# Dynamic Binding and Polymorphism
## Draft Specifications and Illustrative Syntax

### 1. Introduction

These are the specifications with illustrative syntax to satisfy the requirement for the ability to invoke dynamically bound procedures where the actual procedure invoked depends on the runtime type of a single, potentially polymorphic, associated variable. Such procedures are bound to the type of an entity, whereas a procedure component is bound to the contents (value) of an entity.

These dynamically bound procedures are like the "virtual functions" of C++ and friends, the "type-bound" procedures of Oberon-2, and the "methods" of Smalltalk. We will call these **type-bound procedures**, as their invocation is bound to a specific type. This neatly captures the difference between them and the object-bound procedures we call procedure components.

Type-bound procedures are not permitted in SEQUENCE types, because there is no unique definition to trigger the creation of the runtime dispatch table (or equivalent data structure for runtime support).

### 2. Basic Functionality

Type-bound procedures are declared in the definition of the type to which they are bound. They have a unique "component name" by which they are accessed (perhaps "accessor name" is a better term?) which is not necessarily the same as the actual procedure name. This renaming is provided both for convenience and to allow a single module to provide both a type and one or more extensions. The actual procedure is a module procedure from that module.

The first dummy argument of a "type-bound procedure" shall be a scalar non-pointer dummy variable of the type; this is the equivalent of C++'s "this" (implicit) argument. We do not want to have a reserved word (like "this") so allowing users to use their own dummy argument names seems reasonable and friendly. C++ allows access to components of its "this" variable without qualification; we do *not* propose this.

Example:

```
TYPE vector_2d
      REAL x,y
CONTAINS
      PROCEDURE length => length_2d
END TYPE
...
REAL FUNCTION length_2d(v)
      TYPE(vector_2d) v
      length_2d = SQRT(v%x**2+v%y**2)
END FUNCTION
```

Reference to a type-bound procedure looks just like a reference to an object-bound procedure (a.k.a. procedure component), and the object used to access the procedure becomes the first argument, e.g.

```
TYPE(vector_2d) vec
REAL size
...
size = vec%length()     ! Equivalent to "size =
length_2d(vec)",
                        ! provided length_2d is accessible.
```

### 3. With Type Extension

A type-bound procedure may be overridden in an extension of the type, e.g.

```
TYPE vector_3d, EXTENDS TYPE(vector_2d)
      REAL z
CONTAINS
      PROCEDURE length => length_3d
END TYPE
REAL FUNCTION length_3d(self)
      TYPE(vector_3d) self
      length_3d = SQRT(self%x**2+self%y**2+self%z**2)
END FUNCTION
```

When invoked from a TYPE(vector_3d) entity, "length" references "length_3d". Access is still possible to the TYPE(vector_2d) version of "length" (i.e. length_2d) by invoking "length" on the vector_2d part, for example:

```
TYPE(vector_3d) x
...
size = x%length()              ! Invokes length_3d
size = x%vector_2d%length()    ! Invokes length_2d on the
length_2d
                               ! part of x.
```

4. Dynamic Dispatch

Determination of which actual procedure is invoked can often be done at compile-time; it is only necessary to use runtime dispatch when the entities are polymorphic, e.g. given

```
TYPE(vector_2d) v2
TYPE(vector_3d) v3
OBJECT(vector_2d) obj
```

v2%length() invokes the vector_2d length because v2 is always of TYPE(vector_2d).
v3%length() invokes the vector_3d length because v3 is always of TYPE(vector_3d).
obj%length() invokes the vector_2d length when obj is of TYPE(vector_2d), and
vector_3d length when obj is of TYPE(vector_3d).

Note that although "v3%length()" and "length_3d(v3)" are equivalent, there is no straightforward equivalent to "obj%length()" because the actual procedure invoked depends on the runtime type of "obj".

obj%vector_2d%length() invokes the vector_2d length on the vector_2d part of obj
(if obj is of TYPE(vector_2d) that is the entirety of it).

5. Another Example

These procedures may have further arguments following the object dummy. When overriding a type-bound procedure, the names of all arguments and the characteristics of all but the first argument shall be the same.

```
MODULE image_processing
      TYPE image
            ...
      CONTAINS
            PROCEDURE draw_box => image_draw_box
            ...
      END TYPE
CONTAINS
      SUBROUTINE image_draw_box(on,x1,y1,x2,y2)
            TYPE(image) on
            REAL x1,y1,x2,y2
            ...
      END SUBROUTINE
      ...
END MODULE
PROGRAM example
      USE image_processing
      OBJECT(image) view
      ...
      CALL view%draw_box(1.5,1.0,2.5,3.0)
      ...
END PROGRAM
```

6. Abstract Type-Bound Procedures

It can be useful to allow a base type to declare a type-bound procedure name that is not actually bound to anything, forcing the extended types to declare their own version of this operation. This could be provided by syntax like, for example:

```
      TYPE vector_0d
            ! No components necessary, a type-bound procedure is
enough
      CONTAINS
            PROCEDURE length => NULL()
      END TYPE
```

A type extended from "vector_0d" shall contain a declaration for length; this is permitted to confirm that "length" is still abstract or to supply a specific procedure. E.g.,

```
      TYPE vector_0d_special
            INTEGER special_value
      CONTAINS
            PROCEDURE length => null()          ! Still abstract
      END TYPE
      TYPE vector_1d
            REAL x
      CONTAINS
            PROCEDURE length => length_1d       ! We have a
"length()"
      END TYPE
```

However, it is not possible to override an existing procedure binding with the null binding, e.g.

```
      TYPE vector_2d
            REAL y
      CONTAINS
            PROCEDURE length => NULL()          ! Illegal
      END TYPE
```

It is possible to declare entities of TYPE(vector_0d) or OBJECT(vector_0d); however it is a compile-time error to have a reference to "length" in a TYPE(vector_0d) entity, and a runtime

error to execute a reference to "length" from an object that still has the null binding (e.g. the variable is OBJECT(vector_0d) and the runtime type is TYPE(vector_0d_special)).

7. Visibility

The default visibility of a type-bound procedure is PUBLIC, i.e. the default visibility of the section before the "CONTAINS" is separate from the default visibility of the section after the CONTAINS.
This default may be changed with an explicit PRIVATE statement following the CONTAINS, and may be overridden for individual procedures with an accessibility attribute.

E.g.,

```
      TYPE mytype
            PRIVATE
            REAL x,y                        ! secret components
      CONTAINS
            PROCEDURE mean                  ! public type-bound
procedure
            PROCEDURE, PRIVATE :: hypot     ! private type-bound
procedure
      END TYPE

      TYPE another
            LOGICAL available(100)          ! public components
      CONTAINS
            PRIVATE
            PROCEDURE secret                ! private procedures...
            PROCEDURE hidden
            PROCEDURE, PUBLIC :: pub              ! public procedure
      END TYPE
```

The rationale for resetting the default visibility on the CONTAINS is that it is anticipated that it would be common for the user to require private (or mostly private) components but to have public type-bound procedures (e.g. which could include functions which access the private components).