

Implementation Implications of Ada Generics

Gary Bray

Intermetrics, Inc.

ABSTRACT

Generic program units as defined in Ada pose several important design issues for compilers, both in semantic analysis and in runtime implementation. Chief among these issues are separate compilation of generic bodies and the sharing of code among several instantiations of a generic. An implementation is described that allows separate compilation of generic bodies with full semantic checking and that automatically shares instance bodies based on the characteristics of the actual parameters. A single instance body is generated for each "instance class". Instance classes are formed by actual parameters with the same representation attributes.

1. Introduction

Generic packages and subprograms are major abstraction capabilities of Ada*. In addition to parameterization by objects (similar to ordinary subprogram parameterization), generics allow parameterization of packages and subprograms (henceforth called program units) by types and by subprograms and task entries. This capability is useful for algorithms and abstract objects that are applicable to a variety of types and associated operations.

The power of this capability, shown by languages [ELI, CLU, Euclid] that contain related abstraction features, suggests that Ada generics are likely to be used widely, if implementations are acceptably efficient. Common abstract data types, such as stacks, sets, and trees, whose operations are valid for many types, can be written as generic packages. For example, the operations of a stack discipline are meaningful independent of the type of the objects placed on the stack. Using generics, it is possible to construct libraries of proven abstractions which are available for instantiation. If a generic declaration and body have been demonstrated to be correct, instantiations of this generic also are known to be correct.

* Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office).

The work described by this paper was supported partially by Air Force contract number F30602-80-C-0291-SAP006.

This paper discusses several implementation implications of the definition of Ada generics and describes an implementation intended to promote user flexibility and runtime efficiency. The most significant issues to be resolved in a generic implementation are separate compilation of generic bodies and the sharing of code among different instances of the same generic.

The discussion is divided into three parts, corresponding to the major steps in compilation:

1. semantic analysis, which ensures conformance of generics usage with Ada semantics,
2. postsemantic analysis, which makes decisions concerning sharing of instance bodies, and
3. runtime representation, which is the set of runtime structures and operations needed to execute generic instances.

2. Semantic Analysis

Generics are declared in two parts:

1. a declaration, which defines the formal parameters and the program unit specification, and
2. a body (called the template body), which defines the template to be used in creating instances of the generic.

A particular instance of a generic program unit is created by generic instantiation, at which point the programmer supplies the actual parameters to be used to construct the instance. An instantiation creates a new program unit, with the actual parameters substituted for the formal parameters in the template. Logically, instantiation creates both a new instance specification and a new instance body, although for efficiency an implementation may choose to avoid creating a new body. The discussion assumes a program library which records the intermediate results of compilation, e.g., in the form of [DIANA].

Two facts are noteworthy about the definition of generics:

1. the generic declaration alone is sufficient to enable instantiations of the generic and
2. the resulting instance declaration alone is the only information needed by program units that use the instance.

To use generics, information about generic bodies and instance bodies is unnecessary in general. These observations emphasize the level of abstraction that should be preserved in generic implementations.

Like other program units, generic declarations and bodies may be compiled as separate compilation units. The language reference manual [Ada] allows an implementation to require that a generic declaration and its corresponding body appear in the same compilation. Although it can simplify an implementation (for reasons discussed below), this restriction is undesirable, because it is convenient to use generic specifications without prior completion of the body and without introducing compilation order dependence on the body.

During design, the abstraction allowed in Ada by separation of specifications and bodies is a powerful methodological advantage. For an implementation to require that generic specifications and bodies be submitted in the same compilation compromises this key abstraction facility and introduces an irregularity with regard to nongeneric units.

Also, an implementation that requires monolithic compilation of generics introduces unnecessary inefficiencies in compiling other program units. The body cannot be changed without also recompiling all instantiations, even if the generic specification is unchanged. Thus, a change in the generic body forces recompilation of all program units that use the generic. Assuming that generics are used frequently, this cost can be very expensive.

However, allowing separate compilation of generic bodies is more difficult than separate compilation of nongeneric units. Logically, generic instantiation creates a new program unit, both declaration and body. If the body is separately compiled, the body may be unavailable when compiling an instantiation, and therefore no instance body can be created at the point of instantiation. This instance body must be created at some later point when the template body is supplied. If monolithic compilation is required, this problem does not arise because the template body is supplied in the same compilation.

Separate compilation requires potentially deferred creation of instance bodies. To enable deferred creation, the program library records the instantiations of a given generic declaration. This instantiation list is associated with the generic declaration and is used to locate the instantiations which must have instance bodies associated with them. These instance bodies are created when the deferred body template is supplied.

This simple scheme is insufficient, however, because of irregularities in the definition of generics. The irregularities are a result of the possibility of violations of the "contract model". The contract model holds that the generic formal parameters define a "contract" between the generic and its instantiations agreeing that instantiations with matching actual parameters are legal, regardless of the characteristics of the generic specification and body.

The contract model may be violated for certain combinations of actual parameters, formal parameters, and properties of the generic specification and body. For example, an instantiation is illegal if the actual parameter is an unconstrained array type and the corresponding formal is a private type, and the declaration or body declares objects of the formal type. The attempted object declaration has no constraints for the unconstrained array type and therefore is illegal. Thus, the correctness of instantiations depends on the characteristics of the body template, as well as upon the specification.

Checking for contract model violations bears on separate compilation. If the template body is separately compiled, the correctness of certain instantiations can be checked only after the body is supplied. Thus, most semantic checking can be performed at the sites of instantiations, but semantic analysis for certain combinations of actual and formal parameters can be completed only when the body is available.

To enable this checking for separately compiled generic declarations and bodies, a list of "provisos" can be associated with formal parameters that can have actual parameters in violation of the contract model. Again, this list is contained within the program library, one for the declaration and one for the body. If the body has subunits, provisos also are associated with the subunit, because subunits similarly can invalidate instantiations.

A proviso names a property of the specification or body that can invalidate instantiations. Provisos are associated with generic program units during semantic analysis of the body. A proviso refers to a specific generic formal parameter and characterizes the use of that parameter within the program unit. In the above example, a proviso specifies that the template body declares objects of the formal private type. A proviso is defined for each kind of contract model violation possible in Ada. Because few kinds of violations have been identified, the number of provisos is small.

Thus, checking a generic instantiation involves checking both that the actual and formal parameters match and that the actuals violate none of the provisos associated with the generic. For all instantiations, the former check occurs at the instantiation site. For generics having the declaration, body, and instantiations in a single compilation, both checks occur during semantic analysis of the compilation.

For separately compiled generics, a delayed check is performed to complete semantic analysis. The delayed check is simple: the instantiation list discussed above is used to locate instantiation sites and, for each proviso on a formal parameter, the corresponding actual parameter is checked and invalid combinations are diagnosed illegal. This check may be performed by

the linker, by a separate generics checking tool, or by a configuration tool (which may also perform other useful checks concerning completeness and elaboration order). If the programming environment enables execution of programs with parameters from within other programs, the separate generics checker can be used both by the linker and by a configuration tool.

The combination of instantiation lists and provisos enables flexible separate compilation of Ada generics with full semantic checking. The cost of maintaining these lists within the program library and performing the associated checks is small in comparison with the added flexibility.

3. Postsemantic Analysis

Postsemantic analysis of generics is concerned with the generation of executable code for generic instantiations and the resulting instances. The central issue concerns generation of instance bodies. Because a single template body typically is used to create multiple instances, it is advantageous to share the executable code among the instances, reducing the overall program size. Because of differences in actual parameter representations, such sharing may be difficult.

The following subsections outline three approaches to generating instance bodies differing in the amount of code sharing, the efficiency of execution, and the complexity of postsemantic analysis.

3.1 Macro Approach

The macro approach creates a new instance body for each instantiation of a generic. Each instance body is a copy of the template body with the actual parameters substituted for the formal parameters. This approach is similar to macro expansion, because each instantiation results in a copy of the template body.

This approach is simple, because copying the template body adds only minor additional complexity above that necessary for copying the specification. This copying can be done after semantic analysis, so that, in later compiler phases, program units created by generic instantiation are indistinguishable from other units. This uniformity is a major simplification of later phases. Also because of this uniformity, the runtime efficiency of instances is the same as that for other program units.

These significant advantages are gained at the expense of disallowing code sharing. An instance body is generated for every instantiation, which takes no advantage of the similarity of the resulting instance bodies. This approach is considerably more storage inefficient than schemes that achieve sharing (and may discourage wider use of generics).

3.2 Canonical Approach

The canonical approach creates a single instance body for the template body, and this body is shared among all instances of the generic. To create a single body, a canonical representation is adopted for the formal parameters. Actual parameters that do not match this representation are converted both at the instantiation site and at each call and return from a subprogram within the resulting instance.

One method of canonical representation is to treat all generic parameters by reference. Unless all program objects are represented by reference (which is unacceptably inefficient for embedded applications), this representation introduces an undesirable inconsistency between generic and nongeneric program units. This section assumes a canonical approach that, for efficiency, does not treat all program objects by reference.

A canonical scheme has the advantage that instance code is always shared: a single copy of executable code is generated from the generic template. However, the number of conversions, and the potential complexity of them in the presence of representation specifications, are a significant expense, both in compiling and execution. In addition to the executable code to perform the conversions, the representation of formal parameters also is wasteful. Because the generated body must be general, the canonical representation is probably the most execution inefficient. Also, subprogram formal parameters must use generalized calling conventions, even though a hardware operation may exist for the actual parameter.

3.3 Class Approach

The class approach generates a single instance body (the "representative") for each "instance class". An instance class is defined to be a set of instances having actual parameters with similar properties. Properties are associated with the generic formal parameters of the representative and convey information concerning its runtime representation. An example of a property for a formal type might be "occupies a target word". (Properties are unrelated to the semantic provisos discussed in the previous section, although they are analogous in that both properties and provisos indicate conditions for using a given program unit. In the latter case, the condition is necessary for semantic correctness; in the former, the condition merely determines class membership.)

The total set of instance classes for a generic are determined from surveying its instantiations. The classes are built incrementally: for a given instantiation, if the properties of no instance class matches the properties of its actual parameters, a new instance class is created. Executable code is generated for the representative of this class based on the properties of the

current actual parameters.

This approach has the advantage that code sharing is achieved for instantiations within a class and is **not forced** artificially on ones that differ substantially. That is, the representative is shared among similar instantiations; instantiations that differ have their own representative. Thus, the representative is tailored to members of its class, resulting in storage and execution efficiency. The expense of this approach is the storage within the programming environment for the properties stored with representatives and the added checking for instance class membership for each instantiation. Checking for class membership can be delayed as late as linking or it may be performed by a configuration tool.

A mixture of the above approaches is possible. An implementation could treat the kinds of formal entities differently, e.g., it could represent formal subprograms according to the canonical approach and objects and types according to the class approach. Also, the choice of schemes could be governed by implementation-defined pragmas. For example, a pragma could be defined to indicate whether the user prefers code sharing or optimized execution.

4. Runtime Considerations

This section explores several runtime ramifications of the instance class approach to representing instances. The central issues are the representation of each instance and the executable code of the class representative. Each member of the instance class must have its own data, both specification data and body data; a representative must be able to refer to the data of the instance currently executing its code.

To enable data addressing, each instance has an "instance descriptor" allocated to it at runtime, located within the data context of the program unit creating the instance. The instance descriptor contains information describing the actual parameters of this instance and the location of its data area. The kind of information stored in the descriptor for an actual parameter varies with the kind of the associated formal.

Before going further, some observations about generic parameters are helpful. The runtime requirements for formal objects are similar to those for ordinary subprogram parameters of mode in and in out. For types, the basic and predefined operations are known from the formal parameter specification. However, the formal parameter specification does not indicate the subtype constraints of the actual type parameter. For subprograms, the requirements are similar to subprogram renaming.

These observations suggest the format of the instance descriptor. For formal objects, the descriptor stores either the constant itself for a mode in parameter or a reference to the object for a mode in out parameter. For formal types, the descriptor stores a subtype descriptor, giving the constraints applicable to the type. These constraints are used within the body of the representative for constraint checking. For subprograms, the descriptor stores the location of the subprogram and a representation (a static link or a display) of its static addressing environment. For entries passed to subprogram formal, the descriptor stores the location of the entry and the location of the associated task descriptor (or task control block).

Each instance must have an associated data area, which contains the data declared in the instance specification and body. The location of the instance data area may be stored within the instance descriptor. Alternatively, the instance data area may be located within the instance descriptor itself.

Another kind of data addressing must be supported for generics. Objects referred to within the generic, but not declared within the generic, are bound within the static environment of the generic declaration. The instance descriptor also contains the representation (again a static link or a display) for the static addressing environment, to make accessible objects global to the generic.

5. Summary

Decisions about separate compilation and code sharing for generics significantly influence the design of an Ada compiler. This section summarizes the foregoing design, which provides these capabilities, and discusses several advantages and disadvantages.

Separate compilation imposes the requirement to perform delayed semantic checking for combinations of actual parameters, formal parameters, and generic characteristics that violate the contract model. Provisos associated with the formal parameters enable this checking for separately compiled generic bodies. The program library can provide the storage and operations needed to perform this checking efficiently.

To allow code sharing of instance bodies of a single generic template, postsemantic analysis must determine the instances that can share a given instance body. The instances that can share the same body may be grouped together into an instance class. A single instance body is generated for each class, and this body is shared by all members of the class. The characteristics that determine class membership concern the runtime representation of the actual parameters.

For sharing, an instance body must have available at runtime a data structure containing the description of the actual parameters. An instance descriptor is a runtime structure that records the characteristics of a given instance. The instance body uses the instance descriptor to access instance-specific actual parameters.

This design has several disadvantages:

1. The program library (or programming environment) must store additional information to permit cross-compilation unit checking.
2. Additional compilation time is necessary to perform the delayed checking.
3. Execution with sharing can be expected to be somewhat slower than execution with the macro scheme.

The advantages are:

1. Program designers have added flexibility with separate compilation without sacrificing any semantic checking.
2. Sharing conserves storage space at runtime, as compared with the macro scheme.
3. Execution can be expected to be more efficient than execution with the canonical scheme.

This design shifts the costs of generics toward the programming support environment. That is, most of the costs are incurred during analysis within the programming support environment rather than within the execution environment. This advantage is especially important when the target computer is embedded, since embedded computers typically have scarce resources as compared with the programming support environment.

Efficient implementations of Ada generics are a key to widespread use of this important abstraction capability. Generics should be given emphasis in production implementations rather than being treated in a post hoc fashion. The power and flexibility inherent in generics may make them as important in future program designs as subprograms are in current designs.

6. References

[Ada] Reference Manual for the Ada Programming Language, Draft Proposed ANSI Standard Document for Editorial Review, U.S. Department of Defense, July 1982.

[CLU] Liskov, B., Snyder, A., Atkinson, R., "Abstraction Mechanisms in CLU", Communications of the ACM, 20, 8 (August 1977) 564-576.

[DIANA] Goos, G., Wulf, W., Butler, K., (eds.) Draft Revised Diana Reference Manual, Tartan Laboratories Inc., (June 1982).

[EL1] Wegbreit, B., "The Treatment of Data Types in EL1", Communications of the ACM, 17, 5 (May 1974), 251-264.

[Euclid] Lampson, B., Horning, J., London, R., Mitchell, J., Popek, G., "Report on the Programming Language Euclid", SIGPLAN Notices, 12, 2 (February 1977).