Date: 10 November 1998
To: J3
From: Van Snyder
Subject: Specifications and syntax for generic type-bound procedures
References: 97-230r1, 98-136, 98-140, 98-152r1, 98-179r1, 98-007r3

1 Specifications

Allow a specification that a generic identifier (12.3.2.1) is bound to a type. As with specific type bound procedures, all public generic identifiers are accessible if the type is accessible. None can be excluded by using `USE, ONLY`. The specific procedures that implement a generic are not automatically made accessible by accessing the type.

2 Syntax

A generic type-bound procedure is specified by putting
R437a *generic-proc-binding* **is** GENERIC [[, *binding-attr*] ::] *generic-spec* =>■
■ *specific-procedure-name-list*
within a type definition. PASS_OBJ shall not be specified if *generic-spec* is not *generic-name*.

3 Semantics of genericity and overriding

If a *generic-proc-binding* is specified in an extended type and it has the same *generic-spec* as one inherited from the parent type, then any specific procedures associated with the *generic-spec* that correspond in the way specified in 4.5.3.2 (Type-bound procedure overriding) to specific procedures associated with the inherited *generic-spec* override the corresponding specific procedures that are inherited from the parent type. Otherwise, they extend the generic.

In quasi-mathematical notation:

Let $D(T,N)$ be the set of specific procedures **D**eclared in generic procedure bindings having the *generic-spec* N within the declaration of T. Let $S(T,N)$ be the set of **S**pecific procedures associated with the *generic-spec* N and the type T, after accounting for inheritance and overriding. If T is not an extension type then $S(T,N) = D(T,N)$.

If T is an extension type, then let $S(P(T),N)$ be the set of specific procedures associated with the *generic-spec* N inherited into the type T from its parent type $P(T)$. Let $O(T,N) \subseteq S(P(T),N)$ be the set of specific procedures that are **O**verridden by specific procedures in $D(T,N)$ according to the criteria specified in 4.5.3.2 (Type-bound procedure overriding). Then $S(T,N) = D(T,N) \cup S(P(T),N) - O(T,N)$.

See paper 98-171r1 for yet another point of view.

Note

The dependence of overriding on the passed-object dummy argument at [53:1-2] means that overriding doesn't work when PASS_OBJ is not specified. Therefore overriding doesn't work for bindings that don't specify PASS_OBJ, type-bound operators or assignment. Overriding should depend on the first dummy argument of the type.
Define a term **overriding dummy argument** for the first argument of the type, and define overriding in terms of it, instead of in terms of the passed-object dummy argument.

In the definition of overriding at [53:1-2], the characteristics of the passed-object dummy argument are exempt from being the same for the overriding and overridden procedure. Except for the type, the characteristics *should* be the same.	*Note*

If N is the *binding-name* of a *proc-binding* within T then $S(T, N)$ is the specified or implied *binding*.

If N is a defined operator or assignment, define $\Sigma(N) = \bigcup_{\forall T} S(T, N)$.

The procedures that are elements of $S(T, N)$ or $\Sigma(N)$ shall be distinguishable by using the rules specified in 14.1.2.3 (Unambiguous generic procedure references).

If the definition of unambiguous generic procedure references (14.1.2.3) is not changed, then generic type-bound operators or assignments are essentially worthless for extensible types. Consider types T, U an extension of T, V and W not an extension of V. Consider four type-bound operators or assignments with the same generic identifier X, and identify the specific procedures by their dummy argument types. Group the procedures according to the type of their overriding dummy argument, say $X_T = \{(T, V), (T, W)\}$ and $X_U = \{(U, V), (U, W)\}$. If neither V nor T is an extension of the other, then X_U should be considered to be a separate generic set that overrides X_T; $X_U \cup X_T$ should not be considered to be a single generic. It should be required that generic resolution be unambiguous within X_U and separately within X_T. Within $X_U \cup X_T$ it is ambiguous. If W is an extension of V then $X_U \cup X_T$ must be considered as a whole, and is therefore ambiguous. Otherwise the procedures could be grouped as $X_V = \{(T, V), (U, V)\}$ and $X_W = \{(T, W), (U, W)\}$, and dispatching (see below) could yield different results depending on whether the first or second argument is used for the dispatching argument.	*Note*

4 Semantics of dispatch

The explanation of *dispatching* at [238:20-21] is inadequate. Define the **dispatching object** to be the *data-ref* if the procedure is invoked using one, or the actual argument associated with the overriding dummy argument if the procedure is invoked using a type-bound operator or defined assignment.

At a reference to a type-bound procedure, let T be the declared type of the dispatching object, and N be the generic identifier or binding name by which the procedure is invoked. If the procedure is invoked using a *data-ref* and PASS_OBJ is not specified, then the *data-ref* is not associated with a dummy argument. Using all the actual arguments that are associated with dummy arguments, select a specific procedure P from $S(T, N)$ in the usual way of doing generic resolution. A procedure P' is selected from $S(T', N)$, where T' is the dynamic type of the dispatching object, and P' is either P itself, or a procedure that overrides P (directly or indirectly).

Referring to the matrix representation of generic type-bound procedures introduced in paper 98-171r1, a row is selected by T. This row represents $S(T, N)$. Within that row, a column C is selected according to the usual rules for generic resolution. Within that column, again considering the entire array, a new row is selected using T'. This row represents $S(T', N)$. P' is the element at position (T', C) within this array.

The procedure P' is invoked.

In the discussion in the previous section, if U is an extension of T and W is an extension of V, and the type-bound operator or defined assignment is invoked with two polymorphic arguments, then there are two dispatching objects, and the result of dispatching, P', could

be different depending on which one is used for dispatching and which one is used for generic resolution.

Note

The combination of dispatch and argument association has bizarre semantics if it is allowed that the *data-ref* is not associated with a dummy argument: It may be possible to dispatch to a procedure that has no arguments of the type of *data-ref*, or any of its ancestor types.
I propose that the PASS_OBJ be eliminated from procedure bindings within derived types, and the passed-object dummy argument specification be moved to the procedure header (see 98-222). Then the dispatching object is always associated with a dummy argument.

5 This proposal is incomplete

This proposal does not include consideration of KIND type parameters of the dispatching object. In terms of the matrix representation of generic type-bound procedures introduced in paper 98-171r1, a row should probably be selected using the type T *and its kind type parameters.* The definitions of an unambiguous generic collection need work. The sets $D(T, N)$, $S(T, N)$ and $O(T, N)$ above should be parameterized in terms of the kind type parameters of T, in addition to T itself.