Given the following declarations and procedures,

```
TYPE,EXTENSIBLE::tb
  ! components
CONTAINS
  PROCEDURE p1 => p1b
  PROCEDURE q1 => q1b
  PROCEDURE p2 => p2b
END TYPE
TYPE(tb)::vb,vb2
…
SUBROUTINE p1b(x,r); TYPE(tb)::x; REAL r
  ! do the work
END SUBROUTINE p1b
SUBROUTINE q1b(x,i); TYPE(tb)::x; INTEGER i
  ! do the work
END SUBROUTINE q1b
SUBROUTINE p2b(x,y,l); TYPE(tb)::x,y; LOGICAL l
  ! do the work
END SUBROUTINE p1b
```

then these procedure references

```
CALL vb%p1(0.0)
CALL vb%q1(0)
CALL vb%p2(vb2,.false.)
```

can be seen as essentially equivalent to

```
CALL p1b(vb,0.0)
CALL q1b(vb,0)
CALL p2b(vb,vb2,.false.)
```

If we now introduce polymorphic objects

```
OBJECT(tb)::ob,ob2
```

this is expected to be implemented, in part, by creating a dispatch vector for the type

```
TYPE(procedure_pointer)::DV_tb(3)
…
DV_tb(1)%PP=>p1b; DV_tb(2)%PP=>q1b; DV_tb(3)%PP=>p2b
```

and then translating polymorphic procedure references

```
CALL ob%p1(0.0)
CALL ob%q1(0)
CALL ob%p2(ob2,.false.)
```

using the appropriate dispatch vector to locate the procedures

```
PP=>DV(1); CALL PP(ob,0.0)
PP=>DV(2); CALL PP(ob,0)
PP=>DV(3); CALL PP(ob,ob2,.false.)
```

If we now extend our base type

```
TYPE,EXTENDS(tb)::te
  ! additional components
REPLACES
  PROCEDURE p1=>p1e
END TYPE
TYPE(te)::ve,ve2
…
SUBROUTINE p1e(x,r); TYPE(te)::x; REAL r
  ! do the work
END SUBROUTINE p1e
```

the override of p1 means that

```
CALL ve%p1(0.0)
```

is essentially equivalent to

```
CALL p1e(ve,0.0)
```

and the dispatch vector for type te will contain a pointer to p1e in the first position.

Since there was no replacement for q1, the rules say we should continue to use the specific procedure associated with q1 in the base type, i.e. q1b, but mapping

```
CALL ve%q1(0)
```

to

```
CALL q1b(ve,0)
```

would be a type mismatch error, so instead we map it to

```
CALL q1b(ve%tb,0)
```

However, when we access q1b through the dispatch vector, in the general case we will not know whether the base procedure has been replaced or not, so we don't know whether to apply the %tb to the first argument. To work around this problem, we must, in effect, automatically generate a "wrapper" for q1b that handles the application of the %tb subobject selection

```
SUBROUTINE q1e(x,i); TYPE(te)::x; INTEGER i;
  CALL q1b(x%tb,i)
END SUBROUTINE q1e
```

and put the address of the generated q1e in the dispatch vector for type te. [In the typical real implementation, representation tricks are used so we can avoid generating code for q1e, but it exists at a conceptual level to correct the type mismatch.]

The issue of interest is what kind of "wrapper" we need for p2b, since it also was not replaced. A literal interpretation of the wording in the current specification would suggest that it look like

```
SUBROUTINE p2e_A(x,y,l); TYPE(te)::x; TYPE(tb)::y; LOGICAL l
  CALL p2b(x%tb,y,l)
END SUBROUTINE p2e_A
```

on the grounds that y, like l, is a dummy argument different from the distinguished first argument. The suggested alternative is that is should look like

```
SUBROUTINE p2e_B(x,y,l); TYPE(te)::x,y; LOGICAL l
  CALL p2b(x%tb,y%tb,l)
END SUBROUTINE p2e_B
```

on the grounds that in typical applications the significant feature of the original p2b was that the type of y was the same as the type of x, not that it was specifically of type tb, and that therefore that "sameness" is the property that should be preserved as we move to an extension type. [It may seem strange to argue about the interface of a procedure that in real implementations won't be separately manifested, but whichever implied interface we choose for this wrapper will also be the required interface if one wishes to provide a replacement p2, so it will have a significant effect.] The /data subgroup considered the following possible options:

1. Keep the language so it implies alternative A.

2. Tweak the language so it implies alternative B.

3. Disallow procedures with the "dispatching" type repeated in the argument list (i.e., disallow putting p2b in tb in the first place), so it doesn't matter and we can decide between A and B later.

4. Provide a syntactic means for the programmer to dictate which alternative is wanted. (This syntax could be in the original procedure (p2b) or at the point of its binding into the base type (in tb). At the moment, the subgroup appears to be leaning towards the latter.) This option provides the most flexibility, but if you believe, for example, that 99% of the time, the programmer will choose alternative B, then it might be seen as unnecessarily complicated.

(In a brief survey of other languages on this issue, we found some that were like alternative A (e.g., C++ and Java), some that were like alternative B (e.g., Ada 95 and Eiffel), and even some that provided the means to choose, so it should not be too surprising that the subgroup has not reached a consensus on this issue.)

Ω