16.2.3 Edits

To: J3
From: Craig Dedo
Date: August 14, 2002
Subject: 16.2.3 Edits

Following are the JOR responses to the issues raised in 02-252.

[Richard Maine Please Note: The designation *word* is used to indicate that "word" should be set in the code font.]

395:18 Delete "They ensure ... unambiguous."

395:25-26 Replace "has a different ... rank" with "is TKR incompatible".

Do NOT make this edit.
395:27-28 Replace "type incompatible ... parameters" with "TKR incompatible".

Do NOT do this edit.
395:31-396:4 Move to 396:17+ with appropriate renumbering and changes in connecting punctuation and conjunctions.
Reason: The text for C.11.1+1 uses the existing numbers.

396:10-11,14-15 Replace "type incompatible ... different rank" with "TKR incompatible".

396:24+1-14 Delete NOTE 16.6.

397:1- Insert new note:
NOTE 16.7+1

An extensive explanation of the application of these rules may be found in C.11.1+1.
[End Note 16.7+1]

472:33+ Insert new subsection in section notes:

C.11.1+1 Rules ensuring unambiguous generics (16.2.3)

The rules in 16.2.3 are intended to ensure
- that it is possible to reference each specific procedure in the generic collection,
- that for any valid reference to the generic procedure, the determination of the specific procedure referenced is unambiguous, and
- that the determination of the specific procedure referenced can be made before execution of the program begins (during compilation).

Specific procedures are distinguished by fixed properties of their arguments, specifically type, kind type parameters, and rank. A valid reference to one procedure in a generic collection will differ from another because it has an argument that the other cannot accept, because it is missing an argument that the other requires, or because one of these fixed properties is different.

Although the declared type of a data entity is a fixed property, extensible types allow for a limited degree of type mismatch between dummy arguments and actual arguments, so the requirement for distinguishing two dummy arguments is type incompatibility, not merely different types. (This is illustrated in the *BAD6* example later in this note.)

That same limited type mismatch means that two dummy arguments that are not type incompatible can be distinguished on the basis of the values of the kind type parameters they have in common; if one of them has a kind type parameter that the other does not, that is irrelevant in distinguishing them.

Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a procedure is referenced by its specific name. In order to allow rank to always be usable in distinguishing generics, such rank mismatches are disallowed for those arguments when the procedure is referenced as part of a generic. Additionally, the fact that elemental procedures can accept array arguments is not taken into account when applying these rules, so apparent ambiguity between elemental and nonelemental procedures is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.

The concept of TKR incompatibility encapsulates the rules for distinguishing dummy arguments on the basis of any of these properties.

For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that it be possible to distinguish the arguments at one of the argument positions.

For user-defined derived-type input/output procedures, only the ***dtv*** argument corresponds to something explicitly written in the program, so it is the ***dtv*** that is required to be distinguished. Since ***dtv*** arguments are required to be scalar, they cannot differ in rank. Thus, in this rule, TKR incompatibility effectively involves only type and kind type parameters.

For generic procedures identified by names, the rules are more complicated because optional arguments may be omitted and because arguments may be specified either positionally or by name.

In the special case of type-bound procedures with passed-object dummy arguments, the passed-object argument is syntactically distinguished in the reference, so rule (2) can be applied. The type of passed-object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit from being type incompatible. Thus, in this rule, TKR incompatibility effectively involves only kind type parameters and rank.

The primary means of distinguishing named generics is rule (3). The most common application of that rule is a single argument satisfying both (3)(a) and (3)(b):

```
INTERFACE GOOD1
  FUNCTION F1A(X)
    REAL :: F1A,X
  END FUNCTION F1A
  FUNCTION F1B(X)
    INTEGER :: F1B,X
  END FUNCTION F1B
END INTERFACE GOOD1
```

Whether one writes *GOOD1(1.0)* or *GOOD1(X=1.0)*, the reference is to *F1A* because *F1B* would require an integer argument whereas these references provide the real constant 1.0.

This example and those that follow are expressed using interface bodies, with type as the distinguishing property. This was done to make it easier to write and describe the examples. The principles being illustrated are equally applicable when the procedures get their explicit interfaces in some other way or when kind type parameters or rank are the distinguishing property.

Another common variant is the argument that satisfies (3)(a) and (3)(b) by being required in one specific and completely missing in the other:

```
INTERFACE GOOD2
  FUNCTION F2A(X)
    REAL :: F2A,X
  END FUNCTION F2A
  FUNCTION F2B(X,Y)
    COMPLEX :: F2B
    REAL :: X,Y
  END FUNCTION F2B
END INTERFACE GOOD2
```

Whether one writes *GOOD2(0.0,1.0)*, *GOOD2(0.0,Y=1.0)*, or *GOOD2(Y=1.0,X=0.0)*, the reference is to *F2B*, because *F2A* has no argument in the second position or with the name *Y*. This approach is used as an alternative to optional arguments when one wants a function to have different result TKR depending on whether the argument is present. In many of the intrinsic functions, the DIM argument works this way.

It is possible to construct cases where different arguments are used to distinguish positionally and by name:

```
INTERFACE GOOD3
  SUBROUTINE S3A(W,X,Y,Z)
    REAL :: W,Y
    INTEGER :: X,Z
  END SUBROUTINE S3A
  SUBROUTINE S3B(X,W,Z,Y)
    REAL :: W,Z
    INTEGER :: X,Y
  END SUBROUTINE S3B
END INTERFACE GOOD3
```

If one writes *GOOD3(1.0,2,3.0,4)* to reference *S3A*, then the third and fourth arguments are consistent with a reference to *S3B*, but the first and second are not. If one switches to writing the first two arguments as keyword arguments in order for them to be consistent with a reference to *S3B*, the latter two arguments must also be written as keyword arguments, *GOOD3(X=2,W=1.0,Z=4,Y=3.0)*, and the named arguments *Y* and *Z* are distinguished.

The ordering requirement in rule (3) is critical:

```
INTERFACE BAD4  ! this interface is invalid !
  SUBROUTINE S4A(W,X,Y,Z)
    REAL :: W,Y
    INTEGER :: X,Z
  END SUBROUTINE S4A
  SUBROUTINE S4B(X,W,Z,Y)
```

```
      REAL :: X,Y
      INTEGER :: W,Z
    END SUBROUTINE S4B
  END INTERFACE BAD4
```

In this example, the positionally distinguished arguments are *Y* and *Z*, and it is *W* and *X* that are distinguished by name. In this order it is possible to write *BAD4(1.0,2,Y=3.0,Z=4)*, which is a valid reference for both *S4A* and *S4B*.

Rule (1) can be used to distinguish some cases that are not covered by rule (3):

```
  INTERFACE GOOD5
    SUBROUTINE S5A(X)
      REAL :: X
    END SUBROUTINE S5A
    SUBROUTINE S5B(Y,X)
      REAL :: Y,X
    END SUBROUTINE S5B
  END INTERFACE GOOD5
```

In attempting to apply rule (3), position 2 and name *Y* are distinguished, but they are in the wrong order, just like the *BAD4* example. However, when we try to construct a similarly ambiguous reference, we get *GOOD5(1.0,X=2.0)*, which can't be a reference to *S5A* because it would be attempting to associate two different actual arguments with the dummy argument *X*. Rule (3) catches this case by recognizing that *S5B* requires two real arguments, and *S5A* cannot possibly accept more than one.

The application of rule (1) becomes more complicated when extensible types are involved. If *FRUIT* is an extensible type, *PEAR* and *APPLE* are extensions of *FRUIT*, and *BOSC* is an extension of *PEAR*, then

```
  INTERFACE BAD6  ! this interface is invalid !
    SUBROUTINE S6A(X,Y)
      TYPE(PEAR) :: X,Y
    END SUBROUTINE S6A
    SUBROUTINE S6B(X,Y)
      TYPE(FRUIT) :: X
      TYPE(BOSC) :: Y
    END SBUROUTINE S2B
  END INTERFACE BAD6
```

might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed, *BAD6(A_PEAR,A_BOSC)* is a valid reference to both *S6A* and *S6B*.

It is important to try rule (1) for each type present:

```
  INTERFACE GOOD7
    SUBROUTINE S7A(X,Y,Z)
      TYPE(PEAR) :: X,Y,Z
    END SBUROUTINE S7A
    SUBROUTINE S7B(X,Z,W)
      TYPE(FRUIT) :: X
```

```
      TYPE(BOSC) :: Z
      TYPE(APPLE),OPTIONAL :: W
    END SUBROUTINE S7B
  END INTERFACE GOOD7
```

Looking at the most general type, *S7A* has a minimum and maximum of 3 *FRUIT* arguments, while *S7B* has a minimum of 2 and a maximum of three. Looking at the most specific, *S7A* has a minimum of 0 and a maximum of 3 *BOSC* arguments, while *S7B* has a minimum of 1 and a maximum of 2. However, when we look at the intermediate, *S7A* has a minimum and maximum of 3 *PEAR* arguments, while *S7B* has a minimum of 1 and a maximum of 2. Because *S7A*'s minimum exceeds *S7B*'s maximum, they can be distinguished.

In identifying the minimum number of arguments with a particular set of TKR properties, we exclude optional arguments and test TKR compatibility, so the corresponding actual arguments are required to have those properties. In identifying the maximum number of arguments with those properties, we include the optional arguments and test not TKR incompatible (i.e., TKR compatible in either direction), so we include actual arguments which could have those properties but are not required to have them.

These rules are sufficient to ensure that procedures that meet them are distinguishable, but there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```
  INTERFACE BAD8  ! this interface is invalid !
  ! despite the fact that it is unambiguous !
    SUBROUTINE S8A(X,Y,Z)
      REAL,OPTIONAL :: X
      INTEGER :: Y
      REAL :: Z
    END SUBROUTINE S8A
    SUBROUTINE S8B(X,Z,Y)
      INTEGER,OPTIONAL :: X
      INTEGER :: Z
      REAL :: Y
    END SUBROUTINE S8B
  END INTERFACE BAD8
```

This interface fails rule (3) because there are no required arguments that can be distinguished from the positionally corresponding argument, but in order for the mismatch of the optional arguments not to be relevant, the later arguments must be specified as keyword arguments, so distinguishing by name does the trick. This interface is nevertheless invalid so a standard-conforming Fortran processor is not required to do such reasoning. The rules to cover all cases are too complicated to be useful.

In addition to not recognizing distinguishable patterns like the one in *BAD8*, the rules do not distinguish on the basis of any properties other than type, kind type parameters, and rank:

```
  INTERFACE BAD9  ! this interface is invalid !
  ! despite the fact that it is unambiguous !
    SUBROUTINE S9A(X)
      REAL :: X
    END SUBROUTINE S9A
    SUBROUTINE S9B(X)
```

```
      INTERFACE
        FUNCTION X(A)
          REAL :: X,A
        END FUNCTION X
      END INTERFACE
    END SUBROUTINE S9B
    SUBROUTINE S9C(X)
      INTERFACE
        FUNCTION X(A)
          REAL :: X
          INTEGER :: A
        END FUNCTION X
      END INTERFACE
    END SUBROUTINE S9C
  END INTERFACE BAD9
```

The real data objects that would be valid arguments for *S9A* are entirely disjoint from procedures that are valid arguments to *S9B* and *S9C*, and the procedures that valid arguments for *S9B* are disjoint from the procedures that are valid arguments to *S9C* because the former are required to accept real arguments and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran processor need not examine such properties when deciding whether a generic collection is valid. Again, the rules to cover all cases are too complicated to be useful.

472:34-473:15 Move to 467:43+.

References

02-007r2, Fortran 2000 Draft

[End of J3 / 02-252r1]