A trait system for the uniform expression of run-time and compile-time polymorphism in Fortran

Konstantinos Kifonidis, Ondrej Certik, Derick Carnazzola

November 14, 2023

Abstract

Based on conclusions drawn from a survey of modern languages, a trait system for Fortran is developed that is fully backwards compatible with the present Fortran language, and allows for the uniform management of source code dependencies on both user-defined and language-intrinsic types via run-time and compile-time polymorphism, i.e. object-oriented and generic programming. The feature set described here is small enough to facilitate a first prototype implementation in an open source compiler, like LFortran, or LLVM/Flang, but at the same time comprehensive enough to already endow Fortran with polymorphism capabilities that largely equal those of modern programming languages like Swift, Rust, or Go. The discussed extensions 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 1960s 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 defining 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. [11]), as well as in the following quotation from his blog [10]:

> "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 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.

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. That is, types that contain both user-defined state, and implementation code which operates on that (hidden) state. These are the dependencies that Martin is concretely referring to in the above quotation, and it is these dependencies on (volatile) implementation (details) 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 modules, 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 may 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 [11] 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 (i.e. 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, 9], 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 [15]. 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 similar learning process, as that outlined for run-time polymorphism, took 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 first made with the advent of Rust [12].

Rust came with a trait (i.e. polymorphic interface) 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, existing 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 procedure dispatch 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 them. 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 additions 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 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 presented in Chapter 3, we will then recode this 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 "pair-wise" 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 = 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 hard-wired (i.e. rigid) 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 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 function signatures, or *type sets*, like “`int32 | float64`” in the present example. The latter 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 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.

```
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!")
            return
    }

    fmt.Println(avi(xi))
    fmt.Println(avf(xf))
}
```

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 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 an encapsulated 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 way of implementing interfaces requires only 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 the objects `other` and `drv` enables either `SimpleSum` or `PairwiseSum` instances to be plugged into these objects' clients.

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 generic type parameters.

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

```go
package main

import "fmt"

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

type INumeric interface {
   int32 | float64
}

type ISum[T INumeric] interface {
   sum(x []T) T
}

type IAverager[T INumeric] interface {
   average(x []T) T
}

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

```go
// .............

type SimpleSum[T INumeric] struct {
}

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() {
```

```go

    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]{} }
        avf = Averager[float64]{ SimpleSum[float64]{} }
    case 2:
        avi = Averager[int32]{ PairwiseSum[int32]{ SimpleSum[int32]{} } }
        avf = Averager[float64]{ PairwiseSum[float64]{ SimpleSum[float64]{} } }
    default:
        fmt.Println("Case␣not␣implemented!")
        return
    }

    fmt.Println(avi.average(xi))
    fmt.Println(avf.average(xf))
}
```

The main program makes use of Go's built-in structure constructors, and constructor chaining, 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.

3.2 Rust

Like Go, Rust supports both run-time and compile-time polymorphism through polymorphic interfaces, which Rust calls "traits". In contrast to Go, Rust has its programmers implement traits in an explicit manner, by using explicit "`impl`" code blocks to provide a trait's method implementations. These same `impl` blocks also serve to bind methods to a type that aren't a part of some trait, like e.g. user-defined constructors for `structs` (see the functions named "`new`" in the following code Listing 3.2).

Differing from 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. There are, however, a few minor differences, that are listed in the following notes.

- 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.
- Use of a "`Num`" trait from the external "`num`" crate was necessary, in order to enable numeric operations on generic types, which leads to dependency on external library code.
- At times, use of the "`Copy`" trait also had to be 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.
- Despite reliance on external dependencies, conversion to generic types wasn't possible. This led to the necessity to move a type conversion, that in the Go implementation was included in the code of method `average`, to this method's calls in the main program. We also had to import a `zero` generic function from the external `num` crate, in order to initialize the variable `s` that is returned by the `sum` method of the `SimpleSum` ADT.
- 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 (see the functions named `new` 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 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.

```rust
pub mod interfaces {

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

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

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

pub mod simple_library {

    use num::{zero,Num};
    use crate::interfaces::ISum;

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

    pub struct SimpleSum;

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

}

pub mod pairwise_library {

    use num::Num;
    use crate::interfaces::ISum;

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

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

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

    impl<T> ISum<T> for PairwiseSum<T> where T: Num {
        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 num::Num;
    use crate::interfaces::{ISum,IAverager};

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

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

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

    impl<T> IAverager<T> for Averager<T> where T: Num {
```

```
        fn average(&self, x: &[T], length: T) -> T {
            return self.drv.sum(&x) / length;
        }
    }

}

// ..............
// 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, xi.len() as i32));
```

```
    println!("{}", avf.average(&xf, xf.len() as f64));
}
```

The main program in the Rust version is somewhat longer than in the corresponding Go version because of the need to import dependencies from modules (as it would be necessary in realistic situations). Its logic is also somewhat convoluted compared to the Go version, because Rust doesn't allow the programmer to declare variables that aren't initialized upon declaration, and because of the aforementioned necessity to move the required type conversions out of method `average`, and into the calls of this method. Otherwise the codes are pretty much identical[1].

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 trait 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 prints out "`I am 4.9`":

```swift
protocol IPrintable {
    func out()
}
extension Float64: IPrintable {
    func out() {
        print("I am \(self)")
    }
}
var x: Float64 = 4.9
x.out()
```

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,

[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 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.

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 3.3 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:

- Swift uses angled brackets to indicate generic parameter lists.
- Generic type constraints are formulated by supplying a protocol name after a type parameter (separated by a colon).
- Swift does not supply an equivalent to Go's `int32 | float64` syntax and semantics. Hence the user must use a `Numeric` protocol that is defined by the standard library, as a constraint for numeric types. Which leads to reliance on library code.
- Unfortunately, Swift's `Numeric` protocol does *not* support the division operation! Hence the division that would have been required in function `average` of the `Averager` ADT had to be moved out to the calling code of the main program.
- The Swift version makes use of language built-in, default, structure constructors (called "initializers").
- Array slices are not arrays themselves. Hence, an explicit conversion using an `Array()` constructor is required in such cases.
- By default, function and method calls in Swift make use of keyword arguments.
- The syntax for type conversion into a generic type `T` is somewhat peculiar. E.g. Go's `T(0)` is written as `T(exactly:0)!` in Swift (making use of the mandatory keyword "`exactly`" in the function responsible for the type conversion).

Listing 3.3: Encapsulated Swift version of the array averaging example.

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

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

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

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

```

struct SimpleSum: ISum {

    func sum<T: Numeric>(x: [T]) -> T {
        var s: T
        s = T(exactly:0)!
        for i in 0 ... x.count-1 {
            s += x[i]
        }
        return s
    }
}

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

struct PairwiseSum: ISum {
    var other: ISum

    func sum<T: Numeric>(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...]))
        }
    }

}

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

struct Averager: IAverager {
    var drv: ISum

    func average<T: Numeric>(x: [T]) -> T {
        return drv.sum(x: x)
    }
}

// ..............
// main function
// ..............

func main() {
```

```

    let avs = Averager(drv: SimpleSum())
    let avp = Averager(drv: PairwiseSum(other: SimpleSum()))

    var av : IAverager = avs

    let xi: [Int32] = [1,2,3,4,5]
    let xf: [Float64] = [1.0,2.0,3.0,4.0,5.0]

    var key: Int32?

    print("Simple␣␣␣sum␣average:␣1")
    print("Pairwise␣sum␣average:␣2")
    print("Choose␣an␣averaging␣method:␣")
    key = Int32(readLine()!)

    switch key {
    case 1:
        // simple sum case
        av = avs
    case 2:
        // pairwise sum case
        av = avp
    default:
        print("Case␣not␣implemented!")
        return
    }

    print( av.average(x: xi) / Int32(xi.count) )
    print( av.average(x: xf) / Float64(xf.count) )
}

// execute main function
main()
```

Even a casual glance at the Swift version will show that the Swift code is the easiest to read and understand among the encapsulated 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 Swift program 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.

3.4 Conclusions

The use of run-time polymorphism by means of (polymorphic) interfaces is rather similar in all the languages considered here. The most significant differences (that were not concretely explored here) appear to be that Go has stricter limitations on retroactively implementing interfaces for existing types than the other languages. Whereas Rust (with some minor restrictions), and Swift allow the implementation of an interface by some type to be accomplished independently from 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 very similar in that they support fully type checked generics via the mechanism of interfaces. In the following, we will thus only summarize 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. What sets Go apart from the other languages is its built-in, easy to use support for conversion to generic types, and especially its brilliant new notion to interpret type sets as interfaces, along with its syntax to support this notion. This enables the Go programmer to easily tailor constraints on generic types to his specific needs, which is what makes the use of generics in Go pleasant. We consider these latter particular features of Go as "must haves" for Fortran.

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 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, like initializing a generic type. The Rust version of our test case is therefore 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. But even with the functionality provided by such external dependencies, Rust doesn't allow type conversion to generic types within generic routines. A necessary capability for numerical work that is, for instance, built into Go. The points we like most about the language are its idea to decouple trait implementations from a `struct`'s definition through explicit `impl` blocks, 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 particular 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 allows the Swift compiler 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 corresponding OO versions.

We hence consider Swift's generics to be the most attractive model to base Fortran's basic generic 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 basic generics design for Fortran.

Fortran will necessarily do a better job in this respect, both by borrowing Go's idea of formulating interfaces in terms of type sets, so that the user can easily implement his own type constraints. But also by making accessible to the user a set of language-built in interfaces that are truly useful for a variety of numeric operations, that are implemented by Fortran's intrinsic types.

Chapter 4

Fortran additions I: Subtyping

The present and the next chapter describe additions 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, or Go.

4.1 Named abstract interfaces (traits)

The most important of the following additions is the capability to define named abstract interfaces, or traits (i.e. named collections of procedure signatures), and to declare instance variables of them. Named abstract interfaces are the crucial feature that is required in order to uniformly and properly express both run-time and compile-time polymorphism (i.e. generics) 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 types, named versions of them are required, as in the following example, that defines two such named interfaces, `IAddition` and `ISubtraction`, that are intended as abstract blueprints for actual implementations of two type-bound procedures, named `add` and `sub`:

```
abstract interface :: IAddition
   subroutine add(self,b)
      import; implicit none
      class(IAddition), intent(inout) :: self
      real,             intent(in)    :: b
   end subroutine add
end interface

abstract interface :: ISubtraction
   subroutine sub(self,b)
      import; implicit none
      class(ISubtraction), intent(inout) :: self
      real,                intent(in)    :: b
   end subroutine sub
end interface
```

Since named abstract interfaces are merely a simple addition to Fortran, that aims to extend the present use cases of abstract interfaces in the language (which currently serve as bounds on the signatures of procedure pointers, and `deferred`, i.e. abstract, methods), this new feature is fully backwards compatible.

4.1.2 Extends specifier

Abstract interface definitions must allow the programmer to declare new abstract interfaces that inherit procedure signatures from *multiple* simpler interfaces (multiple interface inheritance). In the following example, the interface `IBasicMath` inherits the procedure signatures contained in both the interfaces `IAddition`, and `ISubtraction`, making `IBasicMath` at the same time a *subtype* of both these simpler interfaces. That is, objects that adopt or implement the `IBasicMath` interface (i.e. conform to it), can also be used in settings that require conformance to either the `IAddition`, or `ISubtraction` interfaces:

```
    abstract interface, extends(IAddition,ISubtraction) :: IBasicMath
    end interface
```

4.1.3 Declaration of run-time polymorphic variables (trait objects)

Named abstract interfaces/traits are types in their own right. Their purpose is to allow the programmer to declare (polymorphic) variables in terms of them. Either directly, i.e. as objects of such interfaces in run-time polymorphism, as demonstrated in this section. Or indirectly, as constraints on generic type parameters in compile-time polymorphism (see Sect. 5.3 for examples of the latter).

In order to use abstract interfaces to support run-time polymorphic objects through subtyping, Fortran's `class` specifier for variable declarations needs to be enhanced to accept named abstract interfaces, like in the following declaration of a procedure argument (taken from Sect. 4.1.1)

```
  class(IAddition), intent(inout) :: self
```

or in the declarations of the following two variables:

```
  class(IAddition), allocatable :: adder
  class(IAddition), pointer     :: adderptr
```

The semantics here are that whenever a named abstract interface 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 interface (like `IAddition` in the above examples), will make use of late binding. That is, their calls will be resolved by the run-time system of the language (e.g. through a virtual method table).

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 `self`, `adder`, and `adderptr` in the examples above) must either be arguments of a procedure, or they must be declared using the `allocatable`, or the `pointer` attribute. The proposed additions are therefore backwards compatible with the functionality that is already available in the present language. See also Sect. 5.1.2 for further information on the `class` specifier when used with abstract interfaces.

4.1.4 Short-hand notation for combination of traits

There are often cases where the combined functionality of two (or more) interfaces is required, but where one would not like to go through the labor of explicitly setting up a separate derived interface, like `IBasicMath` of Sect. 4.1.2. This can be useful in variable declarations. In such cases, it should be possible to specify the following

```
   class(IAddition + ISubtraction), allocatable :: addsub
```

instead of having to explicitly derive `IBasicMath` from `IAddition` and `ISubtraction`, as in Sect. 4.1.2, and then use it as follows:

```
   class(IBasicMath), allocatable :: addsub
```

4.2 Multiple interface inheritance for types

The language must make it possible not only for named abstract interfaces to conform to other named abstract interfaces, but also for *other types* to do the same, preferably regardless of whether such types are user-defined, or intrinsic to the language.

4.2.1 `Implements` specifier: Implementing traits for derived types

User-defined (that is derived) types can be made to conform to an interface by introducing an `implements` specifier in derived type definitions. In the following example, the `BasicMath` derived type implements (i.e. conforms to, or adopts) the interface `IBasicMath` that was defined in Sect. 4.1.2, by providing implementations of all the method signatures that are contained in that interface:

```
module basic
   ...

   type, implements(IBasicMath) :: BasicMath
      private
      real :: a
   contains
      procedure, public, pass(self) :: add
      procedure, public, pass(self) :: sub
   end type BasicMath

contains

   subroutine add(self,b)
      class(BasicMath), intent(inout) :: self
      real,             intent(in)    :: b
      self%a = self%a + b
   end subroutine add

   subroutine sub(self,b)
      class(BasicMath), intent(inout) :: self
      real,             intent(in)    :: b
```

```
      self%a = self%a - b
   end subroutine sub

end module basic
```

Notice, how `BasicMath`, by virtue of being an implementer of `IBasicMath`, is now also an implementer of `IAddition`, and `ISubtraction`. Hence, `BasicMath` can be used in client code that requires conformance to either one of the interfaces `IBasicMath`, `IAddition`, or `ISubtraction`.

It is crucial, for flexibility, that the above interface inheritance mechanism allow for a type to implement *multiple* different interfaces. For instance, if one wouldn't have defined the interface `IBasicMath` of Sect. 4.1.2, and would nevertheless need to use objects of type `BasicMath` in settings that require conformance to either the `IAddition` or `ISubtraction` interfaces, then the language must allow one to define the `BasicMath` type as follows (skipping, for brevity, the implementation of the actual methods, that would be done exactly as in the previous example):

```
   ...
   type, implements(IAddition,ISubtraction) :: BasicMath
      private
      real :: a
   contains
      procedure, public, pass(self) :: add
      procedure, public, pass(self) :: sub
   end type BasicMath
   ...
```

In case the "implementing" type is `abstract`, it must be allowed to provide an only partial implementation of the interface(s) that it adopts. However, any non-abstract type that is derived from this abstract type through implementation inheritance (i.e. subclassing via the `extends` specifier) must then provide a full implementation.

4.2.2 `Implements` statement: Retroactively adding functionality to types

In order to avoid having to wrap existing types into wrappers, when unforeseen new use cases result, it should be possible (as in Swift or Rust) to make any type implement new interfaces retroactively, regardless of where its original type definition is located. It would be desirable to provide such a capability for both user-defined and language intrinsic types, which would mean that user-provided methods would have to be allowed also for *intrinsic* types. The `implements` statement that is described below is aimed at ultimately accomplishing these capabilities, but for the purpose of a first prototype implementation it has been restricted here to provide such functionality for user-defined types only, i.e. its use on intrinsic types is presently prohibited.

Notice, that the `implements` statement has *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. The feature is modeled after the "`extension`" feature of Swift, where it is used to enable retroactive implementation of new methods, additional constructors, and especially new interfaces for types, in order to dynamically change an *interface inheritance hierarchy*, and achieve utmost code flexibility.

Swift's `extension` blocks fulfill essentially the same purpose as Rust's `impl` blocks. They have been simplified here (for a first implementation), and adjusted to Fortran's syntax that binds methods to types through declaration blocks, rather than by including the actual implementation bodies themselves into such a block (the implementations need to be supplied as module procedures, as is usual in Fortran). The syntax of the feature is largely symmetric to that of the `contains` section of derived type definitions. Most of the options that are allowed for type-bound procedure declarations in derived type definitions, are therefore also allowed for such declarations within an `implements` statement.

Retroactively adding methods to a type

Suppose that we would like to add, from within a different module, two more methods to the `BasicMath` type of Sect. 4.2.1, in order to give this type some additional functionality. This can be done using an `implements` *statement* as follows,

```
module enhanced

   use basic, only: BasicMath

   implements :: BasicMath
      procedure, public, pass(self) :: mul
      procedure, public, pass(self) :: div
   end implements BasicMath

contains
   ...
end module enhanced
```

where the actual implementations of the `mul` and `div` procedures would be given after the module's `contains` statement, as usual.

Retroactively implementing traits by a type

Assume now, that the purpose of our addition of the previous two methods was to actually make `BasicMath` conform to settings where implementations of multiplication with, or division by, a `real` are needed, and where the required functionality is described by two abstract interfaces called `IMultiplication` and `IDivision`. So far, we have added the code of the required methods, but we haven't made `BasicMath` pluggable yet into code that is written in terms of either one of these latter interfaces. To fix this, we simply state that the `BasicMath` type already has all of the required functionality, by acknowledging this using an `implements` statement for these two interfaces as follows:

```
module double_enhanced
   ...
   use enhanced, only: BasicMath

   implements (IMultiplication,IDivision) :: BasicMath
   end implements BasicMath

end module double_enhanced
```

Alternatively, we could have skipped both of the last two code examples, to instead adopt the interfaces and provide the method implementations simultaneously, using a single `implements` statement obtained as the fusion of the previous two. Or we could have employed two separate `implements` statements in order to conform to one interface at a time, like so:

```
   ...
   use basic, only: BasicMath

   implements IMultiplication :: BasicMath
      procedure, public, pass(self) :: mul
   end implements BasicMath

   implements IDivision :: BasicMath
      procedure, public, pass(self) :: div
   end implements BasicMath

contains
   ...
```

The result would have been the same. Such splitting of `implements` statements can be useful to improve code readability, as it makes the association between the interfaces, 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 interface lists in `implements` statements are optional, but not required.

4.3 Interoperability with subclassing

The present multiple interface inheritance (i.e. subtyping) features are interoperable with the single implementation inheritance (i.e. subclassing) that is already present in the language. That is, code examples like the following are possible:

```
   type :: Parent
   contains
      procedure :: method1
      procedure :: method2
   end type Parent

   type, extends(Parent), implements(IChild) :: Child
   contains
      procedure :: method3
      procedure :: method4
   end type Child
   ...
```

Here, a `Child` type is defined, that inherits two methods (`method1` and `method2`) from a `Parent` type, and adds two further methods of its own (`method3` and `method4`), in order to conform to an interface, `IChild`, that consists of the signatures of all four of these methods. In such use cases, the `extends` specifier shall always precede the `implements` specifier.

Thus, the features described here are backwards compatible with the OO model that is used in the present language. Moreover, since the new `implements` specifier allows for inheritance of *multiple* interfaces (see above), 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").

Chapter 5

Fortran additions 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 enhancements that are required in order to further support compile-time polymorphism, i.e. generics.

5.1 Enhancements to interfaces

5.1.1 Generic procedure signatures

Abstract interfaces should be allowed to contain signatures of generic procedures, as in Swift. The approach taken in Go and Rust to parameterize abstract interfaces themselves, appears not as attractive from a user's perspective. The following code shows, as an example, an abstract interface called `ISum` that contains the signature intended for a generic type-bound procedure (i.e. generic method), named `sum`:

```
abstract interface :: ISum
   function sum{INumeric :: T}(self,x) result(s)
      import; implicit none
      class(ISum), intent(in) :: self
      type(T),     intent(in) :: x(:)
      type(T)                 :: s
   end function sum
end interface
```

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

A similarity is that, in the same way that regular function parameters need to be constrained by a provided type, type parameters need to be constrained by a provided meta-type. This (meta-type) constraint must be an abstract interface name (like `INumeric` in the present example), that precedes the actual type parameter name. The proposed Fortran generics thus support "strong concepts", and can be fully type-checked by the compiler. Both, the type parameter and its constraint, are part of a generic type parameter list that is enclosed in curly braces, and follows immediately behind the procedure's name. Notice that the syntax used above, that 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 Type sets

In order to make the interface based generics facility easy to use for the programmer, it must be possible, as in the Go language, to define generics constraints by means of abstract interfaces that consist of type sets.

Unions of types

The following example shows the simplest form of such a type set, by defining an interface `INumeric`, 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` type:

```
abstract interface :: INumeric
   integer
end interface
```

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

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

that redefines interface `INumeric` 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* an interface consisting of such a type set, if (and only if) it is a member type of this set. For instance, Fortran's various `complex` types are not members of interface `INumeric`'s type set (as it is given above), because they do not appear in its definition. Hence, none of the `complex` types implements the `INumeric` interface. In particular, the `complex` types do not support, i.e. implement, the relational operators `(<)` and `(>)` that are required for conformance to this interface, given that these operators are implemented by the `integer` and `real(real64)` member types.

Kind parameters

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

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

Notice how this makes use of both unions of types, and 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 kind parameters here is merely syntactic sugar that allows one to avoid having to write out a type set for all the involved 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 an interface in this fashion, the smaller the set of intersecting methods will usually become.

Empty interface

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 interface, that matches all types (since any type has at least zero methods):

```fortran
abstract interface :: IAnyType
end interface
```

Implicit 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 define an explicit interface of type sets, like `INumeric` above, and to then have to use it as in Sect. 5.1.1. The following modification of interface `ISum`'s original declaration of Sect. 5.1.1, provides such an example:

```fortran
abstract interface :: ISum
   function sum{integer | real(real64) :: T}(self,x) result(s)
      import; implicit none
      class(ISum), intent(in) :: self
      type(T),     intent(in) :: x(:)
      type(T)                 :: s
   end function sum
end interface
```

The notation within the generic type parameter list in curly braces defines a type set interface implicitly, to be used as a type constraint for type `T`. In this particular case, to admit only the 32 bit `integer`, or 64 bit `real` type for `T`, as discussed above.

Predefined constraints

The facilities described in this section are flexible enough for the user to be able to construct generics constraints himself the way he needs them. Nevertheless, the language should ideally also supply a collection of predefined, commonly used generics constraints in the form of abstract interfaces that are contained in a language intrinsic module, tentatively called `generics_constraints` here. The list of such predefined interfaces could include

- an empty interface of the name `IAnyType` (as shown above),
- some predefined numeric interfaces allowing for different numeric operations, but also
- some predefined interfaces to allow for the use of relational operators with different types.

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

```
module user_code

   use, intrinsic :: generics_constraints, only: INumeric

   abstract interface :: ISum
      function sum{INumeric :: T}(self,x) result(s)
         import; implicit none
         class(ISum), intent(in) :: self
         type(T),     intent(in) :: x(:)
         type(T)                 :: s
      end function sum
   end interface

end module user_code
```

Present restrictions and possible future extensions

Interfaces that are formulated in terms of type sets are presently *exclusively* intended for use as generics constraints. Hence, a compiler must ensure that they are *not* used for any other purpose. In particular, they are not intended to be implemented by derived (i.e. user-defined) types[1], and they cannot be used in variable declarations that involve the `class` specifier (as described in Sect. 4.1.3).

As it was demonstrated by the examples that were given in this section, interfaces that are formulated in terms of type sets often boil down to a set of language intrinsic types. Since the `class` declaration specifier is undefined for intrinsic types, and since – moreover – the semantics of this specifier allow for late binding of methods, which is incompatible with intrinsic types, a compiler will need to ensure that the `class` declaration specifier is not used in conjunction with interfaces that are formulated in terms of type sets. This includes the empty interface, `IAnyType`.

In a future language revision, such interfaces could, however, be admitted for use with the `type` declaration specifier, in order to enable compile-time polymorphism through union (also called sum) types [13, 14]. This would take the present design to its logical conclusion, offer an alternative to generic type parameters for certain use cases, and fill a present gap in the later to be discussed Table 6.1.

5.2 Built-in facility for conversion to generic types

A language like Fortran, that is intended for numeric use, where conversions between different numeric types are required rather frequently, must allow conversions to generic types to be done as easily as it is the case in the Go language. That is, by means that are built into the language, without having to rely exclusively on external library functionality.

For instance, generic routines will often have to initialize the result of reduction operations, as it is, e.g., 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`. In Go, it is easily possible to

[1] One of the problems here is that any new intrinsic function that would be added to the language for some intrinsic type, 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 interfaces which are based on these type sets.

express this initialization by transforming the (typeless) zero constant, `0`, into the corresponding constant of type `T`, i.e. by simply writing `s = T(0)`.

If, for instance, `T` is then instantiated at compile time with the `float64` type, the expression `T(0)` will be transformed by the compiler into `float64(0)`, i.e. a call to the correct conversion function. In Fortran's case, the compiler would have to translate the above expression into the intrinsic function call `real(0,kind=real64)`, which should actually be easy to do, also for all other cases where such conversion is indeed possible. Otherwise, the compiler should emit an error message, and abort compilation at the generics instantiation step.

5.3 Generic procedures, methods, and derived types

As 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.

5.3.1 Generic procedures

Using the syntax 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[2]:

```
 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, is provided by an interface that admits both the `integer` and `real(real64)` types for `T` (see Sect. 5.1.2).

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

```
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. Alternatively, generic instantiation can be accomplished manually by the user, by the additional provision of a generic type argument 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 can be useful for combination with `associate` statements, or for use with procedure pointers, as in the next example:

[2] Generic subroutines can be coded completely analogously.

```
real(real64) :: total1, total2
procedure(sum_real64), pointer :: sumf

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

sumf => sum{real(real64)}

total1 = sumf([1.d0,2.d0,3.d0,4.d0,5.d0])
total2 = sumf([2.d0,4.d0,6.d0,8.d0])
```

5.3.2 Generic methods

The same summation algorithm as that of Sect. 5.3.1, when implemented as a generic method, `sum`, that is bound to a derived type named `SimpleSum`, which implements the interface `ISum` as it was given in Sect. 5.1.1, would instead look as follows:

```
module simple_library
   ...

   type, public, implements(ISum) :: SimpleSum
   contains
      procedure :: sum
   end type SimpleSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      class(SimpleSum), intent(in) :: self
      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
```

Generic methods are used completely analogously to generic procedures. The automatic instantiation use case of Sect. 5.3.1 would, for instance, look as follows:

```
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])
```

5.3.3 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` interface of Sect. 5.1.1 is implemented by another derived-type, named `PairwiseSum` (here, using an `implements` *statement*):

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

   implements ISum :: PairwiseSum{ISum :: U}
      procedure :: 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
         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
```

Notice, how type `PairwiseSum` depends on a generic type parameter, `U`, that is used within type `PairwiseSum` itself in order to declare a field variable of `type(U)`, which is named `other`. As is indicated by the type constraint on `U`, object `other` conforms to the `ISum` interface 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 latters' 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` interface (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 interface can be run-time polymorphic.

The following code snippet finally shows how an object of the `PaiwiseSum` type could be declared and instantiated statically (by substituting its generic type parameter `U` by the `SimpleSum` type of Sect. 5.3.2), and how its generic `sum` method would be employed using automatic type inference:

```
type(PairwiseSum{SimpleSum}) :: pairwise

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

5.4 Updated structure constructors

If a derived type is parameterized over a generic type, as in the example of Sect. 5.3.3, then its structure constructor 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 legal. The following (in this particular case redundant, as demonstrated above) run-time instantiation of object `pairwise` by the structure constructor of type `PairwiseSum` provides such an example:

```
type(PairwiseSum{SimpleSum}) :: 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])
```

As in Sect. 5.3.2, `SimpleSum` is a derived type that implements the `ISum` interface, but (in contrast to the `PairwiseSum` type, as implemented in Sect. 5.3.3) is not parameterized by any generic type parameter itself.

We also propose to make 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 holds the definition of its derived type. Assuming that this scope is a module, then the structure constructor will always be able to access all the components of its derived type, even if these are declared being `private` to the module's scope. Hence, these components could be initialized even by calls to the structure constructor that are being performed from outside the module's scope, 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 afore-given 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 enhanced structure constructors. Lacking these, user-defined constructors would have to be used, leading to overly complex implementations, as e.g. in the Rust example code given in Listing 3.2.

5.5 Associated types

It is often the case, that a method signature that is contained within an abstract interface, needs to declare arguments of a type, that will only be known once that interface is actually implemented. This type could, for instance, be a generic parameter over which a derived type, that is implementing the interface, might need to be parameterized.

The latter problem could, in principle, be solved by making use of separate generic type parameters in both the implementing type *and* its method, and then enforcing the consistency of these parameters through manual instantiation, by passing identical type arguments to them. A much simpler solution, though, that does not require any manual instantiation of generics, can be obtained by allowing "associated types" to be declared within abstract interfaces, as it is possible in both the Rust and Swift languages.

The following example illustrates how an associated type is declared inside an abstract interface, `IAppendable`, that requires any implementing type to provide functionality for appending items to itself. These items are declared here to be of `type(Element)`, where `Element` is an alias, or placeholder, for any actual types that will be provided by implementations of that interface.

```
module vector_library

   use, intrinsic :: generics_constraints, only: IAnyType

   implicit none
   private

   abstract interface :: IAppendable
      type, alias :: Element  ! associated type "Element"
      subroutine append(self, item)
         import; implicit none
         class(IAppendable), intent(inout) :: self
         type(Element),      intent(in)    :: item
      end subroutine append
   end interface

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

contains

```

```
   subroutine append(self, item)
      class(Vector{U}), intent(inout) :: self
      type(U),          intent(in)    :: item
      self%elements = [self%elements, item]
   end subroutine append

end module vector_library
```

An example of such an implementation of interface `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 wish to append to it, 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 interface `IAppendable`'s definition (that doesn't know anything about type `U`), we made use of the placeholder (i.e. associated) type `Element` in the latter interface. Given method `append`'s implementation, the compiler will then infer `Element` to be of the actual type `U`.

5.6 Extensibility to 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 described here, we defer it to a separate document.

Chapter 6

Fortran versions of the test problem

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 both complete functional and encapsulated (i.e. OO) Fortran code versions of the test problem that is used throughout this document.

6.1 Functional versions

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 code version of 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 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 interface `INumeric`, that is implemented as the type set `integer | real(real64)`.
- Interface `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.
- *All* the required instantiations of generic procedures are done 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

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
```

```fortran


program main

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

   ! dependencies on implementations
   use averager_library, only: simple_average, pairwise_average

   implicit none

   ! declarations
   integer,       parameter :: xi(5) = [1, 2, 3, 4, 5]
   real(real64), parameter :: xf(5) = [1.d0, 2.d0, 3.d0, 4.d0, 5.d0]

   integer :: key

   write(*,'(a)') 'Simple␣␣␣sum␣average:␣1'
   write(*,'(a)') 'Pairwise␣sum␣average:␣2'
   write(*,'(a)',advance='no') 'Choose␣an␣averaging␣method:␣'
   read(*,*) key

   select case (key)
   case (1)
      print '(i8)',   simple_average(xi)
      print '(f8.5)', simple_average(xf)
   case (2)
      print '(i8)',   pairwise_average(xi)
      print '(f8.5)', pairwise_average(xf)
   case default
      stop 'Case␣not␣implemented!'
   end select

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 economical, and that type inference by the compiler should be straightforward and therefore reliable. Hence, we believe that the generics features 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. 5.3.1).

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

```fortran
program main

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

   ! dependencies on implementations
   use averager_library, only: simple_average, pairwise_average

   implicit none

   ! declarations
   integer,      parameter :: xi(5) = [1, 2, 3, 4, 5]
   real(real64), parameter :: xf(5) = [1.d0, 2.d0, 3.d0, 4.d0, 5.d0]

   integer :: key
   procedure(average_integer), pointer :: avi
   procedure(average_real64),  pointer :: avf

   abstract interface
      function average_integer(x) result(a)
         integer, intent(in) :: x(:)
         integer             :: a
      end function average_integer
      function average_real64(x) result(a)
         real(real64), intent(in) :: x(:)
         real(real64)             :: a
      end function average_real64
   end interface

   write(*,'(a)') 'Simple   sum average: 1'
   write(*,'(a)') 'Pairwise sum average: 2'
   write(*,'(a)',advance='no') 'Choose an averaging method: '
   read(*,*) key

   select case (key)
   case (1)
      avi => simple_average{integer}
      avf => simple_average{real(real64)}
   case (2)
      avi => pairwise_average{integer}
      avf => pairwise_average{real(real64)}
   case default
      stop 'Case not implemented!'
   end select

```

```fortran
   print '(i8)',   avi(xi)
   print '(f8.5)', avf(xf)

end program main
```

6.2 Encapsulated versions

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 our encapsulated Fortran code version of the test problem, that corresponds closest to the code versions that were presented in Chapter 3 for all the other languages.

- As in all these other versions, three interfaces are used to manage all the source code dependencies in the problem: `INumeric`, `ISum`, and `IAverager`. Interface `INumeric` is defined by the user himself as a type set, similar to the corresponding Go code.
- In contrast to the Go and Rust versions (Listings 3.1 and 3.2), none of the aforementioned interfaces is parameterized itself, since we followed Swift's basic model of generics.
- Interface inheritance is expressed through the presence of the `implements(...)` specifier in a derived-type definition (equivalent to Swift).
- The example code makes use, in the main program, of the new structure constructors, with their enhancements that were discussed in Sect. 5.4, 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: Proposed encapsulated Fortran version of the array averaging example.

```fortran
module interfaces

   use, intrinsic :: iso_fortran_env, only: real64

   implicit none
   private

   public :: INumeric, ISum, IAverager

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

   abstract interface :: ISum
```

```
      function sum{INumeric :: T}(self,x) result(s)
         import; implicit none
         class(ISum), intent(in) :: self
         type(T),     intent(in) :: x(:)
         type(T)                 :: s
      end function sum
   end interface

   abstract interface :: IAverager
      function average{INumeric :: T}(self,x) result(a)
         import; implicit none
         class(IAverager), intent(in) :: self
         type(T),          intent(in) :: x(:)
         type(T)                      :: a
      end function average
   end interface

end module interfaces


module simple_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

   type, public, implements(ISum) :: SimpleSum
   contains
      procedure :: sum
   end type SimpleSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      class(SimpleSum), intent(in) :: self
      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


```

```
module pairwise_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

   type, public, implements(ISum) :: PairwiseSum
      private
      class(ISum), allocatable :: other
   contains
      procedure :: sum
   end type PairwiseSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      class(PairwiseSum), 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

   type, public, implements(IAverager) :: Averager
      private
      class(ISum), allocatable :: drv
   contains
      procedure :: average
   end type Averager

contains

```

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

end module averager_library


program main

   ! dependencies on abstractions
   use interfaces,       only: IAverager

   ! dependencies on implementations
   use simple_library,   only: SimpleSum
   use pairwise_library, only: PairwiseSum
   use averager_library, only: Averager

   implicit none

   ! declarations
   integer :: key
   class(IAverager), allocatable :: avs, avp, av

   ! use of enhanced structure constructors
   avs = Averager(drv = SimpleSum())
   avp = Averager(drv = PairwiseSum(other = SimpleSum()))

   write(*,'(a)') 'Simple   sum average: 1'
   write(*,'(a)') 'Pairwise sum average: 2'
   write(*,'(a)',advance='no') 'Choose an averaging method: '
   read(*,*) key

   select case (key)
   case (1)
      ! simple sum case
      av = avs
   case (2)
      ! pairwise sum case
      av = avp
   case default
      stop 'Case not implemented!'
   end select

   print '(i8)',   av%average([1, 2, 3, 4, 5])
   print '(f8.5)', av%average([1.d0, 2.d0, 3.d0, 4.d0, 5.d0])
```

```fortran

end program main
```

The most important point to note, in Listing 6.3, is how this code makes use of trait objects (cf. Sect. 4.1.3) and generics to avoid rigid dependencies on user-defined implementation and language intrinsic types, respectively, and thus realizes 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 abstract interfaces (see the `use` statements in the above modules).

Notice also, that all the abstract interfaces that are required for this purpose are defined by the user himself (without any need to depend on external libraries), and that not a single manual instantiation of generics is necessary anywhere in the code (as it was already the case in the functional version of Listing 6.1). The encapsulated Fortran version that is described here is therefore as clean as the Go implementation of Listing 3.1 with respect to dependency management, and it is as easy to use, and to read and understand, as the Swift implementation of Listing 3.3.

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 the compiler, to potentially improve code performance. Listing 6.4 gives a *minimally* changed version of the OO Fortran code of Listing 6.3, to effect static dispatch of the different `sum` methods.

The required changes are confined to a parameterization of the `PairwiseSum` and `Averager` derived types, by generic type parameters that conform to the `ISum` interface and are named `U`. 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. This signifies compile-time polymorphism for the objects `other` and `drv` to the compiler, and hence static dispatch of their methods (whose interfaces were declared in the `ISum` interface). This needs to be contrasted 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 `alloctables` and their instantiation at run-time using constructor chaining, 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 case this is possible, as demonstrated in the next section, given that these objects do not contain any other `allocatable` data fields, like arrays). 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!

Listing 6.4: Demonstrates static method dispatch for the `sum` methods.

```fortran
module interfaces

  use, intrinsic :: iso_fortran_env, only: real64
```

[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.

```

   implicit none
   private

   public :: INumeric, ISum, IAverager

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

   abstract interface :: ISum
      function sum{INumeric :: T}(self,x) result(s)
         import; implicit none
         class(ISum), intent(in) :: self
         type(T),     intent(in) :: x(:)
         type(T)                 :: s
      end function sum
   end interface

   abstract interface :: IAverager
      function average{INumeric :: T}(self,x) result(a)
         import; implicit none
         class(IAverager), intent(in) :: self
         type(T),          intent(in) :: x(:)
         type(T)                      :: a
      end function average
   end interface

end module interfaces


module simple_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

   type, public, implements(ISum) :: SimpleSum
   contains
      procedure :: sum
   end type SimpleSum

contains

   function sum{INumeric :: T}(self,x) result(s)
      class(SimpleSum), intent(in) :: self
      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


module pairwise_library

   use interfaces, only: ISum, INumeric

   implicit none
   private

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

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

contains

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

end module averager_library


program main

   ! dependencies on abstractions
   use interfaces,       only: IAverager

   ! dependencies on implementations
   use simple_library,   only: SimpleSum
   use pairwise_library, only: PairwiseSum
   use averager_library, only: Averager

   implicit none

   ! declarations
   integer :: key
   class(IAverager), allocatable :: avs, avp, av

   ! use of enhanced structure constructors
   avs = Averager(drv = SimpleSum())
   avp = Averager(drv = PairwiseSum(other = SimpleSum()))

   write(*,'(a)') 'Simple␣␣␣sum␣average:␣1'
   write(*,'(a)') 'Pairwise␣sum␣average:␣2'
   write(*,'(a)',advance='no') 'Choose␣an␣averaging␣method:␣'
   read(*,*) key

   select case (key)
   case (1)
```

```fortran
         ! simple sum case
         av = avs
      case (2)
         ! pairwise sum case
         av = avp
      case default
         stop 'Case␣not␣implemented!'
      end select

      print '(i8)',   av%average([1, 2, 3, 4, 5])
      print '(f8.5)', av%average([1.d0, 2.d0, 3.d0, 4.d0, 5.d0])

   end program main
```

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. 5.1.2.

As in corresponding Rust and Swift implementations of the test problem, the instantiation of the objects `other` and `drv` through constructor calls in the main program has the benefit that the compiler should be able to infer the correct type arguments that are required to automatically instantiate the involved generic derived types. Notice also, that in Listing 6.4, the `av` object of `IAverager` type 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 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 of 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 merely 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 Static method dispatch and static object declarations

In the present simple example, and taking Listing 6.4 as a baseline, it is actually possible to even avoid some of the run-time memory allocation overhead of the program, by having the field ob-

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

jects `other` and `drv`, that are contained within the `PairwiseSum` and `Averager` types, be statically declared. To accomplish this, the two lines

```
type(U), allocatable :: other
type(U), allocatable :: drv
```

in Listing 6.4 need to be replaced by the following two code lines:

```
type(U) :: other
type(U) :: drv
```

Notice that, because the compiler should be able to automatically infer generic type arguments from the types of regular arguments in constructor calls, the object instantiations that are to be carried out from the main program can remain the same, i.e.:

```
avs = Averager(drv = SimpleSum())
avp = Averager(drv = PairwiseSum(other = SimpleSum()))
```

Alternatively, the two calls of `Averager`'s structure constructor can be manually given generic type arguments to confirm the types of the regular arguments (deleting here the latter's keywords):

```
avs = Averager{SimpleSum}(SimpleSum())
avp = Averager{PaiwiseSum{SimpleSum}}(PairwiseSum(SimpleSum()))
```

Or one can delete the regular arguments to the constructor altogether, and provide only the generic type arguments, as follows (see Sect. 5.4):

```
avs = Averager{SimpleSum}()
avp = Averager{PairwiseSum{SimpleSum}}()
```

The full source code for this version of the test problem can be found in the `Code` subdirectory that is accompanying this document.

6.3 Conclusions

The Fortran extensions, that are described in this document for both modern-day object-oriented and generic programming, resulted from the consistent application of orthogonal language design. That is, new capabilities are provided through the simple extension of existing Fortran features, and their mutual interaction, rather than the addition of (complex) new constructs. Indeed, only in one single case (the `implements` statement of Sect. 4.2.2) did we find it necessary to introduce a new language feature.

The end result is a language that is both significantly easier to use, and much more powerful, than the results of approaches that prefer to abandon the proven philosophy of orthogonal design. The presented extensions are fully backwards compatible with the present Fortran language, while allowing for a consistent use of generics not just in procedural and functional, but also in OO programming settings, including the use of static method dispatch (which can improve the performance of polymorphic methods by facilitating inlining via the compiler).

Taken together, these capabilities will enable the Fortran programmer to uniformly manage source code dependencies on both language intrinsic and user-defined types in his applications, and to thereby 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 rank genericity, 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. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002. ISBN: 978-0-135-97444-5.

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

[11] 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.

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

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

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

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

Acknowledgements

We thank Robert Griesemer of the Go language team for providing the original code version from which Listing 2.2 was derived, and for his and the Go team's inspirational work on type sets in Go generics, on which a good fraction of the present design for Fortran is based. In the same vein, we also thank the many developers of the Swift, Rust, and Carbon languages who, through their work, have also influenced the present design.