Subject: Semantics of the select kind construct are not described, and it appears to be a mess to use; also issues 287-289, part of issue 293
From: Van Snyder
References: 98-179, 00-179, 00-186, 00-194, 00-233, 00-234

1 Introduction

The select kind construct, apparently intended to be used within a derived type definition to select different specific procedures to invoke using an object of derived type, depending on the kind parameters, is not described further than providing its syntax. In particular, the relation between select kind, inheritance, and procedure overriding is not described.

Furthermore it is quite cumbersome to use. Suppose one has a type with three kind parameters, and one anticipates three values for each of those parameters. If one procedure is needed for each combination of kind type parameter values, this results in a requirement to bind 27 procedures to the type. It appears to require 92 statements to do so, using the select kind construct: Three nested select kind constructs are needed. The inner ones needs 8 statements each – the SELECT CASE and END SELECT statements, 3 CASE statements, and 3 procedure declaration statements. Each middle one encloses three of these, and adds five more statements, for a total of 29 statements per middle level case. The outer one has three middle ones, and adds five more statements, for a total of 92 statements. The proposal here would allow one to use one statement – albeit perhaps using more than one line, but not 92 lines.

This is a clumsy explicit replacement for the automatic generic resolution mechanism. (Actually, the intent is to specify how to generate dispatch tables, but the generic mechanism could do that more clearly.)

I propose in this paper to replace the select kind construct with the already-developed generic resolution mechanism.

This strategy has a simple extension to type-bound defined assignment, type-bound defined operators, type-bound derived-type input/output procedures (see 00-233), and type-bound final procedures (see 00-194).

2 Specifications

Several specific procedures may be bound to a type by using one binding name. The specific procedures bound to (not inherited into) a single type-bound procedure name shall be distinguishable according to the rules for unambiguous generic procedure reference (14.1.2.3).

The PASS_OBJ declaration (if present) applies to the binding name, and thereby to all of the specific procedures bound to the type, and all of its extensions, by that name. Therefore we don't need to worry about the case that a binding name has PASS_OBJ in the parent type but not in the type being declared, or vice-versa.

A binding declared in an extension type can override one inherited from the parent if it satisfies the rules in 4.5.3.2. Otherwise, a binding to a generic identifier inherited from the parent type shall be distinguishable from the other bindings to that generic identifier (both those inherited from the parent and bound within the extension) by the rules in subclause 14.1.2.3.

For procedure invocation, the generic resoltion rules from 14.1.2.4 determine which specific binding is referenced; the dynamic type of the object then determines which actual procedure is invoked. From an implementors point of view, each distinct generic resolution of a binding name uses a separate slot in the type's dispatch table.

3 Syntax

The proposed syntax to specify generic type-bound procedures is to specify non-generic procedure bindings by using the PROCEDURE statement, and generic bindings by using a new GENERIC statement.

The PROCEDURE statement is unchanged, and the *proc-binding* is extended to include

R440 *proc-binding* **is** <as at present>
or GENERIC (*proc-interface-name*) ■
■ [, *binding-attr-list*] :: *binding-name* => NULL()
or GENERIC [, *binding-attr-list*] :: ■
■ *binding-name* => *procedure-name-list*

A *binding-name* specified in a PROCEDURE statement shall not be the same as any other binding name specified within the same derived type definition, no matter whether specified in a PROCEDURE or GENERIC statement; if it is the same as an inherited one, the present overriding rules apply – it is not permitted to extend a generic set with a PROCEDURE binding. A binding name specified in a GENERIC statement may be the same as the binding name specified in another GENERIC statement, having the same effect as if the *procedure-name-lists* were combined in a single statement.

Example:

```
TYPE,EXTENSIBLE :: t1
  REAL x
CONTAINS
  GENERIC, PASS_OBJ :: addto => add_real, add_int
END TYPE
TYPE,EXTENDS(t1) :: t2
  REAL y
CONTAINS
  GENERIC, PASS_OBJ :: addto => add_2real, add_complex
END TYPE
...
SUBROUTINE add_real(x,r); CLASS(t1) x; REAL r
...
SUBROUTINE add_int(x,i); CLASS(t1) x; INTEGER i
...
SUBROUTINE add_2real(x,r); CLASS(t2) x; REAL r
...
SUBROUTINE add_complex(x,c); CLASS(t2) x; COMPLEX c
...
CLASS(t1) p; CLASS(t2) p2
! Let p refer to a TYPE(t2) object
CALL p%addto(3)      ! Calls add_int (inherit)
CALL p%addto(3.0)    ! Calls add_2real (override)
CALL p%addto((3,0))  ! Illegal - no specific binding in T1 for this
CALL p2%addto((3,0)) ! Calls add_complex
```