CMU-CS-85-186

Overload Resolution in Ada +

Robert G. Stockton

12 December 1985

Abstract

This paper describes one technique for performing Ada overload resolution. It involves a bottom-up scan of an attributed syntax tree which examines all possible intepretations of an expression and filters out all invalid interpretations.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

1. Introduction

One of the many useful features of the Ada programming language[1] is the capability to overload various symbols. Although this can contribute immensely to the readability of programs, it places a much greater burden upon the compiler, since the meaning of a symbol can not always be uniquely determined based upon its name. In fact, in some cases there might be several equally valid interpretations of the given symbol. The compiler must determine, based on the context of the symbol, which of the possible interpretations is the correct one. This document is an attempt to describe the way in which the Ada+ compiler[2] accomplishes this task. The algorithm presented handles all cases of overload resolution as described in [ARM 83]. It may be considered a development of the algorithm in [Baker 82], but it has been extensively modified and extended.

2. Overloading in Ada

A symbol in an Ada program can come to be overloaded in several different ways. In the simplest case an integer literal may belong to any one of several different integer types or a real literal may belong to either a fixed or floating point type. A more complex case arises when several subprograms are declared with the same name. As long as there is some difference between the parameter or result types, this is perfectly legal, and the compiler must be able to determine which subprogram is appropriate for any instance of the subprogram name. It should be noted that the standard operators are already overloaded in this manner and may be further overloaded by the user's program. Enumeration literals are regarded as functions which return the appropriate values, and can therefore also be overloaded. String literals and the literal **null** are also considered to be overloaded.

Resolution of overloaded subprograms is further complicated by several other factors. Positional or named parameter association or a mixture of the two may be used. A subprogram might have default arguments which may be omitted in the subprogram call. Furthermore, since the syntax of an array reference is a subset of that for subprogram calls, it may be difficult to determine whether a parenthesized list is a set of parameters or a set of array indices.

3. Data Structures

Although the algorithm employed by the Ada+ compiler may be adapted to other systems, the actual implementation is somewhat dependent upon data structures which are part of the overall compiler. This section therefore begins with descriptions of the most important of these. In addition,

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

[2]A general description of the Ada+ compiler may be found in [Barbacci 85]

it contains a description of another structure which is used internally by the overload resolution routines.

3.1. The Syntax Tree

Every compiler for a block structured language such as Ada must have some means of keeping track of its context at any given time. In the Ada+ compiler, this is accomplished by performing all operations upon an attributed syntax tree. This tree is built by an initial syntax processing phase, and is then traversed again by the semantic processing and code generation phases. Each production in the user's program is represented by a *node*, which has many fields which will at some point be loaded with information by the semantic phase. Other fields exist to allow information to be passed as "hidden arguments" to routines in all phases of the compilation.

3.2. Symbol blocks and Type blocks

During semantic processing and overload resolution, many references are made to *symbols* and *types*. These entities are represented by records called, respectively, *symbol blocks* and *type blocks*. A symbol is used to refer to any entity which has a name. Each symbol is maintained in an appropriate symbol table corresponding to the context in which it was declared. It should be noted that a symbol table can contain several different symbols with the same name. A type in Ada+ is considered to include any entity which has a separate context associated with it, including subprograms, packages, and tasks. It also includes the standard Ada notion of types. Note that since each type has a name, it will have a symbol associated with it. Therefore a variable would have a corresponding symbol block which contained information about the variable, such as its type, lexical level, and address in memory. A subprogram would also have a corresponding symbol block, but in addition it would have a type block which contained information about its parameters, return type (if it is a function), and internal variables.

3.3. Interps

During overload resolution, each interpretation of the node which is being considered is represented in an *interp*, a record which holds the following information:

1. A description of the interpretation. This tells whether the interpretation corresponds to a subprogram, an enumeration literal, an array reference, etc.

2. A pointer to the node which is currently being processed.

3. A description of the type and base type of the result (if any).

4. A pointer to the symbol block for the entity (i.e. subprogram, enumeration literal, etc.) to which this interpretation corresponds.

5. A value which indicates the number of implicit conversions required by this interpretation. (See section 7.)

6. A list of interps. These correspond to nodes lower in the syntax tree (i.e. actual parameters, etc.) and indicate what interpretations must be selected for those nodes to make this a valid interpretation for the current node.

Depending upon the type of the interpretation, certain fields may be unused. For example, the result type will be empty in the case of a procedure call.

4. Basic Operation

During the semantic phase the compiler recursively traverses the syntax tree, calling a production specific routine for each node. When dealing with cases in which a symbol may have several interpretations (primarily expressions), these routines call upon one of the overload resolution routines. In general, the routine will have one of three side effects:

1. If a unique interpretation has been discovered, information about that interpretation might be entered in the node. This information will usually consist of a pointer to a type block corresponding to the result type (if any) of the expression, and an (optional) pointer to a symbol block corresponding to the subprogram (or enumeration literal, etc.) which has been selected. In general, this will be done whenever a unique interpretation is found, although it is only necessary if the node corresponds to a complete context. There are, however, a few cases in which it is more convenient not to do so.

2. If several valid interpretations have been found, a list of interps will be stored in the node, in hopes that a later call will resolve the conflict.

3. If no valid interpretations have been found, an error message will be issued, and the node will be marked as erroneous.

In all cases, overload resolution proceeds from the bottom up (i.e returning to the root of the tree). For each node processed, an interp record is generated for every possible interpretation of the node's lexical token. Each of these is then examined to determine whether it is consistent with some set of interpretations that exist at a lower level. An interpretation is considered to be consistent if each of the nodes that constitute the expression (i.e. actual parameters, prefixes, etc.) have interpretations which correspond to the components of the interpretation. If it is consistent, references to these lower level interpretations are stored in the interp. If not, the given interpretation is eliminated from consideration. After all inconsistent interpretations have been eliminated, one of the three cases listed above will apply, and the appropriate action will be taken.

Note that since all consistent interpretations of the lower level nodes are available, this scheme ensures that no consistent interpretation will be eliminated at the current level. In addition, any interpretation that is inconsistent with respect to lower level nodes will be eliminated immediately. Therefore, when the top level is reached, all remaining interpretations are known to be consistent with respect to the entire expression, and are therefore valid.

Figure 4-1 illustrates a simple example of overload resolution. Consider the program fragment:

```
    type color is (red, green, blue, brown);
    type signal is (red, yellow, green);
    function foo(tint: color; light: signal) return integer;
    function foo(i,j: integer) return integer;
```

and consider the semantic processing for the expression

```
    foo(red, green)
```

During semantic processing for the enumeration literals, a routine is called which loads the corresponding nodes with interpretation lists A and B. After the semantic routines for these nodes finish, semantic processing for node 1 resumes and calls another overload resolution routine. This routine finds two functions with the name **foo** (Interp list C) and attempts to find valid interpretations using each of these. It succeeds in finding interpretations lower in the tree (A.2 and B.1) which allow a valid interpretation for Interp C.1. In this case, however, no lower level interpretations are found which are consistent with Interp C.2. Therefore, Interp C.2 is eliminated, and node 1 is left pointing to an interpretation list which contains only the one valid interpretation found (C.1). At this point we know exactly how to interpret the entire expression, and all that remains is to clean up the interp tree and load the nodes with information about this interpretation.

5. Utility Functions

A number of routines are called during overload resolution, but they all perform similar tasks. Therefore, there exist several auxiliary routines which perform actions common to all of these. This section describes the most important of these utility functions.

5.1. Match

Much of the work involved in overload resolution is performed by the function **Match**. The arguments to this function consist of a pointer to a type block and a pointer to a node, and the result is a pointer to a single interp. If there is a single interpretation of the given node which is of the given type, the interp returned will correspond to that interpretation. Otherwise, a **null** pointer will be returned. It should be noted that two types are considered to match if and only if they share a common base type (i.e. they are subtypes of the same type).

The action taken by this routine varies based on the form of the given node. If the node has already been loaded with an interpretation, **Match** checks that the type associated with the node is the same

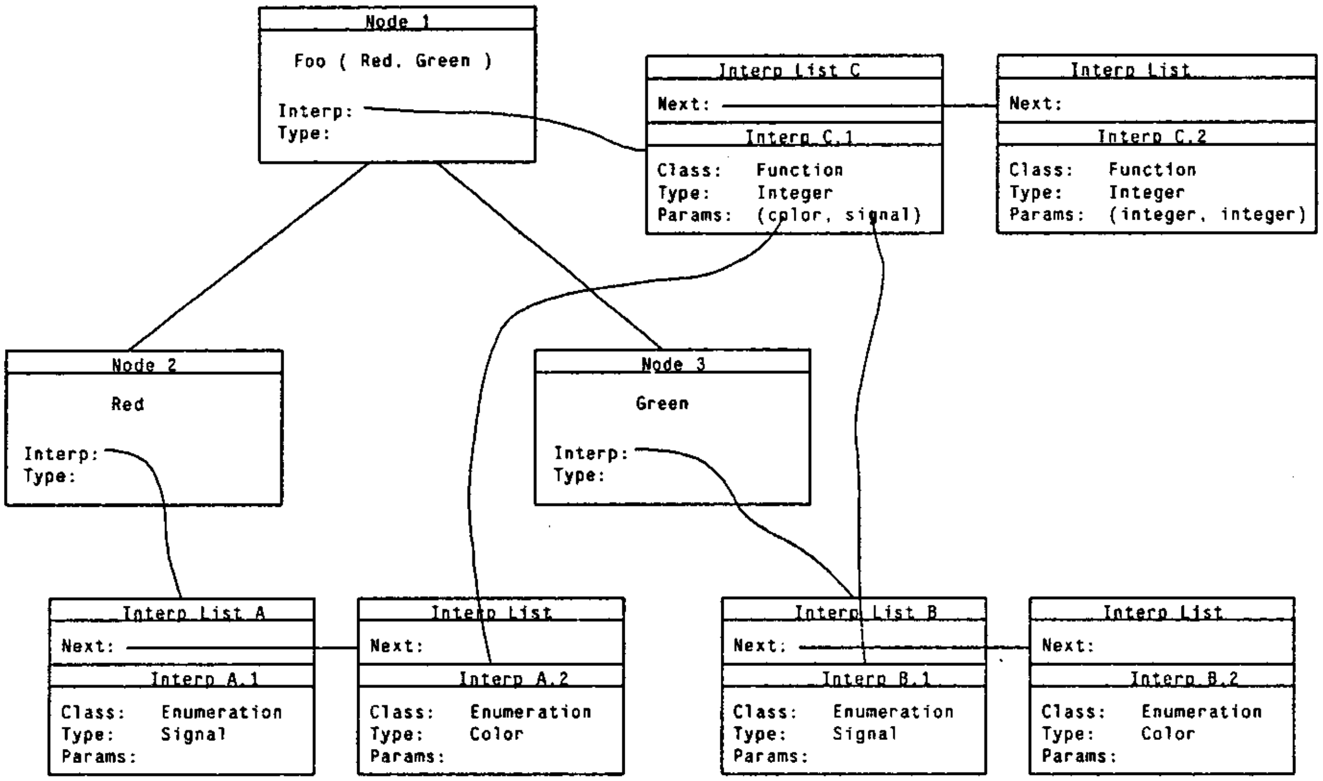

Figure 4-1: An example of overload resolution

as the given type and, if so, creates and returns a dummy interp. This interp contains no information other than an interp class and a node pointer, and exists primarily as a place holder. If, on the other hand, the node contains a list of possible interpretations, **Match** instead scans through this list and attempts to find the appropriate interpretation, which is then returned.

This routine also contains facilities for dealing with various special cases that occur in Ada overload resolution. These special cases include implicit conversions and aggregates, and will be dealt with later in this paper.

5.2. CleanUp

Once a unique interpretation for a given node has been found, information about this interpretation must be installed in the node. In addition, since the interpretation for this node uniquely determines the interpretations of the nodes beneath it, appropriate information should also be entered in them (if this has not already been done). This is done by the function **CleanUp** which takes a valid interp as its single parameter. **CleanUp** accomplishes this by recursively calling itself on each of the interps upon which the given interp depends (which are stored as the last field of the interp. See the description on page 3.) and then actually storing the requisite information (all of which is readily available from the interp) in the node corresponding to the given interp. In addition, **CleanUp** is responsible for deallocating the interp lists used in these nodes, performing validity checks upon the argument lists and, in special cases, performing additional semantic operations on the nodes. Since these special operations may generate errors, **CleanUp** returns a boolean value which is true iff no errors are generated.

A similar procedure, **AmbigClean**, is called whenever a node is found to be ambiguous. It simply deallocates all interps in this node and dependent nodes.

5.3. InterpFind

Each node corresponding to an overloaded symbol must be initially loaded with an interp list corresponding to all interpretations which might be valid. The operations described above can then operate upon this list and filter out the invalid interpretations. This initial list is formed by one of the **InterpFind** functions. Each of these functions searches through the visible symbol tables and locates all symbols which have the right name. The name passed to **InterpFind** will always correspond to a subprogram or enumeration literal (which can be treated as a function with zero parameters), so all symbols located will correspond to subprograms. **InterpFind** filters out some of these symbols based on the properties of the corresponding subprograms, and creates a new interp list with entries corresponding to the remaining symbols. The filtering performed varies between routines, as follows:

1. **InterpFind**: This routine is only called for infix operators, and filters out functions with the wrong number of parameters.

2. **Interp1Find**: This routine is called for subprograms and enumeration literals, and filters out subprograms with too few formal parameters. Note that a subprogram may have default parameters, so this routine accepts subprograms with too many parameters. In addition, if a function is desired then all procedures are filtered out, and vice versa.

3. **Interp2Find**: This routine is called when resolving renaming declarations and is basically a compromise between the first two. Like **InterpFind**, it filters out subprograms with the wrong number of parameters. Like **Interp1Find**, it also filters out all non-procedures or non-functions, depending upon which are desired.

6. Detailed Operation

As mentioned above, a different routine is called for each type of production being handled. Each of these routines incorporates specific knowledge relating to that production. Few assumptions are made, however, about the rest of the tree. The interp list (or exact interpretation) contained within a lower node is assumed to contain all of the information that will be needed concerning that node. Likewise, each of these routines will leave in the node's interp list all of the information which will be required by routines operating upon higher nodes.

All of the routines described below are called from external semantic processing routines which will not be described in this document. Since these external routines already incorporate knowledge concerning the productions which they handle, they are well suited to choose the proper overload resolution routine to call. Each of these external calls is required to take place after the appropriate calls for nodes lower in the tree. Thus, the ordering for the overload resolution routines is externally enforced. Since these routines are intended only for use within a single application (the Ada+ compiler), this is not an unreasonable requirement.

6.1. Procedure calls

Overload resolution for procedure calls is handled by the function **ProcLdType**. It begins by calling **Interp2Find** to compile an interp list with pointers to all procedures with the proper name which do not have fewer arguments than appear in the parameter list. It then compares the formal parameters for each of these procedures with the actual parameters given, as follows: each positional parameter is compared against the corresponding formal, and if a matching interpretation is found it is stored in the interp for the procedure. Likewise, each named parameter is compared against the corresponding formal, if that formal exists and does not already have an actual parameter associated with it; all remaining formal parameters are checked to insure that they have default values. If any mismatches are found in any of these phases then the interp is declared invalid and removed from the interp list. If, after all interps have been examined, there are no interps remaining on the list then the procedure call is invalid and **ProcLdType** returns False to its caller. If there is exactly one interp left, **CleanUp** is called to enter appropriate information in the node. Otherwise, the remaining interps are stored in the node. If the caller has no other way to distinguish the correct interpretation, it will signal an error and call **AmbigClean** to clean up the remaining interps.

6.2. Function calls and Array references

Function calls and array references must be treated together since, as was mentioned before, there is not always any easy way to distinguish between the two. Upon finding an expression consisting of a name (perhaps complex in itself) followed by a parenthesized list of expressions, two possibilites must be considered:

1. The name is either an array object or a function call which returns an array, and the parenthesized expressions constitute a set of array indices.

2. The name indicates the name of a function and the parenthesized expressions constitute a list of actual parameters to the function.

These possibilities are not mutually exclusive since a symbol corresponding to a function name might be either a function call (with no parameters) or merely the name of a function (with parameters following). Therefore, both of these possibilities must be considered simultaneously.

When such an expression is encountered, the name on the left is first resolved, if that name is an expression. If any interpretations are found which yield arrays, they are checked against the parenthesized list of expressions. If each element has an interpretation which matches the corresponding index type, then an interp is created which corresponds to the array reference, containing pointers to the matching interprations for the indices. This is done by the routine **ArrLdType**. Note that since the name on the left need not be a valid interpretation, none of the routines which might be called to resolve it may themselves generate error messages.

In addition, if the name on the left is a symbol corresponding to a function, the procedure **FunLdType** is called. This procedure performs much the same function as **ProcLdType**, but in addition, if the caller has indicated a required return type, it checks that the return types of any functions considered match the given type. It merely stores any valid interpretations without checking for uniqueness.

The interp lists resulting from the two steps above are then combined and the resulting list is treated exactly like the interp list resulting from a procedure call. Note that in this case it is likely, if two or more interps are stored in the node, that these will be further resolved at a higher level based on their types.

6.3. Operators

Overload resolution for operators is handled by the routines **UnLdType** and **BinLdType**, for unary and binary operators respectively. These functions perform the same basic function as **FunLdType**, but are much simpler since they need not be concerned with named parameter association or default parameters, and the number of parameters is fixed.

6.4. Slices

Slices are in many ways similar to array references, and therefore overload resolution for slices is handled by the same routine as is used for array references, **ArrLdType**. This routine can easily distinguish whether it is dealing with a slice or an array by checking whether the first index is a discrete range. The behavior of the routine varies in this case because it must load a different 'return' type and must generate a different type of interp in order to indicate to **CleanUp** that the interpretation should be processed differently.

When **CleanUp** is called to install a correct interpretation, it does some extra processing in order to install the correct array subtype as well as the correct type in the given node.

6.5. Record components and Access values

Overload resolution for record component selection is extremely easy. We need merely examine all possible interpretations of the left side, and dissallow all interpretations which do not satisfy the following criteria:

1. The type of the interpretation must be either a record type or an access type which designates a record type.

2. Neither the record type nor the access type (if any) mentioned above may be private.

3. The record type must have a component corresponding to the selector given.

For each interp which matches these criteria a new interp is created which corresponds to the result of the component selection. A list containing all of these new interps is then stored in the given node.

Selected components of the form *value*.all are handled almost identically by the routines **AccLdType**. Here, however, the criteria used are even simpler:

1. The type of the interpretation must be an access type.

2. This access type must not be private.

6.6. Other constructs

There are several other language constructs which share the same properties with respect to overload resolution, and which are therefore handled together. Assignment statements, discrete ranges, and short circuit operators all involve finding matching interpretations for two different expressions. All of these are handled by the routine **SameLdType**. In some of these cases, however, there are additional criteria which must be met in order for a set of matching interpretations to be accepted. These criteria are as follows:

1. The interpretation for the left hand side of an assignment statement must correspond to an object rather than a value.

2. The types of the interpretations for a discrete range must be discrete.

3. The interpretations for a short circuit operator must be of a boolean type.

SameLdType merely goes through all of the interpretations for each expression and finds all sets which match. All matching sets which meet the criteria given above are collected in an interp list which is handled in the same was as for procedure calls.

6.7. Complete contexts

Upon reaching a complete context, the overload resolution algorithm must take special actions. In particular, it must either find a unique interpretation for the expression being processed or generate an error. In some cases, the context also provides further constraints on the type of the expression. A family of routines exists to perform this task and may be called by the semantic routines for complete contexts. Each of these merely scans the list of interpretations for the expression (or a previously loaded type) and finds the ones which satisfy some given constraint. The remaining interpretations are then examined and the routines either load the unique interpretation or generate an error message, as appropriate.

7. Special Cases

The preceding sections of this document have described the process of overload resolution for standard elements. Ada, however, has several additional features which can sometimes complicate this process. Special handling is required for implicit conversion of universal types. Aggregates, the null expression, and allocators are also given special handling, since the type of these objects must be deduced from a very large set of possible types.

7.1. Universal types

Universal types are somewhat tricky to handle, since implicit conversions to any one of several similar types must be considered, but may not actually be performed unless they are proven to be necessary. The problem may be divided into two parts: generating interpretations which involve these implicit conversions, and determining whether it is necessary to perform them.

The key to the first part lies in the function **Match**. As described in section 5.1, this function is passed a type and a node for which an interpretation is required. If this node falls into one of the few classes which yields a convertible[3] universal type, a check is made. If this check reveals that the universal type can be converted into the given type, a new interp is created which corresponds to the given type and this interp is added to the list for the node. The interp is then returned to the caller. The interps created here are of a special variety, and therefore when **CleanUp** is called upon one of them, it knows to load the result type of the conversion into the node. This indicates clearly to the rest of the compiler that an implicit conversion has taken place.

The second part is somewhat more complex, and requires special handling in several areas:

1. For every interpretation generated, a value is maintained which describes the number of implicit conversions which it requires. This value includes implicit conversion performed in subexpressions. Since the algorithm works up from the bottom nodes of the expression, this value is easy to maintain.

2. In some cases, it is possible for several valid interpretations to exist which return the same type. (For example "(1 = 2)" has three interpretations, all of which return "Boolean".) In these cases, the interpretation which involves the fewest implicit conversion is the correct one, and the others may be ignored. It would be possible to eliminate the other interpretations immediately (or to report an error if there are two equally valid interpretations), but the expense of detecting and eliminating such interpretations was deemed to be too high. Instead, all of these interpretations are retained, which causes special handling to be required in later calls to **Match**. Since this routine is merely trying to find a single interpretation of the given type, it may just ignore all such interpretations which require more than the minimum number of implicit conversions. The interpretation with the minimum number of implicit conversions is guaranteed to be the appropriate one.

3. Each of the routines for handling complete contexts performs basically the same operation described above for **Match**. All valid interpretations which involve more than the minimum number of implicit conversions must be ignored since some of these conversions are "unnecessary".

The first step described above satisfies the requirement that all possible interpretations be

[3]The node may represent a numeric literal, a named number, or an attribute.

generated. The second step is required in order to eliminate all interpretations which involve unnecessary implicit conversions.

7.2. Other special cases

A number of Ada expressions allow such a wide range of results that it becomes infeasible to attempt to enumerate all possible interpretations. For example, an aggregate may correspond to any composite type and it is illegal to attempt to use the contents of the aggregate to eliminate inconsistent interpretations. Other constructs which present similar problems are allocators, string literals, and the null expression. All of these constructs allow the result to be any type which satisfies some constraint. Rather than scanning all types in order to find the ones with the appropriate properties or maintaining tables of such types, the Ada+ compiler employs an alternate method. No interpretations are generated initially for any of these constructs. Instead, when such a construct is passed to the routine **Match**, it examines the target type to see if it has the properties required for the given construct. If it does not, then Match returns false. If the target type has the required properties, then **Match** returns a corresponding interp. This interp may be created by **Match** or it may be an interp which was generated by an earlier invocation and stored in the node.

This approach is feasible for these constructs because all of them place fairly simple constraints on possible result types. It allows these potentially difficult constructs to be handled fairly simply. It must, however, be used with care because of the constraints which it places on the various overload resolution routines. Since there is no complete list of possible interpretations, all routines which may deal with these constructs must do so through the **Match** routine. In addition, routines such as **SameLdType** which attempt to compare two interp lists must take care not only to use **Match** but also to insure that at least one of the interp lists being compared corresponds to some other construct and therefore contains all possible interpretations. This approach would also fail if any of the above constructs could be used as a prefix in indexed components or selected components.

Since the semantic analysis of aggregates is based almost entirely upon the type of the aggregate, it is impossible to perform this analysis until overload resolution is complete. Therefore, when **CleanUp** is called to install this information in the node, it reinvokes the semantic analyzer to complete the processing of the node, which may include semantic processing and overload resolution for the expressions which comprise the aggregate.

8. Conclusion

This paper has attempted to give full details of one particular implementation of Ada overload resolution. The top level algorithm is fairly simple, and comparable schemes have been presented in several other sources. However, the handling of smaller details within the overall scheme (such as the handling of aggregates or array references vs. function calls) have been different in these other sources, or have not been described at all. It is hoped that the approaches described here may prove useful to anyone who might attempt to implement such a system in the future.

It is very difficult to discuss the effectiveness of this algorithm. In general it appears to be fairly efficient, as the amount of work for any given node is dependent only upon the number of possible interpretations for the node and the number of **valid** interpretations in the immediate sons of the node. It is not dependent upon any other nodes beneath these or upon the other components of the expression. It gains further efficiency by performing "lazy evaluation" for some expensive constructs such as aggregates. In addition, it deals fairly neatly with various special cases by encapsulating most of the special handling in a few commonly called routines. Although the routine calls which constitute the overload resolution process are intermixed with other arbitrary semantic processing, the system as a whole has proven remarkably easy to maintain and extend. Most of this may be attributed to the fact that all of the special processing for any given construct is encapsulated within a single routine call, and the description of the resulting interp list is independent of the type of construct.

References

[ARM 83] *American National Standard Reference Manual for the Ada Programming Language*
ANSI/MIL-STD-1815A edition, 1983.

[Baker 82] Baker, T.P.
A One Pass Algorithm for Overload Resolution in Ada.
ACM Transactions on Programming Languages and Systems 4(4):601-14, Oct 1982.

[Barbacci 85] M.R. Barbacci, W.H. Maddox, T.D. Newton, R.G. Stockton.
The Ada+ Front End and Code Generator.
In *Proceedings of the 1985 International Ada Conference: Ada in Use.* Paris, France, May 1985.