NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making
of photocopies or other reproductions of copyrighted material. Any copying of this
document without permission of its author may be prohibited by law.

CMU-CS-86-125

An Implementation of Ada Generics

Thomas D. Newton

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

9 May 1986

Abstract

This paper describes the technique used for implementing generics in the Ada + compiler. It involves performing semantic analysis on generic units, producing code for instantiations by generic expansion, and preserving the results of semantic analysis on a template in its copies.

Copyright © 1986 Thomas D. Newton

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

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

List of Figures

1. Introduction

One of the more interesting features of the Ada[1] programming language is the capability to define *generic* subprograms and packages which can be parameterized by types and subprograms as well as by objects. By allowing the reuse of code, generic units can save programming time and increase reliability. However, while generic units are a nice tool from a programmer's point of view, they pose an added burden for a compiler both in terms of semantic analysis and in terms of code generation. This paper is an attempt to describe how the Ada+ compiler deals with the problems posed by generic units.

2. Background

Generic units are one of the four types of program units defined in the Ada language (the others are subprograms, packages, and tasks). Like packages, generic units always have specifications and usually have bodies. But while a subprogram or package provides resources which can be used immediately, a generic unit cannot be directly used except as a template to produce normal subprograms or packages. If all of the subprograms or packages thus created were identical, generic units would not be very useful; their power lies in their ability to be parameterized by objects, types, and subprograms, which can be put to good use in applications such as writing a sorting subroutine that can be used with a wide range of key and element types.

Although generic units are like macros in the sense that both define templates for later use, they differ from macros in several important respects. One difference is that generic units must be checked for errors even if they are never used, whereas macros typically are not checked until after they are expanded. Another difference is that nonlocal references in an instance of a generic unit are bound in the context of the original declaration, rather than in the context of the instantiation. In general, the semantics of instances are defined in terms of the semantics of templates, and it is sometimes possible to create subprograms and packages using generic instantiation that cannot be declared directly. Such is the case in Figure 2-1, where type aliasing is used to produce a package that contains two subprograms with the same name and parameter/result type profile; normally, such declarations are illegal since there is no way to distinguish between them. And the bodies of generic units, unlike the bodies of macros, may not always be available when a compiler needs to process their instantiations. All of these differences combine to make the implementation of Ada generic units more challenging than the implementation of simple text macros.

---

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

Figure 2-1: One Way in which Generics Differ from Macros

```
generic
    type T1 is private;
    type T2 is private;
package OVERLOAD is
    procedure SAVE(X: T1);
    procedure SAVE(X: T2);      -- legal; T2 is different from T1
    procedure S(X: T2) renames SAVE;
end OVERLOAD;

-- The following instantiation is legal, even though it will
-- produce two subprograms with the same name and profile in
-- one declarative region.  OK.SAVE is permanently ambiguous,
-- but OK.S refers to the second SAVE procedure (its binding
-- is in terms of the generic declaration with substitutions).
package OK is new OVERLOAD(INTEGER, INTEGER);

package NOTALLOWED is
    procedure SAVE(X: INTEGER);
    procedure SAVE(X: INTEGER); -- illegal
end NOTALLOWED;
```

Before discussing how the Ada+ compiler handles generics, it is appropriate to take a brief look at the overall organization of the compiler. The Ada+ compiler processes programs in four major phases, which are split between three programs called the Front End, the Middle End, and the Back End. The Front End performs syntactic and semantic analysis; it is the only one of the three programs that can handle raw text, and it produces compilation database files for its own use and for the use of the Middle and Back Ends. The Middle End performs post-semantic analysis; currently this is all related to generic instantiations, as described later in this report. Finally, the Back End performs code generation, taking the database files produced by the Front and Middle Ends and producing object files which can be linked and run on a PERQ workstation under the Accent operating system.

Generics processing is likewise split between the Front, Middle, and Back Ends. The Front End contains most of the code to handle generic units (the code for processing generic templates, producing instance specifications, and detecting contract violations and circular instantiations). The Middle End contains code for producing instance bodies; since the actions involved in copying specifications and bodies are similar, it shares much code with the Front End. The Back End contains little code specific to generics, since by the time it gains control, the Front and Middle Ends have performed most of the work. Following this section, the rest of this paper will be organized along the lines of the tasks identified above: processing generic templates, producing instance specifications, detecting contract violations and circular instantiations, and producing instance bodies.

While the techniques used to handle generic units in the Ada+ compiler can be adapted for use in other compilers which save the results of semantic analysis in compilation databases, the actual implementation depends upon properties of the data structures the compiler uses to represent programs. Three of the most important of these data structures are described below; other data structures will be introduced as necessary.

- *Nodes.* The Ada+ compiler represents programs internally as syntax trees which are attributed during semantic analysis with pointers to various other data structures. Syntax trees consist of one or more records known as nodes. Each node has an associated production (such as "CaseStatement" or "Name"), space to store several attributes (including a pointer to a *symbol block* and a pointer to a *type block*), and pointers to immediately-surrounding nodes in the tree.

- *Symbol blocks.* Symbol blocks are used primarily to represent entities that have names, such as types, variables, labels, and record components. Each symbol block has a class ("Variable", "Type", etc.) and contains a pointer to a name, a pointer to a *type block*, and various pieces of information. Symbol blocks also contain three pointers which are used to organize the symbol blocks belonging to each context into trinary trees.

- *Type blocks.* Type blocks are used to represent both standard Ada types and entities such as subprograms, packages, and tasks which have associated declarative contexts. Each type block has a class ("Integer", "Record", "Subprogram", etc.) and may contain a pointer to the symbol table used to represent the type's declarative context, a pointer to a symbol block which represents the type's declaration, and various other pieces of information.

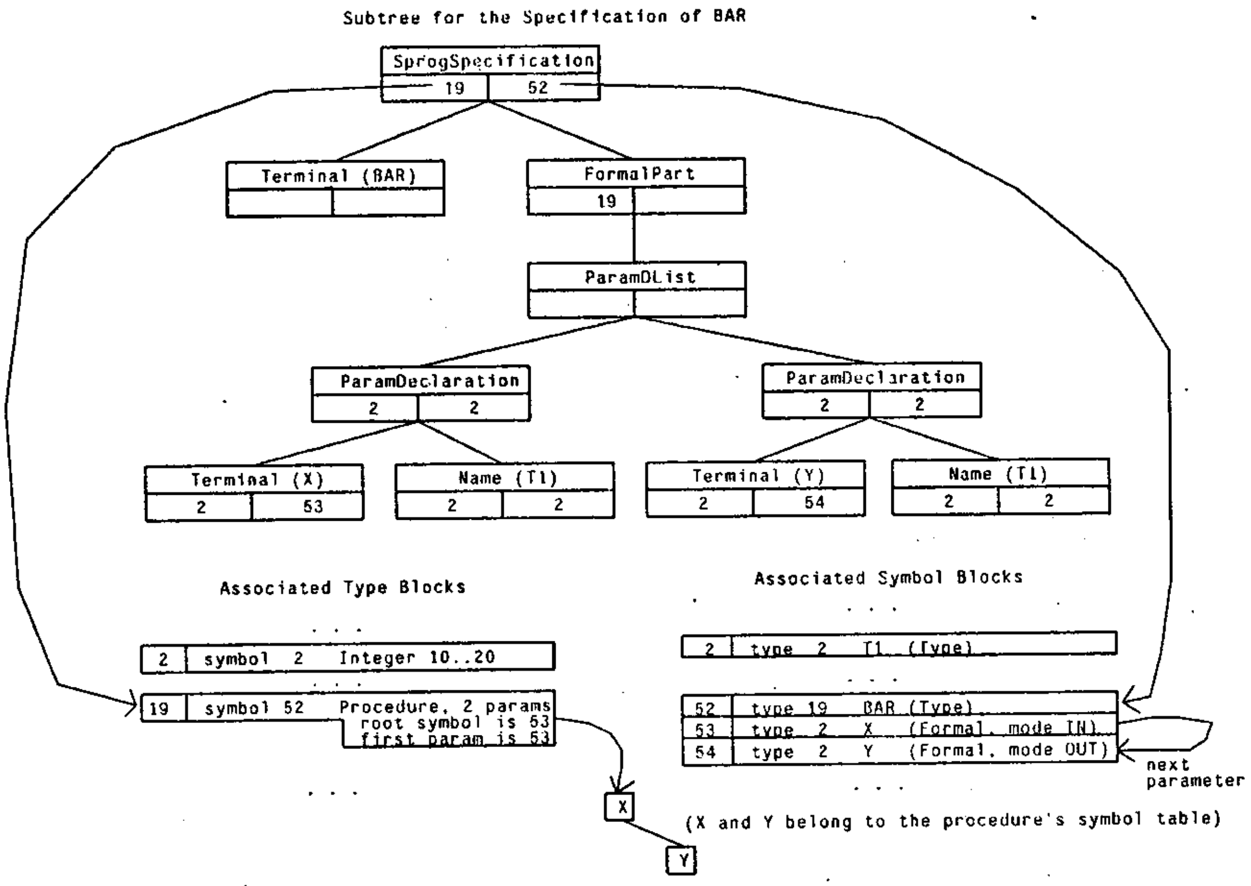
An example of how nodes, symbol blocks, and type blocks are used can be found in Figure 2-2, which shows a small program and the piece of its compilation database that corresponds to the specification of procedure BAR. Nodes are drawn as boxes which contain the name of a production (upper half), a pointer to a type block in the form of an index (lower left), and a pointer to a symbol block, also in the form of an index (lower right). Type blocks and symbol blocks are drawn as boxes containing various bits of information, most notably their indices, their classes, and pointers to associated blocks. Note that the specification of BAR is represented by both a symbol block and a type block, since it has both a name (BAR) and an associated context (the one in which its formals are declared). The database for the specification also includes two symbol blocks which represent the formal parameters; they form a symbol table that hangs from BAR's type block, and are also linked in first-to-last order (as represented by the "first param is 53" in type block 19 and the "next parameter" pointer). Although the symbol and type blocks representing type T1 do not belong to BAR's piece of the database, they are shown because BAR contains references to that type.

More information about the compiler and the data structures it uses can be found in [Barbacci 85].

Figure 2-2: A Subprogram Specification Database

```
procedure FOO1 is
    type T1 is range 10..20;
    A, B: INTEGER;

    procedure BAR(X : T1; Y : OUT T1);
                .   .   .
end FOO1;
```



3. Processing Generic Templates

Since the specification of a generic unit must appear before its body and any instantiations, the first step the compiler takes when processing any given generic unit is to perform semantic analysis upon its specification. This is done for two reasons: (a) to ensure that errors in generic units are found by the compiler whether those units are instantiated or not, as required by the LRM, and (b) to attribute the trees that represent the specifications of generic units with pointers to symbol blocks and type blocks that bind all references, thus making it possible to keep the bindings of non-local names when performing instantiations. For much the same reasons, the compiler performs semantic analysis upon the bodies of generic units; as will be seen later, this task is complicated somewhat because Ada allows generic units to be instantiated before their bodies have been written.

The approach the Ada+ compiler takes to performing semantic analysis on generic units is to treat them as normal subprograms and packages wrapped in "shells" that contain the declarations of their formal parameters and that control how their names are viewed. One motivation for this approach was that the name of a generic unit means different things depending upon whether it is used inside the unit (in which case it refers to a package or subprogram) or outside the unit (in which case it refers to a template). Another motivation for this approach was to simplify the job of copying the "non-generic" part of a generic unit's specification during an instantiation; since generic formal parameters are placed in physically (but not logically) separate symbol tables, it is easy for the compiler to avoid copying them.

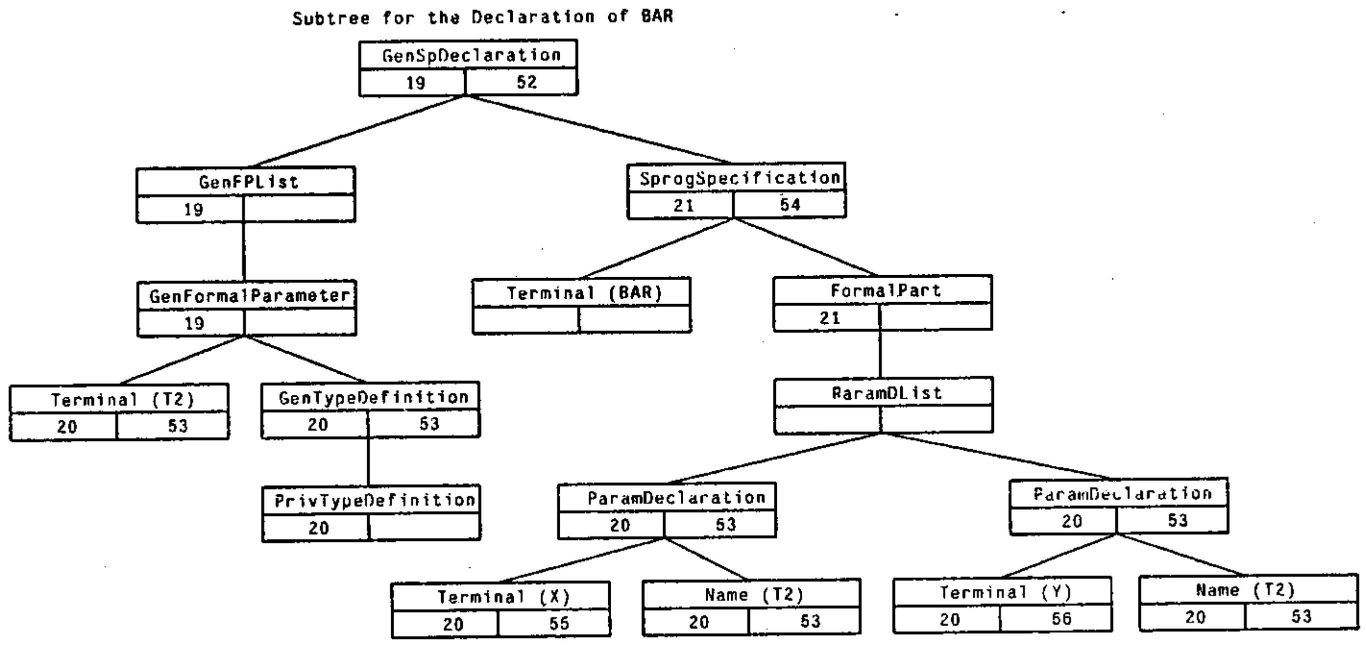
The "shells" mentioned in the previous paragraph take the form of a symbol block of class "Generic" and a type block of class "Generic" for each generic specification or body, in addition to the symbol block/type block pair allocated for a normal subprogram/package specification or body. Symbol blocks of class "Generic" replace symbol blocks of class "Type" for the purpose of entering a generic unit's name into the current symbol table. They receive special treatment from the symbol table lookup routines, which upon finding a "Generic" symbol, return it or the associated "Type" symbol, depending on whether the name currently denotes a template or a normal subprogram/package. Type blocks of class "Generic" hold pointers to the roots of symbol tables which represent generic formal parameters; they are also treated in a special fashion by the symbol table routines, which treat the symbol tables hanging from a generic unit's "Generic" and "Subprogram"/"Package" type blocks as if they define one larger symbol table.

Generic formal parameters are represented by symbol and type blocks much like those used to represent ordinary variables, types, and subprograms. To allow the semantic analyzer to distinguish between generic formals (and subtypes of them or types derived from them) and ordinary entities in the few cases where they must be treated differently, type blocks and some kinds of symbol blocks contain a flag indicating whether or not they are (or are related to) generic formal parameters. During semantic analysis of a generic specification or body, the compiler collects information on the ways in which generic formal private types are used; it uses this information when checking for contract violations, as detailed in section 5.

Figure 3-1: A Generic Subprogram Specification Database

```
procedure F002 is
    type T1 is range 10..20;
    A, B: INTEGER;

    generic
        type T2 is private;
    procedure BAR(X : T2; Y : OUT T2);
         .   .   .
end F002;
```



Associated Type Blocks

19	symbol 52	Generic, root is 53
20	symbol 53	Unknown, private
21	symbol 54	Procedure, 2 params root symbol is 55 first param is 55

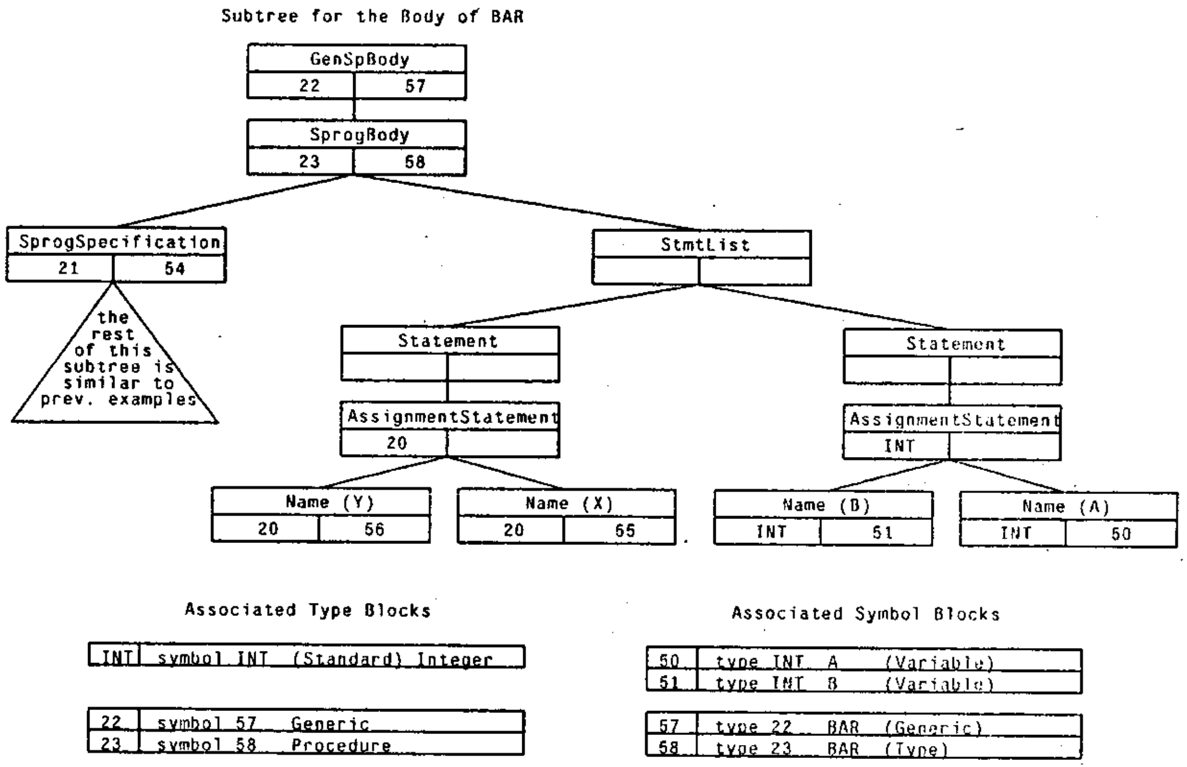
Associated Symbol Blocks

52	type 19	BAR	(Generic)
53	type 20	T2	(Type)
54	type 21	BAR	(Type)
55	type 20	X	(formal, mode IN)
56	type 20	Y	(formal, mode OUT)

Two examples of the databases used to represent generic specifications and bodies can be found in Figures 3-1 and 3-2. Here we have turned the subprogram BAR introduced in Figure 2-2 into a generic subprogram and given it a body. Note that the databases in Figures 2-2 and 3-1 are very similar; while the database for the generic specification has some extra nodes, symbol blocks, and type blocks, the subtrees marked "SprogSpecification" and their associated symbol blocks and type blocks are almost identical. They differ in that the second subtree refers to type T2 instead of type T1

Figure 3-2: A Generic Subprogram Body Database

```
procedure FOO2 is
          .   .   .
    procedure BAR(X : T2; Y: OUT T2) is
    begin
        Y := X;
        B := A;
    end BAR;
          .   .   .
end FOO2;
```



(reflecting a difference in the source) and the symbol blocks and type blocks associated with the two subtrees have slightly different indices (a semantically meaningless difference).

Although it is not shown in the figures, the compiler records information for each generic specification or body which identifies the symbol and type blocks that belong to it and that must be copied when it is instantiated. Unlike nodes, which belong to subtrees, symbol blocks and type blocks are only loosely organized; a few are not even associated with symbol tables. Fortunately, it is not necessary to record the identity of each symbol or type block belonging to a generic unit, due to a useful side effect of the code the compiler uses to read and write compilation databases. This code assigns indices to each symbol and type block, and keeps tables that can be used to find any block given its index. Because the semantic analyzer walks parse trees in a mostly top-down and left-to-right order, the symbol and type blocks representing a generic unit can be identified by ranges of indices. To allow for cases in which one generic unit is nested inside another, and instances of an outer generic unit contain copies of "inner" generic units, ranges are stored in relative form as offsets from known pointers; this works because the compiler fixes pointers when producing instance databases and because ranges of symbol and type blocks are copied in such a way that the copies also have adjacent indices.

4. Producing Instance Specifications

Another action that the compiler must take during semantic analysis is to check instantiations for errors and to produce internal representations of instance specifications to make their public declarations available for use. Since instances of generic bodies cannot, by definition, contain public declarations, the compiler does not need to instantiate generic bodies when performing semantic analysis. This is fortunate because the body of a generic unit may not be available at the time that the unit is instantiated. Despite this, the semantic analyzer cannot completely ignore generic bodies when performing instantiations; they can play a role in causing errors known as contract violations, which are discussed in section 5.

The approach the Ada+ compiler takes to making instance specifications available is to produce subprogram and package specification databases by copying parts of the corresponding generic specification databases and making appropriate substitutions. This method is one instance of a technique which is called *generic expansion* when used to produce code for instantiations. As a code generation technique, generic expansion has both advantages and disadvantages. But regardless of the method a compiler uses to produce code for instantiations, generic expansion is a useful technique for producing instance specifications for the purposes of semantic analysis. It permits the parts of a semantic analyzer not directly connected with generics to ignore them for almost all intents and purposes; this separation of function is especially useful since Ada's semantics would be complex even if the language did not have generics.

The first steps the compiler takes when processing an instantiation are to check that the name used in the instantiation denotes a generic unit of the appropriate type and that the actual parameters match the formal parameters. Ada allows generic formal parameters to denote values, variables, scalar types, array types, access types, unknown types, and subprograms, all of which have their own matching rules; the compiler contains several matching routines which correspond to the various classes of formal parameters. Two of the matching routines (the ones that handle IN OUT objects and subprograms) use code taken with minor changes from code the compiler uses to handle renaming declarations, as Ada's rules for renaming variables and subprograms are very similar to its rules for the formals just mentioned. Most of the other routines are fairly straightforward; the one which handles generic formal private types records information about actual types for use in detecting contract violations, but is otherwise uninteresting.

During the matching process, the compiler sets "forward" fields in the symbol and type blocks representing the generic formals to point to the symbol and type blocks which represent the corresponding actuals; it uses these pointers later when replacing references to formals in the generic specification with references to actuals in the instance specification. Some actual parameters normally are not represented by symbol blocks, such as expressions (actuals for objects of mode IN), record or array components (actuals for objects of mode IN OUT), and attributes (actuals for subprograms). When the compiler encounters one of these actual parameters, it creates a symbol block to represent it; this permits "forward" pointers to be used for all pairs of formals and actuals.

Once the compiler has matched the formal and actual parameters, it produces a subprogram or package specification database for the instance. The copying process, illustrated in Figure 4-1 for a part of the database for BAR's specification (Figure 3-1), involves the following steps:

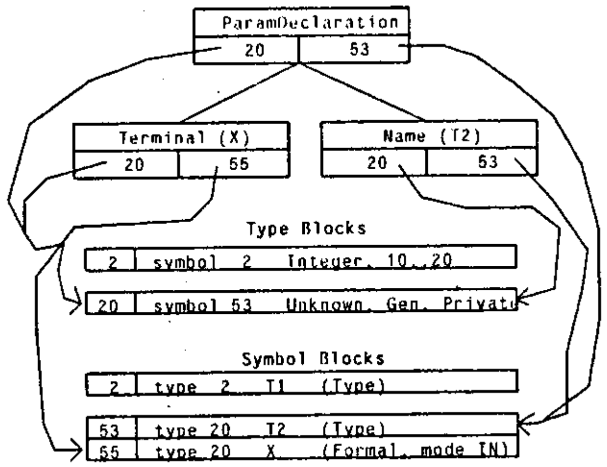
1. Copies are made of all the nodes, symbol blocks, and type blocks belonging to the part of the generic specification's database which corresponds to a normal subprogram or package specification. At the same time, forward pointers in the original nodes, symbol blocks, and type blocks are set to point to their copies (Figure 4-1(b)). At this point, the pointers in the new nodes, symbol blocks, and type blocks are for the most part identical to those in the originals (the exceptions are forward pointers, which are NIL in the copies, and links between the nodes of the new tree, set correctly during the copying process), so many are wrong (Figure 4-1(c)).

2. All pointers in the new nodes, symbol blocks, and type blocks are fixed using the following rule: any field which points to a node, symbol block, or type block whose forward pointer is not NIL is assigned the value of that forward pointer. Since forward pointers are used to map formals to their actuals as well as to map the original specification onto its copy, this has the effect upon the instance specification of fixing cross-references and replacing references to formals with references to actuals. Pointers to pieces of the database that do not correspond to the generic specification are not affected by this process; this is quite useful, since it is precisely these pointers that represent the bindings of nonlocal references. At this point, the instance database is very similar to the database for a normal subprogram or package. One difference is that identifiers in the instance subtree corresponding to generic formals may be wrong; this does not matter, since the compiler uses pointers to symbol and type blocks in preference to identifiers once semantic analysis has been done. Another is that some

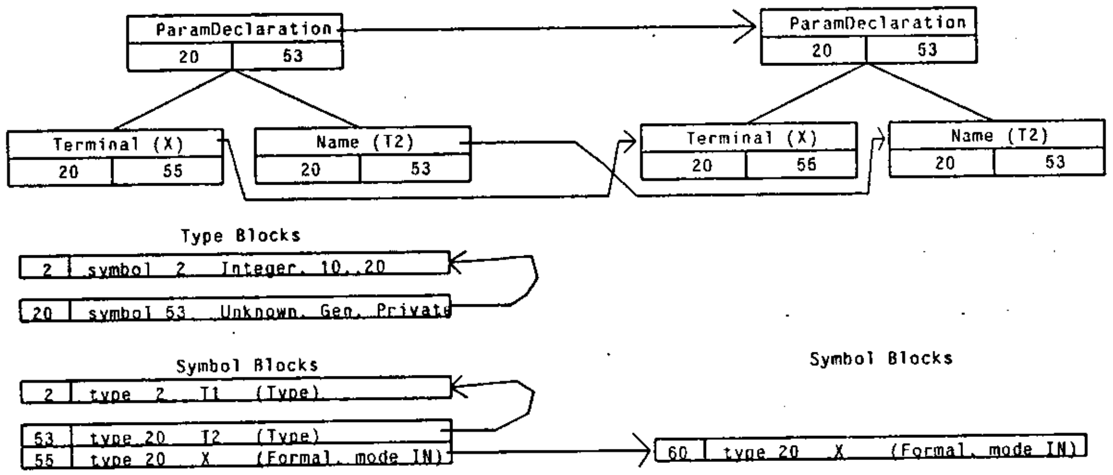
Figure 4-1: Snapshots of a Generic Instantiation

```
procedure FOO2 is
          .   .   .
    procedure ORIG_BAR is new BAR(T1);
          .   .   .
end FOO2;
```

(a) Database for Part of the Declaration of the Generic Unit BAR



(b) Forward Pointers Between Generic/Formals and Instance/Actuals

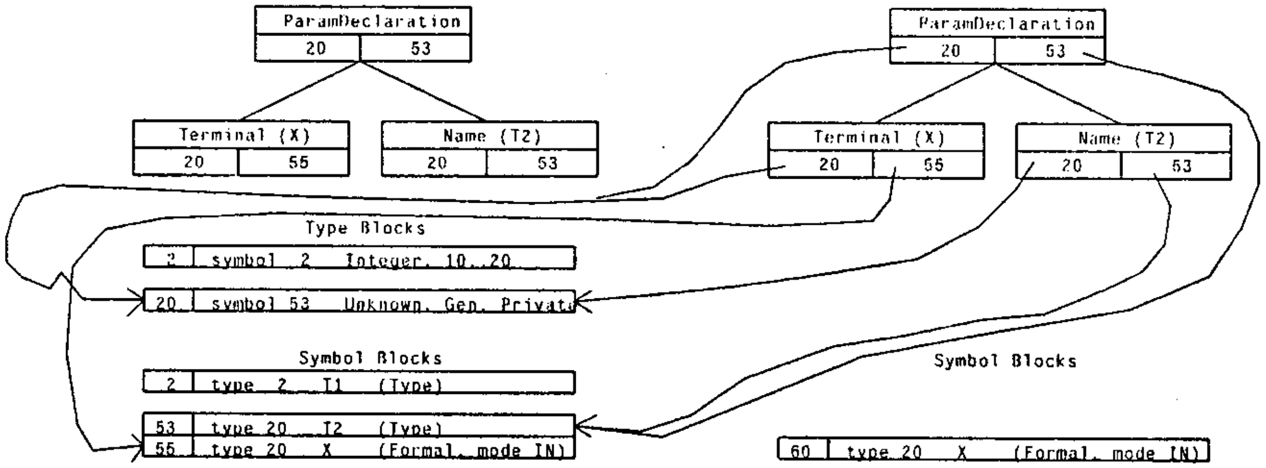


type blocks may need patching, as explained below. The results of fixing the pointers in Figure 4-1(c) are illustrated in Figure 4-1(d).

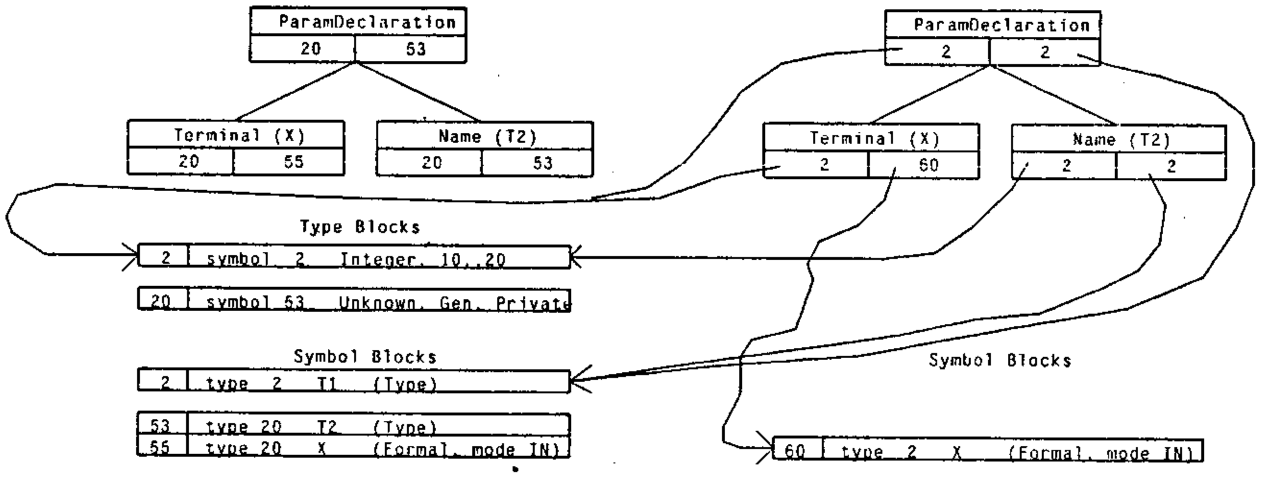
3. Type blocks in the instance database which correspond to subtypes of (or types derived from) the generic formal types in the original specification are patched to include any

Figure 4-1, Continued

(c) Unfixed Pointers in a Part of the Database for BAR's Instantiation


ParamDeclaration
20 53
Terminal (X)
20 55
Name (T2)
20 53
Type Blocks
2 symbol 2 Integer, 10..20
20 symbol 53 Unknown, Gen. Private
Symbol Blocks
2 type 2 T1 (Type)
53 type 20 T2 (Type)
55 type 20 X (Formal, mode IN)
60 type 20 X (Formal, mode IN)

(d) Fixed Database Produced by Following Forward Pointers Set Earlier



extra information associated with the type blocks for the generic actual types. This is not illustrated in Figure 4-1, since BAR does not declare any (sub)types that are related to generic formals.

4. Finally, the instance database is merged into the main database, information about the instantiation is gathered for use in detecting contract violations and circular instantiations, and all of the forward pointers previously set are cleared so they will not interfere with other instantiations.

5. Detecting Contract Violations and Circular Instantiations

So far, we have discussed how the compiler performs semantic analysis on generic units and produces instance specifications without going into detail about how it detects two classes of instantiation-related errors: *contract violations* and *circular instantiations*. Contract violations are caused by some combinations of generic formal private types, actual types, specifications, and bodies; they can happen because Ada allows the name of a generic formal private type to be used where the name of a constrained type is required, and allows an unconstrained type to be the actual for a generic formal private type, but does not allow combining the two. Circular instantiations are just what the name implies: instantiations where a generic unit tries to instantiate itself either directly or indirectly.

Detecting contract violations poses a problem because it is not sufficient merely to check each instantiation at the time that it is compiled. Ada allows generic units to be instantiated before their bodies have been written, and since the body of a generic unit might be the only place where the name of a generic formal private type is used as an unconstrained type mark, compiling the body of a generic unit may make some previously-compiled instantiations illegal. This does not necessarily mean that all of the affected compilation units must be removed from the program library, since it is sufficient merely to reject the unit which was being compiled when an error was detected, but it is still necessary to detect the error in the first place.

To detect contract violations, the Ada+ compiler gathers and keeps information about generic specifications, bodies, and instantiations. When it processes an instantiation, it checks to see whether the actuals for that instantiation would cause a contract violation given what it knows about the specification and body of the generic unit. When it processes a generic body, it checks all records of previous instantiations of the generic unit to see whether or not adding the generic body will cause a contract violation. To minimize the amount of updating that needs to be done as the result of recompiling any unit or set of units, the information is kept on a per-compilation-unit basis with the intent that the compiler should combine information for different units in memory as needed.

Circular instantiations are easier to detect. The compiler maintains lists of which generic units instantiate which others, and after the Front End finishes the bulk of semantic analysis and before it writes the compilation database to disk, it checks for circular instantiations. This consists of building a matrix where the indices correspond to generic units and A[X,Y] = 1 means "X instantiates Y," taking its transitive closure, and checking to see if any values along the A[X,X] diagonal are not equal to zero (which would indicate a circularity). The Front End can perform this check at the end of the compilation without danger of infinite recursion because (a) users cannot build direct circularities --

the name of a generic unit denotes a subprogram or package within that unit, (b) users cannot build indirect circularities involving only generic specifications -- at least one of the names in any "circle" would be undefined or refer to a unit other than the one intended, and (c) programs which contain indirect circularities involving generic bodies cannot cause the Front End to infinitely recurse -- it does not copy generic bodies.

The information used to detect contract violations and circular instantiations is stored as lists of *template records* and *instance records* associated with each compilation unit. Template records describe generic specifications and bodies; each contains a pointer to a list describing how generic formal private types are used in its generic specification or body and a pointer to another list used for detecting circular instantiations. Each entry in the former list contains flags which indicate, when true, that the name of the corresponding formal private type was used as an unconstrained type mark in a derived type definition which was the full declaration of a normal private type, in an object declaration, in a record or array component declaration, or in an allocator. Each entry also lists the formal private types for which the type it describes was used as an actual. Instance records describe instantiations; each contains the unique identifier of the instantiated unit and a pointer to a list describing all the actuals corresponding to generic formal private types. Actual types which are not themselves formal private types are described by a simple "Constrained" flag; actual types which are generic formals are described using the identifier for their templates, to allow the compiler to do the recursive checking necessary in such cases.

Unfortunately, the compiler currently does not detect contract violations or circular instantiations that involve more than one compilation unit, due to inadequate support for keeping track of relationships between units in the program library. In particular, given a compilation unit, the routines which keep track of the program library currently provide no easy way to determine its importers (or the importers of its specification, in the case of a body). The compiler only has access to template and instance records for the current compilation unit, and thus cannot detect cross-compilation-unit errors. But assuming that library support was available, the routines described above for detecting contract violations and circular instantiations would be able to use the extra information to detect errors across compilation boundaries.

6. Producing Instance Bodies

In addition to performing semantic analysis on generic units and their instantiations, an Ada compiler must produce code for those instantiations. Several methods of producing code exist, ranging from producing a separate body for each instantiation to sharing one body between all instantiations; they vary in such things as the amount of knowledge about generic units that a code generator must have and the efficiency of the object code produced. The Ada+ compiler produces code for instantiations by generic expansion; it copies generic bodies to produce instance bodies, which are then fed to the code generator and treated like normal subprogram and package bodies.

Copying of generic bodies is performed by the Middle End, rather than by the Front End. An important difference between the Front End and the Middle End is that the Middle End depends upon the availability of generic bodies to perform instantiations, whereas the Front End does not. Ada does not make many provisions for dependencies of instantiations upon bodies, and the dependency relationships imposed by the Middle End do not fit Ada's order-of-compilation rules. On the other hand, Ada has a very vague definition of compilation: the result of a successful compilation is to enter a unit into a program library, and any unit which contains syntactic or semantic errors must be rejected by a valid compiler. It can be argued that the Front End performs the necessary functions to meet this definition of compilation, and that running the Middle and Back Ends is not a compile-time process, but rather a link-time process which can be performed at compile time in many common cases.

The Middle End was created shortly after the compiler was split into a Front End which performed syntactic and semantic analysis and a Back End which performed code generation. An important motivation for creating it was that at the time, the compiler was copying the bodies of generics as well as their specifications during semantic analysis, in violation of Ada's rules concerning order-of-compilation dependencies. However, this problem could have been solved by having the Back End perform the copying. The motivation for making the post-semantic phase into a separate program was to allow individual Back Ends to choose to produce code for instantiations by some other method if so desired, while providing a generic expansion service that any Back End could utilize to support generics with minimal effort.

While the copying process performed by the Middle End is almost identical to the process performed by the Front End, there are a few differences:

- The Middle End places instance body databases in database files separate from those produced by the Front End, in keeping with the idea that the services of the Middle End should be optional.

- Before copying a generic body, the Middle End sets up forward pointers between the generic formals and the generic actuals and between the generic specification and the instance specification, since the generic body may (and probably will) have references to the generic formals and/or entities in the specification.

- The Middle End may sometimes instantiate bodies in a different order than the Front End instantiates specifications. In particular, the Middle End does not try to complete any instantiations that are located inside generic templates, since it would have problems recording the existence of the extra nodes, symbol blocks, and type blocks associated with them. Instead, after the Middle End instantiates a body, it walks over the parse tree for the instance body looking for any uncompleted instantiations in its nongeneric parts.

The final steps needed to produce code for instantiations are taken by the Back End. When processing an instantiation, the Back End first generates some code to handle the correspondence between generic formals and actuals (such as code to freeze expressions by evaluating them into temporaries). It then generates code for the instance specification and body produced by the Front and Middle Ends as if it was generating code for a normal package or subprogram.

7. Conclusion

This paper has described the techniques used to implement generics in the Ada+ compiler. These techniques include performing generic expansion on attributed syntax trees (so that instance databases will inherit the results of semantic analysis from template databases) and keeping lists to enable the detection of contract violations and circular instantiations. While the ideas behind these techniques are fairly well known, it is hoped that the particular methods used in our compiler may be useful to others who attempt to implement Ada generics.

To a large degree, the choice of generic expansion was driven by the goal of building a generics facility for the Ada+ compiler that would not require much support from the initial code generator, which was intended primarily for use in bootstrapping a second code generator to be written in Ada. One advantage of generic expansion is that it allows such factoring. Another advantage is that the code produced for each instantiation is as good as the code produced for normal subprograms and packages. However, it does have the disadvantage of producing more code than alternate techniques when a generic unit has many instantiations. Two techniques that we might consider if we were to build a more intelligent code generator are described by Rosenberg ([Rosenberg 83]) and by Bray ([Bray 83]).

Rosenberg's thesis addresses the issue of sharing code among instantiations of a generic subprogram; it describes a technique known as *polymorphic routine translation* which involves producing one body that handles all instantiations of a generic subprogram. The technique depends rather heavily upon the use of inline substitution for optimization, and it is apparent that it requires much more support from a code generator than generic expansion does. Some work remains to be done before the method can be applied to the translation of Ada generics; generic packages and constraint checking are mentioned briefly under "directions for future research". But it appears to be a potentially useful technique for compilers with optimizing code generators which can fall back upon some other method when desirable or necessary. Generic expansion seems like a reasonable choice for the backup method, especially since it produces code that is smaller as well as faster for certain types of programs.

Bray describes an approach to handling generic instantiations, which he calls the *class approach*, that falls between the extremes of generic expansion and polymorphic routine translation. Basically, the idea is to generate one body for each group of instantiations that have actual parameters with similar runtime representations. This approach would seem to require more support from a code generator than generic expansion (it must be possible to generate shared bodies) but less than polymorphic routine translation (more is known about the possible users of any given shared body, so extensive optimizations might not be necessary to produce reasonably efficient code). Since the code that it produces is slower than the code produced by generic expansion, a compiler that uses it might also offer generic expansion for cases where speed is important.

Finally, while writing a second code generator would require much work, the current implementation could be modified to recognize special cases which make sharing easy. One such case occurs when a generic subprogram has only object and subprogram parameters; it should be possible to share a single body between all instantiations of such a generic by passing in the generic actuals as implicit parameters. Similar cases exist for generic packages which consist entirely of subprograms. However, since the real power of generic units is in the ability to use type parameters, which pose the real obstacles to code sharing, extending the compiler to handle such special cases probably would not affect the amount of code it produced for most programs.

References

[ARM 83]	*American National Standard Reference Manual for the Ada Programming Language* ANSI/MIL-STD-1815A edition, 1983.
[Barbacci 85]	Barbacci, M. R., Maddox, W. H., Newton, T. D., and Stockton, R. G. The Ada+ Front End and Code Generator. In *Proceedings of the Ada International Conference: Ada in Use.* ACM, Paris, France, May 1985.
[Bray 83]	Bray, Gary. Implementation Implications of Ada Generics. *Ada Letters* III(2):62-71, 1983.
[Rosenberg 83]	Rosenberg, Jonathan. *Generating Compact Code for Generic Subprograms.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, August, 1983.