Date: February 9, 1997

To: X3J3

From: William B. Clodius

Subject: Parametric Procedures and Modules

Introduction

Title: Extended Parametric Polymorphism in Fortran

Basic Functionality: Allow Modules and Procedures to be Parameterized

Rationale: In recent years a variety of languages have provided means of parameterizing the equivalent of Fortran's procedures and modules. The templates of C++ (and the associated Standard Template Library) are perhaps the best known examples of such a parameterization capability, but other examples include Ada's generics, Eiffel's parameterized classes, and SML's functors. Such capabilities, to be termed parametric polymorphism, provide a substantial source of flexibility, while retaining static type checking and permitting a high degree of optimization. The current parameterized derived types proposal provides some of these parameterization capabilities, but the language would benefit by providing these capabilities in as broad an area as possible.

Parameterization in practice has been found to be a particularly useful complement to object oriented programming capabilities. Object oriented capabilities provide polymorphism based on the related structure of different types of objects. Parameterization provides polymorphism based on the external signature (interfaces) of different types of objects.

Usage: Parameterized procedures provide the most useful capabilities of macro substitution in a statically type checked form. Parameterized modules can be used to good effect to define implementations of collection "types", i.e., arrays, lists, stacks, trees, etc., a capability of importance to the high performance computing community. For example, a technique termed "Expression Templates", has been used to create C++ array classes with most of the performance and expressibility of Fortran 90's array capabilities. Another technique, termed "traits", allows C++ templates to be used to define class (derived type) characteristics similar in their flexibility to Fortran 90's intrinsics: DIGITS, EPSILON, HUGE, PRECISION, SELECT_INTEGER_KIND, etc.

Necessity: Because of their flexibility parameterization addresses two main needs of the Fortran programing community.

1. Safe "macro" substitution. One of the controversies involving the standardization of a conditional compilation facility for Fortran has been the lack of a macro facility in the overall preferred alternative, CoCo. This lack has been justified by

noting that macros in C have proved error prone and is strongly deprecated in the C++ community. However, macro usage is deprecated in the C++ community only because templates provide the capabilities of macros in a significantly safer form.

2. Collection types. While arrays have been, and will continue to be, the primary data structure of the Fortran user community, there is an increasing demand for more sophisticated data structures for special purpose applications, e.g., sparse arrays, lists of data, etc. While derived types and modules allow the construction of such structures, it is difficult to exploit the similarities of such structures in the current language. As a result there is an unnecessary amount of code duplication, and no set idioms to for optimizer to recognize and exploit. Fortran would benefit from something like C++'s Standard Template Library in addressing these needs.

Possible syntax: No syntax will be provided here, but examples will be provided by an accompanying paper.

Estimated Impact: There is no doubt that providing this capability would have a large impact on the language in almost any form in which it might be provided. If the language were to be as aggressive about exploiting this capability as has C++ with its Standard Template Library, the impact on the language would be extremely large.

Critical Issues:

There are several points that should to be resolved defining a parameterization scheme. The following attempts to list those points roughly in order of increasing complexity and decreasing priority.

1. Should the syntax be similar to the parameterization of derived types?

2. The parameterized derived types proposal contains many of the characteristics of the parameterization schemes that inspired this proposal. There are two main limitations upon the current parameterized derived types proposal that this proposal attempts to address: first the parameterized derived types proposal allows only parameterization by integers, while this proposal also allows parameterization by types; second, it is not clear how to use the parameterized derived types proposal to implement parameterization of procedures. Extension of the parameterized derived types proposal to allow parameterization by types appears to be straight forward. Should the syntax and semantics of the parameterized derived types proposal be extended to include parameterization by types in general? Can a means be identified for the parameterization of procedures that relies explicitly on parameterized derived types, and should that be the basis of the parameterization of procedures?

3. There is a tradeoff between the ease of usage of polymorphic code and ease of interfacing to code generated by other processors. Interfacing to code generated by other processors is simplified if the global entities have a straightforward

translation to their corresponding names in the “object” code. While Fortran’s modules have complicated the translations to “object” code, the translation remain relatively simple compared to the “name mangling” utilized by C++ systems. Unfortunately, the simplest form of syntax for the usage of polymorphic code provides no means of distinguishing different instantiations except implicitly based on the types of the instantiations. Implementations of languages that use this style of usage must use name mangling to distinguish different (public) instantiations of polymorphic code. As an example of this problem, assume the user has defined a module, EXAMPLE, with a single parameter which the user wants to instantiate with the value, X. In order to instantiate and use this module with this value there are two natural approaches, make instantiation automatic upon use, (which is essentially what C++ does) e.g.,

```
  use EXAMPLE(X)
```

or require that instantiation have an explicit new identifier associated with it before use, i.e.,

```
  module NEW_EXAMPLE = EXAMPLE(X)
```

(which is essentially what Ada requires). Should the syntax for usage of polymorphic code follow the C++ or the Ada model?

4. Although parameterization principally involves the types of objects, it also often involves the “size” of objects, typically expressed in terms of integer parameters. For arrays Fortran now

requires that the types of objects to be statically determined, but the size of objects are dynamically determined for assumed shape arrays. Should similar capabilities be provided for collection “types’ defined through parameterization, i.e., provide a syntax that statically defines the types of elements of a collection, but lets the sizes be determined at run time from a descriptor? Should this capability be extended to parameterized derived types?

5. Parameterization principally involves the “types” of entities, where the term “types” in this context has a more general meaning that Fortran’s data types. Typically the types must be consistent under textual substitution. In this Fortran may be more flexible than most languages that include parameterization. For example, unlike most languages with parameterization, Fortran does not make a clear a distinction between the types of arrays, pointers, and scalars. Similarly, while Fortran makes a distinction between functions and arrays, there is no syntactic distinction between functions and arrays with INTENT(IN). Fortran could therefore significantly increase the flexibility of its parameterization scheme by relaxing this distinction allowing arrays and functions to be treated as equivalent types in parameterization. It is possible, however, that there may be dangers or inefficiencies in such a flexibility. Should the language take advantage of this additional flexibility?

6. A parameterization scheme needs to define how such “types” can be specified. There are two general categories of such

specifications minimal or detailed:

6.a If only minimal separate specification of the types of the entities is allowed, then the allowed types under substitution must be inferred by a global analysis. This form of specification tends to result in implementations that are effectively sophisticated macro schemes where relatively little in the way of precompilation is done and type checking is usually done after the appropriate substitutions are performed. This has a detrimental effect on compilation times and code documentation, but allows flexibility in code development.

6.b. If a detailed separate specification of the important characteristics of the types of the entities is required, then immediate checking of the consistency of the specification section with the code section is possible. Such consistency checks provide the basis for implementations with more extensive precompilation and type checking before actual instantiation is performed. This has beneficial effects on compilation time and code documentation, but users can find it awkward to maintain consistency between the specification part and the main code body.

Which alternative should the language standard choose, no syntax to be provided for the detailed separate specification of the types of the entities, a required syntax for detailed separate specification of the types of the entities, or an optional syntax for detailed separate specification of the types of the entities?

7. Most languages in effect interpret parameterization as a form of macro substitution, and all types must be statically resolved. Some languages either allow (or require) the interpretation of parameterization as dynamic polymorphic procedures (similar to the dynamic polymorphism of class inheritance). The first interpretation generally results in efficient, but bloated, object code, the second tends to result in smaller, but inefficient, object code. Given Fortran's user community the first interpretation must be allowed by the language, but if dynamic polymorphism is allowed in other contexts would it be useful to provide an optional syntax with the second interpretation?

8. Parameterized modules are typically used to defined implementations for collection data "types", i.e., arrays, lists, stacks, trees, etc. One of these "types", arrays, is already an important aspect of Fortran, but, unlike most other languages, the syntax does not represent arrays as types separate from their components and provides a variety of elemental operations for it. It should be possible and desirable to define elemental operations for collection data "types" defined using parameterized modules. Is it possible and desirable to not syntactically represent collection data "types" defined through parameterized modules as separate types from their components?

9. As noted earlier, the work on traits for C++, indicates that with parameterization the characteristics of parameterized types

can be specified in a manner similar to what is currently done with Fortran’s intrinsic types and in distinguish the various KINDs of Fortran’s intrinsic types. The steps necessary to include this capability in Fortran should be identified for regularity of the language. Should the resulting capability for derived types then be used to describe Fortran’s intrinsic types?

10. With parameterization it becomes much more useful to think about syntactic components such as modules and derived type constructs as types in themselves, in the same sense that procedures are objects with types. Should the language in the standard reflect that approach?

References:

Scott W. Haney, “Beating the Abstraction Penalty in C++ Using Expression Templates,” Computers in Physics, Vol. 10, No. 6, Nov/Dec 1996, pp. 552-557

Nathan C. Myers, “Traits: a new and useful template technique,” C++ Report, Vol. 7 No. 5, June 1995 issue.

T. Veldhuizen, "Expression Templates," C++ Report, Vol. 7 No. 5, June 1995, pp. 26-31.