Traits, Generics, and Modern-day OOP for Fortran

Konstantinos Kifonidis, Ondrej Certik, Derick Carnazzola

March 22, 2025

Abstract

Based on conclusions drawn from a survey of modern languages, a traits system for Fortran is developed that allows for the uniform expression of run-time and compile-time polymorphism in the language, and thus for the uniform management of source code dependencies on user-defined and language-intrinsic types. The feature set that is described here is small enough to facilitate a first prototype implementation in an open source compiler (like LFortran, LLVM Flang, or GNU Fortran), but at the same time comprehensive enough to already endow Fortran with polymorphic capabilities that equal those of modern programming languages like Swift, Rust, Go, Carbon, or Mojo. The discussed extensions are fully backwards compatible with present Fortran, and enable modern-day, traits-based, object-oriented programming, and powerful, easy to use, fully type-checked, generics. The latter support seamlessly both the procedural, functional, and object-oriented programming styles, and they largely get by *without* requiring manual instantiations by the user. The presented design could also be naturally extended to support rank-genericity for arrays, structural subtyping, and compile-time polymorphic union/sum types. Its modern capabilities are expected to transform the way both Fortran applications and libraries will be written in the future. Decoupled software plugin architectures with enormously improved source code flexibility, reusability, maintainability, and reliability will become possible, without any need for run-time type inspections, and without any loss in computational performance.

Chapter 1

Introduction

Polymorphism was discovered in the 1960ies by Kristen Nygaard and Ole-Johan Dahl during their development of Simula 67, the world's first object-oriented (OO) language [3]. Their work introduced into programming what is nowadays known as "virtual method table" (i.e. function-pointer) based run-time polymorphism, which is both the first focus of this document, and the decisive feature of all OO languages. Several other forms of polymorphism are known today, the most important of them being parametric polymorphism [1], also known as "generics", which is the second focus of this document, and which has historically developed disjointly from run-time polymorphism since it makes use of compile-time mechanisms.

1.1 The purpose of polymorphism

But what is the purpose of polymorphism in a programming language? What is polymorphism actually good for? One of the more comprehensive answers to this question was given by Robert C. Martin in numerous books (e.g. [12]), as well as in the following quotation from his blog [11]:

> "There really is only one benefit to polymorphism; but it's a big one. It is the inversion of source code and run time dependencies.
>
> In most software systems when one function calls another, the runtime dependency and the source code dependency point in the same direction. The calling module depends on the called module. However, when polymorphism is injected between the two there is an inversion of the source code dependency. The calling module still depends on the called module at run time. However, the source code of the calling module does not depend upon the source code of the called module. Rather both modules depend upon a polymorphic interface.
>
> This inversion allows the called module to act like a plugin. Indeed, this is how all plugins work."

Notice the absence of the words "code reuse" in these statements. The purpose of polymorphism, according to Martin, is the "inversion" (i.e. replacement, or management) of rigid source code dependencies by means of particular abstractions, i.e. polymorphic interfaces (or protocols/traits, as they are also known today). The possibility to reuse code is then merely the logical consequence of such proper dependency management in a polymorphism-based software plugin architecture.

1.2 Source code dependencies in statically typed languages

Which then are the source code dependencies that polymorphism helps us manage? It has been customary to make the following distinction when answering this question:

- Firstly, most larger programs that are written in statically typed languages (like Fortran) have dependencies on *user-defined* procedures and data types. If the programmer employs encapsulation of both a program's procedures and its data, i.e. its state, both these dependencies can actually be viewed as dependencies on user-defined abstract data types. These are the dependencies that Martin is concretely referring to in the above quotation, and it is these dependencies on (volatile) user-defined data and implementations that are particularly troublesome, because they lead to rigid coupling between the various different *parts* of an application. Their results are recompilation cascades, the non-reusability of higher-level source code units, the impossibility to comprehend a large application incrementally, and fragility of such an application as a whole.
- Secondly, every program, that is written in a statically typed language, also has dependencies on abstract data types that are provided by the language itself. Fortran's `integer`, `real`, etc. intrinsic types are examples of *language-intrinsic* abstract data types. While hard-wired dependencies on such intrinsic types do not couple different parts of a program (because the implementations of these types are supplied by the language), they nevertheless make a program's source code rigid with respect to the data that it can be used on.

The most widely used approaches to manage dependencies on language-intrinsic types have so far been through generics, while dependency management of user-defined (abstract data) types has so far been the task of OO programming and OO design patterns. Martin [12] has, for instance, defined object-orientation as follows:

> "OO is the ability, through the use of polymorphism, to gain absolute control over every source code dependency in [a software] system. It allows the architect to create a plugin architecture, in which modules that contain high-level policies are independent of modules that contain low-level details. The low-level details are relegated to plugin modules that can be deployed and developed independently from the modules that contain high-level policies."

1.3 Modern developments

Notice how Martin's modern definition of object-orientation, that emphasizes source code decoupling, is the antithesis to the usually taught "OO" approaches of one class rigidly inheriting implementation code from another. Notice also how his definition does not require some specific type of polymorphism for the task of dependency management, as long as (according to Martin's first quotation) the mechanism is based on polymorphic interfaces.

Martin's statements on the purpose of both polymorphism and OO simply reflect the two crucial developments that have taken place in these fields over the last decades. Namely, the realizations that

- run-time polymorphism should be freed from the conflicting concept of implementation inheritance (to which it was originally bound given its Simula 67 heritage), and be formulated exclusively in terms of conformance to polymorphic interfaces, i.e. function signatures, or purely procedural abstractions, and that

- compile-time polymorphism should be formulated in exactly the same way as well.

These two developments, taken together, have recently opened up the possibility to treat polymorphism, and hence the dependency management of both user-defined and language-intrinsic types, uniformly in a programming language. As a consequence, it has become possible to use the potentially more efficient (but also less flexible) mechanism of compile-time polymorphism also for certain tasks that have traditionally been reserved for run-time polymorphism (in the form of OO programming), and to mix and match the two polymorphism types inside a single application to better satisfy a user's needs for both flexibility and efficiency.

1.4 Historical background

The road towards these realizations was surprisingly long. Over the last five decades, a huge body of OO programming experience first had to demonstrate that the use of (both single and multiple) implementation inheritance breaks encapsulation in OO languages, and therefore results in extremely tightly coupled, rigid, fragile, and non-reusable code. This led to an entire specialized literature on OO design patterns [4, 6, 10], that aimed at avoiding such rigidity by replacing the use of implementation inheritance with the means to formulate run-time polymorphism that are discussed below. It also led to the apprehension that implementation inheritance (but *not* run-time polymorphism) should be abandoned [20]. In modern languages, implementation inheritance is either kept solely for backwards compatibility reasons (e.g. in the Swift and Carbon languages), or it is foregone altogether (e.g. in Rust, and Go).

The first statically typed mainstream programming language that offered a proper separation of run-time polymorphism from implementation inheritance was Objective-C. It introduced "protocols" (i.e. polymorphic interfaces) in the year 1990 [2]. Protocols in Objective-C consist of pure function signatures, that lack implementation code. Objective-C provided a mechanism to implement multiple such protocols by a class, and to thus make classes conform to protocols. This can be viewed as a restricted form of multiple inheritance, namely inheritance of object *specification*, which is also known as *subtyping*. Only a few years later, in 1995, the Java language hugely popularized these ideas using the terms "interfaces" and "interface inheritance" [2]. Today, nearly all modern languages support polymorphic interfaces/protocols, and the basic mechanism of multiple interface inheritance that was introduced to express run-time polymorphism in Objective-C, often in even improved, more flexible, manifestations. The only negative exceptions in this respect being modern Fortran, and C++, which both still stick to the obsolescent Simula 67 paradigm.

A similarly lengthy learning process, as that outlined for run-time polymorphism, took also place in the field of compile-time/parametric polymorphism. Early attempts, notably templates in C++, to render function arguments and class parameters polymorphic, did not impose any constraints on such arguments and parameters, that could be checked by C++ compilers. With the known results on compilation times and cryptic compiler error messages [5]. Surprisingly, Java, the language that truly popularized polymorphic interfaces in OO programming, did not provide an interface based mechanism to constrain its generics. Within the pool of mainstream programming languages, this latter realization was only made with the advent of Rust [13].

Rust came with a traits (i.e. polymorphic interfaces) system with which it is possible for the user to uniformly and transparently express both generics (i.e. compile-time) and run-time polymorphism in the same application, and to relatively easily switch between the two, where possible. Rust's traits are an improved form of protocols/interfaces in that the user can implement them for a type without having these implementations be coupled to the type's actual definition. Thus, ex-

isting types can be made to retroactively implement new traits, and hence be used in new settings (with some minor restrictions on user ownership of either the traits or the types).

Rust's main idea was quickly absorbed by almost all other mainstream modern languages, most notably Swift, Go, and Carbon, with the difference that these latter languages tend to leave the choice between static and dynamic binding of procedures to the compiler, or language implementation, rather than the programmer. C++ is in the process of adopting generics constraints for its "templates" under the term "strong concepts", but without implementing the greater idea to uniformly express *all* the polymorphism in the language through traits. An implementation of this latter idea must today be viewed as a prerequisite in order to call a language design "modern". The purpose of this document is to describe extensions to Fortran, that aim to provide the Fortran language with such modern capabilities.

Chapter 2

Case study: Calculating the average value of a numeric array

To illustrate the advanced features and capabilities of some of the available modern programming languages with respect to polymorphism, and hence dependency management, we will make use here of a case study: the simple test problem of calculating the average value of a set of numbers that is stored inside a one-dimensional array. In the remainder of this chapter, we will first provide an account and some straightforward monomorphic (i.e. rigidly coupled) functional implementation of this test problem, followed by a functional implementation that makes use of both run-time and compile-time polymorphism to manage rigid source code dependencies. In the survey of programming languages that is presented in Chapter 3, we will then recode this standard test problem in an encapsulated fashion, to highlight how the source code dependencies in this problem can be managed in different languages even in more complex situations, that require OO techniques.

2.1 Monomorphic functional implementation

We have chosen Go here as a language to illustrate the basic ideas. Go is easily understood, even by beginners, and is therefore well suited for this purpose (another good choice would have been the Swift language). The code in the following Listing 2.1 should be self explanatory for anyone who is even only remotely familiar with the syntax of C family languages. So, we'll make only a few remarks regarding syntax.

- While mostly following a C like syntax, variable declarations in Go are essentially imitating Pascal syntax, where a variable's name precedes the declaration of the type.
- Go has two assignment operators. The usual = operator, as it is known from other languages, and the separate operator := that is used for combined declaration and initialization of a variable.
- Go has array slices that most closely resemble those of Python's Numpy (which exclude the upper bound of an array slice).

Our basic algorithm for calculating the average value of an array of integer elements employs two different implementations for averaging. The first makes use of a "simple" summation of all the array's elements, in ascending order of their array index. While the second sums in a "pairwise" manner, dividing the array in half to carry out the summations recursively, and switching

to the "simple" method once subdivision is no longer possible. In both cases, the resulting sum is then divided by the array's number of elements, to obtain the desired average.

Listing 2.1: Monomorphic functional version of the array averaging example in Go.

```
package main

import "fmt"

func simple_sum(x []int32) int32 {
   var s int32
   s = int32(0)
   for i := 0; i < len(x); i++ {
     s += x[i]
   }
   return s
}

func pairwise_sum(x []int32) int32 {
   if len(x) <= 2 {
      return simple_sum(x)
   } else {
      m := len(x) / 2
      return pairwise_sum(x[:m+1]) + pairwise_sum(x[m+1:])
   }
}

func simple_average(x []int32) int32 {
   return simple_sum(x) / int32(len(x))
}

func pairwise_average(x []int32) int32 {
   return pairwise_sum(x) / int32(len(x))
}

// ..............
// main program
// ..............

func main() {
   xi := []int32{1,2,3,4,5}

   var key int32

   fmt.Println("Simple   sum average: 1")
   fmt.Println("Pairwise sum average: 2")
   fmt.Print("Choose an averaging method: ")
   fmt.Scan(&key)

   switch key {
   case 1:
      fmt.Println(simple_average(xi))
   case 2:
      fmt.Println(pairwise_average(xi))
   default:
      fmt.Println("Case not implemented!")
   }

}
```

An inspection of Listing 2.1 will readily reveal that this code has three levels of rigid (i.e. hard-wired) dependencies. Namely,

1. function `pairwise_sum` depending on function `simple_sum`'s implementation,
2. functions `simple_average` and `pairwise_average` depending on functions' `simple_sum`, and `pairwise_sum` implementation, respectively, and
3. the entire program depending rigidly on the `int32` data type in order to declare both the arrays that it is operating on, and the results of its summation and averaging operations.

The first two items are dependencies on user-defined implementations, while the third is a typical case of rigid dependency on a language-intrinsic type, which renders the present code incapable of being applied to arrays of any other data type than `int32`s. Given that we are dealing with three levels of dependencies, three levels of polymorphism will accordingly be required to remove all these dependencies.

2.2 Polymorphic functional implementation

Listing 2.2 gives an implementation of our test problem, that employs Go's generics and functional features in order to eliminate the last two of the rigid dependencies that were listed in Sect. 2.1. The code makes use of Go's generics to admit arrays of both the `int32` and `float64` types as arguments to all functions, and to express the return values of the latter. It also makes use of the run-time polymorphism inherent in Go's advanced functional features, namely closures and variables of higher-order functions, to replace the two previous versions of function `average` (that depended on specific implementations), by a single polymorphic version. Only the rigid dependency of function `pairwise_sum` on function `simple_sum` has not been removed, in order to keep the code more readable. In the OO code versions, that will be presented in Chapter 3, even this dependency is eliminated.

A few remarks are in order for a better understanding of Listing 2.2's code:

- In Go, generic type parameters to a function, like the parameter `T` here, are provided in a separate parameter list, that is enclosed in brackets [ ].
- Generic type parameters have a constraint that follows their declared name. Go exclusively uses interfaces as such constraints (like the interface `INumeric` in the following code).
- Interfaces consist of either explicit function signatures, or *type sets*, like `int32 | float64` in the present example. The latter actually signify a set of function signatures, too, namely the signatures of the intersecting set of all the operations and intrinsic functions for which the listed types provide implementations.
- The code makes use of type conversions to the generic type `T`, where required. For instance, `T(0)` converts the (typeless) constant `0` to the corresponding zero constant of type `T`.
- The code instantiates closures and stores these by value in two variables named `avi` and `avf` for later use (Fortran and C programmers should note that `avi` and `avf` are *not* function pointers!).

Listing 2.2: Polymorphic functional version of the array averaging example in Go.

```go
package main

import "fmt"

type INumeric interface{ int32 | float64 }

func simple_sum[T INumeric](x []T) T {
   var s T
   s = T(0)
   for i := 0; i < len(x); i++ {
     s += x[i]
   }
   return s
}

func pairwise_sum[T INumeric](x []T) T {
   if len(x) <= 2 {
      return simple_sum(x)
   }
   m := len(x) / 2
   return pairwise_sum(x[:m+1]) + pairwise_sum(x[m+1:])
}

func average[T INumeric](sum func([]T) T, x []T) T {
   return sum(x) / T(len(x))
}

func main() {
   xi := []int32{1, 2, 3, 4, 5}
   xf := []float64{1, 2, 3, 4, 5}

   var key int32
   var avi func([]int32) int32
   var avf func([]float64) float64

   fmt.Println("Simple   sum average:", 1)
   fmt.Println("Pairwise sum average:", 2)
   fmt.Print("Choose an averaging method: ")
   fmt.Scan(&key)

   switch key {
      case 1:
         avi = func(x []int32) int32 {
                  return average(simple_sum[int32], x)
               }
         avf = func(x []float64) float64 {
                  return average(simple_sum[float64], x)
               }
      case 2:
         avi = func(x []int32) int32 {
                  return average(pairwise_sum[int32], x)
               }
         avf = func(x []float64) float64 {
                  return average(pairwise_sum[float64], x)
               }
      default:
         fmt.Println("Case not implemented!")
```

```
58              return
59      }
60
61      fmt.Println(avi(xi))
62      fmt.Println(avf(xf))
63  }
```

Notice how, in order to instantiate the closures `avi` and `avf` (see the `switch` statement), manual instantiations of the `simple_sum` and `pairwise_sum` generic functions are required – with the arguments `int32` or `float64` being substituted for the generic type parameter, `T`, of these functions.

The motivation to code the example as in Listing 2.2 is that once the two closures, `avi` and `avf`, have been properly instantiated, they may then be passed from the main program to any other client code that may need to make use of the particular averaging algorithm that was selected by the user. This latter client code would *not* have to be littered with `switch` statements itself, and it would *not* have to depend on any specific implementations. It would merely depend on the closures' interfaces. The same holds for the OO code versions that are discussed in the next chapter, with objects replacing the closures (both being merely slightly different realizations of the same idea).

Chapter 3

Survey of modern languages

In the present chapter, we give encapsulated (i.e. OO) code versions of the test problem in various modern languages. As in the functional code version that was presented in Sect. 2.2, we employ run-time polymorphism to manage the dependencies on user-defined implementations (in this case abstract data types), and generics in order to manage the dependencies on language-intrinsic types. This serves to illustrate how both run-time and compile-time polymorphism can be typically used for dependency management in an OO setting in these modern languages. The survey also aims to highlight the many commonalities but also some of the minor differences in the approaches to polymorphism that were taken in these different languages. As a final disclaimer, we do not advocate to code problems in an OO manner that can be easily coded in these languages in a functional way (as it is the case for this problem). However, in more complex cases, where many more nested functions would need to be used, and where state would have to be hidden, the OO programming style would be the more appropriate one. Hence, our test problem will stand in, in this chapter, for emulating such a more complex problem, that would benefit from the OO coding style.

3.1 Go

Go has supported run-time polymorphism through (polymorphic) "interfaces" (and thus modern-day OO programming) since its inception. In Go, encapsulation is done by storing state in a `struct` and by binding procedures, that need to use that state, to this same `struct`. Thereby creating a user-defined abstract data type (or ADT) with methods. Go allows the programmer to implement multiple polymorphic interfaces for such a type (i.e. to use multiple interface inheritance), even though it offers no explicit language statement for this purpose.

Instead, a user-defined type is implicitly assumed to implement an interface whenever it provides implementations of all the interface's function signatures. This structural way of implementing interfaces also merely requires an object reference of the type to be passed to its methods (by means of a separate parameter list, in front of a method's actual name). It is otherwise decoupled from the type's (i.e. the ADT's `struct`) definition. Go, finally, makes it explicit in its syntax that interfaces (like `structs`) are types in their own right, and that hence polymorphic variables (i.e. objects) can be declared in terms of them.

Restrictions in Go are that language-intrinsic types cannot have user-provided methods, and that methods and interfaces cannot be directly implemented for user-defined types whose definitions are located in other packages. That is, the programmer has to write wrappers in the latter case.

Since version 1.18, Go also supports compile-time polymorphism through generics. Go's generics make use of "strong concepts", since they are bounded by constraints that are expressed through interfaces. Hence, the Go compiler will fully type-check generic code. In Go, structures, interfaces, and functions, but not methods, can all be given their own generic type parameters.

3.1.1 Encapsulated version coded in Go

Listing 3.1 gives an encapsulated version of the test problem coded in Go. The two different implementations of the `sum` function have been encapsulated in two different ADTs named `SimpleSum` and `PairwiseSum`, whereas a third ADT named `Averager` encapsulates the functionality that is required to perform the actual averaging. The latter two ADTs contain the lower-level objects `other` and `drv` of `ISum[T]` type as components, to which they delegate calls to these objects' `sum` methods. Notice, how the use of the polymorphic interface `ISum[T]`, for the declarations of these objects, enables them to be initialized with either `SimpleSum` or `PairwiseSum` instances.

A second interface, named `IAverager`, is used to enable polymorphism for different averaging algorithms. Finally, there's a third interface, `INumeric`, that serves exactly the same purpose as in the functional polymorphic version that was given in Sect. 2.2, namely to make all function arguments and return values polymorphic, by admitting as input and output parameters both the `int32` and `float64` intrinsic types.

Hence, three polymorphic interfaces were required in this code, in order to eliminate the three levels of rigid dependencies that were listed in Sect. 2.1. Notice also that, exempting `INumeric`, all the interfaces and all the user-defined ADTs need to take in generic type parameters in this example. In Go, this is required in order to enable all the `sum` and `average` methods to use such generic type parameters.

Listing 3.1: Encapsulated Go version of the array averaging example.

```
1  package main
2
3  import "fmt"
4
5  // ...........
6  // Interfaces
7  // ...........
8
9  type INumeric interface {
10    int32 | float64
11 }
12
13 type ISum[T INumeric] interface {
14    sum(x []T) T
15 }
16
17 type IAverager[T INumeric] interface {
18    average(x []T) T
19 }
20
21 // ..............
22 // SimpleSum ADT
23 // ..............
24
25 type SimpleSum[T INumeric] struct {
26 }
27
```

```
func (self SimpleSum[T]) sum(x []T) T {
   var s T
   s = T(0)
   for i := 0; i < len(x); i++ {
      s += x[i]
   }
   return s
}

// ................
// PairwiseSum ADT
// ................

type PairwiseSum[T INumeric] struct {
   other ISum[T]
}

func (self PairwiseSum[T]) sum(x []T) T {
   if len(x) <= 2 {
      return self.other.sum(x)
   } else {
      m := len(x) / 2
      return self.sum(x[:m+1]) + self.sum(x[m+1:])
   }
}

// ............
// Averager ADT
// ............

type Averager[T INumeric] struct {
   drv ISum[T]
}

func (self Averager[T]) average(x []T) T {
   return self.drv.sum(x) / T(len(x))
}

// ............
// main program
// ............

func main() {
   var avi IAverager[int32]
   var avf IAverager[float64]

   xi := []int32{1,2,3,4,5}
   xf := []float64{1.,2.,3.,4.,5.}

   var key int32

   fmt.Println("Simple   sum average: 1")
   fmt.Println("Pairwise sum average: 2")
   fmt.Print("Choose an averaging method: ")
   fmt.Scan(&key)

   switch key {
   case 1:
      avi = Averager[int32]{ SimpleSum[int32]{} }
```

```
87          avf = Averager[float64]{ SimpleSum[float64]{} }
88      case 2:
89          avi = Averager[int32]{ PairwiseSum[int32]{ SimpleSum[int32]{} } }
90          avf = Averager[float64]{ PairwiseSum[float64]{ SimpleSum[float64]{} } }
91      default:
92          fmt.Println("Case not implemented!")
93          return
94      }
95
96      fmt.Println(avi.average(xi))
97      fmt.Println(avf.average(xf))
98  }
```

The main program makes use of Go's built-in structure constructors, and chaining of their calls, in order to instantiate objects of the required ADTs. In particular, it instantiates run-time polymorphic `Averager` objects (depending on whether simple or pairwise sum averaging is to take place), and it does so for both the `int32` and `float64` types separately, in order to then use these objects on `int32` and `float64` data, respectively.

That *two* such objects are required (one for each language-intrinsic data type) is connected to the aforementioned fact that in order to make methods use generic type parameters in Go, one has to parameterize interfaces, and instantiate these with different actual data types, as in `func main`'s first two code lines. A single (i.e. unparameterized) `IAverager` interface therefore doesn't suffice, which is unfortunate from the user's perspective, as some code duplication in client code cannot be avoided in this way.

It should also be noted, that Go's intuitive way of expressing the generics constraints of interface `INumeric` in terms of a type set, besides having obvious advantages in terms of code clarity and conciseness, comes also at a price. Namely its only *partial* conformance to the Open/Closed Principle (OCP) of OO programming [9]. The latter principle demands that only new code be added to an application in order to extend its functionality, and any already written code to not be changed.

It can be seen, from Listing 3.1, that this principle can only be partly, but not strictly (i.e. fully), complied with in generic object-oriented Go code. Because for the programmer to make this listing work also with, e.g., the `float32` type, he would have to add this type to the type set of interface `INumeric`, and to thus touch already written code. Notice, though, that *none* of the implementation code of Listing 3.1 would be affected. Only the interface `INumeric` would need to be changed, that the remaining client code depends upon as a generics constraint.

If the client code were distributed among different packages or modules, the consequences of such a (weak) violation of the OCP would be confined to the occurrence of a recompilation cascade in the dependent modules. In actual practice, one may quite often regard this as an acceptable (minor) inconvenience, rather than a fundamental maintenance issue – especially in situations where the need to add new types is a relatively rare one.

3.2 Rust

Like Go, Rust supports both run-time and compile-time polymorphism through polymorphic interfaces, which Rust calls "traits". Unlike Go, Rust has its programmers implement traits in a nominal manner, by using explicit `impl` code blocks to provide a trait's method implementations. These same `impl` blocks can also be used to bind so-called "associated functions" to a type, which aren't methods, i.e. which don't take in a `self` (passed-object dummy) argument. A typical example for this are user-defined constructors. See the functions that are named `new` in the following

code Listing 3.2, and are called using `::` syntax.

In contrast to Go, Rust allows the programmer to implement traits for both user-defined *and* language-intrinsic types, and to do so for types that are located in external libraries (called "crates" in Rust), as long as the traits themselves are defined in the programmer's own crate. The reverse, namely implementing an external trait for a user-owned type, is also possible. Only the (edge) case of implementing an external trait for an external type is not allowed (this is called the "orphan rule" [7]). The latter case requires the use of wrappers.

Comparable to Go, Rust's generics model allows for the generic parameterization of functions, traits, and user-defined types like `structs`. Rust does not explicitly forbid generic methods. However, if one defines such a method's signature within a trait, then this will make the trait unusable for the declaration of any "trait objects" [8], i.e. for the employment of run-time polymorphism. Thus, the Rust programmer will in general (need to) parameterize traits and `structs` rather than any methods themselves. Rust generics are fully type-checked at compilation time, i.e. Rust supports "strong concepts".

3.2.1 Encapsulated version coded in Rust

The encapsulated Rust version of our test problem, that is given in the following Listing 3.2, is in its outline quite similar to the corresponding Go version[1]. There are, however, a few differences, that are listed in the following notes, and in our concluding remarks.

- Rust uses angled brackets, `< >`, to indicate generic parameter lists.
- Generics constraints in Rust are typically enforced by specifying the required traits in `impl` blocks using `where` statements.
- Rust does not offer an equivalent to Go's type set syntax and semantics. In order to enable numeric operations on generic types, Rust instead provides a `Num` trait via an external `num` crate (i.e. library). However, the use of this external dependency makes available neither the `AddAssign` (`+=`) operator, nor casts to generic types within generic routines (as they are provided in Go).
- Since the employed algorithm relies on these features, the following code uses a homemade `INumeric` trait, that derives from Rust's `Num` and `AddAssign` traits, and extends them by the requirement to also implement a constructor via an associated function, `new`, that provides the needed type casting (notice that `Self` stands in here for the implementing type). The conformance of the `i32` and `f64` intrinsic types to the `INumeric` trait is then acknowledged via `impl` blocks, that furthermore implement any of the still outstanding functionality of this trait for these two types.
- Where necessary, use of the `Copy` trait is also made, to work around Rust's default move semantics.
- In order to help make all of the source code dependencies explicit, our Rust version employs modules, and `use` statements to import the required functionality.

[1] Notice that the present Rust version makes universal use of dynamic method dispatch via trait objects, in order to correspond most closely to all the other implementations that we provide in both the present chapter, and in Sect. 6.2.1. An alternative, more idiomatic, Rust version that is equivalent to the Fortran version which we'll give in Sect. 6.2.2, and that effects static dispatch of the various `sum` methods through the use of generics, can be found in the `Code` subdirectory that is accompanying this document.

- Rust's default structure constructors suffer from the same flaw as Fortran's. That is, they are unable to initialize from an external scope, structure components that are declared being private to their module. As in Fortran, use of user-defined constructors must be made instead (cf. the `new` functions that are defined in separate `impl` blocks for the ADTs `PairwiseSum` and `Averager`).
- To declare run-time polymorphic variables one has to put so-called "trait objects" into "Boxes", i.e. to declare smart pointers of them, for dynamic instantiation and heap memory allocation (this is the Rust equivalent to using `allocatable` polymorphic objects in Fortran).

Listing 3.2: Encapsulated Rust version of the array averaging example.

```
pub mod interfaces {

    use num::Num;
    use std::ops::AddAssign;

    // ...........
    // Interfaces
    // ...........

    pub trait INumeric: Num + AddAssign {
        fn new(n: usize) -> Self;
    }

    pub trait ISum<T> {
        fn sum(&self, x: &[T]) -> T;
    }

    pub trait IAverager<T> {
        fn average(&self, x: &[T]) -> T;
    }
}

pub mod intrinsics {

    use crate::interfaces::INumeric;

    impl INumeric for i32 {
        fn new(n: usize) -> i32 {
            return n as i32
        }
    }

    impl INumeric for f64 {
        fn new(n: usize) -> f64 {
            return n as f64
        }
    }

}

pub mod simple_library {

    use crate::interfaces::{INumeric,ISum};

    // ..............
```

```rust
    // SimpleSum ADT
    // ..............

    pub struct SimpleSum;

    impl<T> ISum<T> for SimpleSum where T: INumeric + Copy {
        fn sum(&self, x: &[T]) -> T {
            let mut s: T;
            s = T::new(0);
            for i in 0 .. x.len() {
                s += x[i];
            }
            return s
        }
    }

}

pub mod pairwise_library {

    use crate::interfaces::{INumeric,ISum};

    // ................
    // PairwiseSum ADT
    // ................

    pub struct PairwiseSum<T> {
        other: Box<dyn ISum<T>>,
    }

    impl<T> PairwiseSum<T> where T: INumeric {
        pub fn new(other: Box<dyn ISum<T>>) -> PairwiseSum<T> {
            PairwiseSum{
                other: other,
            }
        }
    }

    impl<T> ISum<T> for PairwiseSum<T> where T: INumeric {
        fn sum(&self, x: &[T]) -> T {
            if  x.len() <= 2 {
                return self.other.sum(x);
            } else {
                let m = x.len() / 2;
                return self.sum(&x[..m+1]) + self.sum(&x[m+1..]);
            }
        }
    }

}

pub mod averager_library {

    use crate::interfaces::{INumeric,ISum,IAverager};

    // ..............
    // Averager ADT
    // ..............

```

```
    pub struct Averager<T> {
        drv: Box<dyn ISum<T>>,
    }

    impl<T> Averager<T> where T: INumeric {
        pub fn new(drv: Box<dyn ISum<T>>) -> Averager<T> {
            Averager{
                drv: drv,
            }
        }
    }

    impl<T> IAverager<T> for Averager<T> where T: INumeric {
        fn average(&self, x: &[T]) -> T {
            return self.drv.sum(&x) / T::new(x.len());
        }
    }

}
// ..............
// main program
// ..............
#[macro_use] extern crate text_io;

fn main() {
    use crate::interfaces::IAverager;
    use crate::simple_library::SimpleSum;
    use crate::pairwise_library::PairwiseSum;
    use crate::averager_library::Averager;

    let avsi = Averager::new(Box::new(SimpleSum{}));
    let avsf = Averager::new(Box::new(SimpleSum{}));

    let avpi = Averager::new(Box::new(PairwiseSum::new(Box::new(SimpleSum{}))));
    let avpf = Averager::new(Box::new(PairwiseSum::new(Box::new(SimpleSum{}))));

    let mut avi: Box<dyn IAverager::<i32>> = Box::new(avsi);
    let mut avf: Box<dyn IAverager::<f64>> = Box::new(avsf);

    let xi : [i32;5] = [1,2,3,4,5];
    let xf : [f64;5] = [1.,2.,3.,4.,5.];

    let key: i32;

    println!("Simple   sum average: 1");
    println!("Pairwise sum average: 2");
    scan!("{}\n",key);

    match key {
        1 => {}
        2 => { avi = Box::new(avpi);
               avf = Box::new(avpf); }
        _ => { println!("Case not implemented!");
               return; }
    }

    println!("{}", avi.average(&xi));
    println!("{}", avf.average(&xf));
}
```

The Rust program version is somewhat longer than the corresponding Go version, because user-defined constructors had to be provided, in order to implement the aforementioned custom type conversion functionality and to initialize (opaque) module-hosted abstract data types, and because dependencies from modules had to be imported into the `main` function (as it would be necessary in realistic situations). The `main` function's logic could also not be expressed as concisely as in the Go version, because the various trait objects required "boxing" for their instantiation, because Rust's default move semantics had to be worked around, and because Rust wouldn't allow the declaration of variables without simultaneous initialization.

The most important difference between the two languages, though, is that Rust uses traits (i.e. interfaces) as generics constraints whose function signatures are always specified explicitly, whereas Go's are typically specified implicitly (through type sets). Together with Rust's capability (that Go lacks) to explicitly implement traits even for language-intrinsic types, this allows Rust code to fully (i.e. strictly) conform to OO programming's Open/Closed Principle. In order to enable, for instance, the code of Listing 3.2 to accept also the `f32` type, one would simply need to add a further module, that implements the `INumeric` trait for this new intrinsic type. None of the pre-existing code would have to be changed.

3.3 Swift

Being a successor language to Objective-C, Swift differs slightly from the languages considered so far in that it opted to retain implementation inheritance for backwards compatibility to Objective-C, whereas both Go and Rust do not support implementation inheritance *by design*. Swift therefore supports "classical" classes, but it also allows one to bind methods to structures (which, in contrast to classes, are value types in Swift).

Like Go and Rust, Swift (furthermore) supports a traits system in order to implement both runtime and compile-time polymorphism through polymorphic interfaces, that are called "protocols" in Swift. If the Swift programmer chooses to ignore implementation inheritance and classes, he can therefore very much program with structures and protocols in Swift as he would with structures and interfaces/traits in Go and Rust, respectively.

Given Swift's backwards compatible design, implementation of a protocol (i.e. interface inheritance) is usually done as in classical OO languages, i.e. within a structure's or a class's definition. A colon (`:`) followed by one or more interface names must be supplied for this purpose after the structure's or class's own name. However, a very powerful facility for types to implement protocols retroactively is also provided, through so-called `extensions`, that work even if the types' source code is inaccessible (because one is, e.g., working with a library in binary form). This same facility also allows for protocols to be implemented by language-intrinsic types. For instance, the following little program, given by Listing 3.3, prints out "`I am 4.9`".

Listing 3.3: Swift example of implementing a protocol for an intrinsic data type.

```
protocol IPrintable {
    func output()
}
extension Float64: IPrintable {
    func output() {
        print("I␣am␣\(self)")
    }
}
var y: Float64 = 4.9
y.output()
```

Swift generics support "strong concepts", and are thus fully type-checked at compile time, and their capabilities are on par with those of Go and Rust. In one aspect they are even superior, namely in that Swift allows for parameterized *methods*, instead of parameterized protocols. This has some interesting, positive implications for the Swift programmer, that will be discussed in detail below.

3.3.1 Encapsulated version coded in Swift

Listing 3.4 gives an example of how the encapsulated version of the array averaging test problem can be programmed in Swift. See the following remarks in order to understand this code:

- By default, function and method calls in Swift need to make use of mandatory keyword arguments.
- Array slices are not arrays themselves. Hence, an explicit conversion of such slices via an `Array()` constructor is required to use them as arrays.
- Like Rust, Swift uses angled brackets to indicate generic parameter lists. Type constraints are formulated within these lists by supplying a protocol name after a generic type parameter (separated by a colon).
- Similar to Rust, Swift provides a `Numeric` protocol (that is defined in its standard library) for use as a generics constraint for numeric types, which however does *not* include the division operation!
- It is easy to extend this protocol to also support the division operator, similar to the way we've extended traits in the Rust code example. However, the following code goes a step further than that, and foregoes any dependence on library functionality whatsoever, by defining a custom `INumeric` protocol that prescribes the interfaces of all the required operators and type initializers (i.e. constructors) explicitly.
- The present Swift code makes use of the language's default, built-in, initializers for all the `structs` and intrinsic types that it employs.
- Type conversion into a generic type, `T`, is effected in Swift by calling an appropriate initializer of this type. The call syntax is equivalent to that of Go's generic casts, but is a bit peculiar in that, e.g., Go's `T(0)` is written as `T(exactly:0)!` in Swift (making use of both the initializer's mandatory keyword, `exactly`, and the `!` operator to unwrap the returned optional type). Notice, how the initializer's signature needs to be prescribed in protocol `INumeric`, in order to be able to use this functionality with some conforming type `T`.
- Swift's `Int32` and `Float64` intrinsic types already implement all of protocol `INumeric`'s functionality by default. It therefore suffices to simply acknowledge this fact through empty `extension` statements for these two types.

Listing 3.4: Encapsulated Swift version of the array averaging example.

```
// ...........
// Interfaces
// ...........

protocol INumeric {
    init?(exactly: Int)
    static func += (lhs: inout Self, rhs: Self)
    static func + (lhs: Self, rhs: Self) -> Self
    static func / (lhs: Self, rhs: Self) -> Self
}

protocol ISum {
    func sum<T: INumeric>(x: [T]) -> T
}

protocol IAverager {
    func average<T: INumeric>(x: [T]) -> T
}

// ...........
// Intrinsics
// ...........

extension Int32: INumeric {}
extension Float64: INumeric {}

// ..............
// SimpleSum ADT
// ..............

struct SimpleSum: ISum {

    func sum<T: INumeric>(x: [T]) -> T {
        var s: T
        s = T(exactly:0)!
        for i in 0 ..< x.count {
            s += x[i]
        }
        return s
    }
}

// .................
// PairwiseSum ADT
// .................

struct PairwiseSum: ISum {
    var other: any ISum

    func sum<T: INumeric>(x: [T]) -> T {
        if ( x.count <= 2 ) {
            return other.sum(x: x)
        } else {
            let m = x.count / 2
            return sum(x: Array(x[..<m])) + sum(x: Array(x[m...]))
        }
    }
```

```
58
59 }
60
61 // ............
62 // Averager ADT
63 // ............
64
65 struct Averager: IAverager {
66     var drv: any ISum
67
68     func average<T: INumeric>(x: [T]) -> T {
69         return drv.sum(x: x) / T(exactly: x.count)!
70     }
71 }
72
73 // ..............
74 // main function
75 // ..............
76
77 func main() {
78     let avs = Averager(drv: SimpleSum())
79     let avp = Averager(drv: PairwiseSum(other: SimpleSum()))
80
81     var av : any IAverager = avs
82
83     let xi: [Int32] = [1,2,3,4,5]
84     let xf: [Float64] = [1.0,2.0,3.0,4.0,5.0]
85
86     var key: Int32?
87
88     print("Simple␣␣␣␣sum␣average:␣1")
89     print("Pairwise␣sum␣average:␣2")
90     print("Choose␣an␣averaging␣method:␣")
91     key = Int32(readLine()!)
92
93     switch key {
94     case 1:
95         // simple sum case
96         av = avs
97     case 2:
98         // pairwise sum case
99         av = avp
100    default:
101        print("Case␣not␣implemented!")
102        return
103    }
104
105    print( av.average(x: xi) )
106    print( av.average(x: xf) )
107 }
108
109 // execute main function
110 main()
```

Even a casual glance at the Swift version will show that the Swift code is the easiest to read and understand among the generic object-oriented implementations that were presented in this chapter. This is largely the result of Swift supporting generic methods, and hence not requiring the programmer to parameterize and instantiate any generic interfaces (protocols), in contrast to

both Go and Rust. The consequences are

- that method genericity for an ADT's objects can be expressed using only a single, as opposed to multiple protocols,
- that therefore merely a *single* object instance of that same protocol is required, in order to be able to operate on many different language-intrinsic data types, and
- that this also (largely) *obviates the need for manual instantiations of generics in Swift* (because generic functions/methods are easier to instantiate automatically by the compiler, as it can almost always infer the required types by checking the regular arguments that are passed to a function/method)!

As an example, consider the object `av` in the above Swift code that contains the functionality for array averaging. This object supports two different levels of polymorphism: Firstly, given that it is an instance of the `IAverager` protocol, it can be polymorphically assigned different averaging algorithms (see the `switch` statement). Secondly, because it contains an `average` method that is generic, it can be used on data of different intrinsic types, like `Int32` and `Float64` here.

Notice that the `main` function in the Swift code needs to declare merely a *single* such object variable of `IAverager` type, to make use of all these capabilities. This is a direct consequence of there being only a single (i.e. unparameterized) version of the `IAverager` protocol, and of parameterizing the protocol's method signatures by generic types rather than the protocol itself.

Contrast this with Go's and Rust's model, where not only separate objects of `IAverager` type are required for *every* different intrinsic data type that the programmer wishes to use these objects with. But where also *manual* instantiations of corresponding versions of the generically parameterized `IAverager` interface/trait are required from the programmer, for declaring these objects. Swift's generics model gets rid of all of that complexity, and therefore vastly simplifies client code. We consider this a very significant advantage of the generics approach that is taken in Swift vs. that of Go and Rust.

Otherwise, Swift shares Rust's mechanism of explicitly formulating and implementing interfaces for use as generics constraints. This allows for strict conformance to the Open/Closed Principle, and therefore for full support of modern-day generic OO programming.

3.4 Conclusions

The use of run-time polymorphism by means of (polymorphic) interfaces is rather similar in all the languages that were considered here. The most significant difference in this respect is that Go has stricter limitations than the other languages regarding code ownership, when it comes to retroactively implementing interfaces for existing types. Whereas Rust (with some minor restrictions), and Swift allow the implementation of an interface by some type to be accomplished effectively without regard to the type's definition site. Rust and Swift thereby overcome Haveraaen et al.'s critique [5] of Java regarding this point. In fact, it is *interface inheritance* which makes the uniform polymorphic treatment of both intrinsic and user-defined types possible in the first place in Rust and Swift, that Haveraaen et al. seem to also (rightly) demand. All the considered languages are also quite similar in that they support fully type checked generics via the mechanism of interfaces. In the following, we will thus focus on summarizing the most significant differences in these languages' generics features.

3.4.1 Go

Go's basic model to implement generics allows structures, interfaces, and ordinary functions, but not methods, to be given their own generic type parameters. The lack of true generic methods makes some duplication of instantiation code in clients unavoidable. Nevertheless, generic Go code is quite easy to read and to understand. Go features built-in, easy to use support for conversion to generic types. Yet, where Go truly differs from both Rust and Swift are the unique, and extremely useful, implicit mechanisms for conformance to interfaces that it supports, namely structural subtyping, and the brilliant new notion of formulating interfaces in terms of type sets, to which the member types of these sets conform by definition.

The latter idea, along with the corresponding syntax to support it, enables the Go programmer to formulate generics constraints very naturally and concisely, without having to explicitly implement any methods for this purpose. This is what makes the use of generics in Go pleasant. It is, moreover, essential for supporting concise generic procedural and functional programming. While type sets retain their advantages also in generic object-oriented programming, their disadvantage in this setting is that they allow for only partial, but not full, conformance to object-orientation's Open/Closed Principle.

3.4.2 Rust

Rust's basic model for generics is similar to Go's in that it allows for parameterization of structures, interfaces, and ordinary functions, but not necessarily methods. Hence, what has been said above for Go in this respect holds also for Rust. Rust has, unfortunately, some quirks which render its use for the management of all types of dependencies through polymorphism somewhat sub-optimal when compared to the other languages considered here. The language is unpleasant to use, because of its "borrow checker", its employment of move semantics by default, its *excessive* obsession with type safety, and its overall C++-like philosophy to copiously rely on external dependencies, even for the most basic tasks.

Because of the latter two points, the Rust version of our test case is marred by some dependencies on external libraries, which is quite contrarian to the purpose of programming in a polymorphic fashion, namely to avoid rigid dependencies. Even with the functionality provided by these external dependencies, casts to generic types within generic routines weren't outright possible, and had to be achieved, instead, through some work on the programmer's side. The points we like most about the language are its idea to decouple trait implementations from a `struct`'s definition through `impl` blocks that work with both user-defined and intrinsic types, the full conformance to the Open/Closed Principle that this enables, and the complete control over the use of dynamic vs. static method dispatch (via trait objects and generics, respectively) that Rust affords the programmer. These are the features of Rust that, in our opinion, Fortran should borrow in some form.

3.4.3 Swift

Swift's basic model of implementing generics by allowing parameterized structures, functions, and methods (but not parameterized interfaces) is both the easiest to read, and the easiest to use from a programmer's perspective. Swift's generics design supports casts to generic types, regardless of whether these types are user- or language-defined. Moreover, the Swift compiler is able to instantiate generics largely automatically, through type inference of the regular arguments that are passed to functions, methods, and (structure or class) constructors. In contrast to the other

languages, in Swift, the user almost never has to bother with instantiating any generics.

If the Swift programmer knows how to write generic functions, his knowledge automatically translates into coding generic methods, since generic functions can be transformed into generic methods without requiring changes to their function signatures. This property is helpful for the refactoring of non-OO codes into OO versions. Like Rust, Swift enables explicit and retroactive implementation of interfaces for both user-defined and intrinsic types, and the full conformance to the Open/Closed Principle of OOP that this affords one. Unlike Rust and Go, Swift also allows for constructor, operator, and function overloading. Out of the considered languages, it is therefore the closest to how Fortran works.

For all these reasons, we consider Swift's generics to be the most attractive model to base Fortran's basic generics capabilities on, provided that it can be implemented sufficiently easily. The fact that Swift is a language that does not put emphasis on numerics, and whose present standard library therefore does not provide a truly useful `Numeric` protocol (that supports all the usual numeric operations), is of absolutely *no* consequence for adopting Swift's generics design as a baseline for Fortran. We believe, though, that this baseline design would benefit substantially from being suppplemented with Go's implicit mechanisms for interface conformance.

Especially for properly supporting generic procedural and functional programming in Fortran, i.e. for making them *sufficiently concise*, Go-like type sets should be added. These would also be useful in generic object-oriented programming, even though Swift's design already comes with good support for the latter, right out of the box. Its single minor disadvantage in this respect, namely the need to always provide (possibly empty) explicit `extension` blocks to implement a new interface, even if the involved types already command over all of the functionality that is prescribed by that interface, could be overcome by allowing for Go-like structural, in addition to nominal, subtyping. A conclusion that seems to have been arrived at also in the recent Mojo language [14]. See Sects. 6.2.3 and 7.2 for some further discussion of this point.

Chapter 4

Fortran extensions I: Traits for types

The present and the next chapter, describe a number of simple extensions to Fortran, that we consider essential in order to enable dependency management through polymorphism at a level of functionality that is on par with modern languages like Swift, Rust, Go, Carbon, or Mojo. The present chapter adds general subtyping capabilities to Fortran, while the next chapter aims at providing specific support for generics. The extensions that are related to subtyping concern abstract interfaces, derived types, and the `class` specifier for variable declarations. They also encompass an important, new, Fortran feature: the `implements` statement.

4.1 Traits (named abstract interfaces)

The most essential of all the following extensions is the capability to define named abstract interfaces, or traits (i.e. named collections of procedure signatures), in order to suitably constrain (and hence to type-check) the declarations of polymorphic variables. Traits are the crucial feature that is required in order to uniformly and properly express both run-time and compile-time polymorphism (i.e. modern-day OO and generic programming, respectively) in the language, and to thereby enable a uniform management of dependencies on both user-defined *and* language-intrinsic types.

4.1.1 Definition

Fortran already allows the programmer to define unnamed abstract interfaces. But in order to use these as traits, named versions of them are required. Moreover, and in order to support all of the functionality that is discussed in the next sections, these interfaces must admit signatures of not only ordinary type-bound procedures, but also initializers (i.e. constructors), and operators (including assignments). The following example defines three such traits, `ICastable`, `ICalculable`, and `IPrintable`, that are intended as abstract blueprints for actual type-bound implementations of, respectively, an initializer, `init`, a function, `saxpy`, and a subroutine, `output`:

```
1  abstract interface :: ICastable
2     function init(n)
3        integer, intent(in) :: n
4     end function init
5  end interface ICastable
6
7  abstract interface :: ICalculable
8     function saxpy(a,x,y) result(res)
```

```
9         real, intent(in) :: a, x(:), y(:)
10        real             :: res(:)
11     end function saxpy
12  end interface ICalculable
13
14  abstract interface :: IPrintable
15     subroutine output()
16     end subroutine output
17  end interface IPrintable
```

Examples of traits with signatures of type-bound operators are given in Sects. 5.1.3, 6.2.3, and 6.3. Notice, how initializers within traits are declared as *special* functions, that have a reserved name, `init`, and for which no return type is specified, since the latter is considered to be already fully determined by the initialized type. This is equivalent to how initializers are declared in Swift protocols.

It should also be pointed out that traits contain the signatures of *messages*, which in OO and generic programming may be sent to different receivers. A well-designed traits facility should therefore *not* require such message signatures to contain passed-object dummy arguments, or their types, i.e. any details concerning the receivers. Because in order to be sendable to different receivers, a message must not know anything about its receiver. This is clearly exhibited by the present examples, and also the Go and Swift examples[1] of Listings 3.1 and 3.4, but is violated in present Fortran practice by `deferred` type-bound procedures. We will elaborate further on this in Sect. 4.3.3. There, we will also show how to fix this problem, while achieving interoperability of the new traits facility with the inheritance feature that is already available in the present language.

4.1.2 Overloading

In the examples that were considered so far, traits were shown to contain non-overloaded message signatures. But since Fortran supports "`generic`" overloading, traits must also admit *overloaded* versions of initializer, method, operator, and assignment signatures – that share a generic name, but differ in their arguments lists. The following trait, `IConstructable`, illustrates this for a type-bound initializer, by overloading the reserved name `init` with two such signatures:

```
1  abstract interface :: IConstructable
2     function init(n)
3        integer, intent(in) :: n
4     end function init
5     function init(a)
6        real, intent(in) :: a
7     end function init
8  end interface IConstructable
```

Traits for all the other cases can be formulated analogously – through overloaded use of a method name, or the reserved identifiers `operator(`*`op`*`)`, and `assignment(=)`, respectively (with *`op`* as per [16]). An example of an overloaded multiplication operator signature is given in Sect. 6.3.

4.1.3 Extends attribute

Abstract interface definitions must allow the programmer to define new traits that inherit procedure signatures from *multiple* simpler traits (multiple interface inheritance). In the following

[1] Notice, especially, that despite the fact that Go (like Fortran) uses explicitly passed receiver objects, these appear only in the actual *implementations* of methods, and even there they are clearly *syntactically separated* from the latter's interfaces.

example, the trait `IAdmissible` inherits the procedure signatures that are contained in both the `ICastable` and `IPrintable` traits of Sect. 4.1.1, making `IAdmissible` at the same time a *subtype* of both these simpler traits:

```
1  abstract interface, extends(ICastable,IPrintable) :: IAdmissible
2  end interface IAdmissible
```

That is, objects that implement (or adopt) the `IAdmissible` trait (i.e. conform to it), can also be used in settings that require conformance to either the `ICastable`, or `IPrintable` traits.

The next example finally shows an extended version of the `IAdmissible` trait, that not only inherits the aforementioned procedure signatures, but also adds the signature of a further method, `input`, to them:

```
1  abstract interface, extends(ICastable,IPrintable) :: IAdmissible
2     subroutine input(filename)
3        character(*), intent(in) :: filename
4     end subroutine input
5  end interface IAdmissible
```

4.2 Implements statement

The language must allow not only for traits to conform to other traits (cf. Sect. 4.1.3), but also for both language-intrinsic and user-defined types to do the same. That is, it must be possible for these types to acknowledge their conformance to a trait, if they already command over implementations of all its functionality, or to provide any still outstanding implementations that are required for such conformance. Moreover, this must be possible even retroactively, i.e. regardless of (and without having to touch) any original type definition site[2]. Otherwise, wrapper types would, in general, need to be written, once an already defined type would need to implement some new trait (e.g. in cases where the original type definition is inaccessible). This also means that the provision of user-defined functionality needs to be allowed even for *language-intrinsic* types[3]. The `implements` statement, that is described in this section, provides these capabilities.

4.2.1 Overview

The `implements` statement is modeled after the "`extension`" feature of Swift, to enable retroactive implementation of new methods, initializers, operators, and especially traits for types, in order to *dynamically* change a subtyping (i.e. interface inheritance) hierarchy, and thus achieve utmost code flexibility. Swift's `extension` blocks fulfill essentially the same purpose as Rust's `impl` blocks. They have been somewhat simplified here (for a first implementation), and adjusted to Fortran's idiosyncracies and syntax, that binds procedures to types through declaration blocks, rather than by including the actual implementations themselves into such a block (the implementations need to be supplied as module procedures, as it is usual in Fortran).

The syntax of the feature is an extended subset of that for the `contains` sections of derived type definitions. That is, `implements` statements make use of type-bound procedure declarations. These employ mostly the same attributes (i.e. `public`, `private`, `pass[(arg-name)]`, `nopass`, and `non_overridable`) which are also allowed in derived type definitions. Some new syntax for these declarations is provided, to enable direct overloading of method names, and to thus make it less verbose than using `generic` type-bound procedure declarations (as the Fortran standard calls

[2] Exempting only the case of abstract derived types, as it is discussed below.

[3] With some restrictions. See Sect. 4.2.4.

them). However, the latter are supported as well. Furthermore, a new type of `initial` procedure declarations is available, for specifying type-bound initializers (i.e. constructors).

The main difference to derived type definitions is that the `implements` statement does not accept `deferred` type-bound procedure declarations (which we propose to retire partly or entirely, see Sect. 4.3.3), and that it cannot be used with `abstract` derived types. If interoperability with `abstract` types is required, the later to be discussed `implements` derived type attribute (cf. Sect. 4.3) can be used instead. As a final disclaimer, notice that the `implements` statement has absolutely *no* relation to subclassing, i.e. one derived type being extended into another through (rigid) implementation inheritance. Rather, this is a feature that adds new capabilities to a single, *given*, type.

4.2.2 Binding functionality to a type

The most basic use of the `implements` statement is to (retroactively) bind procedures of varying functionality to a type. The following example shows how to bind the names of an overloaded, user-provided, initializer, and an ordinary method to a derived type with name `MyType`, and how to provide their actual implementations as module procedures:

```
module basic
   ...
   type :: MyType
      private
      real :: a
   end MyType

   implements :: MyType
      initial :: init => init_by_int, init_by_real
      procedure, pass :: output
   end implements MyType

contains

   function init_by_int(n) result(res)
      integer, intent(in) :: n
      type(MyType)        :: res
      res%a = real(n)
   end function init_by_int

   function init_by_real(a) result(res)
      real, intent(in) :: a
      type(MyType)     :: res
      res%a = a
   end function init_by_real

   subroutine output(self)
      class(MyType), intent(in) :: self
      write(*,*) "I am ", self%a
   end subroutine output

end module basic
```

Notice the use of the new declaration of `initial` type, that instructs the compiler to overload the implementing type's structure constructor with, in this case, multiple type-bound initializer implementations of the (reserved) generic name `init`. Notice, also, that the return type of a module function that actually implements an initializer is always the concrete type that it initializes.

The above example also demonstrates how an ordinary method of name `output` can be bound

to MyType, in quite the same manner as this is accomplished in the present language. That is, by declaring the passed-object dummy argument, self, with the class declaration specifier (as it is required for my MyType to be extensible by implementation inheritance).

4.2.3 Implementing traits

Retroactive implementation

Assume now, that the purpose of binding the previous procedures to MyType was to actually make this type compatible with settings where conformance to the IPrintable and IConstructable traits of, respectively, Sects. 4.1.1 and 4.1.2 is required. So far, we have provided both the bindings of the required initializers and method, as well as their implementations – making sure that the signatures of the latter match those that are contained in the IConstructable and IPrintable traits, by ignoring any passed-object arguments (which a compiler will simply skip, in the present design, when checking for trait, i.e. interface, conformance of an implementation).

However, we haven't made MyType pluggable yet into code that is written in terms of these traits. To fix this, we can simply acknowledge (even from a different module, as in the following example) that MyType already has all of the required functionality to implement these traits:

```
module enhanced
   ...
   use basic, only: MyType

   implements (IConstructable,IPrintable) :: MyType
   end implements MyType

end module enhanced
```

It is crucial for flexibility, that the subtyping (i.e. interface inheritance) mechanism, that the implements statement provides, allow for a type to implement *multiple* different traits – as it is demonstrated in this example.

Collective implementation

The last code example has shown how traits can be implemented after the procedure implementations, that they require, were already bound to some type. Of course, it is also possible to do all of this at one fell swoop, as in the following alternative version of the module basic, whose original version was given in Sect. 4.2.2:

```
module basic
   ...
   type :: MyType
      private
      real :: a
   end MyType

   implements (IConstructable,IPrintable) :: MyType
      initial :: init => init_by_int, init_by_real
      procedure, pass :: output
   end implements MyType

contains
...
end module basic
```

To avoid needless repetition, we have omitted here the implementation of the actual procedures, which would be done exactly as it was shown in Sect. 4.2.2.

Split implementation

We could have also employed two separate `implements` statements, in order to implement one trait at a time, and achieve the same effect:

```
module basic
   ...
   type :: MyType
      private
      real :: a
   end MyType

   implements IConstructable :: MyType
      initial :: init => init_by_int, init_by_real
   end implements MyType

   implements IPrintable :: MyType
      procedure, pass :: output
   end implements MyType

contains
   ...
end module basic
```

Here we have skipped again, for brevity, the actual implementations. Such a splitting of `implements` statements can be useful to improve code readability, as it makes the association between the traits and the actual procedures, that are to be implemented for any one of them, immediately obvious. These two statements (together with the actual implementations), could then have been distributed even among different modules and files. Notice, also, how parentheses around traits lists in `implements` statements are optional, but not required.

4.2.4 Implementing traits for intrinsic types

The `implements` statement can be used to provide new functionality also for types that are *intrinsic* to the language – in quite the same fashion as it was already demonstrated for derived types. However, the present language features some restrictions in this respect, that we deem essential to retain. In particular, present Fortran doesn't allow one to overload the intrinsic operators that the language defines for intrinsic types. Which is what makes numeric Fortran code *predictable*. Any present restrictions that serve this latter purpose should therefore continue to be honored.

The following listing, which shows how to code the Swift example of Listing 3.3, is a case that is unaffected by these considerations. It implements the `IPrintable` trait, with its single method `output`, for Fortran's `real(real64)` type, in order to perform printouts for variables of this type:

```
module real64_module

   use, intrinsic :: iso_fortran_env, only: real64

   abstract interface :: IPrintable
      subroutine output()
      end subroutine output
   end interface IPrintable

```

```
10    implements IPrintable :: real(real64)
11       procedure, pass :: output
12    end implements real(real64)
13
14 contains
15
16    subroutine output(self)
17       real(real64), intent(in) :: self
18       write(*,*) "I␣am␣", self
19    end subroutine output
20
21 end module real64_module
22
23 program printy
24
25    use, intrinsic :: iso_fortran_env, only: real64
26
27    real(real64) :: y
28
29    y = 4.9d0
30    call y%output()
31
32 end program printy
```

Notice, that once an `implements` statement has been used to augment the functionality of an existing (intrinsic or derived) type (like the `real(real64)` type in the above example), then this new functionality will be available on *all* instances of that type, even if they were created before this augmentation was defined. As it is the case in the Swift language [19], this prevents the occurrence of any ambiguities regarding the available functionality of such type instances. For the same reason, it is sufficient to import into client code (as usual) either a named constant corresponding to an intrinsic type's kind parameter (like the `real64` constant in the above `printy` example program), or a derived type's (original) definition, in order to make use of the thus augmented functionality.

4.3 Implements attribute for derived types

For reasons of both regularity in the language, and interoperability with Fortran's legacy (i.e. present) OO model, it should be possible to implement traits also directly from within derived type definitions. This can be accomplished by allowing an `implements` *attribute* within such definitions, that – in contrast to the previously discussed `implements` statement – features also interoperability with `abstract` derived types, and their extensions by implementation inheritance.

By its very nature of being bound to derived type definitions, the former attribute, though, necessarily lacks the latter statement's retroactive implementation capabilities, that are important for exploiting all the possibilities that the modern traits-based OO programming model offers. The latter model is a much more flexible, complete, modern-day alternative to both `abstract` types and type extension (i.e. implementation inheritance). Hence, our recommendation is to use the interoperability features, that are described in the following Sects. 4.3.2 and 4.3.3, mainly as migration tools (for modernizing legacy applications incrementally, through refactoring), and to otherwise avoid the use of both `abstract` types and type extension in new programs.

4.3.1 Use with the sealed attribute

The following code shows how the "Collective implementation" example of Sect. 4.2.3, that was written there in terms of an `implements` statement, can be reformulated to use an `implements` derived type attribute:

```
module basic
   ...
   type, sealed, implements(IConstructable,IPrintable) :: MyType
      private
      real :: a
   contains
      initial :: init => init_by_int, init_by_real
      procedure, pass :: output
   end implements MyType

contains

   function init_by_int(n) result(res)
      integer, intent(in) :: n
      type(MyType)        :: res
      res%a = real(n)
   end function init_by_int

   function init_by_real(a) result(res)
      real, intent(in) :: a
      type(MyType)     :: res
      res%a = a
   end function init_by_real

   subroutine output(self)
      type(MyType), intent(in) :: self
      write(*,*) "I␣am␣", self%a
   end subroutine output

end module basic
```

This example also demonstrates the use of a second, new, derived type attribute, the `sealed` attribute. Derived types that are `sealed` are inextensible by type extension, i.e. implementation inheritance. Thus, there's also no need for the passed-object dummy arguments of such types to be polymorphic, and hence for the programmer to declare them with the `class` specifier. One can use the `type` specifier, instead, as it is demonstrated by the implementation of method `output` in the above example.

4.3.2 Use with the extends attribute

The previous Sect. 4.3.1 has illustrated how the `implements` derived type attribute would be typically used in modern code (that avoids the use of implementation inheritance, or subclassing). However, the multiple interface inheritance (or subtyping) features, that are provided by the `implements` derived type attribute[4], can actually be combined with the type extension capabilities which are already present in the language.

In the following example, a `Parent` type is defined, that implements a `method1`. The implementation of the latter is then inherited by a `Child` type, that, in turn, implements an own `method2`. In this way, the `Child` type now commands over implementations of both `method1`, and `method2`, as

[4] And, with some restrictions, also the `implements` statement.

it is required for it to conform to both the `IParent` and `IChild` traits, respectively, that it adopts. Which consist of the signatures of these methods.

```
module extends_parent

   abstract interface :: IParent
      subroutine method1(n)
         integer, intent(in) :: n
      end subroutine method1
   end interface IParent

   abstract interface :: IChild
      subroutine method2(res)
         real, intent(out) :: res
      end subroutine method2
   end interface IChild

   type, implements(IParent) :: Parent
   contains
      procedure, pass :: method1
   end type Parent

   type, extends(Parent), implements(IParent,IChild) :: Child
   contains
      procedure, pass :: method2
   end type Child

contains

   subroutine method1(self,n)
      class(Parent), intent(in) :: self
      integer,       intent(in) :: n
   end subroutine method1

   subroutine method2(self,res)
      class(Child), intent(in)  :: self
      real,         intent(out) :: res
   end subroutine method1

end module extends_parent
```

In cases like the present one, where the `extends` and `implements` derived type attributes are used in combination, `extends` shall always precede `implements`. Since the latter attribute allows for the inheritance of *multiple* interfaces/traits, as it is demonstrated in this example, this also fixes present Fortran's single inheritance limitations, *without* introducing the potential ambiguities that multiple inheritance of implementation would cause (which are also known as "The Diamond Problem").

4.3.3 Use with the abstract attribute

The two previous examples of the `implements` derived type attribute, that were presented in Sects. 4.3.1 and 4.3.2, can, in fact, be coded equally well using `implements` statements. Yet, the `implements` attribute provides one capability that the `implements` statement lacks, namely implementation of traits even if the implementing derived type is `abstract`.

It must be noted, in this context, that the `implements` attribute provides a new, far more general, facility for requiring derived types to provide implementations of abstract methods, than the

present language's *flawed* (non-generalizable) mechanism of declaring methods with the `deferred` attribute. The latter feature's way of prescribing abstract method signatures differs from the traits based `implements` mechanisms that are proposed here, in that it generally requires these signatures to contain entirely *superfluous* passed-object dummy arguments, which unnecessarily and wrongly *intermix* the receiver of a message (i.e. the passed-object), with the interface of the message itself.

This is a fundamental design flaw in the present language that required correction, and couldn't be simply carried over to the design of the traits based facilities, because it would have compromised all the new features that are based upon traits. In order to avoid having (in the long run) two subtly different ways of expressing abstract method signatures in the language, we propose to first make the specification of passed-object dummy arguments, in abstract interfaces of `deferred` methods, optional rather than mandatory, and to subsequently declare obsolescent such mandatory specification. Alternatively, Fortran's `deferred` type-bound procedure declarations could be retired in their entirety, in favor of using traits, as it is demonstrated in the following example:

```
module abstract2b

   abstract interface :: IDeferred
      subroutine method2(res)
         real, intent(out) :: res
      end subroutine method2
   end interface IDeferred

   type, abstract, implements(IDeferred) :: Parent
   contains
      procedure, pass :: method1
   end type Parent

   type, extends(Parent) :: Child
   contains
      procedure, pass :: method2
   end type Child

contains

   subroutine method1(self,n)
      class(Parent), intent(in) :: self
      integer,       intent(in) :: n
   end subroutine method1

   subroutine method2(self,res)
      class(Child), intent(in)  :: self
      real,         intent(out) :: res
   end subroutine method1

end module abstract2b
```

Here, the `abstract` type `Parent` provides an implementation of a `method1`, but it also obliges any type that derives from it to implement the abstract `method2`. In present Fortran, this would be expressed through a `deferred` type-bound procedure declaration within `Parent`, along with the provision of an unnamed abstract interface for `method2`. Whereas in the above example this is formulated in the alternative way of using a trait: `Parent` is made to `implement` the `IDeferred` trait, that contains the abstract signature of `method2`. But since `Parent` is declared, at the same time, with the `abstract` attribute, it can pass this obligation to its children. Hence, the `Child` type, that `extends(Parent)`, is now expected to provide an actual implementation of this method.

More generally, a type that is declared `abstract`, is allowed to provide partial, or no imple-

mentations of the traits that it, or its parents, adopt. Any non-abstract type that `extends` such an `abstract` type through type extension must, however, provide full implementations. This is equivalent to how interfaces work in conjunction with `abstract` classes in the Java language [18].

4.4 Trait objects (run-time polymorphic variables)

The main purpose of named abstract interfaces/traits is to allow the programmer to declare (and thereby constrain) polymorphic variables in terms of them. This can happen either directly, by traits functioning as types in run-time polymorphism (as it is demonstrated in this section), or indirectly, by them acting as constraints on generic type parameters in compile-time polymorphism (see, e.g., Sect. 5.1.1).

4.4.1 Class specifier using a single trait

In order to use traits to declare run-time polymorphic objects through subtyping, Fortran's `class` specifier for polymorphic variable declarations needs to be enhanced to accept traits, like in the declarations of the following two variables (that make use of the `IPrintable` trait of Sect. 4.1.1)

```
1   class(IPrintable), allocatable :: printer
2   class(IPrintable), pointer     :: printerptr
```

or the following declaration of a procedure argument:

```
1   class(IPrintable), intent(in) :: printer
```

The semantics here are that whenever a trait appears within the `class` specifier of an object's declaration, then all the `public` methods of that object whose signatures are prescribed by the adopted trait (like `IPrintable` in the above examples), will make use of dynamic binding. That is, their calls will be resolved by the run-time system of the language (e.g. through a virtual method table). See also Sect. 5.1.4 for further information on the `class` specifier when used with traits.

In accordance with how objects that make use of run-time polymorphism through subclassing (i.e. implementation inheritance) are declared in the present Fortran standard, also "trait objects" (like `printer`, and `printerptr` in the examples above) must either be declared using the `allocatable`, or the `pointer` attribute, or they must be arguments of a procedure. The proposed extensions are therefore backwards compatible with the functionality that is already available in the present language.

4.4.2 Class specifier using trait combinations

The `IAdmissible` trait of Sect. 4.1.3 derives from the `ICastable` and `IPrintable` traits of Sect. 4.1.1. An object that requires the combined functionality of both these latter traits could thus be declared in terms of the `IAdmissible` trait as follows:

```
1   class(IAdmissible), allocatable :: obj
```

It happens often, though, that one needs to declare objects that conform to multiple traits, but where one would like to avoid having to code some (otherwise unneeded) intermediary trait, that derives from these. In such cases, it should be possible to express an object's declaration more directly – by providing a list of trait names to the class specifier, that expresses a traits combination. As in the following example

```
1   class(ICastable,IPrintable), allocatable :: obj
```

in which `obj` again conforms to both the `ICastable` and `IPrintable` traits.

Chapter 5

Fortran extensions II: Generics

The new subtyping features that were discussed in the previous chapter are required in order to uniformly express and support both run-time and compile-time polymorphism in Fortran. We will now proceed with discussing further enhancements that are specifically needed in order to support compile-time polymorphism, i.e. generics.

5.1 Enhancements to abstract interface definitions

5.1.1 Traits with generic procedure signatures

Abstract interface definitions should be allowed to contain the signatures of generically parameterized procedures, as it is the case in the Swift language. The approach that was taken in Go and Rust, to instead parameterize the abstract interfaces themselves, is not as attractive from a user's perspective (cf. Sect. 3.3). As an example, the following code shows a trait that is called `ISum`, which contains a signature that is intended for a generic type-bound function (i.e. generic method) with name `sum`:

```
1  abstract interface :: ISum
2     function sum{INumeric :: T}(x) result(s)
3        type(T), intent(in) :: x(:)
4        type(T)             :: s
5     end function sum
6  end interface ISum
```

The example illustrates the use of a generic type parameter, that is simply called `T` here, in terms of which the regular function arguments are declared. A significant difference of generic type parameters, as compared to regular function arguments, is that the former will be substituted by actual type arguments at compile time, in a process that is called instantiation.

A similarity is that, in the same way that regular function arguments need to be constrained by a provided type, type parameters need to be constrained by a provided meta-type. This (meta-type) constraint must either be the name of a single trait (like `INumeric` in the present example), or a comma separated list of trait names that expresses a traits combination[1]. The proposed Fortran generics thus support "strong concepts", and can be fully type-checked by a compiler.

Both, the type parameter and the constraint which precedes it, and is separated from it by a double colon, are part of a generic type parameter list that is enclosed in curly braces, and follows

[1] For usage examples of traits combinations in run-time and compile-time polymorphism, see Sects. 4.4.2 and 6.3, respectively.

immediately behind the procedure's name. Notice that the syntax that is used here, which deviates slightly from how Fortran's regular function arguments are declared, appears justified, as it reflects that, despite some similarities, in type parameters one is dealing with different entities.

5.1.2 Traits of type sets

In order to make Fortran's traits based generics facility easy to use, the language must allow for generics constraints to be prescribed in an *implicit* manner, via abstract interfaces/traits that are expressed as type sets, as it is possible in the Go language. This is particularly important for admitting the use of selected sets of intrinsic types in generic code (and the multitude of intrinsic procedures that Fortran supports for these types). It is also essential for supporting concise generic procedural and functional programming, where one might wish to avoid having to explicitly implement methods for types (which is, arguably, an OO technique).

Unions of types

The following example shows the simplest form of such a type set. It defines an `INumeric` trait, for use as a generics constraint in the example of Sect. 5.1.1, in order to admit for the type parameter, `T`, that was given there, only the (32 bits wide) default `integer` intrinsic data type:

```
1  abstract interface :: INumeric
2     integer
3  end interface INumeric
```

The above example is actually a special case of specifying entire *unions* of member types as a type set. A type set consisting of such a union of types is demonstrated in the following example

```
1  abstract interface :: INumeric
2     integer | real(real64)
3  end interface INumeric
```

that redefines the `INumeric` trait such as to admit either the default 32 bit `integer`, or the 64 bit `real` type as a generics constraint.

The semantics of such a type set construct are that it implicitly defines a *set of function signatures*, namely the signatures of the intersecting (common) set of all the operations and intrinsic functions (also called methods in the following) that work with all the member types of the type set. This can also be restated, by saying that a type `T` *implements* a trait consisting of such a type set, if (and only if) it is a member type of this set. For instance, the `integer` and `real(real64)` types implement the `INumeric` trait (as it is given above) because they are member types of its type set. In contrast, Fortran's various `complex` types do not implement this trait, because they do not belong to its set of member types. The validity of the latter statement can also be easily checked, by considering that the `complex` types do not support, i.e. implement, the relational operators `(<)` and `(>)`. Implementations of the latter are required for conformance to this trait, because these operators *are* implemented by both the `integer` and `real(real64)` member types.

Assumed kind parameters

Expanding on the previous example, an `INumeric` trait that might be even more useful as a generics constraint, for a number of tasks, could be coded as follows:

```
1  abstract interface :: INumeric
2     integer(*) | real(*) | complex(*)
3  end interface INumeric
```

Notice how this makes use of both unions of types, and assumed (i.e. wildcard) kind parameters for types, to include *all* `integer`, `real`, and `complex` types, that are admitted by the language, in a single `abstract interface` constraint.

The use of assumed kind parameters is here merely syntactic sugar that allows one to avoid having to write out a type set for all the possible kinds of a type. For instance, if the particular Fortran implementation supports `real(real32)` and `real(real64)` as its only `real` types, then `real(kind=*)`, or `real(*)` for short, is understood to mean the type set "`real(real32) | real(real64)`". Notice, also, that the more types are added to a trait in this fashion, the smaller the set of intersecting methods will usually become.

Empty trait

In the limit of adding all possible types to a type set, there won't be any common methods left that are implemented by all its types. This results in the important case of the empty trait, that matches all types (since any type has at least zero methods):

```
1  abstract interface :: IAnyType
2  end interface IAnyType
```

Inline notation

For simple use cases, it should be optionally possible for the programmer to employ a shorter notation for declaring type constraints for generics, than having to separately define a trait of type sets, like `INumeric` above, and to then use it as in Sect. 5.1.1. The following modification of trait `ISum`'s original declaration of Sect. 5.1.1, provides such an example:

```
1  abstract interface :: ISum
2     function sum{integer | real(real64) :: T}(x) result(s)
3        type(T), intent(in) :: x(:)
4        type(T)             :: s
5     end function sum
6  end interface ISum
```

The inline use of a type set, within the generic type parameter list in curly braces, is an alternative, short-hand, notation for defining a type set trait for use as a generics constraint for type `T`. In this particular case, for admitting only the 32 bit `integer`, or 64 bit `real` type for `T`, as it was discussed above.

5.1.3 Traits with associated (deferred) types

In addition to supporting implicitly expressed generics constraints trough type sets, the language must allow such constraints to be formulated also through *explicit* specification of the signatures of ordinary type-bound procedures, initializers, or operators in traits. The proper support of operator signatures, but also of other generic functionality, requires the use of so-called associated types within traits. Associated types (which are also available in Rust and Swift) are essentially aliases. They are employed within traits, in lieu of types whose actual value is not known at the time a trait is formulated, but will be known by a compiler once the programmer has actually implemented that trait for some (concrete derived or intrinsic) data type.

A typical use case for associated types is when a trait needs to refer to the implementing type itself. This case is, in fact, so frequent that the language should provide a *predefined* associated type for it, that we propose to simply call `itself`. The following version of an `INumeric` trait makes use

of this predefined type, in order to explictly declare the signatures of an `operator(+)` for addition, and an `operator(/)` for division, that together with an explicitly declared initializer, `init`, are intended for use as a generics constraints in the `ISum` trait of Sect. 5.1.1:

```
abstract interface :: INumeric
   function init(n)
      integer, intent(in) :: n
   end function init
   function operator(+)(lhs,rhs) result(res)
      type(itself), intent(in) :: lhs, rhs
      type(itself)             :: res
   end function operator(+)
   function operator(/)(lhs,rhs) result(res)
      type(itself), intent(in) :: lhs, rhs
      type(itself)             :: res
   end function operator(/)
end interface INumeric
```

For a more comprehensive usage example, see Sect. 6.2.3, where this trait is employed in the framework of our standard test problem, to achieve strict conformance to the Open/Closed Principle of OO programming.

In general, associated types will have to be declared by the programmer himself, using `deferred` type-definition statements, as it is shown in the following example of the trait `IAppendable`:

```
abstract interface :: IAppendable
   typedef, deferred :: Element
   subroutine append(item)
      type(Element), intent(in) :: item
   end subroutine append
end interface IAppendable
```

This trait requires any implementing type to provide a subroutine for appending certain items (to the type). The type of these items is unknown upon trait formulation, and is therefore declared using the `deferred` (i.e. placeholder) type `Element`, that is left for inference by the compiler. A worked out example that illustrates the actual use of this trait is provided in Sect. 6.4.

5.1.4 Restrictions

The previous sections have illustrated the definitions of three kinds of traits, namely

- "existential" traits, i.e. traits that are *not* made up of type sets, or contain any associated types (examples of this are all the traits of Chapter 4, and the `ISum` trait of Sect. 5.1.1),
- traits of type sets (or "TOTs"), like the `INumeric` traits of Sect. 5.1.2, and,
- traits with associated types (or "TATs"), like the `INumeric` and `IAppendable` traits of Sect. 5.1.3.

Existential traits have the least amount of restrictions connected to them. They can be used together with the `class` specifier in order to declare trait objects (cf. Sect. 4.4), i.e. run-time polymorphic variables, or instances of what in type theory are known as "existential types" (hence their name). But they can *also* serve as generics constraints (as it will be shown in Sect. 5.4.3). These traits allow the programmer to switch from dynamic, to static dispatch of methods (see the later Sect. 6.2.2), and vice versa. In contrast, both TOTs and TATs are (presently) *exclusively* intended for use as generics constraints. Hence, a compiler must ensure that these latter two trait kinds are *not* used for any other purpose.

Notice, in particular, that TOTs cannot be implemented (in the sense of Sect. 4.2.3) by derived (i.e. user-defined) types[2], and that both TOTs and TATs are not intended to be used in variable declarations that involve the `class` specifier, because they require semantics that are based on the static binding of methods. Since the semantics of the `class` declaration specifier are based on dynamic binding, a compiler will need to ensure that the `class` declaration specifier is not used in conjunction with such traits. This includes the empty trait, `IAnyType`.

In a future language revision, both traits of type sets (TOTs), and traits with associated types (TATs), could be admitted for use with the `type` declaration specifier, in order to enable compile-time polymorphism through union (also called sum) types [15, 17], as it is further elaborated on in Sect. 7.3.

5.2 Predefined generics constraints

The facilities that were described in Sect. 5.1 are flexible enough to enable the user to easily construct generics constraints himself, the way he needs them. Nevertheless, the language should ideally also supply a collection of predefined, frequently used, generics constraints, in the form of traits that are contained in a language-intrinsic module (tentatively called `generics_constraints` here). A list of such predefined traits could include

- the empty trait `IAnyType` that admits any type (see above),
- some predefined numeric traits allowing for different numeric operations, but also
- some predefined traits to allow for the use of relational operators with different types.

Such traits could then be imported from user code through a `use` statement, like in the following example, that assumes the existence of a language defined trait `INumeric`:

```
1  module user_code
2
3     use, intrinsic :: generics_constraints, only: INumeric
4
5     abstract interface :: ISum
6        function sum{INumeric :: T}(x) result(s)
7           type(T), intent(in) :: x(:)
8           type(T)             :: s
9        end function sum
10    end interface ISum
11
12 end module user_code
```

5.3 Casts to generic types

A language like Fortran, that is intended for numeric use, where conversions between different types are required rather frequently, must allow one to carry out type conversions also in generic code, in a manner that is similarly user-friendly as in the Swift and Go languages, without having to rely on external library functionality. Moreover, this should be possible *regardless* of whether the

[2]One of the problems here is that any new intrinsic function for an intrinsic type, that would need to be added to the language, would change the set of methods of all the type sets of which this type is a member. This would break any user-defined types that would implement traits which are based on these type sets.

type to be cast to is an intrinsic or user-defined type. This requires language built-in mechanisms (that were already discussed) for requiring the availability of initializer implementations for types, in a manner that is either implicit (see the Sect. 5.1.2 on type sets), or explicit (see the language features that were described in Sects. 4.1, and 4.2).

As an example, consider that generic routines will often have to initialize the result of reduction operations, as it is the case in the test problem implementation of Sect. 2.2. There, a reduction variable for summation, `s`, needs to be initialized to the zero constant of type `T`. It should be possible, in Fortran, to effect this initialization by simply writing an expression like the following:

```
s = T(0)
```

Which (in this example) implies a cast of the zero constant of the `integer` type into the corresponding zero constant of type `T`, by a call to an appropriate initializer of this latter generic type.

Notice that if the generic type parameter, `T`, is instantiated at compile time with a language-intrinsic type, then the "initializer" to be called by the compiler would actually be Fortran's built-in function for casts to this type. That is, if `T` is instantiated with, say, the `real(real64)` type, then the compiler would transform `s = T(0)` into the following, final, expression:

```
s = real(0,kind=real64)
```

If `T` is instantiated, instead, with a derived type, then an appropriate initializer of this type would need to be invoked. This could either be the type's default "structure constructor", or a user-provided initializer, either of which would need to take in an `integer`, and return an instance of type `T`. In the absence of an appropriate (language or user-provided) initializer implementation, the compiler should simply emit an error message, and abort compilation.

5.4 Generic procedures, methods, and derived types

As it was already mentioned, Fortran's basic generics design should allow both ordinary and type-bound procedures (i.e. methods), and derived types to be given their own generic type parameters.

5.4.1 Generic procedures

Using the syntax that is proposed in this document, an implementation of a Fortran function for array summation that is generic over the type of its input array argument would look as follows[3]:

```
function sum{INumeric :: T}(x) result(s)
   type(T), intent(in) :: x(:)
   type(T)             :: s
   integer :: i
   s = T(0)
   do i = 1, size(x)
      s = s + x(i)
   end do
end function sum
```

The following examples will assume that the `INumeric` generics constraint, that is used here for type `T`, is provided by a trait that prescribes either implicitly (see Sect. 5.1.2), or explicitly (see Sect. 5.1.3), an appropriate initializer and addition operator, that are implemented by the `integer` and `real(real64)` types.

To actually use the `sum` generic function, one simply needs to pass to it (via its regular arguments list) an argument of one of these two admitted "numeric" types, as in the following calls:

[3] Generic subroutines can be coded completely analogously.

```
integer      :: integer_total
real(real64) :: float_total

integer_total = sum([1,2,3,4,5])
float_total   = sum([1.d0,2.d0,3.d0,4.d0,5.d0])
```

Here, the compiler will *automatically* instantiate appropriate versions of the `sum` generic function by using type inference on the regular arguments lists. That is, the programmer can straightforwardly use his generic routine on any of the admitted types.

Although the present design *doesn't* typically require it, the programmer can accomplish generic instantiation also manually, by the additional provision of generic type arguments in curly braces, as in the following two calls:

```
integer_total = sum{integer}([1,2,3,4,5])
float_total   = sum{real(real64)}([1.d0,2.d0,3.d0,4.d0,5.d0])
```

Such manual instantiation of generic procedures is only needed in the relatively rare cases where a regular arguments list is either unavailable or outright inappropriate, e.g. for use in procedure pointer assignments, or in `associate` statements, as in the next example:

```
real(real64) :: dtot(2)
real(real32) :: stot(2)
procedure(sum_real64), pointer :: dsum

abstract interface
   function sum_real64(x) result(s)
      real(real64), intent(in) :: x(:)
      real(real64)             :: s
   end function sum_real64
end interface

dsum => sum{real(real64)}

dtot(1) = dsum([1.d0,2.d0,3.d0,4.d0,5.d0])
dtot(2) = dsum([2.d0,4.d0,6.d0,8.d0])

associate( ssum => sum{real(real32)} )
   stot(1) = ssum([1.,2.,3.,4.,5.])
   stot(2) = ssum([2.,4.,6.,8.])
end associate
```

5.4.2 Generic methods

The same summation algorithm as that of Sect. 5.4.1, when implemented as a generic method, `sum`, that is bound to a derived type named `SimpleSum`, which implements the trait `ISum` as it was given in Sect. 5.1.1, would instead look as follows:

```
module simple_library
   ...

   type, public :: SimpleSum
   end type SimpleSum

   implements ISum :: SimpleSum
      procedure, nopass :: sum
   end implements SimpleSum

```

```
contains

   function sum{INumeric :: T}(x) result(s)
      type(T), intent(in) :: x(:)
      type(T)             :: s
      integer :: i
      s = T(0)
      do i = 1, size(x)
         s = s + x(i)
      end do
   end function sum

end module simple_library
```

We have used here the `implements` *statement* for implementing a trait by a derived type, and will continue to consistently do so for the remainder of this chapter. The alternative way of employing the `implements` derived type *attribute* for the same purpose, is demonstrated on the same examples in Chapter 6.

Generic methods are used completely analogously to generic procedures. The automatic instantiation use case of Sect. 5.4.1 would, for instance, be written:

```
  type(SimpleSum) :: simple

  integer_total = simple%sum([1,2,3,4,5])
  float_total   = simple%sum([1.d0,2.d0,3.d0,4.d0,5.d0])
```

Whereas the corresponding manual instantiation case would take the form:

```
  type(SimpleSum) :: simple

  integer_total = simple%sum{integer}([1,2,3,4,5])
  float_total   = simple%sum{real(real64)}([1.d0,2.d0,3.d0,4.d0,5.d0])
```

5.4.3 Generic derived types

In addition to procedures and methods, generic type parameter lists must also be allowed for derived type definitions, as in the following example, in which the `ISum` trait of Sect. 5.1.1 is implemented by another derived-type, named `PairwiseSum`:

```
module pairwise_library
   ...
   type, public :: PairwiseSum{ISum :: U}
      private
      type(U) :: other
   end type PairwiseSum

   implements ISum :: PairwiseSum{ISum :: U}
      procedure, pass :: sum
   end implements PairwiseSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      class(PairwiseSum{U}), intent(in) :: self
      type(T),               intent(in) :: x(:)
      type(T)                           :: s
      integer :: m
      if (size(x) <= 2) then
```

```
20          s = self%other%sum(x)
21       else
22          m = size(x) / 2
23          s = self%sum(x(:m)) + self%sum(x(m+1:))
24       end if
25    end function sum
26
27 end module pairwise_library
```

Notice, how type `PairwiseSum` depends on a generic type parameter, `U`, that is used within `PairwiseSum` in order to declare a field variable of `type(U)`, which is named `other`. As it is indicated by the type constraint on `U`, object `other` conforms to the `ISum` trait itself, and therefore contains its own implementation of the `sum` procedure.

The above example furthermore demonstrates, how a derived type's generic parameters are brought into the scope of its type-bound procedures via the latter's passed-object dummy arguments. In this example, type `PairwiseSum`'s method, `sum`, has a passed-object dummy argument, `self`, that is declared being of `class(PairwiseSum{U})`. Hence, method `sum` can now access `PairwiseSum`'s generic parameter `U`. This allows the method to make use of two independently defined generic type parameters, `T` and `U`, which grants it increased flexibility. This also means that there is *no* implicit mechanism of bringing generic parameters of a derived type into the scope of its methods. If a type-bound procedure needs to access the generic parameters of its derived type, it must be provided with a passed-object dummy argument.

Notice, that the declaration `class(PairwiseSum{U})` does not imply any ambiguities or contradictions with respect to compile-time vs. run-time polymorphism, because substitution semantics apply. At compile time, the compiler will substitute a set of different type arguments for the generic parameter `U`. Hence, the notation `PairwiseSum{U}` really refers to a set of multiple, related, but *different* `PairwiseSum` types, whose *only* commonality is that they all implement the `ISum` trait (and furthermore contain different field components that do the same). Of course, passed-object dummy arguments of any of the different `PairwiseSum` types of this set can then be run-time polymorphic. In exactly the same manner that passed-object dummy arguments of other derived types that implement the same trait can be run-time polymorphic.

The following code snippet finally shows how an object of the `PaiwiseSum` type could be declared and instantiated manually (by substituting its generic type parameter `U` by the `SimpleSum` type of Sect. 5.4.2), and how its generic `sum` method would be employed using automatic type inference:

```
1  type(PairwiseSum{SimpleSum}) :: pairwise
2
3  integer_total = pairwise%sum([1,2,3,4,5])
4  float_total   = pairwise%sum([1.d0,2.d0,3.d0,4.d0,5.d0])
```

5.5 Updated structure constructors

If a derived type is parameterized over a generic type, as in the `PairwiseSum` type example of Sect. 5.4.3, then the structure constructor of this derived type must also be assumed to be parameterized over the same generic type. Hence, calls of structure constructors, with specific argument types substituting the generic type parameters of their derived types, must be valid.

For an illustration of this, and all the other possibilities that are available to the programmer when calling the structure constructors of generic derived types, consider the following variation of Sect. 5.4.3's last code example:

```
type(U) :: pairwise

pairwise = PairwiseSum{SimpleSum}()

integer_total = pairwise%sum([1,2,3,4,5])
float_total   = pairwise%sum([1.d0,2.d0,3.d0,4.d0,5.d0])
```

Here, we have assumed that the type parameter `U` (in terms of which the object `pairwise` is generically declared) has been constrained by the `ISum` trait in the enclosing scope of this code, and that moreover both the `PairwiseSum` and `SimpleSum` types implement this trait.

Notice the call to `PairwiseSum`'s structure constructor in the above code, that the compiler will take advantage of (via type inference) to *automatically* instantiate object `pairwise` with the `PairwiseSum` type. Notice, also, how `PairwiseSum`'s structure constructor is generically parameterized, as there is a type argument, `SimpleSum`, that is passed to this constructor within curly braces. This type argument will be used by the compiler to *manually* instantiate the generic field object `other`, that is embedded within the `PairwiseSum` type (cf. Sect. 5.4.3).

One of the great advantages of constructors, though, is that through their use one can rely *exclusively* on type inference for the instantiation of generic types. In the above example, one would merely have to drop the curly braces in the aforegiven constructor call, and to rewrite this call in the following standard OO way of chaining the structure constructors of the two involved types:

```
pairwise = PairwiseSum(SimpleSum())
```

The compiler would then automatically infer the generic type argument, that `PairwiseSum`'s structure constructor needs, from the type of its regular argument. Thus, making manual instantiation of any generic types unnecessary.

However, it is also possible, in the last call, to provide the appropriate generic type argument manually in curly braces, for confirmation purposes, as follows:

```
pairwise = PaiwiseSum{SimpleSum}(SimpleSum())
```

Hence, all three of these call variants are valid, and give the same result.

We will close with proposing a further, small, but extremely important addition to structure constructors: namely to introduce the notion that a structure constructor is implicitly defined *within the same scope that hosts the definition of its derived type*. This would make it possible for the structure constructor to access even `private` components of its derived type, through host association. Meaning that if the derived type is hosted within the specification part of a module, such components would be accessible, and thus could be initialized, even by calls to the structure constructor that are being performed from outside this module's scope. Which would be in complete analogy to how user-defined constructors work in Fortran.

In this way, it would become possible to initialize `private`, `allocatable` derived type components by structure constructors, which is absolutely crucial for concise OO programming. Since such an extension would merely add to the capabilities of the language, it would be fully backwards compatible. The elegance of the aforegiven Go and Swift code versions, but also of the Fortran code examples that are presented in the next chapter is largely due to the use of such constructors. Lacking these, user-defined constructors would have to be employed, leading to overly complex implementations, as e.g. in the Rust example code given in Listing 3.2.

Chapter 6

Fortran examples

In order to comprehensively illustrate how the new features, that were discussed in the last two chapters, would be used in practice, we will give in the present chapter several worked out examples. The first two subsections contain both functional and OO Fortran code versions of the standard test problem that is used throughout this document. While the last two subsections deal with how to use operator overloading, and associated types with generics.

6.1 Functional versions of the standard test problem

Fortran presently lacks support for advanced functional programming capabilities, like closures and variables of higher-order functions, that are, e.g., available in Go and other modern languages. In contrast to the Go version of the standard test problem that is given in Sect. 2.2, the functional Fortran code versions that are presented in this section therefore make no attempt to eliminate rigid dependencies on user-defined function implementations, and content themselves with demonstrating how the new generics features can be used to eliminate rigid dependencies on language-intrinsic types.

6.1.1 Automatic instantiation of generic procedures

Listing 6.1 shows a straightforward generic functional implementation of the standard test problem, that uses automatic type inference by the compiler. The following additional remarks should help to better understand this code:

- To express type genericity for the arguments and return values of our different generic functions, we make use of a type constraint expressed by the trait `INumeric`, that is implemented as the type set `integer | real(real64)`.
- Trait `INumeric` is defined by the user himself. Thus, there is no need for an external dependency.
- Any required conversions to generic types are done using explicit casts, as in Go (or Swift).
- *All* the required instantiations of generic procedures are performed automatically by the compiler, based on type inference of the regular arguments that are passed to these procedures.

Listing 6.1: Fortran version of the array averaging problem with automatic generics instantiation.

```
module averager_library

   use, intrinsic :: iso_fortran_env, only: real64

   implicit none
   private

   public :: simple_average, pairwise_average

   abstract interface :: INumeric
      integer | real(real64)
   end interface INumeric

contains

   function simple_sum{INumeric :: T}(x) result(s)
      type(T), intent(in) :: x(:)
      type(T)             :: s
      integer :: i
      s = T(0)
      do i = 1, size(x)
         s = s + x(i)
      end do
   end function simple_sum

   function pairwise_sum{INumeric :: T}(x) result(s)
      type(T), intent(in) :: x(:)
      type(T)             :: s
      integer :: m
      if (size(x) <= 2) then
         s = simple_sum(x)
      else
         m = size(x) / 2
         s = pairwise_sum(x(:m)) + pairwise_sum(x(m+1:))
      end if
   end function pairwise_sum

   function simple_average{INumeric :: T}(x) result(a)
      type(T), intent(in) :: x(:)
      type(T)             :: a
      a = simple_sum(x) / T(size(x))
   end function simple_average

   function pairwise_average{INumeric :: T}(x) result(a)
      type(T), intent(in) :: x(:)
      type(T)             :: a
      a = pairwise_sum(x) / T(size(x))
   end function pairwise_average

end module averager_library

program main

   ! dependencies on intrinsic constants
   use, intrinsic :: iso_fortran_env, only: real64

   ! dependencies on implementations
```

```
58    use averager_library, only: simple_average, pairwise_average
59
60    implicit none
61
62    ! declarations
63    integer,      parameter :: xi(5) = [1, 2, 3, 4, 5]
64    real(real64), parameter :: xf(5) = [1.d0, 2.d0, 3.d0, 4.d0, 5.d0]
65
66    integer :: key
67
68    write(*,'(a)') 'Simple   sum average: 1'
69    write(*,'(a)') 'Pairwise sum average: 2'
70    write(*,'(a)',advance='no') 'Choose an averaging method: '
71    read(*,*) key
72
73    select case (key)
74    case (1)
75       print '(i8)',   simple_average(xi)
76       print '(f8.5)', simple_average(xf)
77    case (2)
78       print '(i8)',   pairwise_average(xi)
79       print '(f8.5)', pairwise_average(xf)
80    case default
81       stop 'Case not implemented!'
82    end select
83
84 end program main
```

The example demonstrates that using the new generics features together with a functional (or procedural) programming style is easy, that the syntax is concise, and that type inference by the compiler should be straightforward and therefore reliable. Hence, we believe that the generics features, that are described here, will place no burden on the programmer.

6.1.2 Manual instantiation of generic procedures

It is actually possible to make the Fortran code version that was given in Listing 6.1, resemble the Go code version of Listing 2.2 a bit closer, by having two procedure pointers stand in, within the `select case` statement of the main program, for the closures that were used in the Go code. As this is a good example for demonstrating how generics can be instantiated manually by the programmer, we give in Listing 6.2 an alternative form of the main program of Listing 6.1 that makes use of both procedure pointers and such manual instantiation (as it was discussed in Sect. 5.4.1).

Listing 6.2: Main program using procedure pointers and manual generics instantiation.

```
1  program main
2
3     ! dependencies on intrinsic constants
4     use, intrinsic :: iso_fortran_env, only: real64
5
6     ! dependencies on implementations
7     use averager_library, only: simple_average, pairwise_average
8
9     implicit none
10
11    ! declarations
12    integer,      parameter :: xi(5) = [1, 2, 3, 4, 5]
13    real(real64), parameter :: xf(5) = [1.d0, 2.d0, 3.d0, 4.d0, 5.d0]
```

```
14
15    integer :: key
16    procedure(average_integer), pointer :: avi
17    procedure(average_real64),  pointer :: avf
18
19    abstract interface
20       function average_integer(x) result(a)
21          integer, intent(in) :: x(:)
22          integer             :: a
23       end function average_integer
24       function average_real64(x) result(a)
25          real(real64), intent(in) :: x(:)
26          real(real64)             :: a
27       end function average_real64
28    end interface
29
30    write(*,'(a)') 'Simple   sum average: 1'
31    write(*,'(a)') 'Pairwise sum average: 2'
32    write(*,'(a)',advance='no') 'Choose an averaging method: '
33    read(*,*) key
34
35    select case (key)
36    case (1)
37       avi => simple_average{integer}
38       avf => simple_average{real(real64)}
39    case (2)
40       avi => pairwise_average{integer}
41       avf => pairwise_average{real(real64)}
42    case default
43       stop 'Case not implemented!'
44    end select
45
46    print '(i8)',   avi(xi)
47    print '(f8.5)', avf(xf)
48
49 end program main
```

6.2 Object-oriented versions of the standard test problem

The present section will demonstrate that being able to use the new generics features seamlessly and easily even within a modern-day OO programming setting is one of the great strengths of the present design.

6.2.1 Dynamic method dispatch

Listing 6.3 shows an encapsulated (i.e. OO) Fortran code version of the standard test problem, that makes use of dynamic method dispatch, and is therefore comparable to the code versions that were presented in Chapter 3 for all the other languages.

- As in all these other versions, three interfaces/traits are used to manage all the source code dependencies in the problem: `INumeric`, `ISum`, and `IAverager`. Trait `INumeric` is defined here by the user himself as a type set, similar to the corresponding Go code (for a version that does not use type sets, see the later Sect. 6.2.3).

- In contrast to the Go and Rust versions (Listings 3.1 and 3.2), none of the aforementioned interfaces/traits is parameterized itself, since we followed Swift's basic model of generics.
- Interface inheritance is expressed through the presence of the `implements` attribute in derived-type definitions (equivalent to the Swift version, Listing 3.4). Alternatively, `implements` statements could be used (cf. Sects. 4.2, 5.4.2, and 5.4.3).
- All our derived types are `sealed`, which makes them inextensible to implementation inheritance, and thus enables us to declare the passed-object dummy arguments of their type-bound procedures with the `type` specifier.
- The example code makes use, in the main program, of the new structure constructors, with their enhancements that were discussed in Sect. 5.5, for the classes `Averager`, `SimpleSum`, and `PairwiseSum`.
- This Fortran version makes use of modules and `use` statements with `only` clauses, in order to make explicit the source code dependencies of the different defined classes/ADTs.

Listing 6.3: Fortran OO version of the array averaging example with dynamic method dispatch.

```
module interfaces

   use, intrinsic :: iso_fortran_env, only: real64

   implicit none
   private

   public :: INumeric, ISum, IAverager

   abstract interface :: INumeric
      integer | real(real64)
   end interface INumeric

   abstract interface :: ISum
      function sum{INumeric :: T}(x) result(s)
         type(T), intent(in) :: x(:)
         type(T)             :: s
      end function sum
   end interface ISum

   abstract interface :: IAverager
      function average{INumeric :: T}(x) result(a)
         type(T), intent(in) :: x(:)
         type(T)             :: a
      end function average
   end interface IAverager

end module interfaces

module simple_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

   public :: SimpleSum
```

```
38
39    type, sealed, implements(ISum) :: SimpleSum
40    contains
41       procedure, nopass :: sum
42    end type SimpleSum
43
44 contains
45
46    function sum{INumeric :: T}(x) result(s)
47       type(T), intent(in) :: x(:)
48       type(T)             :: s
49       integer :: i
50       s = T(0)
51       do i = 1, size(x)
52          s = s + x(i)
53       end do
54    end function sum
55
56 end module simple_library
57
58 module pairwise_library
59
60    use interfaces, only: ISum, INumeric
61
62    implicit none
63    private
64
65    public :: PairwiseSum
66
67    type, sealed, implements(ISum) :: PairwiseSum
68       private
69       class(ISum), allocatable :: other
70    contains
71       procedure, pass :: sum
72    end type PairwiseSum
73
74 contains
75
76    function sum{INumeric :: T}(self,x) result(s)
77       type(PairwiseSum), intent(in) :: self
78       type(T),           intent(in) :: x(:)
79       type(T)                       :: s
80       integer :: m
81       if (size(x) <= 2) then
82          s = self%other%sum(x)
83       else
84          m = size(x) / 2
85          s = self%sum(x(:m)) + self%sum(x(m+1:))
86       end if
87    end function sum
88
89 end module pairwise_library
90
91 module averager_library
92
93    use interfaces, only: IAverager, ISum, INumeric
94
95    implicit none
96    private
```

```
97
98    public :: Averager
99
100   type, sealed, implements(IAverager) :: Averager
101      private
102      class(ISum), allocatable :: drv
103   contains
104      procedure, pass :: average
105   end type Averager
106
107contains
108
109   function average{INumeric :: T}(self,x) result(a)
110      type(Averager), intent(in) :: self
111      type(T),        intent(in) :: x(:)
112      type(T)                    :: a
113      a = self%drv%sum(x) / T(size(x))
114   end function average
115
116end module averager_library
117
118program main
119
120   ! dependencies on abstractions
121   use interfaces,       only: IAverager
122
123   ! dependencies on implementations
124   use simple_library,   only: SimpleSum
125   use pairwise_library, only: PairwiseSum
126   use averager_library, only: Averager
127
128   implicit none
129
130   ! declarations
131   integer :: key
132   class(IAverager), allocatable :: avs, avp, av
133
134   ! use of enhanced structure constructors
135   avs = Averager(drv = SimpleSum())
136   avp = Averager(drv = PairwiseSum(other = SimpleSum()))
137
138   write(*,'(a)') 'Simple   sum average: 1'
139   write(*,'(a)') 'Pairwise sum average: 2'
140   write(*,'(a)',advance='no') 'Choose an averaging method: '
141   read(*,*) key
142
143   select case (key)
144   case (1)
145      ! simple sum case
146      av = avs
147   case (2)
148      ! pairwise sum case
149      av = avp
150   case default
151      stop 'Case not implemented!'
152   end select
153
154   print '(i8)',   av%average([1, 2, 3, 4, 5])
155   print '(f8.5)', av%average([1.d0, 2.d0, 3.d0, 4.d0, 5.d0])
```

```
156
157 end program main
```

The most important point to note, in Listing 6.3, is how this code makes use of trait objects (cf. Sect. 4.4) and generics to avoid rigid dependencies on user-defined implementation and language-intrinsic types, respectively, and to thus realize a plugin architecture that allows for a maximum of code reuse. Notice, in particular, how the main program is the only part of the code that (necessarily) depends on implementations. The *entire* rest of the code is decoupled, i.e. it depends merely on traits (i.e. abstract interfaces, see the `use` statements in the above modules).

Notice also, that all the traits that are required for this purpose are defined by the user himself, without any need to depend on external libraries. The OO Fortran version that is presented here is therefore as clean with respect to dependency management, and as easy to use, read, and understand, as the Go and Swift implementations of Listings 3.1 and 3.4, respectively.

6.2.2 Static method dispatch

One of the greatest benefits of the present design is that, through the use of generics, polymorphic methods in OO programming can be made to use static dispatch[1]. This will enable inlining of polymorphic methods by Fortran compilers, to potentially improve code performance. Listing 6.4 gives minimally changed, alternative, implementations of the modules `pairwise_library` and `averager_library`, with which the original versions of these modules in Listing 6.3 would need to be replaced, in order to effect static dispatch of the different `sum` methods.

Notice, that the required changes are confined to a parameterization of the `PairwiseSum` and `Averager` derived types, by generic type parameters, `U`, that conform to the (existential) `ISum` trait. These type parameters are then used in order to declare the field objects `other` and `drv` of the `PairwiseSum` and `Averager` types, respectively, by means of the `type` specifier. Taken together, these changes signify compile-time polymorphism for the objects `other` and `drv` to the compiler, and hence static dispatch of their methods (whose signatures were declared in the `ISum` trait). Contrast this with the `class` specifier, that was used previously for the declaration of these former trait objects, in order to effect run-time polymorphism and dynamic dispatch of their methods. See also Table 6.1 for a summary of rules regarding method dispatch.

Everything else, especially the declaration of these object variables as `allocatables` (and their instantiation using chaining of constructor calls in the main program), was kept the same in order to demonstrate that static method dispatch does *not* mean that the actual object instances that contain the methods must be initialized and their memory allocated at compile-time. (Although in this particular example this is possible, by simply deleting the `allocatable` attribute from their declarations, given that all the eligible types for instantiating these objects do not contain any other `allocatable` data fields.) Notice, also, that none of the source code dependencies in the `use` statements have changed. That is, the code is *still* fully decoupled, despite making now use of static dispatch!

[1] This can be achieved in a compiler by implementing generic polymorphism through the (compile-time) technique of monomorphization, that relies on static method dispatch. In contrast to traditional (run-time) polymorphism, that relies on virtual method tables and dynamic method dispatch.

Listing 6.4: Minimal required changes to effect static method dispatch of the `sum` methods.

```
module pairwise_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

   public :: PairwiseSum

   type, sealed, implements(ISum) :: PairwiseSum{ISum :: U}
      private
      type(U), allocatable :: other
   contains
      procedure, pass :: sum
   end type PairwiseSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      type(PairwiseSum{U}), intent(in) :: self
      type(T),              intent(in) :: x(:)
      type(T)                          :: s
      integer :: m
      if (size(x) <= 2) then
         s = self%other%sum(x)
      else
         m = size(x) / 2
         s = self%sum(x(:m)) + self%sum(x(m+1:))
      end if
   end function sum

end module pairwise_library

module averager_library

   use interfaces, only: IAverager, ISum, INumeric

   implicit none
   private

   public :: Averager

   type, sealed, implements(IAverager) :: Averager{ISum :: U}
      private
      type(U), allocatable :: drv
   contains
      procedure, pass :: average
   end type Averager

contains

   function average{INumeric :: T}(self,x) result(a)
      type(Averager{U}), intent(in) :: self
      type(T),           intent(in) :: x(:)
      type(T)                       :: a
      a = self%drv%sum(x) / T(size(x))
   end function average
```

```
58
59 end module averager_library
```

object declaration	dynamic dispatch	static dispatch
`class(Interface)`	always	never
`class(DerivedType)`	if `DerivedType` is `extended`	if `DerivedType` is `unextended`[2]
`type(DerivedType)`	never	always
`type(Interface)`	—	—

Table 6.1: Correspondence between object declarations and method dispatch strategies that would be typically employed by an optimizing Fortran compiler according to the present document. A dash indicates that the case in question is presently undefined, but could be used for a future extension as discussed in Sect. 7.3.

The complete source code of this (statically dispatched) version can be found in the `Code` subdirectory that accompanies this document (see the file `Code/Fortran/static.f90`). Notice, from this complete listing (or from the original reproduction of the main program, given in Listing 6.3), that since only the `pairwise_library` and `averager_library` modules have changed, the `av` object of `IAverager` type in the main program still needs to make use of run-time polymorphism, because it is initialized in a `select case` statement by the main program. This object cannot be made to employ compile-time polymorphism, as it is initialized within a statement that performs a run-time decision.

As in corresponding (static method dispatch) Swift and Rust implementations of the standard test problem, that are not reproduced here (see the `Code` directory for this), the instantiation (in the main program) of the objects `other` and `drv` through constructor calls, has the benefit that the compiler can infer the correct type arguments, that are required to automatically instantiate all the involved generic derived types (cf. Sect 5.5). Hence (and similar to both the functional and dynamically dispatched OO code versions of Listings 6.1 and 6.3, respectively), not a single manual instantiation of generics is necessary, anywhere, also in the statically dispatched version.

As a final remark, we'd like to emphasize that, on readability grounds, a coding style as that given in Listing 6.3 is generally preferable over that shown in Listing 6.4. The use of numerous generic type parameters can quickly make code unreadable. We'd therefore recommend the default use of run-time polymorphism for managing dependencies on user-defined types, and the employment of generics for this latter task mainly in cases where profiling has shown that static dispatch would significantly speed up a program's execution (by allowing method inlining by the compiler). Of course, the use of generics to manage dependencies on language-intrinsic types remains unaffected by this recommendation.

6.2.3 Strict conformance to the Open/Closed Principle

In the examples that were presented so far in this chapter, we made predominant use of generics constraints that, in the form of the `INumeric` trait, were expressed as (Go-like) type sets of intrinsic types. As it was already discussed in Sect. 3.1.1, the conciseness of formulation, that type sets permit, comes at the price of having to give up strict conformance to the Open/Closed Principle (OCP) of object-oriented programming.

[2] Or the method is declared as `non_overridable`.

However, the present design offers the programmer the possibility to formulate generics constraints also in the alternative manner of always specifying both the signatures of procedures in abstract interfaces, and their implementations, explicitly. An example of this was already given in the form of the (existential) `ISum` trait, that was used as a generics constraint for the derived types `PairwiseSum` and `Averager` in Listing 6.4. The explicit approach has the advantage of allowing generic object-oriented code to strictly conform to the OCP. This can be accomplished independently of the employed (dynamic or static) method dispatch strategy. Even though we will confine ourselves here to illustrating this using only Listing 6.3, i.e. the dynamically dispatched code version of the standard test problem, as a baseline.

The following Listing 6.5 gives the changes that are required to make the code of Listing 6.3 strictly conform to the OCP. The `INumeric` abstract interface is now formulated as a trait with associated types, through explicit specification of the signatures of the type-bound initializer (i.e. cast function), and the addition and division operators, that the different summation or averaging methods of Listing 6.3 require.

Notice, that Fortran's `integer` and `real(real64)` types already come with implementations of all this functionality that is predefined for them by the language. Hence, it is sufficient to acknowledge their conformance to the `INumeric` trait through empty `implements` statements (that are provided here in the separate new module `intrinsics`). This is fully equivalent to the Swift implementation that was presented in Listing 3.4. It is also equivalent to the Rust example of Sect. 3.2.1, in that now no pre-existing code needs to be modified, to make this Fortran version work with additional, new, types. Simply adding a new module that implements the `INumeric` trait for any such new type is now sufficient.

Listing 6.5: Required changes to enforce strict conformance of Listing 6.3 to the OCP.

```
module interfaces

   implicit none
   private

   public :: INumeric, ISum, IAverager

   abstract interface :: INumeric
      function init(n)
         integer, intent(in) :: n
      end function init
      function operator(+)(lhs,rhs) result(res)
         type(itself), intent(in) :: lhs, rhs
         type(itself)             :: res
      end function operator(+)
      function operator(/)(lhs,rhs) result(res)
         type(itself), intent(in) :: lhs, rhs
         type(itself)             :: res
      end function operator(/)
   end interface INumeric

   abstract interface :: ISum
      function sum{INumeric :: T}(x) result(s)
         type(T), intent(in) :: x(:)
         type(T)             :: s
      end function sum
   end interface ISum

   abstract interface :: IAverager
      function average{INumeric :: T}(x) result(a)
```

```
31          type(T), intent(in) :: x(:)
32          type(T)             :: a
33       end function average
34    end interface IAverager
35
36 end module interfaces
37
38 module intrinsics
39
40    use, intrinsic :: iso_fortran_env, only: real64
41    use interfaces, only: INumeric
42
43    implements INumeric :: integer
44    end implements integer
45
46    implements INumeric :: real(real64)
47    end implements real(real64)
48
49 end module intrinsics
```

When comparing Listing 6.5 to the more compact (type set based) code of Listing 6.3, the need for explicit provision of empty `implements` statements, for all the types that conform to trait `INumeric`, might appear long-winded. Especially in cases where one would have to deal with a relatively large number of such types. Such `implements` statements could be completely avoided, though, by adding also structural (i.e. implicit) subtyping to the design, as it is supported in Go. See Sect. 7.2 for a further discussion.

6.3 Use of overloaded operators with generic procedures

The following Listing 6.6 provides a worked out example of how, in the present design, traits with overloaded operator signatures can be used with generic routines. This is demonstrated here in terms of two types, namely a derived type, `MyType`, and the `real` intrinsic type. Both of these conform to the traits `IReducible` (with its overloaded multiplication operator signatures), and `IPrintable`, as they are defined in the module `basic_interfaces`.

Since `MyType` and `real` conform to the same traits, a set of common operations can be performed on them. This includes (`elemental`) multiplication of a type with itself, and with `integer` instances, along with casting to, as well as customized printing of, this type. The generic routines `products` and `prod` take advantage of this conformance, to carry out a series of array multiplication and reduction operations, and to print out their results in an appropriate format, regardless of whether their argument arrays are of type `real` or `MyType`.

Listing 6.6: Use of an overloaded multiplication operator with different generic procedures.

```
1  module basic_interfaces
2
3     implicit none
4     private
5
6     public :: IReducible, IPrintable
7
8     abstract interface :: IReducible
9        function init(n)
10          integer, intent(in) :: n
11       end function init
12       elemental function operator(*)(lhs,rhs) result(res)
```

```
         type(itself), intent(in) :: lhs, rhs
         type(itself)             :: res
      end function operator(*)
      elemental function operator(*)(lhs,rhs) result(res)
         type(itself), intent(in) :: lhs
         integer,      intent(in) :: rhs
         type(itself)             :: res
      end function operator(*)
   end interface IReducible

   abstract interface :: IPrintable
      subroutine output()
      end subroutine output
   end interface IPrintable

end module basic_interfaces

module my_type

   use basic_interfaces, only: IReducible, IPrintable

   implicit none
   private

   type, public, sealed :: MyType
      private
      integer :: n
   end type MyType

   implements (IReducible,IPrintable) :: MyType
      initial :: init
      procedure, nopass :: operator(*) => multiply, multiply_by_int
      procedure, pass   :: output
   end implements MyType

contains

   function init(n) result(res)
      integer, intent(in) :: n
      type(MyType)        :: res
      res%n = n
   end function init

   elemental function multiply(lhs,rhs) result(res)
      type(MyType), intent(in) :: lhs, rhs
      type(MyType)             :: res
      res%n = lhs%n * rhs%n
   end function multiply

   elemental function multiply_by_int(lhs,rhs) result(res)
      type(MyType), intent(in) :: lhs
      integer,      intent(in) :: rhs
      type(MyType)             :: res
      res%n = lhs%n * rhs
   end function multiply_by_int

   subroutine output(self)
      type(MyType), intent(in) :: self
      write(*,'(a,i8)') "I␣am:␣", self%n
```

```
72    end subroutine output
73
74 end module my_type
75
76 module real_type
77
78    use basic_interfaces, only: IReducible, IPrintable
79
80    implicit none
81    private
82
83    implements (IReducible,IPrintable) :: real
84       procedure, pass :: output
85    end implements real
86
87 contains
88
89    subroutine output(self)
90       real, intent(in) :: self
91       write(*,'(a,f8.5)') "I am: ", self
92    end subroutine output
93
94 end module real_type
95
96 program main
97
98    use basic_interfaces, only: IReducible, IPrintable
99    use my_type,          only: MyType
100
101   implicit none
102
103   integer      :: i
104   type(MyType) :: at(4)     ! Array of MyType
105   real         :: ar(4)     ! Array of real type
106
107   ! initializations
108   at = [(MyType(i),i=1,4)]  ! Use of user-defined initializer/constructor
109   ar = [(real(i),i=1,4)]    ! Use of built-in initializer (i.e. cast)
110
111   call products(at,ar)
112
113 contains
114
115   subroutine products{IReducible,IPrintable :: T,R}(at,ar)
116      type(T), intent(in) :: at(4)
117      type(R), intent(in) :: ar(4)
118      type(T) :: st                      ! Scalar of type(T)
119      type(R) :: sr                      ! Scalar of type(R)
120      st = prod( (at * at) * [4,3,2,1] ) ! Reduce arrays to scalars after
121      sr = prod( (ar * ar) * [4,3,2,1] ) ! elementwise multiplication operations
122      call st%output()                   ! Print the results
123      call sr%output()                   ! (13824 in both cases)
124   end subroutine products
125
126   function prod{IReducible :: T}(arr) result(res)
127      type(T), intent(in) :: arr(:)
128      type(T)             :: res
129      integer :: i
130      res = T(1)                         ! Use initializer for cast
```

```
      do i = 1, size(arr)
         res = res * arr(i)              ! Use operator(*) for reduction
      end do
   end function prod

end program main
```

Of note, here, is that (in contrast to `MyType`) the `real` type already came with suitable, language-defined, implementations of both an intrinsic initializer for casts to this type, and an elemental multiplication operator – as they are required by the `IReducible` trait. We therefore had to merely acknowledge `real`'s conformance to this trait. Whereas we had to provide an own implementation of procedure `output`, to make `real` conform to the `IPrintable` trait.

Furthermore, and as it was discussed in Sect. 4.2.4, the present Fortran language *rightly prohibits* the programmer (for reasons of predictability) from overloading the intrinsic numeric operators for language-intrinsic types! Any attempt by the programmer to provide some alternative implementation(s) of `operator(*)` for the `real` type should thus be treated as an error. It should be obvious from the above listing, that the present generics design makes it easy for a compiler to detect, and ultimately reject, such an error.

6.4 Use of associated types with generic implementing types

Lastly, we will demonstrate, now, how associated types in traits (as they were introduced in Sect. 5.1.3) can be used in conjunction with implementing types that are parameterized over some generic parameter. The following example makes use of the `IAppendable` trait, that was defined in Sect. 5.1.3, in order to require an implementing type to provide functionality for appending items to itself. This trait contains an associated type, `Element`, acting as a placeholder for the type of the appendable items, that is to be inferred from an actual implementation of that trait.

```
module vector_library

   implicit none
   private

   abstract interface :: IAnyType
   end interface IAnyType

   abstract interface :: IAppendable
      typedef, deferred :: Element
      subroutine append(item)
         type(Element), intent(in) :: item
      end subroutine append
   end interface IAppendable

   type, public, implements(IAppendable) :: Vector{IAnyType :: U}
      private
      type(U), allocatable :: elements(:)
   contains
      procedure, pass :: append
   end type Vector

contains

   subroutine append(self,item)
      class(Vector{U}), intent(inout) :: self
```

```
27        type(U),           intent(in)    :: item
28        self%elements = [self%elements,item]
29     end subroutine append
30
31  end module vector_library
```

An example of such an implementation of trait `IAppendable` is provided here by the derived type `Vector`, that stores an `elements` array of generic type `U`, where the type parameter `U` conforms to the `IAnyType` constraint. Notice, that in order to maintain consistency between the type of the `elements` array, and any new `item` that we might wish to append to this array, we must force the `item` argument of method `append` to be of type `U` as well, as it is shown in this method's actual implementation. In order to accomplish this enforcement without contradicting trait `IAppendable`'s definition (that doesn't know anything about type `U`), we made use of the placeholder (i.e. associated) type `Element` in this trait. Given method `append`'s implementation, the compiler will then infer `Element` to be of the actual type `U`.

Associated types thus allow one to make traits/abstract interfaces generic, without having to parameterize either them or their method signatures by actual generic type parameters, and to thereby entail the instantiation needs of the latter, and the potential duplication of client code that this can result in (cf. Sect. 3.1.1). Associated types are therefore an important element of the present generics design, and its philosophy not to burden the user with superfluous manual instantiations of generics.

Chapter 7

Outlook and final conclusions

This final chapter lists a number of foreseeable extensions, that could be added to the present design once a prototype of it has already been successfully implemented.

7.1 Rank genericity

Fortran's special role, as a language that caters to numerical programming, demands that any generics design for the language must allow for the possibility to also handle genericity of array rank. The present design offers a lot of room in this respect, but since this is a largely orthogonal issue, and since we consider a discussion of such functionality to be non-essential for the purpose of a very first prototype implementation of the generics features that were described here, we defer it to a separate document.

7.2 Structural subtyping

A pretty natural extension of the present design would be to *supplement* the currently used nominal subtyping with structural subtyping. The latter could be easily incorporated, by making lists of interface/trait names optional (rather than mandatory) when providing trait implementations. The `implements` statement, for instance, would still serve to bind methods (retroactively) to types, and to provide optional nominal subtyping, whose use would remain highly recommended, for documentation purposes [14].

But in all cases where such nominal subtyping would be intentionally skipped, e.g. because it might appear more cumbersome, the compiler would take over. It would infer any traits that a type conforms to, given the functionality that the type supports. This would, for instance, allow one to delete the entire `module intrinsics` from the code example of Listing 6.5, and to thereby achieve greater conciseness of formulation, while still maintaining strict conformance to the Open/Closed Principle. We ranked structural subtyping as an optional feature, here, mainly in order to ease prototype implementations of the present design.

7.3 Union types

Another natural extension would be to allow both type sets, and traits that are formulated in terms of them (cf. Sect 5.1.2), to be used directly within the `type` specifier of variable declarations. This would offer additional compile-time polymorphism support, via union types. The example of a

generic `sum` function, that was given in Sect. 5.4.1, could for instance be re-written in the following, schematic, manner:

```
function sum(x) result(s)
   type(integer(*) | real(*)), intent(in) :: x(:)
   typeof(x)                              :: s
   integer :: i
   s = typeof(x)(0)
   do i = 1, size(x)
      s = s + x(i)
   end do
end function sum
```

This no longer employs a generic type parameter list, and instead makes use of both, a union type, and Fortran 2023's `typeof` declaration specifier, in order to allow this code to be used with any of the `integer` or `real` types that the language supports. An extension of the present design by union types would take it to its logical conclusion, offer an alternative to generic type parameters for certain use cases, such as the one shown above, and fill the gap that is present in Table 6.1.

7.4 Conclusions

The Fortran extensions, that are described in this document for both run-time and compile-time polymorphism, resulted from the consistent application of orthogonal language design. That is, significant new capabilities are provided through the systematic extension of already existing Fortran features, and their mutual interaction, rather than the piecemeal addition of mutually incompatible (and superfluous) new constructs. Indeed, only in one single case (the `implements` statement of Sect. 4.2) did we find it necessary to introduce a new language feature. But even then, it was ensured that the new feature would be of *equal* utility in supporting both compile-time and run-time polymorphism, so that orthogonality, again, prevailed.

The end result is a language that is both far less verbose, much easier to use, and incomparably more powerful than that of competing non-holistic approaches, which prefer to abandon the proven philosophy of orthogonal design. The presented extensions are fully backwards compatible with the present Fortran language, and (in stark contrast to competing approaches) allow for a consistent use of fully type-checked generics not just in procedural and functional, but *also* in OO programming settings, *including* the support of static method dispatch (which can improve the performance of polymorphic methods by facilitating inlining via the compiler).

Taken together, these capabilities will allow for the uniform management of source code dependencies on both language-intrinsic and user-defined types in Fortran. They will enable the Fortran programmer to write unprecedented modular code, that is on par with the most modern languages in terms of reusability, and moreover does not sacrifice any computational performance. The present design achieves all of this largely *without* requiring manual instantiations of generics, and it furthermore provides a solid foundation for a number of possible future extensions, like support for array-rank genericity, structural subtyping, and compile time polymorphism via union (sum) types.

Bibliography

[1] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys* 17 (4 Dec. 1985), pages 471–523. URL: https://dl.acm.org/doi/pdf/10.1145/6041.6042.

[2] Brad J. Cox, Steve Naroff, and Hansen Hsu. "The Origins of Objective-C at PPI/Stepstone and Its Evolution at NeXT". In: *Proceedings of the ACM on Programming Languages* 4 (June 2020), pages 1–74. URL: https://dl.acm.org/doi/10.1145/3386332.

[3] Ole-Johan Dahl. "The Birth of Object Orientation: the Simula Languages". In: *From Object-Orientation to Formal Methods*. Edited by O. Owe, S. Krogdahl, and T. Lyche. Berlin: Springer, 2004. URL: https://www.mn.uio.no/ifi/english/about/ole-johan-dahl/bibliography/the-birth-of-object-orientation-the-simula-languages.pdf.

[4] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, 1994.

[5] Magne Haveraaen, Jaakko Järvi, and Damian Rouson. *Reflecting on Generics for Fortran*. July 2019. URL: https://j3-fortran.org/doc/year/19/19-188.pdf.

[6] Allen Holub. *Holub on Patterns, Learning Design Patterns by Looking at Code*. Apress, 2004. ISBN: 1-59059-388-X.

[7] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. URL: https://doc.rust-lang.org/stable/book/ch10-02-traits.html#implementing-a-trait-on-a-type.

[8] George Lyon. URL: https://gist.github.com/GeorgeLyon/c7b07923f7a800674bc9745ae45ddc7f.

[9] Robert C. Martin. "The Open/Closed Principle". In: *C++ Report* (Jan. 1996). URL: https://web.archive.org/web/20060822033314/http://www.objectmentor.com/resources/articles/ocp.pdf.

[10] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002. ISBN: 978-0-135-97444-5.

[11] Robert C. Martin. *OO vs FP*. Nov. 2014. URL: https://blog.cleancoder.com/uncle-bob/2014/11/24/FPvsOO.html.

[12] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017. ISBN: 978-0-13-449416-6.

[13] Nicholas D Matsakis and Felix S Klock II. "The Rust language". In: *ACM SIGAda Ada Letters*. Volume 34. ACM. 2014, pages 103–104.

[14] *Mojo Manual: Traits*. URL: https://docs.modular.com/mojo/manual/traits/.

[15] Benjamin C. Pierce. *Programming with intersection types, union types, and polymorphism*. URL: https://api.semanticscholar.org/CorpusID:118907019.

[16] PL22.3 and WG5. *J3/18-007r1 (F2018 interpretation document)*. Oct. 2018. URL: https://j3-fortran.org/doc/year/18/18-007r1.pdf.

[17] Ian Lance Taylor. URL: https://github.com/golang/go/issues/45346.

[18] *The Java Tutorials. Abstract Methods and Classes*. URL: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html.

[19] *The Swift Programming Language (6 beta). Extensions*. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/extensions/.

[20] Wolfgang Weck and Clemens Szyperski. *Do We Need Inheritance?* URL: https://www.researchgate.net/publication/2297653_Do_We_Need_Inheritance.

Acknowledgements

The present extensions for Fortran could not have been devised without the guidance we had through the prior work of countless developers on the Swift, Go, Rust, Carbon, and Java languages. We wish to thank all these developers, even though we can only mention here a few of them explicitly by name, who have particularly influenced the present effort. Chris Lattner's design of the Swift language provided the foundational ideas upon which we built the majority of our language extensions for Fortran, including much of the modern OO features, and the generics. While Robert Griesemer's, Ian Lance Taylor's, and the entire Go team's inspirational work on type sets accounts largely not only for our remaining generics design, but also for much of its ease of use. Robert also kindly provided the original code version from which Listing 2.2 was derived, and answered the many questions we had regarding Go's generics. We are also grateful to everyone else who gave us their feedback to our work, in particular Chris Lattner, Magne Haveraaen, and Brad Richardson.