Date: 7 May 1998
To: J3
From: Van Snyder and Malcolm Cohen
Subject: Edits for R.6a – Inheritance
References: 97-196r2 98-133

Edits refer to 98-007r1. Page and line numbers are displayed in the margin. Remarks for the editor are noted in the margin, or appear between [and].

[*component-def-stmt* ...] [39:3-4]

R423 *derived-type-stmt* **is** TYPE [[, *type-qualifier-list*] ::] *type-name* [39:6]

R424a *type-qualifier* **is** *access-spec*
or EXTENSIBLE [39:9+]
or EXTENDS(*parent-type-name*)

Constraint: A derived type shall not have both the EXTENSIBLE and EXTENDS qualifiers. [39:13+]
Constraint: *parent-type-name* shall be the name of an accessible extensible type ([new section] 4.5.3).

Constraint: If *derived-type-def* defines an extensible type ([new section] 4.5.3), SEQUENCE shall not be present. [39:17+]

Constraint: A component declared with the OBJECT keyword ([new section] 5.1.1.8) shall have the POINTER attribute. [39:30+]

4.5.3 Extensible types [46:36+]

Insert new section, add **base type**, **descendant type**, **extensible type**, **extension type**, and **inherit** to the index. *Editor*

A type that is declared with the EXTENSIBLE or EXTENDS(*parent-type-name*) qualifier is an **extensible type**. Note: Intrinsic types are not extensible.

New types may be derived from extensible types by adding zero or more components.

A type that is declared with the EXTENSIBLE qualifier is a **base type**.

The next sentence indirectly defines "parent type." Should "parent type" be in bold face type? If so, it should be in the index, too. *Malcolm*

A type that is declared with the EXTENDS(*parent-type-name*) qualifier is an **extension type** of the specified parent type. A type is a **descendant type** of another type if and only if it is the same as the other, or is a direct or indirect extension of it.

The above definition of **descendant type** avoids "the same type as ... or a type extended from ..." in numerous places. This usage is common practice in theoretical computer science, and textbooks on algorithm design. The latter can be used if J3 prefers it. *J3 Note*

An extensible type is not required to have any components. Note: An extension type is a new type even if it declares no additional components.

An extension type includes all of the components of its parent type. The components of the parent type are said to be **inherited** by the extension type. Additional components may be declared. For purposes of intrinsic input/output (9.4.2) and value construction ([existing section] 4.5.4), the order of the components of an extension type is the components inherited from the parent type, followed by the components of the extension type, in the order declared.

An extension type has a component name that is the same name and has the same type as its parent type. This is not an additional component; it denotes a subobject that has the parent type, and that consists of all of the components inherited from the parent type.

Note: The subobject denoted by the parent type name has the same accessibility as the additional components of the extension type, even if the components of the parent type are not accessible.

A component declared in an extension type shall have neither the same name as any accessible component of its parent type nor the same name as the parent type name.

Note 4.5.3a

Examples:

```
TYPE, EXTENSIBLE :: POINT           ! A base type
  REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: COLOR_POINT  ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT
```

R436 *structure-constructor* **is** *derived-type-spec*([*component-spec-list*]) [47:24]

[47:42+] A *structure-constructor* for an extension type may use a nested form or a list form. In the **nested form** a single value is provided for the component that has the same name as the parent type.

If every component of the parent type has a default initialization, does this constitute a default initialization for the component that has the same name as the parent type? *Malcolm*

In the **list form** a separate value is provided for each component of the parent type. The list form shall not be used if any components of the parent type are inaccessible in the scoping unit in which the *structure-constructor* appears (4.5.1).

In the absence of a component name keyword, values for the parent type shall be provided before values for the additional components of the extension type. *Necessary?*

Note 4.4.4a

Examples of equivalent values (see note 4.5.3a):

```
! Create values with components x == 1.0, y == 2.0, color == 3:
TYPE(POINT) :: PV = POINT(1.0, 2.0) ! Assume components of TYPE(POINT)
                                    ! are accessible here
...
COLOR_POINT( PV, 3 )                ! Nested form, available even if
                                    ! TYPE(POINT) has PRIVATE
                                    ! components.
COLOR_POINT( POINT(1.0, 2.0), 3 )   ! Nested form, components of
                                    ! TYPE(POINT) must be accessible.
COLOR_POINT( 1.0, 2.0, 3 )          ! List form, components of
                                    ! TYPE(POINT) must be accessible.
```

or OBJECT(*type-name*) [51:25+]

Constraint: An entity declared with the OBJECT keyword shall be a dummy argument or have the POINTER attribute. [52:1+]

Constraint: If an entity is declared with the OBJECT keyword, the *type-name* shall be an extensible type ([new section] 4.5.3).

5.1.1.8 Polymorphic objects [Editor: new section]

[56:16+]

An OBJECT type specifier is used to declare objects that can, during program execution, assume any derived type that is a descendant type ([new section] 4.5.3) of the type specified by the *type-name*. The type specified by *type-name* is the **declared type** of the polymorphic object. The type assumed at any instant during program execution is the **run-time type** at that instant. The run-time type of a fixed-type object is the same as its declared type.

Note: Only components of the declared type of a polymorphic object may be referenced by component selection (6.1.2).

When (if) a SELECT TYPE construct is added, mention it here. *J3 note*

The next two paragraphs are an extension of specifications. It seems more useful for a disassociated polymorphic pointer object to have the declared type than an undefined type, and it would have been difficult to describe a NULL() intrinsic returning a pointer with no type. *J3 note*

Polymorphic objects acquire their run-time type from associated actual arguments (12.4.1.2), as a result of pointer assignment (7.5.2), or as a result of successful execution of an ALLOCATE (6.3.1), NULLIFY (6.3.2), or DEALLOCATE (6.3.3) statement.

The type of a polymorphic pointer with a pointer association status of disassociated or undefined is the declared type.

[Editor: change "type" to "declared type"]. [82:42]

An *allocate-object* that is a polymorphic object ([new section] 5.1.1.8) is allocated with its run-time type equal to the declared type. [88:22+]

Note: When a NULLIFY statement is applied to a polymorphic object ([new section] 5.1.1.8) the run-time type becomes the declared type. [91:8+]

An **intrinsic assignment statement** is an assignment statement wherein the shapes of *variable* and *expr* conform, and which is one of the following. [118:29-40]

(1) A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of numeric type. *New wording for existing material*

(2) A **character intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type character and have the same kind type parameter.

(3) A **logical intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type logical.

(4) A **derived type intrinsic assignment statement** is an intrinsic assignment statement for which the following are true.

  (i) There is no accessible defined assignment (12.3.2.1.2) for objects of the declared type of *variable* and objects of the declared type of *expr*.

The following were not covered by specifications. Choose one alternative of the three. *J3 Note*

  (ii) The declared type of *expr* is a descendant type ([new section] 4.5.3) of the declared type of *variable*; if either *variable* or *expr* is polymorphic ([new section] 5.1.1.8), the run-time type of *expr* shall be a descendant type of the run-time type of *variable*. *Alt. 1*

  (ii) The declared type of *variable* shall be the same as the declared type of *expr* [within specs]. If either *variable* or *expr* is polymorphic, the run-time type of *expr* shall be the same as the run-time type of *variable* [an extension of specs]. *Alt. 2*

(ii) The declared type of *variable* shall be the same as the declared type of *expr* [within specs]. The *variable* shall not be polymorphic [within specs]. If *expr* is polymorphic, the run-time type of *expr* shall be the same as the type of *expr* [an extension of specs]. *Alt. 3*

Notwithstanding Werner Schulz's objections, anything other than Alt. 1 requires a pile of nested SELECT TYPE constructs, or even worse if we don't do SELECT TYPE. *Malcolm*

[Editor: Move table 7.9 to be between lines 17 and 18.] [119:1-8]

Specs didn't discuss the following paragraph. It is germane if we don't choose Alt. 3 above. *J3 note*

For a derived type intrinsic assignment statement, if the type of *expr* is not the same as the type of *variable*, only the part of *expr* that is inherited ([new section] 4.5.3), directly or indirectly, from the type of *variable* is assigned. Note: This is analogous to the case of *variable* being of type real and *expr* of type complex, wherein only the real part of *expr* is assigned to *variable*. If either *variable* of *expr* are polymorphic (5.1.1.8) the run-time types determine the part of *expr* that is assigned to *variable*. [120:0+]

Specs didn't discuss the following paragraph. *J3 note*

I used "assigned from" instead of "assigned the value of" because this sentence asserts that assignment is done, not that value assignment is done. In fact, the following paragraph states the condition under which pointer assignment is done. *Malcolm*

A derived type intrinsic assignment is performed as if each component of *variable* is assigned from the corresponding component of *expr*. Note: if *variable* is polymorphic ([new section] 5.1.1.8) the components are determined by the run-time type of *variable* [unless we choose Alt. 3 above]. Note: If the type of *expr* is different from the type of *variable*, additional components of *expr* are ignored [unless we choose Alt. 2 or Alt. 3 above]. [120:34-37]

For pointer components, pointer assignment is used.

For non-pointer components that are not allocatable arrays, intrinsic assignment is used.

For allocatable array components the following sequence of operations is applied:

[Editor: change "types" to "declared types".] [121:25]

Constraint: If *pointer-object* is a data object or a function procedure pointer, the declared type of *target* shall be a descendant type ([new section] 4.5.3) of the declared type of *pointer-object*. [122:14-15]

Constraint: If *pointer-object* is a data object or a function procedure pointer, the rank and kind type parameters of *target* shall be the same as the rank and corresponding kind type parameters of *pointer-object*.

Do we need to be more careful in the two previous constraints about "the type" of a function procedure pointer? *Malcolm*

If the *target* is polymorphic (5.1.1.8), the run-time type of the *target* shall be a descendant type ([new section] 4.5.3) of the declared type of the *pointer-object*. [122:33+]

The following paragraph extends the specifications. See the second J3 note at [56:16+]

If the *pointer-object* is polymorphic, the *pointer-object* assumes the run-time type of the *target*.

[Note to Editor: Replace "If ... object," by:] [225:41]

If a dummy argument is a dummy data object of fixed type and the associated actual argument is of fixed type, If either or both of a dummy argument and its associated actual argument are polymorphic, their data types are not required to be exactly the same.

The run-time type of an actual argument shall be a descendant type ([new section] 4.5.3) of the declared type of the dummy argument. If the dummy argument is polymorphic it assumes the run-time type of the corresponding actual argument.

Malcolm

The call to SA is legal, but results in allowing an illegal pointer assignment, or requiring a run-time check. The call to SB is illegal, but if it were allowed, nothing bad would happen in its body. Because intent(out) dummy arguments must be pointer-assigned before they can be referenced, the argument association rule for intent(out) pointer dummy arguments should be that the declared type of the dummy argument shall be a descendant type of the declared type of the actual argument (opposite to all the other cases). Then, the call to SA is illegal, and no run-time check is needed, while the call to SB is legal. The effect is to allow a slightly larger class of legal and meaningful programs, and to obviate the need for a run-time check of the constraint "In a pointer assignment to a polymorphic dummy argument, the run-time type of the target shall be a descendant type of the declared type of the associated actual argument" in the case of intent(out) pointer dummy arguments; it is still needed for intent(inout) and unspecified intent. This has the effect of requiring the declared type of the actual argument to be part of the dope vector.
We need the constraint in any case. Should it be in (7.5.2 – Pointer assignment) or here?

```
  TYPE, EXTENSIBLE :: A ...
  TYPE, EXTENDS(A) :: B ...
  OBJECT(A), POINTER :: PA
  OBJECT(B), POINTER :: PB
  SUBROUTINE SA ( XA )
    OBJECT(A), POINTER, INTENT(OUT) :: XA
    XA => PA
  END SUBROUTINE SA
  CALL SA ( PB ) ! Indirectly, the XA => PA in SA violates pointer
                 ! assignment rules.
  SUBROUTINE SB ( XB )
    OBJECT(B), POINTER, INTENT(OUT) :: XB
    XB => PB
  END SUBROUTINE SB
  CALL SB ( PA ) ! Violates argument association rules, but
                 ! XB => PB inside SB is OK.
```

Examples: Given

```
TYPE(POINT) :: T2  ! See note 4.5.3a
TYPE(COLOR_POINT) :: T3
OBJECT(POINT) :: P2
OBJECT(COLOR_POINT) :: P3
! Dummy argument is polymorphic and actual argument is of fixed type
SUBROUTINE SUB2 ( X2 ); OBJECT(POINT) :: X2; ...
SUBROUTINE SUB3 ( X3 ); OBJECT(COLOR_POINT) :: X3; ...

CALL SUB2 ( T2 ) ! Legal -- The declared type of T2 is the same as the
                 !          declared type of X2.
CALL SUB2 ( T3 ) ! Legal -- The declared type of T3 is extended from
                 !          the declared type of X2.
CALL SUB3 ( T2 ) ! Illegal -- The declared type of T2 is neither the
                 !          same as nor extended from the declared type
                 !          type of X3.
CALL SUB3 ( T3 ) ! Legal -- The declared type of T3 is the same as the
                 !          declared type of X3.
! Actual argument is polymorphic and dummy argument is of fixed type
SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2
SUBROUTINE TUB3 ( D3 ); TYPE(COLOR_POINT) :: D3

CALL TUB2 ( P2 ) ! Legal -- The declared type of P2 is the same as the
                 !          declared type of D2.
CALL TUB2 ( P3 ) ! Legal -- The declared type of P3 is extended from
                 !          the declared type of D2.
CALL TUB2 ( P2 ) ! is legal only if the run-time type of P2 is the same
                 !          as the declared type of D2, or a type
                 !          extended therefrom.
CALL TUB3 ( P3 ) ! Legal -- The declared type of P3 is the same as the
                 !          declared type of D3.
! Both the actual and dummy arguments are of polymorphic type.
CALL SUB2 ( P2 ) ! Legal -- The declared type of P2 is the same as the
                 !          declared type of X2.
CALL TUB2 ( P3 ) ! Legal -- The declared type of P3 is extended from
                 !          the declared type of X2.
CALL TUB2 ( P2 ) ! is legal only if the run-time type of P2 is the same
                 !          as the declared type of X2, or a type
                 !          extended therefrom.
CALL TUB3 ( P3 ) ! Legal -- The declared type of P3 is the same as the
                 !          declared type of X3.
```

Note 12.4.1.2a *Maybe this should be in Annex C.*

13.10 Polymorphic type inquiry functions [Editor: new section] [248:15+]

I changed the functions to allow either argument to be polymorphic or fixed. This allows asking EXTENDS_TYPE_OF either way, without needing to put .NOT. in front of it. SAME_TYPE_AS is changed for symmetry. I don't think we need a note about pointlessness if they're both of fixed type, any more than we need a note about pointlessness of SQRT(1.0). *Malcolm*

These intrinsic functions are slightly changed from specifications. *J3 note*

The function SAME_TYPE_AS inquires whether two objects of extensible type ([new section] 4.5.3) have the same run-time type. The function EXTENDS_TYPE_OF inquires whether the run-time type of one object of extensible type is a descendant type ([new section] 4.5.3) of the run-time type of another object of extensible type.

13.11.21 Polymorphic type inquiry functions [Editor: new section] [252:29+]

EXTENDS_TYPE_OF(A, B)	Same run-time type or an extension
SAME_TYPE_AS(A, B)	Same run-time type

13.13.37 EXTENDS_TYPE_OF(A, B) [Editor: new section] [268:33+]

Description. Inquires whether the run-time type of A is a descendant type ([new section] 4.5.3) of the run-time type of B.

Class. Inquiry function.

Arguments.

A	shall be an object of extensible type.
B	shall be an object of extensible type.

Result Characteristics. The result is of type default logical scalar.

Do you want "direct or indirect extension" instead of "descendant" below, and in the summary (13.10) above? *Malcolm*

Result Value. The result is true if and only if the run-time type of A is a descendant type ([new section] 4.5.3) of the run-time type of B.

13.14.92 SAME_TYPE_AS(A, B) [Editor: new section] [291:37+]

Description. Inquires whether the run-time type of A is the same as the run-time type of B.

Class. Inquiry function.

Arguments.

A	shall be an object of extensible type.
B	shall be an object of extensible type.

Result Characteristics. The result is of type default logical scalar.

Result Value. The result is true if and only if the run-time type of A is the same as the run-time type of B.

[Note to Editor: Replace "different type" by "declared type that is not a descendant type ([new section] 4.5.3) of the declared type of the other dummy argument."] [305:26,30]

---

base type ([new section] 4.5.3): An extensible type that is not an extension of another type. A type that is declared with the EXTENSIBLE type qualifier. [341:37+]

---

descendant type ([new section] 4.5.3): An extensible type that is the same as another extensible type, or a direct or indirect extension of it. [342:38+]

---

extensible type ([new section] 4.5.3): A type from which new types may be derived by adding components. [344:28+]

extension type ([new section] 4.5.3): A type derived from an extensible type by adding components.

---

fixed-type object ([new section] 5.1.1.8): An object for which the type cannot change during execution of a program. [344:38+]

---

inherit ([new section] 4.5.3): Components of an extension type are automatically acquired [345:13+]

from the parent type, without requiring explicit declaration in the extension type.

---

parent type ([new section] 4.5.3): The extensible type from which an extension type is derived. [346:37+]

---

polymorphic object (5.1.1.8): An object for which the type may vary during program execution. [347:7+]

---

run-time type (5.1.1.8): The actual type of a polymorphic object during execution of a program. The run-time type of a fixed-type object is the same as its declared type. [347:36+]

Components of an object of extensible type that are inherited from the parent type may be accessed, as a whole, by using the component name that is the same as the parent type name, or individually, either with or without qualifying them by the component name that is the same as the parent type name. Continuing note 4.5.3a: [361:43+]

```
TYPE, EXTENSIBLE :: POINT           ! A base type
  REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: COLOR_POINT  ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT) :: PV = POINT(1.0, 2.0)
TYPE(COLOR_POINT) :: CPV = COLOR_POINT(PV, 3) ! Nested form constructor

PRINT *, CPV%POINT                  ! Prints 1.0 and 2.0
PRINT *, CPV%POINT%X, CPV%POINT%Y   ! And this does, too
PRINT *, CPV%X, CPV%Y               ! And this does, too
```