Date: February 8, 1997

To: X3J3

From: William B. Clodius

Subject: Critical Issues for Object Orientation in Fortran

1. Introduction

In recent months I have been an active, some might justifiably say too active, participant in the sc22wg5-data email discussion group. This group was intended to provide a focus for the development of ideas for object orientation. Unfortunately, the imminent cutoff date for proposals for requirements, and the crush to prepare for the February meeting led to a spate of proposals of features, without first establishing a consensus about overall goals and requirements that appear (in retrospect) to be necessary in evaluating such features.

This paper attempts to provide a basis for establishing such a consensus. It identifies what I believe are the most important issues, posed as a series of questions, and provides information in accompanying comment sections that I believe will be useful in reaching a decision on those issues. It is likely that I have missed some issues and have failed to include some pertinent information, and I encourage the data subgroup to issue a

revised version of this paper with whatever additions they believe to be useful. Note that in order to minimize prejudicing analysis of these aspects, I have tried to avoid injecting my personal opinions in the main text. However, because I am unable to attend this meeting, and I want participants in the meeting to be aware of my opinions, I have summarized my opinions in an appendix.

2. Issues in Object Orientation

In order to give an overview of the issues, this section simply divides the issues into four general categories commonly recognized as important to object oriented programming, i.e., polymorphism, dispatch, inheritance, and encapsulation, and lists the issues in the order in which they will be discussed in detail latter.

I. Polymorphism

A. What are the semantic restrictions on polymorphism that should be recognized in the language?

B. What are the sources of polymorphism that should be recognized in the language?

C. How should the existence of polymorphic relationships be identified in the language?

D. What polymorphic relationships should be allowed between intrinsic and derived types?

E. What term(s) should be used to identify polymorphic types? This issue has the related issues

1. Should the term KIND be extended to allow a more abstract usage?

2. Should the intrinsic KIND values be allowed to include derived types?

F. Can closed polymorphic relationships (relationships involving a fixed set of types) be defined?

G. What should be the semantics of polymorphic objects?

II. Dynamic Dispatch

A. Should the language include dynamic dispatch?

B. Should multiple dispatch be allowed?

C. Should dispatch only be allowed for “tagged” types?

D. How should object dispatch be determined by the language?

III. Questions involving inheritance.

A. How should an inheritance relationship be indicated?

B. What is the relationship between polymorphism and inheritance?

C. Should the language allow overriding of inherited “methods”? This has the related issues

1. In procedures that have not been overridden invoke a method that has been overridden, is the new or the overridden procedure invoked by default?

2. Does the language provide a means of overriding the default for overridden procedures?

D. Should multiple inheritance be allowed? This has the related issues

1. If multiple inheritance is allowed, how are name conflicts handled?

2. If only single inheritance is initially defined, should it be extensible later to multiple inheritance?

IV. Encapsulation

A. How should inheritance affect encapsulation?

B. Should encapsulation be made more flexible in general?

3. Comments on Issues in Object Orientation

This section examine in moderate detail the issues listed in the previous section.

I. Polymorphism

Polymorphism (sometimes termed abstraction) is one of the most important aspects of object oriented languages. In its essence it means that a variety of different types can be used in some contexts and still result in valid expressions. This is usually described in terms of the similarity of the signatures (the set of procedures and operators having entities of that type as arguments) of the polymorphic types. Because Fortran's intrinsic types, overloaded operators, and elemental operations already provide some polymorphism, the central questions is: how to extend it to derived types? This requires answering a number of questions:

A. What are the semantic restrictions on polymorphism that should be recognized in the language?

Comments on I.A

There are a number of polymorphic relationships with subtly different semantics that are recognized in the literature. Among them are:

Subtype/Supertype: A is a subtype of supertype B if in any valid expression containing entities of type B, any of those entities can be replaced by an entity of type A and still yield a valid expression.

Equivalence: Type A is equivalent to type B if A is a subtype of B and B is a subtype of A.

Matching: Type A matches type B if in any valid expression containing entities of type B, the expression will remain valid if all of those entities are replaced by an entities of type A.

Incidental: Types A and B are incidentally polymorphic if in some some valid expressions containing entities of type B remain valid after appropriate substitution of entities of type A.

Incidental polymorphism is not sufficiently well defined to warrant recognition by the language.

The differences between the other three relationships are

most noticeable when the related types include multi-methods, i.e., procedures which have more than one argument with intent(IN) whose type satisfies the relationship. By far the most important such multi-methods are binary methods such as Fortran's binary operators, ==, /=, <=, >=, >, <, +, -, *, /, **, and //. It is well recognized in the literature that the general subtype/supertype relationship has trouble systematically handing binary methods. For example, the equality comparison operation, A == B is type safe only if A and B have equivalent or the same types. As matching ensures that binary methods can only be invoked if the two arguments have the same type, binary methods are also safe under matching.

While type equivalence can handle binary methods, if binary methods are present the number of procedure definitions that are implicitly required grows in proportion to the square of the number of equivalent types. This makes equivalence among more than a few types impractical. I am not aware of any language that explicitly provides for equivalent types.

In practice therefore the question becomes should the language recognize only matching polymorphism, only subtype polymorphism, or allow both? If it allows both should it be necessary for the user to indicate which is necessary. Most object oriented languages in current use apparently recognize only subtype polymorphism, but they were designed before the distinction was well recognized and appear to have made that choice without recognizing its implications. Ada implicitly allows both, but the restrictions on the allowed arguments to

procedures for matching types is stronger than what is strictly necessary.

Note: In Fortran, the different “KIND”s of each intrinsic type are in essence equivalent types, i.e., a 32 bit REAL has a type equivalent to that of a 64 bit REAL, similarly a 32 bit INTEGER has a type equivalent to that of a 64 bit INTEGER.

Note: In the above I did not discuss covariance a type relationship that can be thought of as a less type safe version of matching that works well in practice. TO the best of my knowledge Eiffel is the only language that uses covariance.

Note: Ada requires that either only one argument (the first) may be of a tagged (dispatching) type, or all arguments must be of the same tagged type (=> matching). Slightly more flexibility can be achieved and still retain matching, if the constraint is that all arguments that are tagged must have the same type. Significantly more flexibility can be achieved if the language allows the specification of detailed matching criteria, e.g.

```
    SUBROUTINE UPDATE_B(A, NEW_B)
      TAGGED(ATYPE) :: A  ! A possible syntax for indicating
                          ! that A is a dispatching type
      MATCH(A%B) :: NEW_B ! A possible syntax for indicating
                          ! that UPDATE_B can only be called
                          ! in contexts where NEW_B is the
                          ! same type as the component B
                          ! of the type ATYPE
```

```
  A%B = NEW_B
END SUBROUTINE UPDATE_B
```

B. What are the sources of polymorphism that should be recognized in the language?

Comments on I.B.:

There are a number of sources of polymorphism. Among them are:

Inheritance (see below)

Ad Hoc - The user constructs a type so that its signature (collection of procedure and operator interfaces) is the same as that of another type, although the details of their implementations may differ significantly from one another. Already provided in Fortran by overloading. (Note a REAL and an INTEGER type might be considered ad hoc implementations of the "type" scalar with the operations, +, -, *, ==, >=, <=)

Parametric - Types share a common signature due to their common definition in terms of a parametric type.

Accidental - Means what it says. A data type MARKSMAN in the code for a video game with the associated procedures, MOVE, DRAW, POINT, is probably accidentally polymorphic with the data type POLYGON, with the associated procedures, MOVE, DRAW, POINT.

C. How should the existence of polymorphic relationships be identified in the language?

Comments on I.B.

There are several possible ways for languages to identify polymorphic relationships: not recognize them, implicitly recognize them, or provide programmers with a syntax for indicating the expected relationship (which the compiler can statically verify), or provide programmers with a syntax for indicating the expected relationship (which the compiler only can dynamically verify) connotations of this identification. There are also several possible aspects of the relationship that are of interest to the language and its implementation.

1. The semantics of the relationship: Matching vs. subtype/supertype. Need not be indicated if the relationship is implicit.

2. The source of the relationship. Need not be specified if due to inheritance or parametric polymorphism. Must be specified if ad hoc.

3. Whether the relationship results in dispatching or is statically resolved.

Finally should the recognition be considered a fundamental

aspect of the type of an entity, or an attribute of the entity.

Note Fortran uses KIND for ad hoc equivalent relationships that are resolved statically, and indicated syntactically as an aspect of the type.

D. What polymorphic relationships should be allowed between intrinsic and derived types?

Comments on I.D.

This is actually several questions.

Should users be allowed to indicate that certain derived types implement new KINDs of intrinsic types?

Should users be allowed to group the intrinsic types into additional categories, or should the language recognize the most intuitive categories?

Obvious additional categories include

Object - only assignment (and perhaps equality) are recognized.

Ordered - has assignment and the operations ==, <=, >=, >, <, /=

Number - has assignment and the operations +, -, *, ==, /=

Scalar - has assignment and the operations +, -, *, ==, <=, >=, >, <, /=

Should dynamic dispatch be allowed for relationships including intrinsic types?

Note - such relationships are special cases of ad hoc polymorphism and their definition should use the same notation.

E. What term(s) should be used to identify polymorphic types? This issue has the related issues

1. Should the term KIND be extended to allow a more abstract usage (i.e., not have an integer value)?

2. Should the intrinsic KIND values be allowed to include derived types?

Comments on I.E.

Fortran currently uses the term KIND to identify different intrinsic types that are polymorphic with respect to one another. The current specification of KIND as an integer value is prone to portability problems and is difficult to extend to derived types. Alternatives include CATEGORY, CLASS, FAMILY, GENUS, MATCH, SUBTYPE, TYPE_SET, and DISPATCH_TYPE. Extending

the usage of KIND implies complicating its definition and usage. Adding other keywords has its own disadvantages. A sophisticated polymorphic scheme may need more than one keyword.

F. Can closed polymorphic relationships (relationships involving a fixed set of types) be defined?

Comments on I.F.

Such restrictions clarify the semantics implied by the relationship and potentially yield useful optimizations, but do complicate the language.

G. What should be the semantics of polymorphic objects?

Comments on I.G.

There are a variety of commonly used semantics for polymorphic objects: as references, as values by default, or “pure functional” access.

Reference semantics implies that except when a special keyword is used (typically the keyword NEW), any use of assignment between two objects actually results in a pointer assignment. Reference semantics minimizes the amount of copying and temporaries, but has the danger that the user can accidentally modify objects, i.e.

```
A = B ! => A points at B
CALL MODIFY_B(B) ! the value associated with A is
                 ! implicitly modified
```

Values by default implies, any use of assignment between two objects must behave as though all data in the object on the right hand side of the assignment is copied to the left hand side. Pointer assignment must be specifically invoked to avoid this behavior. Values by default semantics minimizes the danger of unintentionally modifying an object, but has the potential of increasing the amount of copying and temporaries generated by the code. The user will find it awkward that the vast majority of time he will want to override the default in order to use reference semantics.

Pure functional access implies that no object may be modified by a modification of another object. This retains the safety of values by default, and usually should have close to the efficiency of reference semantics. However, there will be cases where compile time analysis is unable to ensure safety. For such cases, objects will have to have run time tags and that whenever they are modified there is the potential for a cascade of copyings that can result in difficult to understand losses of performance.

Question I.5. Should the language include “dynamic dispatch” for polymorphic objects?

Comment on Question I.5. Dispatch occurs when procedure to be invoked depends not only on the procedure's name but also on the type of at least one of the arguments to the procedure. In principle a complete inter-procedural analysis can eliminate the need for dynamic dispatch, but in practice such an analysis is impractical for all but small codes. Dynamic dispatch is then required if the amount of local information is insufficient to determine the specific type of entities. Fortran's KIND values currently define static dispatch for polymorphic objects.

My preference is to have dynamic dispatch, with an explicit notation used to indicate the arguments upon which the dispatch is based.

II. Dynamic Dispatch

Dynamic dispatch occurs if the local information is insufficient to statically determine the detailed types of all the entities that are nominally arguments to a procedure. In such a case the processor must provide a means of determining at run time the detailed types of the arguments so that the appropriate procedure can be invoked (the last part is sometime stated "so that the appropriate message can be dispatched"). Dynamic dispatch can be though of as dynamically resolved polymorphism.

Note: I have a minor quibble with this definition, as it is based on an implementation technique and not the underlying semantics. It is possible that with sufficient global analysis a compiler will statically determine the procedure to be invoked although there may be insufficient for the programmer to determine that dynamic dispatch is not used.

Note: This category involves more knowledge of the implementation details then the other categories and would greatly benefit from input from a compiler writer.

A. Should the language include dynamic dispatch?

Comments on II.A.

The main tradeoffs here seem to be between a simpler language definition and increased flexibility in its usage.

B. Should multiple dispatch be allowed?

Comments on II.B.

Multiple dispatch occurs when the specific procedure to be invoked can only be determined by determining the types of more than one of the arguments to the procedure to be invoked. Multiple dynamic dispatch increases the language's flexibility, but, if utilized, requires a great deal of additional coding on

the user's part, and also increases the dispatch cost. It is needed if multi-methods are allowed and the matching constraint is not enforced.

CLOS and a few other functional object oriented languages implement this by requiring the processor to automatically determine the appropriate dispatching. Most other object oriented languages always dispatch nominally on the type of a single object. However, C++ and a few other object oriented languages allow the programmer to directly access the run time type information in order to resolve the dispatching for the compiler. It is almost universally claimed that multi-methods of this form tend to be brittle and break encapsulation, but I have no direct experience with this capability.

Note: Fortran's overloaded procedures provide a form of multiple (static) dispatch.

C. Should dispatch only be allowed for "tagged" types?

Comments on II.C.

As I understand it tagging involves attaching type information to an object so that this information is available for dispatching and run time type inference. It is usually simplest to encode this information as an integer, so that the storage cost of tagging is typically four bytes per object.

D. How should object dispatch be determined by the language?

Comments on II.D.

There are at least two possible means of determining dispatch for an object: either the object of a given type should be dispatching throughout its existence, or there should be a syntactic keyword to distinguish between contexts where the object is non-polymorphic and contexts where it is polymorphic. This appears to be solely a tradeoff between linguistic complexity and efficiency.

III. Questions involving inheritance.

Inheritance is the definition of a new type (or object) as an extension of another type (or object)

A. How should an inheritance relationship be indicated?

Comments on III.A.

This is a question of syntax that is influenced by the answers to several questions.

Should it be possible to rename or override components on inheritance? If so then it is syntactically convenient to have the keyword indicate the start of a special statement or appear at the end of a statement with a syntax similar to the USE statement.

Should it be necessary to indicate for a type that it can be inherited, but does not inherit?

B. What is the relationship between polymorphism and inheritance?

Comment on III.B.

There are several possible relationships between polymorphism and inheritance.

Dynamic (or static) polymorphic relationships are always specified by means independent of any inheritance relationship.

Dynamic (or static) polymorphic relationships can be specified by means independent of any inheritance relationship.

Dynamic (or static) polymorphic relationships are always created by an inheritance relationship.

Dynamic (or static) polymorphic relationships are optionally created by an inheritance relationship.

Dynamic (or static) polymorphic relationships are only created by an inheritance relationship.

Identifying specific polymorphic “types” with an inheritance relationship, appears to allows optimizations otherwise unavailable, i.e., knowledge that a type relationship involves an inheritance hierarchy allows the compiler to make optimizing assumptions about the layout of the type structure. but if required in general reduces the flexibility of the language. Allowing polymorphic relationships to be specified by a means independent of any inheritance relationship increases the flexibility of the typing system.

C. Should the language allow overriding of inherited “methods”?

Comments on III.C.

Most object oriented languages allow overriding of inherited “methods”. This is a useful capability, but complicates, for the user, the interactions of the inheritance tree because the degree of similarity of classes defined along the inheritance path is reduced by overriding.

The natural way to implement method overriding in Fortran is via renaming, etc. upon USE of a module containing the “type” to be inherited. In fact it appears to be very awkward to attempt

to prevent the user from overriding upon use of a module. Therefore any implementation of inheritance in Fortran requires careful definition of its interaction with the USE statement. The importance of this interaction is reduced but not eliminated if module code can be accessed by means other than USE, e.g., the child units of Ada. Should additional syntax be added to allow overriding type components? This has the related issues

1. In procedures that have not been overridden invoke a method that has been overridden, is the new or the overridden procedure invoked by default?

Comments on III.C.1.

Either case can be surprising to the user

2. Does the language provide a means of overriding the default for overridden procedures?

Comments on III.C.2.

This increases the number of aspects of inheritance that have to be learned.

D. Should multiple inheritance be allowed?

Comments on III.D.

Some languages restrict inheritance to a single line of ancestors, others allow multiple inheritance lines. Multiple inheritance is almost necessary if the inheritance hierarchy is tightly linked to the type hierarchy. Its usage is problematic in the absence of this constraint.

1. If multiple inheritance is allowed, how are name conflicts handled?

Comments on III.D.1.

The most detailed work on this problem appears to be that for Cecil. Cecil does not allow any default resolution of name and method conflicts, requires the compiler to recognize any conflicts and provides the programmer with constructs to resolve such conflicts.

2. If only single inheritance is initially defined, should it be extensible later to multiple inheritance?

IV. Encapsulation

Fortran has the minimal encapsulation capabilities required of an object oriented language. However most object oriented

languages provide additional capabilities beyond those provided by Fortran.

A. How should inheritance affect encapsulation?

Comments on IV.A.

Entities defined along the inheritance path benefit from having less restricted to a module than other entities. Object oriented languages have therefore commonly provided an additional level of accessibility between public and private either specified explicitly, e.g., C++'s protected, or implicitly, e.g., Ada's distinction between private entities defined in the "specification package" and private entities defined in the "body package," and its associated child packages.

B. Should encapsulation be made more flexible in general?

Comments on IV.A.

A number of additional levels (or modes) of encapsulation have been provided by a number of languages. The include READ_ONLY (sometimes known as INTENT(IN)), WRITE_ONLY (sometimes known as INTENT(OUT)), ONCE (functions that calculate their values once and afterwards always return the same values (useful for initialization)). Also some languages provide fine tuning of the encapsulation of the equivalent of Fortran's

derived types by letting some components be marked PRIVATE, some PUBLIC, some READ_ONLY.

Appendix: The author's opinions on the issues.

I. Polymorphism

A. What are the semantic restrictions on polymorphism that should be recognized in the language?

I would strongly encourage some form of the matching relationship simply because of the large importance of binary methods.

B. What are the sources of polymorphism that should be recognized in the language?

I say all three, Inheritance, Ad hoc, and parametric.

C. How should the existence of polymorphic relationships be

identified in the language?

Relationship	Identification
Matching & Inheritance	Implicit by default. A special notation for recognizing sophisticated forms of matching might be desirable
Subtype & Inheritance	Implicit as a special case of matching if all methods are unary, not recognized otherwise
Matching & Parameterization	See Matching & Inheritance
Subtype & Parameterization	Not recognized
Matching & Ad hoc	Must have an explicit notation
Subtype & Ad hoc	Not recognized?
Dispatching/non-dispatching	Non dispatching by default. dispatching should have a special notation. Probably best made a fundamental aspect of the type

D. What polymorphic relationships should be allowed between intrinsic and derived types?

I strongly believe that it would be useful for the users to be able to declare that a given derived type implements a form of intrinsic type. Such a capability reduces the need for the language to explicitly deal with issues such as large integers, extended precision arithmetic, and would provide a structure for dealing with special cases such as interval arithmetic.

I believe that dynamic dispatch should be allowed for relationships including intrinsic types if and only if the syntax for invoking dispatch is sufficiently obvious that any reasonable user will not invoke it by accident.

While I believe the other capabilities listed here are useful, I do not consider them to be critical.

E. What term(s) should be used to identify polymorphic types?

My (weak) preference is to extend the usage of the term KIND as this has the potential of reducing its portability problems as well providing a basis for identifying other forms of polymorphism. However this will make the text in the standard more awkward.

F. Can closed polymorphic relationships (relationships involving a fixed set of types) be defined?

I do not have strong feelings on this point.

G. What should be the semantics of polymorphic objects?

I slightly lean towards functional semantics.

II. Dynamic Dispatch

A. Should the language include dynamic dispatch?

I would say yes.

B. Should multiple dispatch be allowed?

I would say no. If matching is made part of Fortran's semantics, the no would be a strong one, if matching were not my preference would be to have a syntax that can be readily extended to multiple dispatch, but to begin with a single dispatch language. In the short term I do not consider the gain in flexibility to be worth the increase in language and processor complexity.

C. Should dispatch only be allowed for "tagged" types?

This is aspect of object orientation with which I have little familiarity. I am aware that Ada 95 uses the keyword tagged to identify polymorphic types. I know that a strongly typed object oriented language requires maintaining information with the object and tagging implies the presence of this information. I also believe that an object oriented Fortran should be strongly typed. It is not clear to me whether tagging is a generic term for maintaining this information, a special means of maintaining this information that is of interest to the compiler writer but should not be required by the standard, or has sufficient advantages that it should be implicitly or explicitly required by the standard. I hope Malcolm Cohen has sufficient information on this to allow adequate discussion of this topic.

D. How should object dispatch be determined by the language?

My belief is that in this case the increase in linguistic complexity is small, and the potential performance gains are sufficiently large that non-dispatching (non-polymorphic) objects should be allowed.

III. Questions involving inheritance.

A. How should an inheritance relationship be indicated?

While not critical for F2000 I would prefer a syntax that can be extended to allow renaming or overriding of components on inheritance. I am concerned that requiring an indication that a type can be inherited, but does not inherit may have problems similar to C++'s virtual and friends.

B. What is the relationship between polymorphism and inheritance?

My preference is to allow polymorphic relationships to be specified independent of any inheritance hierarchy, but that it be possible to indicate that the actual relationship satisfies additional structural constraints, i.e., lies within an inheritance hierarchy.

C. Should the language allow overriding of inherited "methods"?

No preference at this time.

1. In procedures that have not been overridden invoke a method that has been overridden, is the new or the overridden procedure invoked by default?

No preference at this time.

2. Does the language provide a means of overriding the default for overridden procedures?

No preference at this time.

D. Should multiple inheritance be allowed?

My preference is to not tie polymorphism strongly to inheritance. As this makes multiple inheritance much less useful (while not simultaneously reducing its problems) I would not at this time include multiple inheritance.

1. If multiple inheritance is allowed, how are name conflicts handled?

Use Cecil's methods if necessary.

2. If only single inheritance is initially defined, should it be extensible later to multiple inheritance?

I see no reason to rule it out for the future and would encourage the choice of a syntax that could be extended later to allow multiple inheritance. This mostly implies a syntax that can be extended to allow renaming and overriding of components.

IV. Encapsulation

A. How should inheritance affect encapsulation?

An intermediate level of encapsulation should be provided, preferably along the lines of Ada's distinction between private entities defined in the "specification package" and private entities defined in the "body package," and its associated child packages.

B. Should encapsulation be made more flexible in general?

I would appreciate having READ_ONLY module entities, and more selective encapsulation of the components of derived types, but neither capability is high on my list of priorities.