Public Review Comment #8 by Rob James

To: J3
From: Craig Dedo
Date: November 11, 2002
Subject: Public Review Comment #8 by Rob James

Comment on the Fortran 200x Committee Draft
==
Rob James
==

Contents:

1. The magical "same value" behaviour of some intrinsics
2. The POS= specifier in a data transfer statement
3. Allocatable result variables and ENTRY
4. Function prefixes and specification inquiries
5. Problems with Type Aliases
6. Additional constants for the ISO_C_BINDING module
7. IOSTAT_END should be split into two separate constants
8. IEEE issues
9. Specifying "private" in an extends clause
10. A plea for the poor, deferred bindings

==
1. The magical "same value" behaviour of some intrinsics

There are problems with passing structures that have allocatable components to some of the transformational intrinsics defined in section 13. The affected intrinsics are CSHIFT, MERGE, PACK, RESHAPE, SPREAD, and UNPACK.

In the "Result Value" section of the descriptions of each of these intrinsics, it says that result either "has the same value as", or simply "is", an argument (or a subobject of an argument). There is no explanation of how the result gets this value, or how it could simply be the same object as the argument (which seems blatantly wrong), as the case may be. It just seems to be this way by The Magic of Fortran.

The problem is that, for structure components that are allocatable, the simple bitwise copy semantics that are implied by these statements are simply wrong. Intrinsic assignment, on the other hand, is well defined for structures containing allocatable components. If it could be said that they are copied as if by intrinsic assignment, that would cover this situation.

At first glance, saying they get the values "as if by intrinsic assignment" seems alright, at least for structure components that are allocatable. But is this really what is intended, especially in light of type-bound defined assignments that could be called during an intrinsic assignment? Depending on the semantics of those type-bound defined assignments, the result may not really have the "same value" as the argument.

However, if that's alright, then the descriptions of the result values of these intrinsics can simply be changed to say that it is as if an intrinsic assignment had taken place. Otherwise, I think structures that have allocatable components need to be prohibited, as well as structures that have components whose types have a type-bound defined assignment. Either way, the bitwise semantics that are implied by these descriptions are just wrong.

===

2. The POS= specifier in a data transfer statement

In 9.5.1.10, it is said (at page 190, lines 14-15 of the Committee Draft) that the POS= specifier "may appear only in an input/output statement that specifies a unit connected for stream access". This is not quite true.

It's correct that the POS= specifier may appear in a data transfer statement that specifies a unit connected for stream access, and that it may not appear in any other data transfer statement. However, it may also appear in an INQUIRE statement, whether or not that INQUIRE statement specifies a unit connected for stream access.

The intended effect could be achieved by changing the sentence in 9.5.1.10 that was quoted above to say the following:

> This specifier may not appear in a data transfer statement that specifies a unit connected for sequential or direct access.

===

3. Allocatable result variables and ENTRY

The following code seems to be allowed when it probably shouldn't be:

```
FUNCTION foo()
INTEGER, ALLOCATABLE :: foo
INTEGER :: bar
...
ENTRY bar()
...
END FUNCTION
```

Page 279, lines 18-22 of the Committee Draft state:

> If the characteristics of the result of the function named in the ENTRY statement are the same as the characteristics of the result of the function named in the FUNCTION statement, their

> result variables identify the same variable, although their names need not be the same. Otherwise, they are storage associated and shall be scalars without the POINTER attribute and one of the types: default integer, default real, double precision real, default complex, or default logical.

The code above fits into the "otherwise" situation. So, they must be scalars (yes, they are) without the POINTER attribute (no POINTER attribute here), and one of the types mentioned (default integer is in the list). Both foo and bar fit those criteria and are storage associated. But, for foo, what storage are we talking about? Certainly not the dope vector. Does it mean that the storage allocated for foo needs to be the same as the storage for bar? Does it mean that they can be separate, but need to be kept synchronized with each other (i.e. the same value at all times)?

All of these suggestions are terrible. And besides, wasn't this the reason behind prohibiting the POINTER attribute in this situation? The only acceptable solution is to prevent these function results from having the ALLOCATABLE attribute as well as the POINTER attribute. This can be achieved by changing "scalars without the POINTER attribute" to "scalars that do not have the ALLOCATABLE or POINTER attribute" on page 279, line 21 of the Committee Draft.

===
4. Function prefixes and specification inquiries

The following code looks like it's allowed, according to the Committee Draft:

```
type(dt(1)) function f()
integer, parameter :: i = f%x
type :: dt(x)
integer, kind :: x
...
end type
...
end function
```

The initializer for i (f%x) appears to be a perfectly fine specification expression, at least according to 7.1.7 (Initialization expression). However, the type and type parameters of f haven't even been defined yet. Sure, they're specified previously, in the function statement, and that's all that seems to be required.

In Fortran 95, there didn't need to be a requirement that the type of a variable needed to be defined. The only types that had type parameters were intrinsic types, and therefore always defined. With the arrival of derived type parameters in Fortran 200x, however, the type of an object whose type parameter is being inquired about could be defined later, if the type and type parameters of the object are specified on a function statement.

The same problem gets some nastier symptoms if you use a type alias, rather than a parameterized derived type. Consider the following code fragment, which again looks legal according to the Committee Draft:

```
type(ta) function f()
integer, parameter :: i = f%kind
typealias :: ta => integer(i)
...
end function
```

From a legal type parameter inquiry, we get a nasty little circular definition.

1. The type and type parameters of f are those aliased by the type alias ta.
2. The type aliased by ta is integer with kind type parameter value equal to the value of i.
3. The value of i is the kind type parameter of f.
4. Go to 1.

This problem can be fixed in one of three ways:

1. Fix the description of specification inquiries to prohibit these situations.
2. Fix the description of type parameter inquiries to prohibit these situations.
3. Fix 5.1.1.7 to say that, if a derived type is specified on a function statement and the derived type is defined within the body of the function, the effect is the same as if the result variable was specified with that derived type in a type declaration statement appearing immediately after the derived type definition. Something analogous would also need to be added for type aliases.

Personally, I prefer option 3, as it seems a bit cleaner and it specifies something that Fortran 95 did not: the exact effect of specifying a derived type in a function prefix when that derived type is defined inside the body of the function.

==

5. Problems with Type Aliases

Type alias names and derived type names should behave essentially the same in most situations. They are both just names of types. They are both used in the same way to declare objects. They should abide by the same rules, as far as declaring an entity goes. Right now, they don't.

Type Alias Problem #1:

Type aliases may not be extended. Don't say otherwise.

Note 4.45 (page 54, after line 5 in the Committee Draft) says that the name of a parent type might be a type alias name. This was true in a previous draft, but not in the Committee Draft. The words "a <type-alias> name or" should be removed from this note.

Type Alias Problem #2:

It's not clear, except from an example in a note, that a <declaration-type-spec> in a TYPEALIAS statement can specify a type alias.

Sure, I suppose that if a <declaration-type-spec> in a TYPEALIAS statement specifies a type alias, it is really specifying the type and type parameters being aliased by that type alias. After all, it does say below that declaration of an entity using a type alias name has that effect (page 61, lines 12-13 of the Committee Draft). But it would really make things more clear if type alias names were explicitly allowed in this case.

I suggest that constraint C480 (page 61, lines 10-11 of the Committee Draft) be changed to the following:

> (R453) A <declaration-type-spec> shall specify an intrinsic type, a previously defined derived type, or a previously defined type alias.

This also covers situations like the following:

```
TYPE :: dt
...
END TYPE
TYPEALIAS :: ta => TYPE(ta2)
TYPEALIAS :: ta2 => TYPE(dt)
```

Without this change to the constraint, there is nothing prohibiting the user from specifying a type alias that is defined later in that scoping unit (there is such a rule for derived types).

Type Alias Problem #3

If a derived type is specified in a function prefix, it can be defined inside the function. Can the same be done with type aliases?

I would think the same could be done with type aliases. The syntax looks the same for derived types and type aliases on a function prefix. One would think that if you specify TYPE(NAME) in a function prefix, then NAME could be legally defined inside the function, whether it's a derived type or a type alias.

As it stands, there is nothing that says this is allowed for type aliases. I suggest that 5.1.1.7 be renamed to "TYPE" rather than "Derived type". 5.1.1 is called "Type specifiers". "TYPE" is the type specifier for derived types and type aliases. (In a similar vein, 5.1.1.8 should be renamed to "CLASS", because it talks about the CLASS type specifier.) If this is done, something about type aliases could easily be added into 5.1.1.7 (such text doesn't already exist in this chapter). The part about derived types that are specified on FUNCTION statements could be expanded to include type aliases.

===

6. Additional constants for the ISO_C_BINDING module

The C_INT_LEAST*_T, C_INT_FAST*_T, and C_INTMAX_T constants that are provided in the ISO_C_BINDING module are important, but there are other C99 integral types provided by stdint.h that are not accounted for in this module.

The types int8_t, int16_t, int32_t, and int64_t will probably be more widely used than their "least" and "fast" counterparts, because they provide exactly the number of bits requested. Admittedly, these types need not exist in a given C99 implementation, but they are still very useful and desirable.

There is also one other signed integer type provided in stdint.h: intptr_t. This type, like the int*_t types, does not need to exist in a C99 implementation. It could be useful to provide a constant in the ISO_C_BINDING module that corresponds to this type as well. Aside from being useful, it would make this module more complete, in that the module would provide constants corresponding to all of the signed integral types provided by C99's stdint.h.

The values of these additional constants could be -2 if the C companion processor does not define the types corresponding to these constants (as opposed to -1, which would mean that the type is defined by the companion C processor, but there is no corresponding Fortran integer kind).

===

7. IOSTAT_END should be split into two separate constants

The IOSTAT_END constant in the ISO_FORTRAN_ENV module is a useful thing, but requiring that a processor can only have one IOSTAT value for end-of-file is overly restrictive. It invalidates one implementation I can think of (gee, I wonder whose implementation that could be).

In order to avoid invalidating this mystery company's implementation, two things could be done. The first option (the one I'm obviously in favour of) is to split this constant into two separate ones: one for the end-of-file condition for external files, and one for the end-of-file condition for internal files.

The second option is to simply remove the constant from the module.

<guilt-trip>
Of course, there is implicitly a third option, forcing Company X's Fortran runtime to return a different value than it previously did for some end-of-file conditions in order to be compliant with Fortran 200x.
</guilt-trip>

===
8. IEEE issues

IEEE issue #1: IEEE_SELECTED_REAL_KIND

The example for IEEE_SELECTED_REAL_KIND is incorrect. I believe that it was blindly copied from the description of SELECTED_REAL_KIND, where it was also stated that this was only true for a particular hexadecimal real model, which is obviously not an IEEE-compliant model. Changing the second argument in the example to a nice safe number like 30 will fix this example.

IEEE issue #2: What is "normal execution"?

Page 358, lines 2-5 of the Committee Draft specify what happens "if an intrinsic procedure or a procedure defined in an intrinsic module executes normally". This is far too vague and could be interpreted in many ways.

Does "executes normally" mean that there was no hardware failure during execution of the procedure? It could be interpreted this way, even if it is a bit extreme. If there is no hardware failure, then even if you pass arguments that are nonsensical for that procedure (for example, ieee_rem(0.0, 0.0)), it is doing what could be considered "normal" for those arguments.

Does "executes normally" mean that it produces a value that is mathematically valid? This would narrow the scope of the statement and its prescribed behaviour to an acceptable point.

Or does "executes normally" mean that it produces an exact representation of a mathematically valid value? This narrows the scope too much, as almost no floating-point operation would execute "normally".

I believe that my second interpretation was probably the one that was intended. But whether it is or not, the "executes normally" bit is far too vague to provide for any consistent interpretation.

==

9. Specifying "private" in an extends clause

If a parent component is specified as private in an extends clause, what is the effect? This is never explained in the normative text of the Committee Draft.

Given the example in C.1.3, it appears that the parent component and all components inheritance associated with it are private in the resulting type. This is never said anywhere in the normative text.

Even if this is fixed, there is the problem of type bound procedures. Are these all private in the resulting type unless they are overridden and declared to be public? If this is the case, and if a type bound procedure that was declared as public in the parent type is not overridden and declared to be public in the child type, then this makes the type bound procedure private in the child type, which is exactly what page 55, line 18 was trying to prevent ("If the inherited binding is PUBLIC then the overriding binding shall not be PRIVATE.").

In light of the fact that this feature is very poorly explained, maybe it would be best to remove it from this revision of Fortran.

==

10. A plea for the poor, deferred bindings

In the last J3/WG5 meeting before the Committee Draft was produced, deferred bindings were removed. Deferred type-bound procedure bindings and their non-Fortran equivalents (pure virtual member functions in C++, abstract methods in Java, etc.) are an important part of an object-oriented programmer's code. The loss of this feature is quite a large pain for such programmers.

It's true that the effect that deferred bindings have could be achieved by specifying an actual procedure for a deferred binding, and having the procedure issue an error message and stop the program. That will work, but it's hardly as self-documenting as deferred bindings were. Not to mention that OO purists are likely to shake their heads in disapproval and develop migraines at the sight of such a workaround. As a friend of mine once said, when it comes to Object-Oriented Programming, do it right or do it in C.

Of course, that's not to say that there were no problems with deferred type-bound procedure bindings as they were before they were removed. The syntax was unsightly (ungodly, some might say) and unintuitive. Although I personally liked the syntax because of its parallel to C++, I do concede that it was not right for Fortran.

Also, the fact that you could attempt to invoke a procedure through a binding that was deferred in the dynamic type of an object was something that I didn't like one bit. In (for example) C++ or Java, if you attempt to invoke a virtual function/ nonfinal method and the compiler doesn't complain, then you know that the function/method exists for the dynamic type of the object, no matter what. That's because you can't instantiate an object of a type that contains a pure virtual function/abstract method. Something similar should really be done in Fortran, if only to keep users from shooting themselves in the foot. Assuming, of course, that deferred bindings make it back into Fortran 200x or the next revision.

References

02-007r3, Fortran 2000 Committee Draft

[End of J3 / 02-319]