Subject: Comments on C interoperability
References: 98-170r1
From: Van Snyder

There is no pretense offered that remarks in this paper constitute complete edits necessary to correct problems or answer questions noted here. Some of the alleged problems may not be problems at all. Some of them should perhaps turn into unresolved issues.

Page and line numbers refer to 99-007r1.

The constraint "If BIND(C) is present, there shall be no *proc-component-def-stmts* in the type definition" compromises the usability of C interoperability. It is not unusual in C programs that *structs* have "pointer to function" components. In contrast, I don't see that *proc-component-def-stmts* cause any trouble for C interoperability, so why prohibit them? Maybe we need a BIND(C) attribute for procedure pointers, and a constraint that BIND(C) pointers can only be pointed at BIND(C) procedures, and non-BIND(C) pointers can only be pointed at non-BIND(C) procedures, for it all to hang together. The ability to interface to the X-windows system has been suggested as a "success test" for C interoperability, but interfacing to the X-windows system *requires* the ability to put "pointer to function" components in *structs* (see the *typedef* for the *struct* `XImage` in `Xlib.h`.) One of the reasons offered for not using POINTER(C) or POINTER,BIND(C) for C pointers was that pointer-to-void is difficult to represent (see remarks concerning 407:28 below). There is no similar problem for procedure pointers. Procedure pointers are allowed to have an empty interface, which means they can have either functions of any result type or subroutines as targets. 42:20-21

The fact that a type alias *does not* define a new type will cause mutability and portability problems when objects used for generic resolution are defined by using type aliases. The reason for using type aliases in C programs is usually *not* for the purpose of "hiding" the type, as asserted at 63:23, but rather to make programs more mutable. Due to defects in the C language, it is usually necessary to change the declaration of an object for portability reasons, for example from "int" to "long int". The purpose of type aliases is to allow to make this change in one place, instead of numerous places (consider the ubiquity of the Unix type "time_t".) Presumably, the same reasons will arise in writing interfaces to C programs. If a type alias does not define a new type, and one is required to change the kind parameter of a C-interoperable type from, say, C_SHORT to C_LONG, and the kind parameter C_LONG corresponds to a Fortran kind, generic resolutions may have surprisingly different results, or fail altogether. If type aliases are used in the declaration of procedure interfaces, generic sets of interfaces that are unambiguous may become ambiguous. The fact that type aliases do not create new types in C is not a problem in C because C does not have generic procedures. 63:21-23

If there's more to this constraint than just saying the length shall be 1 (see unresolved issue 89), then "assumed" needs to be "assumed or deferred". 72:1-4

I don't see any problem with a dummy argument that has the VALUE attribute becoming associated with a pending I/O sequence. There may, however, be a problem if the associated actual argument is associated with a pending I/O sequence – in that case, the copy-in takes place at a time not well-determined with respect to the progress of the asynchronous I/O. Similar considerations apply to a VOLATILE actual argument associated to a VALUE dummy argument. 72:19-22

Why put "that has the BIND(C) attribute" in normative text? That is, what's wrong with allowing the BIND(C) attribute for non-BIND(C) procedures? It ought to be enough to put it after the word "argument" at 85:42. See also unresolved issue 87. 85:26

If the VALUE attribute is allowed for non-BIND(C) procedures, it would be useful that the VALUE attribute *does not* imply all of the properties of INTENT(IN). In particular, it should be OK to change a VALUE dummy argument, but that change is not reflected in the associated actual argument. This would also require changes to definition of argument association in the VALUE case, because changes in the dummy argument would have no effect on the corresponding (not associated) actual argument.

In any case, add VALUE to the index.

Add VALUE to the index. 92:34

We probably need the VALUE attribute in this list. 267:20-29

Allowing C_PTR to be a type alias is an invitation for portability problems, especially if used for generic resolution. Yet another reason type aliases ought to introduce new types, not new names for existing ones. 404:3

Couldn't we provide a means to convert between C pointers and Fortran pointers? E.g. 407:28

```
F_pointer => TRANSFER(C_pointer, <mold>)
```

together with some extensions of RESHAPE, and a new intrinsic, say

```
C_pointer = Transfer_To_C (F_pointer)
```

?

The latter would strip off everything that can't be represented in a C pointer – that is, everything but the base address. I continue to prefer the combination of BIND(C) and POINTER attributes, preferably spelled POINTER(C), as advocated in 98-170r1. As I understand it, the only problem was C's pointer-to-void. Could this be finessed using an intrinsic derived type (not object) C_VOID, together with the above suggested extension of TRANSFER? E.g.

```
TYPE(C_VOID), POINTER(C) :: FOO => NULL() ! A C pointer, initially null
TYPE(MY_C_STRUCT), TARGET :: BAR          ! A C struct that can be a target
TYPE(MY_C_STRUCT), POINTER(C) :: BAZ      ! A C pointer
FOO => TRANSFER ( BAR, FOO )              ! C pointer assignment
BAZ => BAR                                ! C pointer assignment
NULLIFY ( FOO )                           ! instead of FOO = C_NULL
FOO => NULL()                             ! ditto
```

If so the `Transfer_To_C` intrinsic wouldn't be needed, nor would the C_PTR type, the VALUE attribute, the LOC intrinsic function, or the C_NULL object be needed, as remarked in 98-170r1.

I can't figure out what the word "original" does in these two places. 409:33,34