A One-Pass Algorithm for Overload Resolution in Ada

T. P. BAKER
Florida State University

A simple method is presented for detecting ambiguities and finding the correct interpretations of expressions in the programming language Ada. Unlike previously reported solutions to this problem, which require multiple passes over a tree structure, the method described here operates in one bottom-up pass, during which a directed acyclic graph is produced. The correctness of this approach is demonstrated by a brief formal argument.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classification—*Ada*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Overloading, translators, intermediate code

1. INTRODUCTION

The possibility of "overloading" certain kinds of symbols is one of the most interesting features of the programming language Ada [8, 11]. For an overloaded symbol in an Ada program, "there may be several meanings acceptable at a given point, and the ambiguity must be resolved by the rules of overloading" [8]. Operator symbols, subprogram and entry identifiers, and enumeration literals may be overloaded, and by extension, so may names and expressions that include instances of overloaded symbols.

For example, consider the following Ada declarations:

```
type T1 is (A, B);
type T2 is (A, B);
type T3 is (A, B);
function "+"(X: T2; Y: T3) return T1 is ...;
function "+"(X: T3; Y: T1) return T2 is ...;
function "+"(X: T1; Y: T2) return T3 is ...;
X: T1;
```

These declarations overload A and B as literals of three distinct types and + as three distinct operators. Let A_i and B_i denote the meanings of A and B as of type

This work was supported in part by the U.S. Air Force under contract F08635-81-0062 and in part by the National Science Foundation under grant MCS-79-24583.
Author's address: Department of Mathematics and Computer Science, Florida State University, Tallahassee, FL 32306.

Ti, respectively, for i = 1, 2, 3. Let $+_i$ denote the definition of + shown that returns type Ti, for i = 1, 2, 3. Within the visibility range of the definitions above (provided no further meanings are defined for A, B, and +), the expression A + A + A + A has three possible interpretations:

(a) $(((A_1 +_3 A_2) +_2 A_1) +_1 A_3)$;
(b) $(((A_2 +_1 A_3) +_3 A_2) +_2 A_1)$;
(c) $(((A_3 +_2 A_1) +_1 A_3) +_3 A_2)$.

The statement X := A + A + A + A; has only one legal interpretation, however, since in this context the type of A + A + A + A is required to match that of the simple variable X, namely, T1. Thus, assuming there are no other visible declarations of A and +, the interpretation of A + A + A + A in X := A + A + A + A; is unambiguously (a).

For each expression or name, a compiler for Ada must determine whether it has zero, one, or more legal interpretations, and if there is only one, it must produce the correct interpretation. This is made more difficult by a number of factors:

(1) **use** clauses, which make declarations of operators and literals from a package visible within an expression only if there is no consistent interpretation of that expression which does not use those declarations;[1]
(2) contexts that do not completely specify the types of the component expressions, but do impose some restriction, such as ranges (where both expressions must be of the same type) and assignments;
(3) operatorlike constructs that are unnamed, such as array and record aggregates, which are far more complex but have characteristics similar to function calls;
(4) constructions such as F(**new** A'RANGE(B + C)), where overloading in B + C must be completely resolved, so that it may be evaluated, before overloading of F may be resolved;[1]
(5) implicit type conversions;
(6) literal expressions, which always have a unique interpretation, independent of operator overloadings.[1]

Determining whether an expression has a unique meaning and, in the case that it does, identifying the types and definitions of all the constituent components (referred to hereafter as simply overload resolution) clearly is a complex task. For reliability and efficiency, care must be taken in the design of a "front end" processor for Ada that it does not become any more complex than is actually necessary.

A number of papers have appeared on the subject of overload resolution algorithms and their implementation [1–3, 4–6, 9, 10]. One reason for all this activity is that the algorithm originally offered by the Ada language design team [4] is clearly more complicated than necessary. In essence, it was suggested that an indefinite number of alternating passes be made up and down the tree of an expression, pruning away overloadings that fail to meet the requirements of

[1] Recent language changes appear to have eliminated this.

context, until convergence is reached. The first published improvement of this algorithm is given in [3], where it is shown that four passes over the tree are sufficient to reach convergence. Subsequent papers reduce the number of passes to two or three.

These multipass algorithms can be roughly divided into two groups. In the first, exemplified by Ganzinger and Ripken [3], the tree is processed three times: top-down, bottom-up, and top-down. During the first pass, the set of types allowed by context is used to select from all the visible declarations of each operator those that have permissible result types. The second pass checks the possible types of argument expressions against formal parameter types, pruning out operator interpretations that cannot apply to the given arguments. At the conclusion of this pass there will be a unique interpretation left for the operator at the root of the expression tree, provided the expression has exactly one legal interpretation. The third pass carries the consequences of this unique interpretation back down the tree, essentially repeating the processing done in the first pass. The algorithms in the second group, including those given in [6, 7], make only two passes, corresponding to the second and third passes described above. It would appear at first that these algorithms are more efficient, but Pennello et al. [6] observe that it might turn out that even though a preliminary top-down pass is not *needed* it may prove useful as a heuristic, should it turn out that this first pass is usually successful in finding a unique interpretation for an expression. Thus the question of which group of algorithms is better remains open until a good statistical comparison is done.

All the algorithms discussed so far make use of lists of interpretations which are attached to the nodes of a parse tree, or an abstraction of one, and so require the use of extra working storage, which is presumably reclaimed when the analysis of each expression has been completed. Cormack [2] proposes a radically different recursive algorithm that begins from the root of an expression tree and enumerates all possible interpretations of the tree. The algorithm is simple and requires as working storage only the stack used in implementing the recursion. On the other hand, its worst case running time appears to be of the order $(n^{\log_m k+1})(km^2)$, where n is the number of operators in an expression, k is the number of interpretations for each operator, and m is the number of arguments to each operator. This is compared to order kmn for the other algorithms. Despite this potentially longer running time, it is claimed that Cormack's algorithm takes very little time on the kind of expressions encountered in actual programs, especially when this time is viewed relative to the time spent doing all the other processing that is part of the compilation process.

The algorithm for overload resolution described in this paper is based on a viewpoint similar to that expressed by Cormack—that expression analysis is a small part of the work done by a compiler and that inefficiency on a few expressions is tolerable, so long as it permits reducing the overhead for simple expressions and the complexity of the compiler itself. Our viewpoint is different in two respects, however, owing to a bias created by the implementation context in which we expect overload resolution to take place. First, the implementation is viewed as parsing and producing intermediate code trees from the source in one bottom-up pass. Second, the symbol table and intermediate code are viewed

as belonging to a library file that resides on a secondary storage device, from which small pieces are brought into central memory when needed for processing. Our goal is therefore to perform as much of the semantic processing as possible in parallel with the parsing and construction of the intermediate code tree and to avoid backtracking whenever possible, since it might require bringing portions of the symbol table or code back into central memory.

As it turns out, this goal can be largely accomplished. It is possible, in one bottom-up pass, to read in an expression, parse it, check for a unique legal interpretation, and translate it to an intermediate code tree (or evaluate it, if static). This is accomplished by a technique that may be viewed as a refinement of the multipass methods derived from [4]. The essence of the refinement is that when a set of possible operators is constructed for a node it does not merely represent an unstructured set of possibly allowed operator definitions. Rather, each operator points to those operators in the operator lists of the actual parameter subexpressions that are legal operands of the operator. The pointers make superfluous the final top-down pass which two-pass methods such as those in [6, 7] must make to distribute the final operator information over the expression tree, provided operator trees or other pointer-based intermediate codes are acceptable as an intermediate form for the next stage of the compilation process.

Following precedent established by earlier papers on this subject, for the sake of clarity, the proposed algorithm for one-pass overloading resolution is presented in a simplified form, restricted to the cases of subprogram calls and expressions built up from elementary data objects by means of operators and function calls. Among the language features specifically neglected are aggregates, default parameters, and named parameter associations, implicit type conversions, evaluation of static expressions, and **use** clauses. It is not that extending the algorithm described here to take all such complicating factors into account is impossible. Indeed, most of the required extensions are discussed in Section 6, after the basic method has been explained.

Section 2 gives an informal explanation of the overloading resolution method. Section 3 introduces some formal definitions that are then used to prove two lemmas, which are the method's theoretical basis. Section 4 defines a directed acyclic graph (DAG) constructing function, *gen_calls*, in detail. Time and space requirements for overloading resolution using *gen_calls* are discussed in Section 5. Section 6, the concluding section, deals briefly with extensions of the basic algorithm to permit handling the full generality of Ada constructs, including such things as static expressions and declarations made visible by **use** clauses.

2. INFORMAL DESCRIPTION OF THE METHOD

Overload resolution is assumed to take place during a bottom-up left-to-right parse of the source program, during which a tree structure representing the meaning of the program is produced. Although the *final* goal of this process is the production of a tree structure, at intermediate stages the structures produced are likely to be DAGs with multiple roots, each root corresponding to a different interpretation of the Ada source construct. More precisely, the semantic structure representing a source construct is a set of trees, one for each meaning, which may share subtrees, representing shared interpretations of subexpressions. These

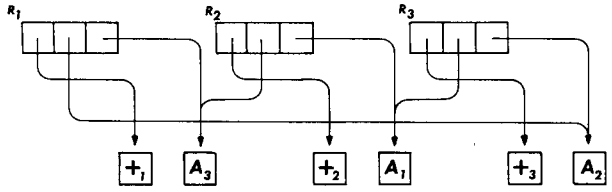

Fig. 1. Expression-DAG for A + A.

DAGs, or overlapping trees, are referred to as *expression-DAGs* here (even though they may in fact also represent constructs other than expressions, such as procedure calls).

The construction of expression-DAGs is performed by a function, *gen_calls*, which is described informally here and more formally in Section 4. *Gen_calls* takes two parameters. The first is a set V of operator definitions. The second is a sequence $(S_1, S_2, \ldots, S_n)$ of sets of roots of expression-DAGs representing actual parameters. Each S_i is a set of roots of an expression-DAG representing the ith actual parameter of an operator. *Gen_calls*(V, $(S_1, S_2, \ldots, S_n)$) returns a set of roots of an expression-DAG which represents all legal interpretations of the operators in V applied to the actual parameter-expressions in $(S_1, S_2, \ldots, S_n)$. In practice, V would ordinarily be the set of visible instances of some operator name and each S_i would be the set of visible instances of a simple actual parameter or the result of an earlier evaluation of *gen_calls* for an actual parameter expression.

To save memory, and to avoid risking exponential growth of expression-DAGs in the case that *gen_calls* discovers more than one applicable operator definition with the same result type, say T, all such interpretations are represented by the abbreviation "ambiguous(T)." No more information need be retained since the surrounding context can only be used to select from multiple interpretations on the basis of result type. When there is more than one interpretation for a subexpression with the same result type, such interpretations can never be part of an unambiguous interpretation of any surrounding expression. On the other extreme, if *gen_calls* cannot discover a set of legal actual parameter interpretations for any of the operator definitions, the special value "void" is returned.

Example. For the Ada program fragment given in Section 1, when the expression A + A is parsed, *gen_calls*($\{+_1, +_2, +_3\}$, ($\{A_1, A_2, A_3\}$, $\{A_1, A_2, A_3\}$)) would be evaluated, returning the set of roots $\{R_1, R_2, R_3\}$ of the DAG shown in Figure 1. (Recall that A_1, A_2, and A_3 are literals of types T1, T2, and T3, respectively, and that $+_1$, $+_2$, and $+_3$ are operators that return types T1, T2, and T3, respectively.)

Moving up the parse tree, when (A + A) + A is parsed, *gen_calls*($\{+_1, +_2, +_3\}$, ($\{R_1, R_2, R_3\}$, $\{A_1, A_2, A_3\}$)) would be evaluated, returning the set of roots $\{Q_1, Q_2, Q_3\}$ of the DAG shown in Figure 2.

Eventually a level is reached in the context of each expression where overload resolution must be completed. Three basic cases arise:

(1) An expression of a specific type, say T, is required. In this case, the root with result type T is selected from the set of roots provided by *gen_calls*, and the DAG is reduced to a single tree. If no root with result type T is found, or if ambiguous(T) is found, there is an error.

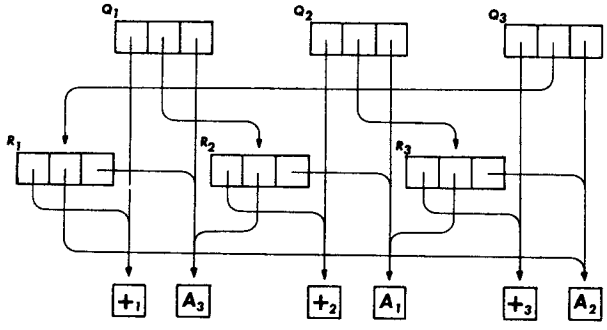

Fig. 2. Expression-DAG for (A + A) + A.

(2) A context, such as a range, that requires two expressions of the same type is reached. The sets of roots of the DAGs for the two expressions are compared to determine whether there is any type T for which each expression has a root. If there is exactly one such T and neither root of the pair is ambiguous(T), the correct interpretations of the two expressions have been found. Otherwise, an error has been detected.

(3) An operator, such as a procedure call, that returns no result is reached. Although it is not specifically pointed out in the definition of *gen_calls* (below), *gen_calls* return either void, a single tree, or ambiguous(no_type), in this case, according to whether the syntactic construct has zero, one, or more legal interpretations.

3. FORMAL FOUNDATIONS

The overload resolution method described in this paper is based on two heuristics proved in this section. In order to state these heuristics in a form that can be proved, some formal definitions are necessary.

Let Object_symbols be a set of atomic expressions.

Let Operator_symbols be a set of operator symbols. (It is intended here to include all Ada constructs that take parameter expressions, such as functions, type conversions, and aggregates, as well as the usual operator symbols.)

Expressions are constructed of *instances* of symbols. Suppose that there is an infinite set of instances for each object symbol and each operator symbol, and a mapping taking each instance to the unique symbol of which it is an instance. Any functions defined on symbols are implicitly extended to all instances of symbols.

An *expression* is either

(1) an instance of an object_symbol or
(2) an ordered sequence $(X, Y_1, \ldots, Y_k)$, where X is an instance of an operator symbol, $Y_1, \ldots, Y_k$ are expressions, $X, Y_1, \ldots, Y_k$ have no instances in common, and $k \geq 0$.

Instances of the same symbol are distinguished by the use of subscripts. For example, in $(F_1, A_1(F_2, A_2, B_1))$ there are two instances of F and two instances of A.

One expression S is a *subexpression* of another expression E if either

(1) E = S or
(2) for some subexpression $(X, Y_1, \ldots, Y_k)$ of E, $Y_i = S$.

Let Types be a set of types, including a special element, no_type.

Let Object_definitions be a set of possible denotations of object symbols, with special elements void, denoting an object with no legal interpretation, and ambiguous(T) for each type T, denoting an ambiguous object of type T.

Let Operator_definitions be a set of possible denotations of operator symbols.

Let out_type(X) give the type of X, if X is an object definition, and the result type of X if it is an operator definition.

Let number_of_parameters(D) give the number of formal operands of operator definition D.

Let parameter_type(F, I) give the type of the Ith operand of operator definition F.

Let visible(S) give the set of visible definitions of a symbol S.

An interpretation I for an expression is a mapping giving each instance X of a symbol in the expression a unique definition I(X).

Function out_type can now be extended to interpreted expressions:

out_type(E, I) = T if E is an object symbol and out_type(I(E)) = T;
out_type(E, I) = T if $E = (X, Y_1, \ldots, Y_k)$ and out_type(I(X)) = T.

An interpretation I for E is *legal* if for every instance X in E, I(X) is in visible(X) and

(1) X is an instance of an object symbol and I(X) is an object definition or
(2) X is an instance of an operator symbol F occurring in a subexpression $(X, Y_1, \ldots, Y_k)$ of E, I(F) is an operator definition, k = number_of_parameters(I(F)), and for $i = 1, \ldots, k$,

$$\text{out\_type}(I, Y_i) = \text{parameter\_type}(I(F), i).$$

An expression is *void* if it admits no legal interpretation. An expression is *ambiguous* (with respect to type T) if it admits more than one distinct legal interpretation (for which E has a result of type T).

We are now prepared to state and prove the two heuristics to be used in overload resolution.

LEMMA 1 (VOID HEURISTIC). *If E is an expression with subexpression S and S is void, then so is E.*

PROOF. Suppose S is a subexpression of E and E is not void. Any legal interpretation of E, when restricted to S, gives a legal interpretation for S, so S cannot be void. □

LEMMA 2 (AMBIGUITY HEURISTIC). If E is an expression with a unique legal interpretation I, then for every subexpression $(X, Y_1, \ldots, Y_k)$ of E and every Y_i $(1 \leq i \leq k)$, there is no other legal interpretation J of Y_i for which out_type(Y_i, J) = parameter_type(I(X), i).

PROOF. The proof is by induction on the definition of subexpression. As a basis,

suppose E = (X, Y_1, . . . , Y_k). Then if there is another legal interpretation J of some Y_i for which out_type(Y_i, J) = parameter_type(I(X), i), consider the interpretation J′ obtained by extending J to all of E, using the definitions given by I for instances not in Y_i. J′ ≠ I, since J and I differ on Y_i, but J′ must be legal, since J′ and I assign the same result type to Y_i and J and I are legal. This contradicts the hypothesis that E has no more than one legal interpretation. In general, suppose (A, B_1, . . . , B_n) is a subexpression of E and B_j = (X, Y_1, . . . , Y_k) for some j. Suppose I is the unique legal interpretation of E, and by induction, the restriction of I to B_j is the unique legal interpretation of B_j. If there is another legal interpretation J of Y_i (that differs from the restriction of I to Y_i) for which out_type(Y_i, J) = parameter_type(I(X), i) then, as in the basis case, J can be extended to give a new legal interpretation for all of B_j, and hence to all of E—a contradiction. □

4. DEFINITION OF *GEN_CALLS*

Let Expressions be the set of "interpreted expressions," that is, expressions in which every instance of a symbol has been replaced by a definition of that instance according to some legal interpretation of the expression. The function *gen_calls*, described informally in Section 2, and two other functions, matching_argument and gen_call, which it uses are defined below.

```
function matching_argument(T: types; ARGS: set of Expressions) return Expressions
    is
-- Return the unique expression in ARGS with out_type T, if possible.
-- If more than one, return ambiguous(T).
-- If none, return void.
        A: vector of Expressions;
        E: Expressions;
begin   A := null_vector;
        for each element E of ARGS
        loop
            if T = out_type(E) then A := A & X; end if;
        end loop;
        -- A contains all the interpreted expressions in ARGS
        -- that return type T.
        if length(A) ≥ 2 then return ambiguous(T);
        elsif A = null_vector then return void;
                                  else return A(1); end if;
end matching_argument;

function gen_call(F: Operator_definitions; ARGS: vector of set of Expressions)
    return Expressions is
-- ARGS is a vector with one component for each formal parameter of F.
-- The ith component of ARGS is a set of interpreted expressions,
-- one for each legal interpretation of the ith actual parameter expression.
        E: Expressions;
        I: Numbers;
        S: vector of Expressions;
        T: Types;
        ISAMBIGUOUS: Boolean;
begin   S := null_vector; ISAMBIGUOUS := false;
        for I in 1 . . number_of_parameters(F)
        loop T := parameter_type(F, I);
```

```
            T := matching_argument(T, ARGS(I));
            if E = void then return void;
            else ISAMBIGUOUS := ISAMBIGUOUS or E = ambiguous(T); end if;
            S := S & E;
        end loop;
        if ISAMBIGUOUS then return ambiguous(out_type(F));
        else return F & S; end if;
-- Void is returned if Lemma 1 guarantees the expression is void.
-- Ambiguous(T) is returned if Lemma 2 guarantees the expression has more
-- than one interpretation returning a value of type T.
-- Otherwise an interpreted expression is returned, using operator_definition
-- F, and matching argument_interpretations from ARGS.
end gen_call;

function gen_calls(FUNCS: set of Operator_definitions; ARGS: vector of set of Expres-
        sions) return Set of Expressions is
        F: Operator_definitions;
        S: Set of Expressions;
begin   S := empty_set;
        for each element F of FUNCS
        loop S := S ∪ {gen_call(F, ARGS)};
        end loop;
        return S;
end gen_calls;
```

5. ANALYSIS

Function matching_argument performs one iteration for each element of ARGS. Function gen_call performs at most one iteration for each formal parameter of F and calls matching_argument once in each iteration. Function *gen_calls* performs one iteration for each operator definition in FUNCS, and each iteration involves one call of gen_call. All together, the worst case running time of one call on *gen_calls* is thus bounded above by a constant times

$$\sum_{F \in FUNCS} \sum_{I=1}^{number\_of\_parameters(F)} |ARGS(I)|.$$

This could be improved slightly by using a more sophisticated scheme than linear search to find elements in ARGS that return type T, but such a change is likely to increase overhead without improving efficiency for the size sets likely to be encountered in use. Note that $|ARGS(I)|$ is the number of distinct possible result types for the Ith actual parameter.

Concerning worst case storage requirements, the key observation is that each call on *gen_calls* returns a set of expressions no larger than the number of distinct result types of the operator definitions in the set FUNCS, which consists of all the visible overloadings of the operator symbol under consideration. Each of these expressions requires additional storage proportional to the number of parameters of the corresponding operator definition. Since each operator definition in FUNCS corresponds to a visible definition of an operator, and each call of *gen_calls* corresponds to an actual use of an operator in an expression, even the most pathologically ambiguous program can use no more storage than the product of the number of operator and formal parameter definitions times the number of

uses of operators in the program. The example given earlier can be modified to exhibit this kind of worst case behavior.

Example

```
type T1 is (A, B);
        ⋮
type Tk is (A, B);
function "+"(X, Y: T1) return T1 is ...;
                    ⋮
function "+"(X, Y: Tk) return Tk is ...;
```

Within this context, A + A ··· + A (k operators) would expand to a code-DAG of height k, with k^2 internal nodes and leaves. It is expected that such situations will not arise frequently in practice, and that when they do, k will not be extremely large.

It is worth emphasizing that the storage requirements under discussion so far are for working storage, most of which can be collected after analysis of an expression has been completed. Continuing with the example above, if the expression occurred in the context of an assignment

$$X := A + A + \cdots + A;$$

k − 1 of the roots of the code-DAG generated could be discarded as soon as the complete statement is processed, reducing the storage used for the final interpretation to k internal nodes and one leaf. In the author's implementation, where code trees are kept on a secondary storage device, no attempt has yet been made to reclaim this storage, since the space consumed by intermediate code structures that are no longer needed has so far been tolerable. It is planned to add a garbage collector for this storage, and for other storage that may be freed as a result of library updates, later. Such a garbage collector might be viewed as a deferred "second pass," which would mean the algorithm proposed here must be a two-pass algorithm. Alternatively, a true second pass could be performed over the code structures created during the analysis of an expression, after analysis is complete, provided that multiple interpretations actually were constructed. Since such a pass would not access the symbol table and need not always be performed, it might be argued that it would still be preferable to the kind of second pass performed by methods such as those given in [6, 7].

Of course, it is no more clear that a one-pass method for overloading resolution is superior to a two-pass method than it is clear that a two-pass method is superior to a three-pass method. Choosing the correct method for an implementation requires balancing conflicting considerations, such as worst case versus average case behavior, time versus working storage versus storage for compiler code, and reliability versus efficiency. There does not appear to be any way of being certain of making the right choice short of exhaustive experimentation.

A few generalizations can be made, however. With the exception of [2], the worst case asymptotic orders of complexity of the published overload resolution algorithms are the same. The multipass methods make more node visits, but the one-pass method may incur greater overhead on its one pass than is incurred by

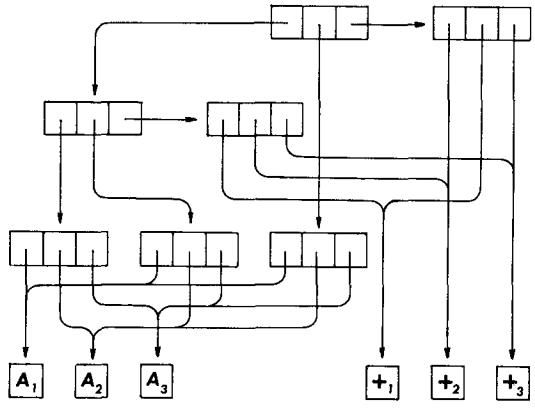

Fig. 3. Decorated expression tree for (A + A) + A.

the others on any single one of their passes, since it must save the formal–actual parameter correspondences that the multipass methods may discard (and recompute in part later). So far as size complexity of the algorithm is concerned, Cormack [2] appears to offer the simplest algorithm, with second place going to the one-pass method described here. With respect to requirements for working storage, the one-pass method appears equivalent or marginally better on expressions that are very small or do not contain many locally ambiguous symbols, but will require considerably more working storage on expressions that are harder to analyze. This can be seen by comparing the "decorated" code tree shown in Figure 3, which might be produced by an implementation of a multipass algorithm such as given in [6], with the structure shown in Figure 2, which would be produced by the algorithm described here on the same input. The one-pass algorithm seems to be able to save space on leaves (though this is an implementation detail), but it uses one extra pointer for each formal parameter of each operator interpretation.

Since, as of this writing, there are no validated Ada compilers yet operational, it appears to be too early to say which approach to overloading resolution will prove superior in use, if any.

6. CONCLUSIONS

We have explained how to resolve overloads in Ada expressions in a single bottom-up pass, which may be carried out during a bottom-up parse of a program. The author and some of his students are at work on an Ada implementation in which this algorithm is used in conjunction with an LALR(1) parser.

As mentioned earlier, several complicating factors have not been explicitly covered. The Ada language definition introduces a multitude of special cases and quite a few of these are encountered in the process of overloading resolution. It is not appropriate to go into all such details in this paper, but a few extensions to the algorithm presented here deserve mention.

First, the problem of performing static expression evaluation during overload resolution is trivially solved by modifying the final **return** statement of function gen_call to first check whether the function and actual parameter expressions

constitute a static expression and if so to evaluate it, returning a leaf representing this value rather than the vector (F, S_1, . . . , S_k).

Second, the handling of **use** clauses can be accomplished, still in one pass, by tagging each internal node of the DAG with one of six possible states. We represent each state by a two-digit sequence, where the first digit indicates the number of meanings of the node with **use** clauses and the second digit indicates the number of meanings of the node without **use** clauses. A zero indicates no meanings, a 1 indicates one meaning, and a 2 indicates two or more meanings. The six such states needed are

(00) void, independent of **use** clauses;
(10) void without **use** clauses, but representing a unique interpretation if names made visible by **use** clauses are taken into account;
(11) representing a unique interpretation, independent of **use** clauses;
(20) ambiguous, representing more than one interpretation with this result type, if **use** clauses are taken into account, but void otherwise;[2]
(21) ambiguous with **use** clauses, but representing a unique interpretation without **use** clauses;
(22) ambiguous, independent of **use** clauses.

Note that the Ada visibility rules, which state that a **use** clause may make additional names directly visible but may never cause names otherwise visible to be hidden, specifically rule out states that would correspond to (01), (02), and (12) by the numbering scheme used above. With slight modifications, the state of each node can be incorporated into the DAGs returned by function *gen_calls* described above.

Implicit type conversions can be supplied, when needed, by the function matching_argument. Similarly, handling default parameters and named parameter associations makes the functions gen_call and matching_argument more complicated, but the tree structures produced need not be any more complicated, and the detection of ambiguous and void constructs is no different.

Literal expressions can be handled by letting *gen_calls* discard all interpretations returning other types whenever an expression has an interpretation that returns a universal type.

Finally, there are aggregates. Although aggregates are in many ways like anonymous function calls (with more complex syntax), the type resolution of aggregates, and therefore of overloaded symbols in expressions which include or are a part of aggregates, touches on some details of the Ada language definition that are inadequately specified in the proposed standard at the time of this writing. It appears likely that changes or clarifications to the standard will be adopted which may improve the situation, but during the interim implementations must still identify the types of aggregates and do so in a manner totally consistent with the language definition.

[2] State 20 need not be distinguished from state 22 at the end of the analysis, but the two must be distinguished during the calculation of the collective state of a set of interpretations from the states of the individual interpretations, since adding an interpretation which is independent of **use** clauses (state 11) to a set of interpretations that are ambiguous but entirely dependent on **use** clauses (state 20) yields a set of interpretations with state 21, which is unambiguous.

The "brute force" application of the approach described here for functions to aggregates, making suitable allowance for the additional complexity of matching record and array components to expressions over matching formal parameters, leads to an implementation that can handle aggregates, though it may be very cumbersome in some cases. That is, all array and record types that are within scope at the point where an aggregate is found must be considered and checked against the aggregate for conformance. Code constructs for all possible interpretations are passed on up the tree. Clearly there are two problems: (1) in a context where many similar array and record types are available, the number of possibilities considered for the type of an aggregate may be very large; (2) for large aggregates, in deeply nested contexts where the required type cannot be locally determined, a large amount of storage may be required. These are not problems which can be avoided by multipass schemes, except in so far as they may be able to apply a greater number of heuristics. Expressions such as (A, B, C) = (D, E, F) provide no more information to a top-down multipass algorithm than they do to the bottom-up algorithm presented here. On such expressions a multipass scheme might use less storage, but it would have to process the same number of cases and so would take comparable time.

In the author's opinion, an Ada compiler must provide a fully general scheme for resolving the types of aggregates, but it must employ as many heuristics as possible to avoid using the general algorithm when it is not needed. In particular, it is possible to avoid consideration of more than one type for an aggregate in a number of contexts, including qualified expressions, allocators, and places where an initial or default value is given in a declaration. Ultimately, barring language changes to eliminate problem contexts such as (A, B, C) = (D, E, F) entirely, coding Ada programs so that the type of an aggregate is always available from a nearby context may make a difference between short and long compilation times (as well as making programs easier for humans to read).

ACKNOWLEDGMENTS

The author is grateful to D. Dunkle for implementing a multipass overload resolution algorithm for Ada which helped to motivate and demonstrate the greater simplicity of a one-pass approach, and to the U.S. Air Force for supporting the development of a prototype Ada semantic analyzer which demonstrated the approach's practicality.

REFERENCES

1. BELMONT, P. Type resolution in Ada: An implementation report. *SIGPLAN Notices* (ACM) *15*, 11 (Nov. 1980), 57–61.
2. CORMACK, G.V. An algorithm for the selection of overloaded functions in Ada. *SIGPLAN Notices* (ACM) *16*, 2 (Feb. 1981), 48–52.
3. GANZINGER, H., AND RIPKEN, K. Operator identification in Ada: Formal specification, complexity, and concrete implementation. *SIGPLAN Notices* (ACM) *15*, 2 (Feb. 1980), 30–42.
4. ICHBIAH, J., HELIARD, J., ROUBINE, O., BARNES, J., KRIEG-BRUECKNER, B., AND WICHMANN, B. Rationale for the design of the Ada programming language. *SIGPLAN Notices* (ACM) *14*, 6, pt. B (June 1979), 7-8–7-12.
5. JANAS, J.M. A comment on "Operator identification in Ada" by Ganzinger and Ripken. *SIGPLAN Notices* (ACM) *15*, 9 (Sept. 1980), 39–43.

6. PENNELLO, T., DEREMER, F., AND MEYERS, R. A simplified operator identification scheme for Ada. *SIGPLAN Notices* (ACM) *15*, 7 (July 1980), 82–87.
7. PERSCH, G., WINTERSTEIN, G., DAUSMANN, M., AND DROSSOPOULOU, S. Overloading in preliminary Ada. *SIGPLAN Notices* (ACM) *15*, 11 (Nov. 1980), 47–56.
8. *Reference Manual for the Ada Programming Language* (proposed standard), U.S. Department of Defense, Washington, D.C., Nov. 1980.
9. SHERMAN, M. A flexible semantic analyzer for Ada. *SIGPLAN Notices* (ACM) *15*, 11 (Nov. 1980), 62–71.
10. WALLIS, R.J., AND SILVERMAN, R.W. Efficient implementation of the Ada overloading rules. *Inf. Process. Lett. 10*, 3 (Apr. 1980), 120–123.
11. WEGNER, P. *Programming with Ada: An Introduction by Means of Graduated Examples.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

Received May 1981; revised November 1981 and March 1982; accepted March 1982