Date: 4 February 1999
To: J3
From: Van Snyder
Subject: Comments and questions concerning 99-007
References: 98-221, 99-007, 99-105, 99-106

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.

The preference to use syntax terms instead of descriptive names begs for a way to get them into the index.

Is the meaning of the phrase "does not imply that the object has the SAVE attribute" explained somewhere? In the absence of default initialization, the distinction between having and not having the SAVE attribute is clear. What does the absence of the SAVE attribute imply in the context of default initialization? Does it mean that the object is re-initialized every time it comes into existence? (I think so.) Or does it mean that the object has the value given by initialization the first time it comes into scope, and it's indeterminate thereafter? It should be spelled out. If it is spelled out somewhere, a cross-reference at this point would help the reader. 39:16

It would clarify later discussions, that will be noted below, if R422 were changed to 39:18-23

R422 *derived-type-def* **is** *derived-type-stmt*
[*data-component-part*]
[*type-bound-procedure-part*]
end-type-stmt

R422A *data-component-part* **is** [*private-sequence-stmt*] ...
[*type-param-def-stmt*] ...
[*component-def-stmt*] ...

It would also help one remember that type parameters can't have accessibility control if the *private-sequence-stmts* were put after the *type-param-def-stmts*

It would more clearly express the intent if "CONTAINS" were replaced by *type-bound-procedure-part*. 39:46

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 common 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 must be pointed at BIND(C) procedures, and non-BIND(C) pointers must 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`.) 40:15-16

Syntax rule R429 is improperly type-set. 40:25-26

Using the definition of D-expression given in discussion of 40:46-47, add the constraints "The character length specified by the *char-selector* in a *restricted-type-spec* (5.1, 5.1.1.5) in a *data-component-def-stmt* shall be a D-expression" and "A *type-param-value* used in a *derived-type-spec* in a *data-component-def-stmt* shall be a D-expression."

The constraint "Each bound of an *explicit-shape-spec* shall be a constant specification expression (7.1.6)" essentially nullifies the value of parameterized data types. It also conflicts with 43:11-13. It must surely be an oversight that this has not been liberalized. (By the way, a "constant specification expression" seems to be no different from a "constant expression" – i.e. (123:36) "A **constant specification expression** is a constant expression that is also a specification expression." I can't find anything about specification expressions that constrains constant expressions any more than they are by their normal definition.) It appears that we need a term for an expression that consists of constants and type parameter names, perhaps with the restriction on the exponentiation operator that appears in the definition of initialization expressions. "Constant expression" and "initialization expression" are clearly too restrictive, while "specification expression" would allow accessing dummy and common variables directly from within a type declaration. It would be better to require them to be accessed by the correspondence between type parameter values and type parameter names. For want of thinking of a better name quickly, I will call expressions with the appropriate restrictions and privileges "D-expressions." Then, the constraint should be rephrased "Each bound of an *explicit-shape-spec* shall be a D-expression." 40:46-47

Using the definition of D-expression given in discussion of 40:46-47, replace the constraint by "The character length specified by the *char-length* in a *component-decl* shall be a D-expression." 41:9-14

A similar constraint is needed for *type-param-values* in *restricted-type-specs* within derived type declarations.

"nonpointer dummy variable" → "nonpointer nonallocatable dummy argument". 41:33

If the syntax of *proc-binding* were changed to 42:4-6, 14-19, 21-24

R439 *proc-binding* **is** PROCEDURE [[, *binding-attr-list*] ::] ■
■ *binding-name* [=> *procedure-name*]
or PROCEDURE [([*proc-interface-name*])] ■
■ [[, *binding-attr-list*] ::] ■
■ *binding-name* => NULL()

Constraint: The procedure name specified by *procedure-name* or *binding-name* shall be the name of an accessible module procedure or external procedure that has an explicit interface. If PASS_OBJ is specified, it shall have a scalar nonpointer nonallocatable dummy argument of type *type-name*. The first such dummy argument is called the passed-object dummy argument and shall be polymorphic if and only if *type-name* is extensible.

Constraint: The *proc-interface-name* shall be specified if and only if the NULL() binding is not overriding (4.5.3.2) an inherited (4.5.3.1) binding.

Constraint: The *proc-interface-name* shall be the name of an accessible abstract explicit interface. If *proc-interface-name* and PASS_OBJ are both specified, *proc-interface-name* shall have a scalar nonpointer nonallocatable dummy argument of type *type-name*. The first such dummy argument is called the passed-object dummy argument and shall be polymorphic if and only if *type-name* is extensible.

then the amazing requirement to define a procedure only for the purpose of using it in a NULL intrinsic in order to declare that there is no procedure to which to bind the *proc-binding* would be eliminated. This form would also be more like the usual procedure declaration statement.

It would also be necessary to change *binding* to *procedure-name* at line 6, to eliminate R441, and to delete the constraints at 42:14-19 and 42:21-24 (they've been moved and reworded).

If the change suggested for 42:4-6, 14-19, and 21-24 is not accepted, "procedure or abstract interface" needs to be "procedure or *proc-entity-name*" and "nonpointer dummy variable" needs 42:16-19

to be "nonpointer nonallocatable dummy argument".

We probably need a constraint that all of the nonkind parameters of the passed-object dummy argument shall be assumed, and if the passed-object dummy argument has kind parameters the *proc-binding* shall appear within a SELECT KIND construct (or constructs) that account for all of the kind parameters. These problems could be avoided if it were allowed to declare type-bound procedures within the body of the type (see 98-221), so that the argument declarations and procedure body have access to the parameters – and then SELECT KIND wouldn't be necessary.	42:11+
Note 4.19 would be more clearly explanatory if placed at 41:35+.	42:35-37
The word "parent" is almost certainly wrong here.	43:41-43
The sentence "The *proc-entity-name* ... parent type" should be deleted, because it's covered by the constraint at 42:23-24 (although there's no explanation for the reason for the constraint at that point).	47:14-16
The editor previously objected to the construction "The default is A but it may be changed to B." A construction parallel to 47:31-34 would presumably be less objectionable.	47:22-23
The phrase "If a type ... *private-sequence-stmt*" could be more clearly written "If the *data-component-part* of a type definition statement contains a PRIVATE statement" if the change suggested for 39:18-23 were made.	47:31
It would be clearer to use a construction parallel to 47:31-34.	47:40-42
The phrase "PRIVATE statement that is a *private-sequence-stmt*" could be more clearly written "PRIVATE statement in a *data-component-part*" if the change suggested for 39:18-23 were made.	48:5-6
"inany" should be "in any".	48:44
Is it a problem to allow a dummy pointer with intent(OUT) to have an assumed type parameter. Maybe not: The corresponding parameter of the associated actual argument can't be deferred (if edits from 99-106 are accepted).	53:4ff
The second alternative for R466, *viz.* **or** *type-alias-name* appears to be redundant to 61:41. Is it needed in both places?	53:18
The phrase "that is an alias for a derived type" significantly cripples the usability of type aliases for C interoperability. I could not find a similar constraint in section 5.	53:14-15
Is it necessary to impose a restriction on what assumed parameters can be used for? For example, is it OK to assume the value of a nonkind parameter that is used to provide an array dimension other than the last one?	53:34+
The fact that a type alias *does not* define a new type will cause mutability and probably 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 line 40, 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	55:38-40

fail altogether. If type aliases are used in the declaration of procedure interfaces, generic sets of interfaces that are unambiguous may become ambiguous.

Given the sentence "Each value is converted to the type and type parameters of the *array-constructor* in accordance with the rules of intrinsic assignment," why is kind type parameter equality required? What is "(724)"? Should it be (7.5.1.5)? — 58:30-31

Replace "used" by "restricted to use in" at line 25, and delete lines 32-33. — 61:25, 32-33

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". — 65:8-11

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 VOLATILE. — 65:25-29

The sentence should begin "In a type declaration statement, a...". In line 40, "specification expression" should be "type declaration statement". — 65:38,40

The note suggests that a dummy function can have a result with a character length parameter of "*". This is not currently allowed by normative text at lines 14-25, which are specified at line 13 to be the only ways allowed. Also see remarks at 78:30ff. — 68:27-28

The phrase "or DEALLOCATE (6.5.3)" should probably be removed. I don't see how a polymorphic object can acquire a concrete type by execution of a DEALLOCATE statement. — 69:26-27

Why put "that has the BIND(C) attribute" in normative text? It ought to be enough to put it after the word "argument" at line 28. — 78:12

"when" → "if" or "where". — 78:27

Is it permitted for the result type of a dummy function or dummy function procedure pointer to have an assumed type parameter? Allowing an asterisk for the length parameter of the character type of a dummy function is printed in small type at 67:44-45. If we go so far as to allow a dummy function or dummy function pointer result type to have a type parameter that is a specification expression (not just an initialization expression), I don't see any extra difficulty in the additional step of allowing the result type to be declared to have an assumed type parameter, so long as it is spelled out that the parameter value is assumed from the associated actual argument's function declaration, not somehow assumed into the function associated as an actual argument from the context of its invocation. If it's OK, we also need a constraint at or near 79:10+: — 78:30ff

Constraint: If *proc-interface* is present and declares the declared procedure pointers to be functions, and the result type has an assumed type parameter, the functions shall be dummy procedures or dummy procedure pointers.

I agree that the result of a non-dummy function or function procedure pointer should not be permitted to have an assumed nonkind parameter, except for the "grandfathered" case of character length. This is different, however, from the case of deferred type parameters of a function result, which indicate that the function sets those parameters – this is not a problem because such a result is required to be allocatable or a pointer.

The same remarks apply to assumed shape of the result of a dummy function. That is, if we allow it, the shape is assumed from the associated actual argument, not somehow assumed into the function when it is invoked.

Set "procedure pointer" in bold face, and add it to the index. — 78:31

The presence of any of the attributes mentioned in the constraint at lines 1-2 implies that the *proc-entity* in some sense behaves more like data, and therefore they don't make sense for external or dummy procedures. I don't know precisely the words to say that, but that's the reason for the constraint. 79:2+

The phrase "except in the scoping unit of the main program" appears to be prohibiting to specify the SAVE attribute for a common block in the main program, while I think the real intent is that it's not necessary to specify the SAVE attribute for a common block in the main program. 81:38

Do we allow type parameters in IMPLICIT that depend on, say, dummy arguments? I.e., is the following allowed? 86:42

```
subroutine SUB ( N )
  implicit integer(N), my_type(N)(A-C)
```

Does it make sense to inquire about nondeferred parameters of the result type of a function procedure pointer or dummy function? If we allow assumed parameters for the result type of a dummy function procedure pointer or dummy function (see remark at 78:30ff above), it would at least be useful to inquire about them, so why not allow the general case. 100:13+

In many other languages, array bounds are what we call "nonkind type parameters." I suggest that we bundle together what we now call type, parameters, and array bounds, and call that something like "complete type". Use something like "element type" for the type and type parameters of an array, if we ever need to discuss them while excluding the array bounds. 119:1-8

"character lengths (R510)" should be "nonkind type parameters". 121:18

"or type parameter" is included in the list at 122:1-2, but not here. Is this a problem? 123:29-39

"or type parameter" is included in the list at 122:1-2, but not here. Is this a problem? 124:32-33

It should be specified whether intrinsic or defined assignment is used when objects of derived type appear in input lists in READ statements, or after *name*= in namelist input, or for purposes of the implied copy if a dummy argument has the VALUE attribute. 7.5.1.3

Do we need to say anything special about passed-object dummy arguments? 141:2+

Should type parameters participate in type selection? It seems more reasonable that they should than that they shouldn't. If a type guard is TYPE IS, and the type (sans parameters) is the same as the type of the *type-selector*, all the parameters are available for testing. Similarly, If a type guard is TYPE IN, and the type is a descendant type of the declared type of the *type-selector*, all the parameters are available for testing. Furthermore, allowing arbitrary combinations of type parameters in the type guard statements gives a form of conditional compilation, viz.: 156:31-33

```
SELECT TYPE ( FOO )
TYPE IS ( DOUBLE PRECISION )
  special double precision code -- not compiled if FOO is REAL
TYPE IS ( REAL )
  special default real code -- not compiled if FOO is DOUBLE PRECISION
END SELECT
```

If we do specify kind parameters, we will need a place-holder for nonkind parameters. I suggest either asterisk or colon.

Remove one "the". 157:8

What are the type parameters of the *associate-name*?	157:24-31
It would be nice to be able to change between sequential and stream access without closing the file, at least for asterisk units.	167:20-21
"is" should be "may be". Otherwise, if sentences that begin "for example" are normative, this sentence states it is always possible to reposition stream files, and always possible to write – both of which contradict statements elsewhere. We also need to take care at line 29 to say that "any file storage unit may be read" providing the file was not opened with `ACTION='write'`.	168:28
The PAD= specifier can be specified in an OPEN statement, but not in a READ statement. It therefore can't be specified for standard input. The most efficient way to discover the length of an input record is to specify `PAD="no"`, `SIZE=`*var* (unfortunately, the latter requires using non-advancing input – see remark at 180:10-11). The former can't be done for standard input if 99-105 is not adopted. (This is a very old problem, not introduced in 99-007.)	162:31
Is there any benefit to restricting use of `SIZE=` to the case of `ADV="no"`? (This is a very old problem, not introduced in 99-007.)	180:10-11
Replace "I/O on" by "I/O so that default rounding can be specified on".	180:30
Perhaps "not equivalent" should be "not necessarily equivalent".	186:36
The caveat "when execution of the statement begins" begs the question whether it is allowed for a user-defined derived-type input/output routine to close or reopen the file. There appears not to be a prohibition in 9.5.4.4.3. Should this be "throughout execution"?	189:36-37
Should "data transfer" be "data transfer and INQUIRE by output list"?	192:20
Begin the sentence "A preconnected file that has not been opened or an internal file...." As it is, it makes one wonder if an internal file can be opened.	223:43-44
Put "value separator" in the index.	225:10
Remove slash from the list, and call it a terminator of some kind. Then we can use "separator" instead of repeatedly spelling out "The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal edit mode is COMMA."	225:11-17
The *r* should almost certainly be *r**.	225:17
It's OK to have blanks or end-of-record between the real and imaginary parts of a complex number, but not between the numbers and the parentheses. It would simplify the description, processors, and users' preparation of data if complex numbers could be separated by comma, semicolon or blank, and if blanks could appear anywhere except within the numbers.	225:43-2
Remove "The separator ... COMMA."	225:47-1
Replace "slashes, blanks, or commas" by "separators".	226:3-4
Replace "characters blank, comma, and slash" by "separator characters".	226:12
Replace "separators blank, comma, slash" by "separator".	226:14
Replace "slash or comma" by "nonblank separator".	227:7
Replace "one or more blanks or by a comma, or a semicolon if the decimal edit mode is comma" by "separator".	227:31-32
Insert "nonblank" before "separator". (But "nonblank" ought not to be necessary – see remarks at 225:43-2.)	227:45
Remove the sentence "The separator ... COMMA."	227:46-1

Add (10.9.1.4) after "values"	230:9
Replace "comma" by "nonblank separator". Come to think of it, is there a special reason a blank wouldn't be OK? (See remarks at 225:43-2.)	230:30, 34
Replace "characters blank, comma, and slash" by "separator characters".	230:45
Replace "slashes, blanks, equals, or commas" by "separators or equals".	230:36
Replace "slash or comma, or a semicolon if the decimal edit mode is comma" by "nonblank separator".	231:31-32
Insert "nonblank" before "separator".	232:42
Delete the sentence "The separator ... COMMA."	232:43-44
Is it really necessary to have BIND(C) in this list?	244:5
Is it really necessary to have the VALUE attribute in this list?	244:15
Don't we need the VALUE attribute in this list?	245:20-29
Discussions of pointer or nonpointer probably need to be extended to include allocatable or nonallocatable. Do we need to address the question whether an allocatable dummy can be associated with a pointer actual, or vice-versa? If this is addressed elsewhere, it probably belongs in 12.4.1.2.	12.4.1.2
"may" → "shall".	257:21
There is no discussion of what a dummy argument is. Should there at least be a reference to 12.4.1?	12.5
Replace the first "or" by a comma, and add a comma before the second "or".	300:40
Replace "then" by a comma.	301:4, 11
We could just say that trailing blanks are significant. If they're not, one can always use TRIM().	301:9-14
Replace "the definition ... (4.5.1)" by "the *data-component-part* of the type definition does not include a PRIVATE statement (4.5.1)" if the change suggested for 39:18-23 is made.	340:37-38
Replace "as" by "is".	341:22
Should (v) be in the list?	364:38
Replace "IEE" by "IEEE".	371:3
Replace "COMPLES" by "COMPLEX".	378:4
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.	378:27
Replace "compatable" by "compatible with".	379:5
Couldn't we provide a means to convert between C pointers and Fortran pointers? E.g. `F_pointer => TRANSFER(C_pointer, <mold>)` together with some extensions of RESHAPE, and a new intrinsic, say `C_pointer = Transfer_To_C (F_pointer)`?	380:27
I can't figure out what the word "original" does here.	382:37
Capitalize "Fortran".	383:14