This is intended to be a slightly more realistic illustration of the polymorphism issue described in J3/97-261, with some estimation of the costs of the alternatives and tradeoffs. Let us begin with an extensible type representing 2D vectors with two type-bound operations — obtaining the length of a single vector and computing the cosine of the angle between two such vectors:

```
TYPE,EXTENSIBLE::vector_2d
  REAL::x,y
CONTAINS
  length=>length_2d
  corr=>corr_2d
END TYPE
…
FUNCTION length_2d(v) RESULT(length)
  TYPE(vector_2d)::v; REAL:: length
  length=sqrt(v%x**2+v%y**2)
END FUNCTION length_2d
FUNCTION corr_2d(v1,v2) RESULT(corr)
  TYPE(vector_2d)::v1,v2; REAL::corr
  corr=(v1%x*v2%x+v1%y+v2%y)/(v1%length()*v2%length())
END FUNCTION corr_2d
```

Using this type, we can write a subroutine that prints out all the cosines for all combinations of vectors drawn from a list of vectors:

```
SUBROUTINE print_corr_table(vlist)
  OBJECT(vector_2d)::vlist(:); INTEGER::j,k
  DO(j=1,size(vlist))
    print *,(vlist(j)%corr(vlist(k)),k=1,j)
  END DO
END SUBROUTINE print_corr_table
```

There are two costs to declaring vector_2d with OBJECT rather than type (thus making this procedure polymorphic:

1. The reference to vlist(j)%corr must be made through a run-time dispatch table, at the cost of an additional memory reference per function reference. In this particular example, a smart compiler might keep that the function address will be invariant in the loop on j and keep that function address in a register.

2. For alternative B (the one similar to Ada 95), a naive analysis of the types in the reference to vlist(j)%corr would suggest the possibility of a run-time type mismatch. If you wish to run in a mode where such errors are caught, the necessary code (assuming an appropriate representation of the type information) would involve verifying that one integer in memory is greater than or equal to another integer in memory and then than one address stored in memory is equal to another address stored in memory. In this particular case, the two integers would come from the same place memory, as would the two addresses, so a smart compiler might recognize that the correctness condition is always true and optimize the test away.

   Under alternative A (the one similar to C++), a naive analysis of the types in that reference will show them to be safe, so no safety test would be necessary.

Let us now extend this type (and these operations) to a 3D vector type. Under alternative B (like Ada 95), this is easy:

```
TYPE,EXTENDS(vector_2d)::vector_3d
  REAL::z
REPLACES
  length=>length_3d
  corr=>corr_3d
END TYPE
…
FUNCTION length_3d(v) RESULT(length)
  TYPE(vector_3d)::v; REAL:: length
  length=sqrt(v%x**2+v%y**2+v%z**2)
END FUNCTION length_3d
FUNCTION corr_3d(v1,v2) RESULT(corr)
  TYPE(vector_3d)::v1,v2; REAL::corr
  corr=(v1%x*v2%x+v1%y+v2%y+v1%z*v2%z)/(v1%length()*v2%length())
END FUNCTION corr_3d
```

The polymorphic procedure print_corr_table can operate just as efficiently on an array of type vector_3d as it does on one of type vector_2d.

Under alternative A (like C++), things do not go so smoothly. The required type for v2 in our vector_3d version of corr would be vector_2d. We can force a type match with this type, but we would then not have access to the z component of v2, and we need that access to compute the right formula. To get around this limitation, we must go back to corr_2d and change the declaration of v2 from TYPE(vector_2d) to OBJECT(vector_2d), permitting us to associate a vector_3d actual argument without losing access to its additional component. If we made no further changes to corr_2d, we would have changed the reference v2%length() from something that can be resolved at compile-time to something requiring dynamic dispatch, but we can fix that problem by rewriting that reference as v2%vector_2d%length(). The text for corr_3d cannot corrected so easily. It is still not syntactically correct to reference v2%z. We must first coerce v2 from OBJECT(vector_2d) to vector_3d. One way to do that would be as follows:

```
FUNCTION corr_3d(v1,v2) RESULT(corr)
  TYPE(vector_3d)::v1; OBJECT(vector_2d)::v2; REAL::corr
  corr=corr_3d_helper(v1,v2)
END FUNCTION corr_3d
FUNCTION corr_3d_helper(v1,v2) RESULT(corr)
  TYPE(vector_3d)::v1,v2; REAL::corr
  corr=(v1%x*v2%x+v1%y+v2%y+v1%z*v2%z)/(v1%length()*v2%length())
END FUNCTION corr_3d_helper
```

The costs here are an extra level of procedure reference and a safety check that the OBJECT(vector_2d) v2 in corr_3d can be correctly associated with the TYPE(vector_3d) v2 in corr_3d_helper. Presumably, a smart compiler could eliminate the cost of the extra level of procedure reference by inlining corr_3d_helper in corr_3d.

Let us summarize the differences between the alternative A (like C++) and alternative B (like Ada 95) versions of this example:

1. They involved a similar number of safety checks, albeit at different levels. In this particular example, the alternative B safety checks could be optimized away, but I am certain there would be many variants where this would not be the case.

2. The direct implementation costs for alternative A are slightly higher (because of the need to coerce the OBJECT(vector_2d) to TYPE(vector_3d)), but it should be possible optimize away those extra costs.

3. The alternative A version is a bit more verbose (more work to write).

The reason alternative B is less expensive for this example is that the natural extension of our operation between two 2D vectors is an operation between two 3D vectors. If we had chosen an operation whose natural extension was an operation between a 3D vector and 2D vector, alternative A would have been cheaper and less verbose. (I haven't been able to think of such an example, but maybe you can.)

Thus, it would appear that the following questions are relevant in deciding among the alternatives:

- How important is it that Fortran the example of well-known languages like C++ in this regard?

    * What fraction of Fortran programmers will know these languages?

    * Of them, how many will know these particular details?

  [I suspect that relatively few people that have learned C++ or Java well will choose to switch to Fortran for doing object-oriented programming, so I feel following their lead is not particularly important in avoiding surprise.]

- What proportion of real programs are like our example, better served by alternative B?

  [I have been unable to think of a real example that wasn't better served by alternative B.]

- How significant are the differences in implementation speed and program verbosity?

  [The implementation speed differences are small enough to be ignored in nearly all cases. It is the differences in verbosity that disturb me.]

Ω