if PARM_COMBOS ever becomes zero. This means that once a parameter of a function cannot be matched, that function is discarded from further consideration; we need not attempt to match the remainder of its parameters. Note that we may NOT terminate the loop when PARM_COMBOS exceeds one.

This modification reduces the processing required to reject invalid QQ's (note that, in any valid expression, all but one QQ are invalid). If there is one invalid parameter, we have to examine, on average, only half of the parameters before the invalid one is encountered. If more than one parameter is invalid, as will often be the case, shorter searches occur. This optimization applies at each level of recursion and thus causes exponential improvement with the depth of the parse tree. Further heuristics may be applied, such as matching parameters of uncommon types first, or matching parameters with smaller parse trees first. These techniques are not investigated in this paper.

A similar optimization may be applied to the loop beginning "for every QQ". The loop may be aborted when NUMBER_SOLUTIONS exceeds 1; once the expression is found to be ambiguous we don't care how ambiguous it is. Note that this optimization changes the result of SELECT; we no longer receive a true count but can still determine whether the expression was inconsistent, correct, or ambiguous. As with the previous example, heuristics may be applied to the order of the loop; as with the previous example, these heuristics have not been pursued.

3.2 AN OPTIMIZATION WHICH USES EXTRA STORAGE

With the above optimizations, the algorithm investigates fewer possibilities than [ICH 79] and many fewer than [DAU 79]. However, many recursive calls may be made with exactly the same arguments. For example, we may have:

```
PROCEDURE F(A:INTEGER; B:REAL);
PROCEDURE F(A:INTEGER; B:INTEGER);
```

In resolving the expression

```
F(1+2,3)
```

SELECT is called twice with DESIRED_TYPE = INTEGER and the parse tree for "1+2". This duplication can be avoided by attaching to each node of the parse tree a list of each DESIRED_TYPE for which SELECT has been called. The result of the calls must be stored also. The storage required would be equal to that required by [ICH 79] if the optimizations described in section 3.1 were not applied. If the optimizations were included, the storage required would be considerably less. Furthermore, running out of storage merely defeats the optimization; it need not cause termination of the algorithm.

4. YIELDING THE SELECTED FUNCTIONS

If the expression is unambiguous, it is necessary to set a pointer to the appropriate symbol table entry for each function in the parse tree. The algorithm, as discussed so far, has merely indicated whether or not an unambiguous filled-in tree exists. A trivial method of obtaining the filled-in parse tree would be for SELECT to return a filled-in subtree from every successful invocation. However, this section describes a modified algorithm which sets the pointers in the original parse tree, thus avoiding the storage cost of the previously mentioned modification.

```
SELECT(DESIRED_TYPE,PARSE_TREE,MARK) returns NUMBER_SOLUTIONS;
   if PARSE_TREE is a leaf then
      if type of PARSE_TREE is DESIRED_TYPE then
         NUMBER_SOLUTIONS := 1
      else NUMBER_SOLUTIONS := 0
   else
      NUMBER_SOLUTIONS := 0
      let Q = name of proc at root of parse tree
      let R = list of actual parameters to Q (as parse trees)
      for every proc QQ with name Q which yields DESIRED_TYPE do
         let S = list of formal parameters to QQ
         if R matches S in terms of length, parameter names, and
            default values
         then
            PARM_COMBOS := 1
            for I from 1 to length of R do
               PARM_COMBOS := PARM_COMBOS * SELECT(
                  type of S[I],R[I],MARK and NUMBER_SOLUTIONS=0)
            if MARK and PARM_COMBOS > 0 and NUMBER_SOLUTIONS = 0 then
               set pointer in root of PARSE_TREE to point to QQ
            NUMBER_SOLUTIONS := NUMBER_SOLUTIONS + PARM_COMBOS
 END SELECT
```

We have added a boolean parameter, MARK, which indicates whether or not PARSE_TREE should should be marked with pointers to the appropriate symbol table entries. The pointers continue to be filled in until a solution is found. Subsequent calls merely check for ambiguity and therefore do not request that the parse tree be marked. If there is one solution, the tree is filled in appropriately; if there is more than one solution, the first one is filled in; if there are no solutions, the pointers which are filled in in the parse tree are not meaningful.

The optimizations discussed in section 3.1 can be applied to this algorithm as easily as to the first. The optimization described in section 3.2 may be applied also, but the symbol table pointers for the root of the tree must be stored along with the list of desired types for which SELECT has been called.

5. CONCLUSIONS

The major advantage of the algorithm presented is simplicity, both as a statement of the rules for overloading and as a guide for implementation. It is suggested that the execution time of the algorithm is not important. Nevertheless, the speed of three algorithms has been analysed. [DAU 79] follows different paths than [ICH 79] or SELECT (the algorithm presented herein), however there is little reason to believe that it is faster or uses less storage than [ICH 79]. Three optimizations to SELECT, which are mechanical and do not change the algorithm, make SELECT considerably faster than [ICH 79].

SELECT is clearly superior in what we consider to be a more important constraint, namely, memory size. Two of the optimizations do not affect memory usage, while the third may be selected at run-time whenever there is surplus storage available.

The algorithm (without optimizations) has been implemented using PL/I. The effort required was very small and the algorithm was found to be effective.

REFERENCES

[ICH 79] Ichbiah,J. et al., Rationale for the Design of the ADA Programming Language. Sigplan Notices, Vol. 14, No. 6, June 1979.

[DAU 79] Dausmann,M. et al., Overloading in ADA. Institut fur Informatik II, Universitat Karlsruhe, Bericht Nr. 23/79