Improved run-time polymorphism for Fortran

Proposal to J3 for inclusion into Fortran 202y

K. Kifonidis

February 9, 2020

Abstract

A case is made for extending Fortran's support for object-oriented (OO) programming (i.e. run-time polymorphism) in order to enable and encourage the use of best practices for the design and implementation of flexible OO software, and to improve the safety and efficiency of the language. It is proposed to introduce multiple inheritance of specification, in order to supplement the single inheritance of implementation that is already contained in the language. The new feature extends Fortran's abstract interfaces and derived types by functionality akin to what is present in Java, and some other, recent, languages. It is essential for writing code that conforms to the dependency inversion principle of OO programming. This will encourage the development of software that forgoes dependency upon concretions in favor of dependency on abstractions. The decoupling of classes that can be achieved in this way will lead to significantly more flexible and extensible numerical codes, that are also more efficient. The new feature enables a programming style that dispenses with the use of implementation inheritance, and therefore helps to eliminate type conflicts whose resolution requires "select type" (down-casting) statements. These, and their associated run-time overhead, have been a frequent gripe of Fortran OO programmers since the introduction of Fortran 2003. As a bonus, the feature should be relatively easy to implement in Fortran compilers that adhere to at least the Fortran 2003 standard.

1 Introduction

Over the last decades, a significant body of experience has accumulated concerning the practical use of object-oriented programming (OOP). This experience has markedly shaped the design of some very recent programming languages, like Rust and Go, and it has also led to some notable revision of our notions of what actually constitutes OOP. In this modern view, the very essence of OOP, and in fact its ultimate upshot, is the use of (run-time) polymorphism to manage (i.e. minimize) code dependencies [3, 1]. Viewed in this way, OOP is simply a means to organize code so that it achieves a maximum amount of *decoupling*.

The crucial idea in this context is that polymorphism has the unique ability to allow one to invert the flow of dependency in a computer code, which normally points from higher level modules to lower level modules (read "classes" in the OOP context). Polymorphism makes it possible to replace ("invert") the dependency of a higher level class upon a lower level class by a dependency of both these classes upon a pure abstraction, which is called an interface (or a contract, protocol, or trait). Thereby, high-level policy gets decoupled from low-level implementation detail. This is known as the dependency inversion principle [3]. If strictly followed, this principle leads to the development of loosely coupled, easily extensible, easily testable, modular, maintainable, and thus future-proof software. It is this decoupling which allows for the re-usability and flexibility of well-written OO code.

The situation encountered in practice is usually a different one, though. Experience has shown that OO software which is written in Fortran 2003 (and its successors) seldom displays these desirable characteristics. It will be argued in the following that it is the combination of a lack of appropriate features in the language, for encouraging (or even enforcing) a programming style that upholds the dependency inversion principle, and the consequent indiscriminate use of features that the language *does* provide (in particular inheritance of implementation), that has to be blamed for this situation.

2 Deficiencies in Fortran 2018 addressed by this proposal

2.1 No explicit distinction between different inheritance types

In statically typed languages, like Fortran, run-time polymorphism is tightly bound to inheritance. In order to manage dependencies in the way it was described above, the users of these languages must employ some form of inheritance mechanism. It is important to realize that it is *inheritance of (object) specification*, i.e. inheritance of interfaces, that is required for this purpose. It is this (restricted) form of inheritance that is today viewed as being indispensable for OOP. Interface inheritance allows for the aforementioned interface-based programming style, that embodies the dependency inversion principle.

Fortran, on the other hand, makes exclusive use of *inheritance of implementation*, i.e. inheritance of classes. A class encompasses both an interface (given by the union of its methods' signatures, through which it communicates with the external world), *and* the implementation details of an OO algorithm (in the form of data variables, and the implementation bodies of its methods). A class is therefore not a proper abstraction (for other classes or procedures) to depend upon. It is not sufficiently generic. It contains implementation-dependent detail, which other classes end up being needlessly coupled to, if they make use of it via either inheritance or object composition. Only the class's interface is free from such detail and should be depended upon for these purposes, to achieve a maximum amount of decoupling in an application.

Fortran presently makes no explicit distinction between implementation inheritance and specification inheritance. The latter can only be *emulated* in Fortran by *re-purposing* class inheritance, and exploiting (or rather abusing) the fact that classes intermingle implementation with specification. Fortran offers abstract types (i.e. abstract classes) for related use cases, which can contain deferred (i.e. abstract) procedures, for which "abstract interfaces" have to be specified by the user. However, abstract classes (as all other classes) are explicitly allowed to *also* contain variables and concrete methods, i.e. state, as well as implementation code, which makes abstract classes dangerous to depend upon for inheriting specification from.

In contrast to Java, C#, D, Swift, Rust, Go and some other modern languages, Fortran does not offer a means to inherit specification directly from its "abstract interfaces", which — though available, and guaranteed to be completely free of implementation detail — play only a subordinate, instead of a cardinal role in the language. Hence it is not possible to enforce, at the language level, the strict separation of the implementation of an OO algorithm from its interface, that is required for adherence to the dependency inversion principle.

2.2 Problems due to the lack of distinction between inheritance types

Interface inheritance has the sole purpose of enabling polymorphism, in order to invert dependencies and thereby make parts of an algorithm interchangeable (i.e. to achieve high-level code reuse [2]). This is also called *sub-typing*. In contrast, class inheritance is meant to be used primarily for sharing common implementation code between parent and child classes. This is also called *sub-classing*, and can be viewed as a form of low-level code reuse.

There is a significant body of evidence that demonstrates that sub-classing is a dangerous practice, which, moreover, is not essential to OOP. Snyder [4] showed already in the eighties that sub-classing breaks encapsulation. Weck and Szyperski re-emphasized this [5]. They proposed to abolish sub-classing, since they regard it as being prone to abuse. Even moderately incautious

use of sub-classing typically leads to rigid, brittle code, that is tightly coupled to the details of concrete (implementation) classes, and is therefore unmaintainable and in-extensible. This is especially the case when sub-classing is used in combination with method overriding, or changes to a base object's state [5]. Moreover, when sub-classing is employed along with object composition in an application (which is inevitable in practice), type conflicts can often result. These can then only be resolved by down-casting, i.e. by the use of `select type` statements in Fortran, which lead to unnecessary run-time overhead. This can be viewed as resulting from a violation of the dependency inversion principle, which states that inheritance should be used *exclusively* for sub-typing.

All these issues could be avoided, if Fortran would be extended to allow for (multiple) inheritance of interfaces, in order to provide its programmers with easy and safe access to sub-typing. Sub-classing (i.e. type extension) could then be mostly shunned in the development of new codes. Clear guidelines could then be given to Fortran programmers to rely instead on object composition for low-level, and on the features proposed in this proposal for high-level code reuse, in the vast majority of their code. A coding style that is in accordance with the dependency inversion principle would thus be encouraged that would eliminate type conflicts, the need for `select type` statements, and their run-time overhead. A potential drawback of object composition is the need for some boilerplate code, i.e. for methods that delegate functionality to composed classes. However, efficiency can still be expected to improve over present Fortran OOP codes, especially in cases where `select type` statements appear in low-level code, e.g. in computational kernels.

The importance of such a coding style is actually not some entirely new insight. Among the very first recommendations that one finds in the pioneering work of Gamma et al. [2] on OO design patterns are the statements "program to an interface not an implementation" and "favor object composition over class inheritance". What had apparently not been recognized widely is that these are also recommendations for language design. This has changed recently, with the availability of the Rust and Go languages. In these languages, interface inheritance is the *only* type of inheritance that is supported. These languages therefore *enforce* an OOP style that is in accordance with the dependency inversion principle.

3 Proposed additions to Fortran 202y

The new features proposed to solve the problems discussed in the last section are

- A possibility to declare *named* versions of Fortran's abstract interfaces, from which derived types would then be able to inherit specification.
- An `extends` attribute-specifier for the declaration header of these named abstract interfaces, to allow hierarchies of such (named) abstract interfaces to be built[1].
- A new `implements` type-attribute-specifier for derived types, to enable these types to inherit specification from one or more named abstract interfaces.
- An extension of Fortran's `class` declaration-type-specifier for polymorphic variables, to accept named abstract interfaces. This would enable one to declare polymorphic variables of named abstract interfaces, in order to make use of sub-typing polymorphism.

3.1 Extended declaration syntax for abstract interfaces

The following listing shows an example of an extended syntax for interface declarations. In a module named `interfaces`, four named abstract interfaces are declared: `ISolver`, `IPrinter`, `IClient`, and `IRelaxationSolver`. `ISolver` and `IPrinter` contain the declarations of subroutines `solve_system`, and `print_result`, respectively. Notice that `IClient`, on the other, hand contains declarations for *two* procedures: subroutines `solve_system`, *and* `print_result`, which it inherits from `ISolver` and `IPrinter`, respectively, by means of the `extends` attribute-specifier.

[1] Hierarchies of interfaces do not pose the dangers that sub-classing poses, as only *specification* is shared, see [5].

```
module interfaces
  abstract interface :: ISolver
      subroutine solve_system(a,b,x)
         real, dimension(:,:), intent(in)  :: a
         real, dimension(:),   intent(in)  :: b
         real, dimension(:),   intent(out) :: x
      end subroutine solve_system
   end interface ISolver

   abstract interface :: IPrinter
      subroutine print_result(x)
         real, dimension(:), intent(in) :: x
      end subroutine print_result
   end interface IPrinter

   abstract interface, extends(ISolver,IPrinter) :: IClient
   end interface IClient

   abstract interface :: IRelaxationSolver
      import :: ISolver
      subroutine driver(self,d,rhs,x)
         class(IRelaxationSolver), intent(in)  :: self
         real, dimension(:,:,:,:), intent(in)  :: d
         real, dimension(:,:,:),   intent(in)  :: rhs
         real, dimension(:,:,:),   intent(out) :: x
      end subroutine driver
      subroutine relax(blksolver,d,rhs,x)
         class(ISolver),           intent(in)  :: blksolver
         real, dimension(:,:,:,:), intent(in)  :: d
         real, dimension(:,:,:),   intent(in)  :: rhs
         real, dimension(:,:,:),   intent(out) :: x
      end subroutine relax
   end interface IRelaxationSolver
end module interfaces
```

Finally, the declaration of the `IRelaxationSolver` interface illustrates that a named abstract interface may depend on other named abstract interfaces, not only via an `extends` attribute-specifier (as shown above), but also through the procedure declarations that it contains. Namely, the latter may contain polymorphic arguments that are declared with the help of the `class` declaration-type-specifier (see Sect. 3.3), to give these procedures access to either the data fields (variables) of the type that *implements* the abstract interface (i.e. to the descendant of that interface) or to the descendants of *other* abstract interfaces. In the latter case, Fortran's `import` statement would be required in order to bring such external names into the abstract interface's own scope.

Other ways to express and implement the proposed functionality are possible, and need to be considered. The one discussed here has the advantages that abstract interfaces are already a part of the language and are merely extended here by functionality that does not affect their other uses. It does, however, elevate their status in the language as it makes them central to expressing interface inheritance. This syntax also makes the distinction between sub-typing and sub-classing apparent, which is a major plus. It has the further advantage that named abstract interfaces could also be used in future extensions of the language, e.g. in connection with *intrinsic* types, and with generics. This is the approach taken in the Swift programming language where (named) abstract interfaces (called "protocols" there) are so central to its functionality, that Swift has been called a "protocol-based" language. It is not our purpose here to delve deeper into syntactic or extensibility issues. We rather need some syntax that is able to express the following code examples, and the present one does so satisfactorily.

3.2 Facility to implement interfaces

Once a named abstract interface has been declared, it can be implemented by a derived type that needs to conform to that interface, by using the new `implements` type-attribute-specifier as follows

```
module lu_decomp
   use interfaces, only: ISolver
   type, implements(ISolver) :: LUDecomposition
   contains
      procedure, nopass :: solve_system
   end type LUDecomposition
contains
   subroutine solve_system(a,b,x)
      real, dimension(:,:), intent(in)  :: a
      real, dimension(:),   intent(in)  :: b
      real, dimension(:),   intent(out) :: x
      ! Implementation of LU decomposition goes here
   end subroutine solve_system
end module lu_decomp
```

In case the derived type is abstract, it is allowed to provide an only partial implementation of the interface(s) that it implements. Any non-abstract derived type that extends the abstract type must, however, provide a full implementation.

3.3 Polymorphism via sub-typing

To make actual use of the sub-typing polymorphism that we have just set up, polymorphic variables of named abstract interfaces would then be declared with the help of the (extended) `class` declaration-type-specifier; for instance, as arguments of a procedure, or within another derived type. They could then be used in the actual implementation of this derived type's bound procedures, e.g. as follows:

```
type, implements(IClient) :: Client
   class(ISolver),  allocatable :: solver
   class(IPrinter), pointer     :: printer => null()
contains
   procedure, nopass :: solve_system
   procedure, nopass :: print_result
end type Client
```

Here, two polymorphic variables, `solver` and `printer`, that conform to the `ISolver` and `IPrinter` interfaces, respectively, are declared for the purpose of object composition within type `Client`, which itself implements the `IClient` interface, and thus has to provide implementations for the procedures `solve_system` and `print_result` (whose actual code is omitted here, and would simply employ delegation to `solver` and `printer` to provide the required functionality).

Notice also, that the proposed features allow multiple interface inheritance. In case `Client` has to conform to either `ISolver` or `IPrinter`, one could, for instance, provide an implementation much as in the last example, but would use the `implements` specifier as follows

```
type, implements(ISolver,IPrinter) :: Client
   class(ISolver),  allocatable :: solver
   class(IPrinter), pointer     :: printer => null()
contains
   procedure, nopass :: solve_system
   procedure, nopass :: print_result
end type Client
```

The next example shows, in detail, how sub-typing polymorphism would be used together with delegation, and also how type-bound procedures with the `pass` attribute would be handled

```
module jacobi
   use interfaces, only: IRelaxationSolver, ISolver
   type, implements(IRelaxationSolver) :: BlockJacobi
      class(ISolver), allocatable :: blksolver
   contains
      procedure, pass   :: driver
      procedure, nopass :: relax
   end type BlockJacobi
contains
   subroutine driver(self,d,rhs,x)
      class(BlockJacobi),       intent(in)  :: self
      real, dimension(:,:,:,:), intent(in)  :: d
      real, dimension(:,:,:),   intent(in)  :: rhs
      real, dimension(:,:,:),   intent(out) :: x
      call self%relax(self%blksolver,d,rhs,x)
   end subroutine driver
   subroutine relax(blksolver,d,rhs,x)
      class(ISolver),           intent(in)  :: blksolver
      real, dimension(:,:,:,:), intent(in)  :: d
      real, dimension(:,:,:),   intent(in)  :: rhs
      real, dimension(:,:,:),   intent(out) :: x
      integer :: i, j
      do j = 1, size(x,3)
         do i = 1, size(x,2)
            call blksolver%solve_system(d(:,:,i,j),rhs(:,i,j),x(:,i,j))
         end do
      end do
   end subroutine relax
end module jacobi
```

Notice that `BlockJacobi` is an implementor (i.e. descendant) of `IRelaxationSolver` and that therefore the above signature of method `driver` is in accordance with its declaration in module `interfaces`. To use `BlockJacobi`, one would have to simply inject into it, via a constructor, an instance of `LUDecomposition` (or any other object that implements the `ISolver` interface) in order to intialize the allocatable, polymorphic variable `blksolver`.

3.4 Combination of sub-classing with sub-typing

The new features need to be inter-operable also with type extension (i.e. sub-classing). They need to enable one to declare derived types that conform to some interfaces, and inherit at the same time implementation from some other derived type (e.g. the procedure `solve_system` implemented by type `LUDecomposition` in the following example):

```
type, extends(LUDecomposition), implements(ISolver,IPrinter) :: Client
   class(IPrinter), pointer :: printer => null()
contains
   procedure, nopass :: solve_system
   procedure, nopass :: print_result
end type Client
```

In the Java language, from which this idea is borrowed, the convention is that `extends` shall precede `implements`. The combination of sub-typing and sub-classing will allow advanced users of the language, who have a good understanding of OO design, to employ some advanced techniques that make use of both the flexibility that sub-typing affords one, with the raw reduction of code lines that sub-classing permits.

4 Example of a use case: a Taylor series application

To illustrate the difficulties that were discussed in Sect. 2.2, and to demonstrate how the use of the new features will contribute to the solution of these problems, we will present here a small case study that exhibits a pattern that is encountered often in practical applications, namely the coupling of two separate inheritance hierarchies through object composition. It is truly only the pattern that is of importance here, but in order to provide a concrete example, we will present it in the framework of a somewhat fictional application that calculates the Taylor series expansion of some function up to certain orders of accuracy.

4.1 The basic design

Suppose that we have an application that needs to calculate the Taylor series of a function $F(x)$ for some concrete x — and it needs to do so only up to the linear term, i.e. it only needs access to the first derive, $F'(x)$, of F with respect to x. Ignoring all the details of how this first derivative might be calculated, a code skeleton for using it within our application might look as follows. Declare a derived type `DerivF` that contains a procedure `deriv1` that calculates $F'(x)$, and a separate type, named `Taylor`, that makes use of type `DerivF` via object composition in order to `evaluate` the actual Taylor series approximation itself (whose details, like calculation of the zeroth order term, etc., are immaterial here):

```
module derivs
   type :: DerivF
   contains
      procedure, nopass :: deriv1 => deriv1f
   end type DerivF
contains
   subroutine deriv1f()
      write(*,*) ' 1st derivative of function F!'
   end subroutine deriv1f
end module derivs

module series
   use derivs, only: DerivF
   type :: Taylor
      type(DerivF), allocatable :: calc
   contains
      procedure :: term1
      procedure :: evaluate
   end type Taylor
contains
   subroutine term1(self)
      class(Taylor), intent(in) :: self
      call self%calc%deriv1()
   end subroutine term1

   subroutine evaluate(self)
      class(Taylor), intent(in) :: self
      write(*,*) 'Evaluating Taylor series using'
      call self%term1()
   end subroutine evaluate
end module series
```

Our client application would then set up an object `teval` of type `Taylor` in order to evaluate the Taylor series approximation it needs, e.g. as follows

```
program client
   use derivs, only: DerivF
   use series, only: Taylor
   type(Taylor), allocatable :: teval
   teval = Taylor( DerivF() )
   call teval%evaluate()
end program client
```

Suppose now, that our requirements on the application have changed. We need some additional functionality. We want the application to be able to also calculate, if required, a higher-order accurate approximation to the Taylor series of F, say up to the quadratic term. In addition, we also want it to be able to use some other function, say G, instead of F. Moreover, we do not want to change any of the present code structure, if possible. We only wish to extend it by the new functionality.

4.2 Extension by sub-classing

We will concentrate on implementing the functionality related to the higher-order-accuracy capability first. To accomplish this, we will use Fortran's `extends` feature for derived types (i.e. sub-classing). We extend type `DerivF` by a child type, `HDerivF`, that contains a procedure `deriv2` to calculate also the higher (i.e. second) order derivative of F.

```
type :: DerivF
contains
   procedure, nopass :: deriv1 => deriv1f
end type DerivF

type, extends(DerivF) :: HDerivF
contains
   procedure, nopass :: deriv2 => deriv2f
end type HDerivF
```

We also extend the type `Taylor` by a child type, `HTaylor`, to deal with the evaluation of the higher order Taylor series:

```
type :: Taylor
   type(DerivF), allocatable :: calc
contains
   procedure :: term1
   procedure :: evaluate
end type Taylor

type, extends(Taylor) :: HTaylor
contains
   procedure :: term2    => hterm2
   procedure :: evaluate => hevaluate
end type HTaylor
```

Here, we inherited method `term1` from the parent type, and added method `term2` to enable us to calculate also the second-order Taylor term. We also had to override the `evaluate` method of the parent type, since the higher-order evaluation of the complete series needs to account for this additional, i.e. second-order, term. We have thus implemented the skeleton of the high-order functionality, using types that make up two separate inheritance hierarchies. However, we are still bound to exclusive use of the function F. To implement the capability to use different functions, we are going to employ the strategy pattern [2]. We introduce a further inheritance hierarchy made up of two new types, `DerivG`, and `HDerivG`, to provide the same functionality for function G, as it was done above for function F. We also need to make these hierarchies, connected to F and G, interchangeable, so that we can use either of them within the code that evaluates the Taylor series. For this purpose, we introduce the following abstract type

```
type, abstract :: Deriv
contains
   procedure(pderiv), deferred, nopass ::  deriv1
end type Deriv

abstract interface
   subroutine pderiv()
   end subroutine pderiv
end interface
```

Now we can make both the inheritance hierarchies connected to F and G derive from this abstract type, and hence merge them into a single hierarchy. Its G-related branch, for instance, looks as follows

```
type, extends(Deriv) :: DerivG
contains
   procedure, nopass :: deriv1 => deriv1g
end type DerivG

type, extends(DerivG) :: HDerivG
contains
   procedure, nopass :: deriv2 => deriv2g
end type HDerivG
```

whereas the F-related branch looks completely analogous. The last step, in implementing the strategy pattern, is to transform the variable `calc` in type `Taylor` into a polymorphic variable of `class(Deriv)`

```
type :: Taylor
   class(Deriv), allocatable :: calc
contains
```

in order to accept `Deriv` types from either branch of our merged hierarchy. The complete code, that we end up with, is given in Appendix A. This code[2] produces the following output

```
 Evaluating Taylor series using
  1st derivative of function G!

 Evaluating Taylor series using
  1st derivative of function G!
  2nd derivative of function G!
```

That is, the code calculates first the low-order approximation of G, and subsequently its high-order approximation, as intended. It works correctly. Yet, it is extremely bad, rigid code! The most conspicuous symptom of its rigidity is the appearance of the `select type` statement in the subroutine `hterm2`, that we reproduce here for illustration:

```
subroutine hterm2(self)
   class(HTaylor), intent(in) :: self
   select type ( calc => self%calc )
   class is ( HDerivF )
      call calc%deriv2()
   class is ( HDerivG )
      call calc%deriv2()
   class default
      write(*,*) 'Unknown type!'
   end select
end subroutine hterm2
```

Obviously, this routine expects the `calc` instance to be of either type `HDerivF` or `HDerivG`, whereas this has been inherited from the parent type `Taylor` being of type `class(Deriv)`,

[2] compiled with gfortran Version 9

```
type :: Taylor
   class(Deriv), allocatable :: calc
contains
```

Our abstract type `Deriv` does not contain a declaration for the method `deriv2` that the subroutine needs. Since the compiler can't know at compile time whether at run-time `calc` will be of the right type-extension to contain the `deriv2` method, it will refuse to compile subroutine `hterm2` if the `select type` statement is omitted. By coupling two inheritance hierarchies (based on `Deriv` and `Taylor`) together, via object composition, we have created a (potential) type conflict that now requires a type check every time we run through the subroutine. If we had written a real production application along these lines, that would evaluate the Taylor series for a single scalar x per call, the type-checking would have completely killed performance!

Type extension (i.e. sub-classing) has locked us into a straight-jacket of rigid inheritance hierarchies that denies us the flexibility to provide our procedures with the right data types, and has thereby forced us to circumvent the static type system of the language, in order to get the code to work at all. But this is not the only problem. The entire code relies on concrete derived types, i.e. on concrete implementations. Except for type `Deriv`, it does not use any abstractions. Should the implementation of some concrete type need to be changed, all types that depend on it would need to be checked and possibly changed, too, in order not to break the program.

4.3 Extension by sub-typing and object composition

Now, let us go back to the point we were at at the end of Sect. 4.1. Let us, furthermore, suppose that Fortran doesn't support sub-classing, and instead supports only object composition and sub-typing (via the features proposed in Sect. 3), akin to Rust or Go. We will again focus first on the extension of our application to higher order of accuracy. We still contemplate to introduce this functionality through four derived types `DerivF`, `HDerivF`, `Taylor`, and `HTaylor`. But since we no longer have sub-classing at our disposal, the only way to make these types cooperate, on solving the task, is the following

```
type :: Taylor
   type(DerivF), allocatable :: calc
contains
   procedure :: term1
   procedure :: evaluate
end type Taylor

type :: HTaylor
   type(HDerivF), allocatable :: calc
contains
   procedure :: term1    => hterm1
   procedure :: term2    => hterm2
   procedure :: evaluate => hevaluate
end type HTaylor
```

that is, we need to employ object composition *twice*, once in `Taylor` and once in `HTaylor`. This is already sufficient to avoid the appearance of a `select type` statement in subroutine `hterm2`, because now our instance of `calc` will be of type `HDerivF`, i.e. of the proper type for doing high-order calculations. We will, of course, see to it that it contains the procedure `deriv2` which calculates the second order derivative required by the high-order Taylor series term.

The above code fragment still has a problem, though. It relies on the concrete types `DerivF` and `HDerivF`. Our copy of "Fortran 202y Explained" tells us that we should rely on abstractions instead of concretions, and depend on polymorphic variables of abstract interfaces within our derived types, instead. We therefore think about the functionality that these two concrete types need to provide to their clients. We already mentioned that `HDerivF` will have to provide an implementation of subroutine `deriv2`. However, and as it was the case in Sect. 4.2, `HDerivF`

also needs to implement procedure `deriv1`, while `DerivF` needs to implement *only* the latter procedure. Hence, we distill the signatures of these methods into two abstract interfaces

```
abstract interface :: IDeriv
   subroutine deriv1()
   end subroutine deriv1
end interface IDeriv

abstract interface, extends(IDeriv) :: IHDeriv
   subroutine deriv2()
   end subroutine deriv2
end interface IHDeriv
```

and then make use of these interfaces to modify our code for the Taylor types as follows

```
type :: Taylor
   class(IDeriv), allocatable :: calc
contains
   procedure :: term1
   procedure :: evaluate
end type Taylor

type :: HTaylor
   class(IHDeriv), allocatable :: calc
contains
   procedure :: term1    => hterm1
   procedure :: term2    => hterm2
   procedure :: evaluate => hevaluate
end type HTaylor
```

With this single stroke, we have just solved our entire problem, including the requirement of being able to calculate the Taylor series of different functions! Moreover, we have accomplished this goal with ease. The only things that remain to be done are to make `DerivF` and `HDerivF` simply implement the two abstract interfaces, e.g. as follows

```
type, implements(IDeriv) :: DerivF
contains
   procedure, nopass :: deriv1 => deriv1f
end type DerivF

type, implements(IHDeriv) :: HDerivF
contains
   procedure, nopass :: deriv1 => deriv1f
   procedure, nopass :: deriv2 => deriv2f
end type HDerivF
```

Listing 1: Implementation of interfaces for derivatives of function F.

and to add two more types, `DerivG`, and `HDerivG`, in exactly the same fashion, in order to provide implementations of the same functionality for function G.

We are finished! Since both `DerivF` and `DerivG` now conform to the abstract `IDeriv` interface, we can easily swap instances of either of them into type `Taylor` at will, in order to calculate the Taylor series of either F or G. Similar functionality is exhibited by the types that implement the high-order series. We didn't even have to spend a single thought on using the strategy pattern for these purposes. We used it naturally without thinking about it because it was encouraged by the features contained in the language.

The complete code, that uses this sub-typing approach, is given in Appendix B. It is an almost literal translation of a Java as well as a Rust code, that were used to test and verify the presented ideas. The code in Appendix B is vastly superior compared to the one that uses sub-classing and is given in Appendix A. Both codes have roughly the same number of lines,

but in the one relying on sub-typing there are *no* `select type` statements, *no* sub-classing hierarchies, and *no* dependencies on concrete types. We have inverted all the dependencies in the algorithm, i.e. the only dependencies are on abstract interfaces. We have thus attained the highest possible degree of decoupling. The *only* part of the code that depends on concrete types is the main client program, whose sole purpose is to organize the entire process of injecting instances of the correct concrete types via (Fortran default) constructors into the lower level code.

This decoupling of types leads to multiple benefits. For instance (and provided we have compiled the `interfaces` module), we gain the freedom to compile the `derivs` and `series` modules in any order we like, even in parallel if we wish to. We thereby neutralized the biggest drawback of a module system (like Fortran's) for separate compilation, namely serialization of the compilation process. We also avoid re-compilation cascades in case we need to change the implementation of some concrete type, as long as we leave its (abstract) interface intact. We are, moreover, free to take any of the aforementioned modules and reuse it in another application, without having to drag along numerous dependencies. In summary, using the new features we have not only accomplished to make the code more efficient, but also much more flexible, much more maintainable, and much more reusable.

4.4 Combination of the two inheritance types

The code given in Appendix B strictly conforms to the dependency inversion principle. By relaxing this requirement, and allowing for the use of both sub-typing and sub-classing, we could have written the code fragment in Listing 1 above somewhat more concise as

```
type, implements(IDeriv) :: DerivF
contains
   procedure, nopass :: deriv1 => deriv1f
end type DerivF

type, extends(DerivF), implements(IHDeriv) :: HDerivF
contains
   procedure, nopass :: deriv2 => deriv2f
end type HDerivF
```

This saves us the labor of providing an implementation for procedure `deriv1` within type `HDerivF`, at the significant cost of `HDerivF` depending now on a concretion, i.e. on type `DerivF`. In this particular, trivial, example, we have only saved a single line of code in this way. But had `DerivF` contained half a dozen or so methods, which we would have been forced to implement also in `HDerivF`, the convenience that the combination of `extends` and `implements` would have afforded us for reducing the number of code-lines could hardly have been passed up.

4.5 Summary

The features proposed in this document would significantly enhance Fortran's OOP capabilities without affecting backwards compatibility. Moreover, Fortran compilers that adhere to at least the Fortran 2003 standard could implement them relatively easily, as most of the functionality required to support them is already present in some form in such compilers for the support of sub-classing, i.e. the extension of derived types.

References

[1] Bert Bates, Kathy Sierra, Eric Freeman, and Elisabeth Robson. *Head First Design Patterns.* O'Reilly Media, 2009.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Prentice Hall, 1994.

[3] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson, 2002.

[4] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 Proceedings*, pages 38–45. ACM, September 1986.

[5] Wolfgang Weck and Clemens Szyperski. Do We Need Inheritance? `https://www.researchgate.net/publication/2297653_Do_We_Need_Inheritance`. Accessed February 4, 2020.

Appendix A Taylor series example based on sub-classing

```
module derivs

   type, abstract :: Deriv
   contains
      procedure(pderiv), deferred, nopass ::  deriv1
   end type Deriv

   abstract interface
      subroutine pderiv()
      end subroutine pderiv
   end interface

   type, extends(Deriv) :: DerivF
   contains
      procedure, nopass :: deriv1 => deriv1f
   end type DerivF

   type, extends(DerivF) :: HDerivF
   contains
      procedure, nopass :: deriv2 => deriv2f
   end type HDerivF

   type, extends(Deriv) :: DerivG
   contains
      procedure, nopass :: deriv1 => deriv1g
   end type DerivG

   type, extends(DerivG) :: HDerivG
   contains
      procedure, nopass :: deriv2 => deriv2g
   end type HDerivG

contains

   subroutine deriv1f()
      write(*,*) ' 1st derivative of function F!'
   end subroutine deriv1f

   subroutine deriv2f()
      write(*,*) ' 2nd derivative of function F!'
   end subroutine deriv2f

   subroutine deriv1g()
      write(*,*) ' 1st derivative of function G!'
   end subroutine deriv1g
```

```
46
47    subroutine deriv2g()
48       write(*,*) ' 2nd derivative of function G!'
49    end subroutine deriv2g
50
51 end module derivs
52
53 module series
54
55    use derivs, only: Deriv, HDerivF, HDerivG
56
57    type :: Taylor
58       class(Deriv), allocatable :: calc
59    contains
60       procedure :: term1
61       procedure :: evaluate
62    end type Taylor
63
64    type, extends(Taylor) :: HTaylor
65    contains
66       procedure :: term2    => hterm2
67       procedure :: evaluate => hevaluate
68    end type HTaylor
69
70 contains
71
72    subroutine term1(self)
73       class(Taylor), intent(in) :: self
74       call self%calc%deriv1()
75    end subroutine term1
76
77    subroutine evaluate(self)
78       class(Taylor), intent(in) :: self
79       write(*,*) 'Evaluating Taylor series using'
80       call self%term1()
81    end subroutine evaluate
82
83    subroutine hterm2(self)
84       class(HTaylor), intent(in) :: self
85       select type ( calc => self%calc )
86       class is ( HDerivF )
87          call calc%deriv2()
88       class is ( HDerivG )
89          call calc%deriv2()
90       class default
91          write(*,*) 'Unknown type!'
92       end select
93    end subroutine hterm2
94
95    subroutine hevaluate(self)
96       class(HTaylor), intent(in) :: self
97       write(*,*) 'Evaluating Taylor series using'
98       call self%term1()
99       call self%term2()
100   end subroutine hevaluate
101
102 end module series
103
104 program client
```

```
105
106    use derivs, only: DerivG, HDerivG, Deriv
107    use series, only: Taylor, HTaylor
108
109    class(Deriv),  allocatable :: derv
110    class(Taylor), allocatable :: teval
111
112    derv  = DerivG()
113    teval = Taylor(derv)
114    call teval%evaluate()
115
116    write(*,*)
117
118    derv  = HDerivG()
119    teval = HTaylor(derv)
120    call teval%evaluate()
121
122 end program client
```

Appendix B Taylor series example based on sub-typing

```
1  module interfaces
2
3     abstract interface :: IDeriv
4        subroutine deriv1()
5        end subroutine deriv1
6     end interface IDeriv
7
8     abstract interface, extends(IDeriv) :: IHDeriv
9        subroutine deriv2()
10       end subroutine deriv2
11    end interface IHDeriv
12
13 end module interfaces
14
15 module derivs
16
17    use interfaces, only: IDeriv, IHDeriv
18
19    type, implements(IDeriv) :: DerivF
20    contains
21       procedure, nopass :: deriv1 => deriv1f
22    end type DerivF
23
24    type, implements(IHDeriv) :: HDerivF
25    contains
26       procedure, nopass :: deriv1 => deriv1f
27       procedure, nopass :: deriv2 => deriv2f
28    end type HDerivF
29
30    type, implements(IDeriv) :: DerivG
31    contains
32       procedure, nopass :: deriv1 => deriv1g
33    end type DerivG
34
35    type, implements(IHDeriv) :: HDerivG
36    contains
37       procedure, nopass :: deriv1 => deriv1g
```

```
38       procedure, nopass :: deriv2 => deriv2g
39    end type HDerivG
40
41 contains
42
43    subroutine deriv1f()
44       write(*,*) " 1st derivative of function F!"
45    end subroutine deriv1f
46
47    subroutine deriv2f()
48       write(*,*) " 2nd derivative of function F!"
49    end subroutine deriv2f
50
51    subroutine deriv1g()
52       write(*,*) " 1st derivative of function G!"
53    end subroutine deriv1g
54
55    subroutine deriv2g()
56       write(*,*) " 2nd derivative of function G!"
57    end subroutine deriv2g
58
59 end module derivs
60
61 module series
62
63    use interfaces, only: IDeriv, IHDeriv
64
65    type :: Taylor
66       class(IDeriv), allocatable :: calc
67    contains
68       procedure :: term1
69       procedure :: evaluate
70    end type Taylor
71
72    type :: HTaylor
73       class(IHDeriv), allocatable :: calc
74    contains
75       procedure :: term1    => hterm1
76       procedure :: term2    => hterm2
77       procedure :: evaluate => hevaluate
78    end type HTaylor
79
80 contains
81
82     subroutine term1(self)
83        class(Taylor), intent(in) :: self
84        call self%calc%deriv1()
85     end subroutine term1
86
87     subroutine evaluate(self)
88        class(Taylor), intent(in) :: self
89        write(*,*) 'Evaluating Taylor series using'
90        call self%term1()
91     end subroutine evaluate
92
93     subroutine hterm1(self)
94        class(HTaylor), intent(in) :: self
95        call self%calc%deriv1()
96     end subroutine hterm1
```

```
97
98      subroutine hterm2(self)
99         class(HTaylor), intent(in) :: self
100        call self%calc%deriv2()
101     end subroutine hterm2
102
103     subroutine hevaluate(self)
104        class(HTaylor), intent(in) :: self
105        write(*,*) 'Evaluating Taylor series using'
106        call self%term1()
107        call self%term2()
108     end subroutine hevaluate
109
110 end module series
111
112 program client
113
114    use derivs, only: DerivG, HDerivG
115    use series, only: Taylor, HTaylor
116
117    type(Taylor),  allocatable :: teval
118    type(HTaylor), allocatable :: hteval
119
120    teval = Taylor( DerivG() )
121    call teval%evaluate()
122
123    write(*,*)
124
125    hteval = HTaylor( HDerivG() )
126    call hteval%evaluate()
127
128 end program client
```