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

Konstantinos Kifonidis, Ondrej Certik, Derick Carnazzola

August 1, 2023

Abstract

TBD.

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 [2]. 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. [4]), as well as in the following quotation from his blog:

> "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

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 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, i.e. 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 also has dependencies on abstract data types that are provided by the programming language in which it is written. 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 [4] 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 somewhat less flexible) mechanism of compile-time polymorphism also for a number of 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 has been 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, that aimed at avoiding or mitigating the effects of 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. In modern languages, implementation inheritance is either kept solely for backwards compatibility reasons (e.g. in the Swift language), or it is foregone altogether (e.g. in Rust, Go, and Carbon).

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. 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 under the monikers "interfaces" and "interface inheritance". 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, albeit in 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. 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 [5].

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, ex-

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

Rust's main idea was quickly absorbed by almost all other mainstream modern languages, most notably Go, Swift, 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 case 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. 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 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 "pairwise" manner, dividing the array in half to carry out the summations recursively, and switching to the "simple" method once subdivision is no longer possible.

As a result, 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 `int` 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 `int`s. Given that we are dealing with three levels of dependencies, three levels of polymorphism will accordingly be required to remove all these dependencies.

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

```go
package main

import "fmt"

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

func pairwise_sum(x []int) int {
    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 []int) int {
    return simple_sum(x) / len(x)
}

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

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

```go
33   // ..............
34
35   func main() {
36
37      xi := []int{1,2,3,4,5}
38
39      var key int
40
41      fmt.Println("Simple___sum_average:_1")
42      fmt.Println("Pairwise_sum_average:_2")
43      fmt.Print("Choose_an_averaging_method:_")
44      fmt.Scan(&key)
45
46      switch key {
47      case 1:
48         fmt.Println(simple_average(xi))
49      case 2:
50         fmt.Println(pairwise_average(xi))
51      default:
52         fmt.Println("Case_not_implemented!")
53      }
54
55   }
```

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 (we thank Robert Griesemer of the Go team for providing the original code of this particular version of the example). The code makes use of Go's generics to admit arrays of both the int 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 (see the interface INumeric in the present example).

- Interfaces consist of either function signatures, or *type sets*, like "int | float64" in the present example. The latter signify a set of function signatures, too, namely the signatures of the in-

tersecting set of all the operations/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 integer constant 0 to the corresponding zero constant of type T.

- The code instantiates closures and stores these by value in two variables named avi and avf for later use (Fortran and C programmers should note that avi and avf are *not* function pointers!).

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

```go
1  package main
2
3  import "fmt"
4
5  type INumeric interface{ int | float64 }
6
7  func simple_sum[T INumeric](x []T) T {
8      var s T
9      s = T(0)
10     for i := 0; i < len(x); i++ {
11         s += x[i]
12     }
13     return s
14 }
15
16 func pairwise_sum[T INumeric](x []T) T {
17     if len(x) <= 2 {
18         return simple_sum(x)
19     }
20     m := len(x) / 2
21     return pairwise_sum(x[:m+1]) + pairwise_sum(x[m+1:])
22 }
23
24 func average[T INumeric](sum func([]T) T, x []T) T {
25     return sum(x) / T(len(x))
26 }
27
28 func main() {
29     xi := []int{1, 2, 3, 4, 5}
30     xf := []float64{1, 2, 3, 4, 5}
31
32     var key int
33     var avi func([]int) int
34     var avf func([]float64) float64
35
36     fmt.Println("Simple⎵⎵⎵sum⎵average:", 1)
37     fmt.Println("Pairwise⎵sum⎵average:", 2)
```

```
38    fmt.Print("Choose_an_averaging_method:_")
39    fmt.Scan(&key)
40
41    switch key {
42       case 1:
43          avi = func(x []int) int {
44                  return average(simple_sum[int], x)
45              }
46          avf = func(x []float64) float64 {
47                  return average(simple_sum[float64], x)
48              }
49       case 2:
50          avi = func(x []int) int {
51                  return average(pairwise_sum[int], x)
52              }
53          avf = func(x []float64) float64 {
54                  return average(pairwise_sum[float64], x)
55              }
56       default:
57          fmt.Println("Case_not_implemented!")
58    }
59
60    fmt.Println(avi(xi))
61    fmt.Println(avf(xf))
62 }
```

The motivation to code the example as in Listing 2.2 is that once the two closures avi, and avf, are properly instantiated (by means of the switch statement), they may 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 implementations, in various modern languages, of encapsulated (i.e. OO) code versions of the test problem. 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 also 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 hence 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. Thus 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 definition (i.e. the ADT's struct definition). Limitations in Go are that language intrinsic types cannot have 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. Go, finally, makes it explicit in its type definition syntax that interfaces (like struct's) are types in their own right, and that hence polymorphic variables (i.e. objects) can be declared in terms of them.

Since version 1.18, Go also supports compile-time polymorphism through generics. Go's gener-

ics 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 example 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] in the declarations of "other" and "drv" enables either SimpleSum or PairwiseSum objects to be plugged into their higher-level 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 given in Sect. 2.2, namely to make all function arguments and return values polymorphic, by admitting as input and output parameters both the int 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. This is required in order to enable all the sum and average methods to use generic type parameters in Go.

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

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

```
23  // ..............
24
25  type SimpleSum[T INumeric] struct {
26  }
27
28  func (self SimpleSum[T]) sum(x []T) T {
29      var s T
30      s = T(0)
31      for i := 0; i < len(x); i++ {
32          s += x[i]
33      }
34      return s
35  }
36
37  // ...............
38  // PairwiseSum ADT
39  // ...............
40
41  type PairwiseSum[T INumeric] struct {
42      other ISum[T]
43  }
44
45  func (self PairwiseSum[T]) sum(x []T) T {
46      if len(x) <= 2 {
47          return self.other.sum(x)
48      } else {
49          m := len(x) / 2
50          return self.sum(x[:m+1]) + self.sum(x[m+1:])
51      }
52  }
53
54  // .............
55  // Averager ADT
56  // .............
57
58  type Averager[T INumeric] struct {
59      drv ISum[T]
60  }
61
62  func (self Averager[T]) average(x []T) T {
63      return self.drv.sum(x) / T(len(x))
64  }
65
66  // ..............
67  // main program
68  // ..............
69
70  func main() {
```

13

```
71
72      var avi IAverager[int]
73      var avf IAverager[float64]
74
75      xi := []int{1,2,3,4,5}
76      xf := []float64{1.,2.,3.,4.,5.}
77
78      var key int
79
80      fmt.Println("Simple___sum_average:_1")
81      fmt.Println("Pairwise_sum_average:_2")
82      fmt.Print("Choose_an_averaging_method:_")
83      fmt.Scan(&key)
84
85      switch key {
86      case 1:
87         avi = Averager[int]{ SimpleSum[int]{} }
88         avf = Averager[float64]{ SimpleSum[float64]{} }
89      case 2:
90         avi = Averager[int]{ PairwiseSum[int]{ SimpleSum[int]{} } }
91         avf = Averager[float64]{ PairwiseSum[float64]{ SimpleSum[float64]{} } }
92      default:
93         fmt.Println("Case_not_implemented!")
94      }
95
96      fmt.Println(avi.average(xi))
97      fmt.Println(avf.average(xf))
98   }
```

The main program makes use of Go's built-in structure constructors, and 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 int and float64 types separately, in order to then use these objects on int and float64 data, respectively. The fact that two such objects are required (one for each language intrinsic data type) is connected to the fact that in order to obtain generic methods in Go, one has to parameterize interfaces by generic parameters, and instantiate them 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).

14

In contrast to Go, Rust allows the programmer to implement traits for both user-defined *and* language intrinsic types, and to do so for types that are located in external libraries (called "crates" in Rust), as long as the traits themselves are defined in the programmer's own crate. The reverse, namely implementing an external trait for a user-owned type, is also possible. Only the case of implementing an external trait for an external type is not allowed (this is called the "orphan rule"). 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 within a trait, then this will make the trait unuseable for the declaration of any "trait objects", 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". The code in the following Listing 3.2 is in its outline quite similar to the Go code version. However, see the following notes for the differences.

- Rust uses angled brackets to indicate generic parameter lists.

- Use of a `Num` trait was necessary to enable numeric operations on generic types, which leads to dependency on external library code. Nevertheless, conversion to generic types wasn't possible. Hence, some type conversions had to be moved to the main program.

- Generics constraints in Rust are typically enforced by specifying the required traits in `impl` blocks using `where` statements.

- To declare run-time polymorphic variables one has to put so-called "trait objects" into "Boxes", i.e. to declare smart pointers of them (this is the Rust equivalent to using `allocatable` polymorphic objects in Fortran).

3.2.1 Encapsulated version coded in Rust

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

```rust
1  pub mod interfaces {
2
3      // ...........
4      // Interfaces
5      // ...........
6
7      pub trait ISum<T> {
8          fn sum(&self, x: &[T]) -> T;
9      }
10
11     pub trait IAverager<T> {
12         fn average(&self, x: &[T], length: T) -> T;
13     }
14 }
15
16 pub mod simple_library {
17
```

```rust
        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 {
                let n = 2;
                let l = x.len();
                if  l <= n {
```

```rust
                return self.other.sum(x);
            } else {
                let m = x.len() / 2;
                return self.sum(&x[0..m+1]) + self.sum(&x[m+1..l]);
            }
        }
    }

}

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};
```

17

```
114    use crate::pairwise_library::{PairwiseSum};
115    use crate::averager_library::{Averager};
116
117    let avsi = Averager::new( Box::new( SimpleSum{} ));
118    let avsf = Averager::new( Box::new( SimpleSum{} ));
119
120    let avpi = Averager::new(Box::new(PairwiseSum::new(Box::new(SimpleSum{}))));
121    let avpf = Averager::new(Box::new(PairwiseSum::new(Box::new(SimpleSum{}))));
122
123    let mut avi: Box<dyn IAverager::<i32>> = Box::new(avsi);
124    let mut avf: Box<dyn IAverager::<f64>> = Box::new(avsf);
125
126    let xi : [i32;5] = [1,2,3,4,5];
127    let xf : [f64;5] = [1.,2.,3.,4.,5.];
128
129    let key: i32;
130
131    println!("Simple    sum average: 1");
132    println!("Pairwise sum average: 2");
133    scan!("{}\n",key);
134
135    match key {
136        1 => {}
137        2 => { avi = Box::new(avpi);
138               avf = Box::new(avpf);
139        }
140        _ => { println!("Case not implemented!");
141        }
142    }
143
144    println!("{}", avi.average(&xi, xi.len() as i32));
145    println!("{}", avf.average(&xf, xf.len() as f64));
146 }
```

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.

The ":" operator followed by one or more interface names must be supplied for this purpose after the structure's or class's own name. 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 the implementation of protocols for language-intrinsic types, e.g. the following little code prints out "I am 4.9":

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

Swift generics support "strong concepts", and are thus fully type-checked, and their capabilities are on par with those of Go and Rust. In one aspect they are even superior, namely in that Swift allows for parameterized *methods*, instead of parameterized protocols. This has some interesting, postive consequences 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.

- 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 int | float64 syntax. Hence the user must use a Numeric protocol 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 struct 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. So 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.

```swift
1   // ...........
2   // Interfaces
3   // ...........
4
5   protocol ISum {
6       func sum<T: Numeric>(x: [T]) -> T
7   }
8
9   protocol IAverager {
10      func average<T: Numeric>(x: [T]) -> T
11  }
12
13  // ..............
14  // SimpleSum ADT
15  // ..............
16
17  struct SimpleSum: ISum {
18
19      func sum<T: Numeric>(x: [T]) -> T {
20          var s: T
21          s = T(exactly:0)!
22          for i in 0 ... x.count-1 {
23              s += x[i]
24          }
25          return s
26      }
27  }
28
29  // ...............
30  // PairwiseSum ADT
31  // ...............
32
33  struct PairwiseSum: ISum {
34      var other: ISum
35
36      func sum<T: Numeric>(x: [T]) -> T {
37          if ( x.count <= 2 ) {
38              return other.sum(x: x)
39          } else {
40              let m = x.count / 2
41              return sum(x: Array(x[..<m])) + sum(x: Array(x[m...]))
42          }
43      }
44
45  }
46
```

```swift
47  // ...........
48  // Averager ADT
49  // ...........
50
51  struct Averager: IAverager {
52      var drv: ISum
53
54      func average<T: Numeric>(x: [T]) -> T {
55          return drv.sum(x: x)
56      }
57  }
58
59  // ...........
60  // main program
61  // ...........
62
63  let avs = Averager(drv: SimpleSum())
64  let avp = Averager(drv: PairwiseSum(other: SimpleSum()))
65
66  var av : IAverager = avs
67
68  var xi: [Int] = [1,2,3,4,5]
69  var xf: [Float64] = [1.0,2.0,3.0,4.0,5.0]
70
71  var key: Int?
72
73  print("Simple    sum average: 1")
74  print("Pairwise sum average: 2")
75  print("Choose an averaging method: ")
76  key = Int(readLine()!)
77
78  switch key {
79  case 1:
80      // simple sum case
81      av = avs
82  case 2:
83      // pairwise sum case
84      av = avp
85  default:
86      print("Case not implemented!")
87  }
88
89  print( av.average(x: xi) / xi.count )
90  print( av.average(x: xf) / Float64(xf.count) )
```

Even a casual glance at the Swift version will show that the Swift code is the easiest to read and understand among all the encapsulated implementations. 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 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 always infer the required types by checking the actual arguments that are passed to a function/method)!

As an example, consider the `IAverager` protocol in the above Swift code. There's only a single (i.e. unparameterized) version of this protocol. Consequently, there's only a need in the main program to declare a single object variable, av, of that protocol (that enables av to be polymorphically assigned different `structs` that implement `IAverager`). Because it contains an "average" method that is generic, this *single* object can then be straightforwardly used on data of *both* the `Int` and `Float64` types!

This vastly simplifies client code that needs to make use of objects such as av, especially if such client code needs to work on *many* more types than just `Int` and `Float64`. Contrast this with Go's and Rust's model, where manual instantiation of a different version of `IAverager` is required for *every* different generic type parameter that the user wishes to employ. Notice also, how there's *not a single manual instantiation* of generics code required in the Swift example! We consider these significant advantages 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 interfaces is rather similar in all the languages considered here. The most significant difference appears 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 [3] 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. In the following we will make some final comments on the differences in all 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 interfaces as type sets, 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. 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 *excessive* obsession with type safety, its employment of move semantics by default, 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 somewhat 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 dispatch 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 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 inspection of the argument types that are passed to functions and methods. In contrast to the other languages, in Swift, the user basically never has to bother with instantiating a generic function, a method, and often not even a structure or a class.

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 any 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 interpreting type sets as interfaces, 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 numeric operations, and are implemented by Fortran's intrinsic types.

Chapter 4

Fortran additions I: Basic enhancements

4.1   Sealed derived types

4.2   Improved handling of constructors

4.2.1   Structure constructors

4.2.2   User-defined constructors

4.3   File extension with new defaults

Chapter 5

Fortran additions II: Subtyping

5.1 Named abstract interfaces (traits)

The most important of all the new features is the possibility to declare named abstract interfaces, or traits (of collections of procedure signatures). Fortran already allows the programmer to declare unnamed abstract interfaces, but in order to use these as types in their own right, named versions of them are required, as in the following example:

```
1    abstract interface :: IAddition
2       function add(a,b) result(c)
3          real, intent(in) :: a, b
4          real             :: c
5       end function add
6    end interface
7
8    abstract interface :: ISubtraction
9       function sub(a,b) result(c)
10         real, intent(in) :: a, b
11         real             :: c
12      end function sub
13   end interface
```

Named abstract interfaces are the central feature that is required in order to uniformly express both compile-time and run-time polymorphism (i.e. generics).

5.2 Enhanced type declarations

Named abstract interfaces/traits are types in their own right. Hence, (polymorphic) instance variables can be declared in terms of them. To enable the Fortran programmer to do so, Fortran's type declaration specifier needs to be enhanced to accept named abstract interfaces.

5.3 Multiple interface inheritance

5.3.1 Implements specifier

```
1  module basic
2
3     type, implements(IAddition,ISubtraction) :: BasicMath
4     contains
5        procedure, nopass :: add
6        procedure, nopass :: sub
7     end type BasicMath
8
9  contains
10
11    function add(a,b) result(c)
12       real, intent(in) :: a, b
13       real             :: c
14       c = a + b
15    end function add
16
17    function sub(a,b) result(c)
18       real, intent(in) :: a, b
19       real             :: c
20       c = a - b
21    end function sub
22
23 end module basic
```

5.3.2 Extensions

The extensions described below are a feature that is modeled after extensions in Swift, in order to make it possible to retroactively implement new methods, constructors, and especially traits, for types, regardless of where the original type definition is located. Swift's extensions fulfill essentially the same purpose as Rust's impl blocks in this respect. They have been adjusted here 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 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 extensions.

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 from above, in order to give this type some additional functionality. This can be done using an extension as follows:

```
1  module extended
2
3     use basic, only: BasicMath
4
5     extension :: BasicMath
```

```
6        procedure, nopass :: mul
7        procedure, nopass :: div
8      end extension BasicMath
9
10 contains
11     ...
12 end module extended
```

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 or division of `reals` are required, and where the required functionality is described by two interfaces called `IMultiplication` and `IDivision`. So far we have added the code of the required methods, but we haven't made `BasicMath` pluggable into code that is written in terms of either one of these latter interfaces. To fix this, we combine the addition of our two new methods to the `BasicMath` type with an `implements` specification for these two interfaces by this type, as follows:

```
1      use basic, only: BasicMath
2
3      extension, implements(IMultiplication,IDivision) :: BasicMath
4        procedure, nopass :: mul
5        procedure, nopass :: div
6      end extension BasicMath
7      ...
```

We could also split this single `extension` statement into two, to achieve the same effect, like so:

```
1      use basic, only: BasicMath
2
3      extension, implements(IMultiplication) :: BasicMath
4        procedure, nopass :: mul
5      end extension BasicMath
6
7      extension, implements(IDivision) :: BasicMath
8        procedure, nopass :: div
9      end extension BasicMath
10     ...
```

This can be useful to improve readability, if the different interfaces contain multiple procedure signatures. These two statements, together with their corresponding implementations, could then even be distributed among different modules and files.

Adding new initializers

Like derived type definitions, extensions can contain declarations of user-defined constructor functions with the `initial` specifier, in order to overload Fortran's default structure constructor with these user-defined functions.

27

Chapter 6

Fortran additions III: Generics

The new 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 some further enhancements that are exclusively required in order to further support compile-time polymorphism, i.e. generics.

6.1 Interfaces containing generic procedure signatures

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

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

The example illustrates the use of a generic type parameter, i.e. a meta-type, or a *type of types*. In this example, this type parameter is simply called `T`, and it is preceded by the name of an abstract interface that expresses a constraint on the type parameter. Fortran generics thus support "strong concepts". 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, since `T` is a meta-type, the syntax used above, that deviates slightly from how Fortran's usual function arguments are declared, appears justified as it reflects that, in type parameters, one is dealing with different entities.

6.2 Interfaces as type sets

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

Different "length" parameters for character variables would be handled in the same fashion. The following interface would, for the lack of a better example, be used to only admit `character` variables with a length of either 4 or 8 characters:

```
1    abstract interface :: IPrintable
2       character(len=4) | character(len=8)
3    end interface
```

For simple use cases, it should be optionally possible for the programmer to employ a shorter notation for declaring type constraints than having to write a separate interface like `IPrintable`, or `INumeric` above. As in the following modification of interface `ISum`'s declaration, that was given previously in Sect. 6.1:

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

The above notation would then define an abstract interface implicitly, to be used as a type constraint for type `T`. In this particular case, to admit only the default `integer`, or `real(real64)` types, for `T`.

6.3 Predefined interfaces for expressing common constraints

The language should ideally supply some predefined, commonly used generic constraints in the form of abstract interfaces that are contained in a language intrinsic module. The actual implementation of these interfaces could then, of course, employ the "interfaces-as-type-sets" syntax that was described above. For instance, a more general `INumeric` interface than the one given above, could be implemented as follows:

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

Notice how this makes use of kind parameters to include all `integer`, `real`, and `complex` types, admitted by the language, in a single `abstract interface` constraint. Such language provided, interfaces could then be used from user code through a `use` statement like in the following example for `INumeric`

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

```
10        end function sum
11    end interface
12
13 end module user_code
```

for use as a constraint in function and derived type implementations, or in other interfaces, like ISum here, that require the functionality of INumeric.

6.4 Conversions to generic types

6.5 Generic type parameters for methods and procedures

As already mentioned above, LFortran's design of generics should follow Swift's, if possible, and allow both ordinary and type-bound procedures (i.e. methods) to be given their own generic type parameters. An implementation of the generic method sum of Sect. 6.1, that is bound to a derived type with name SimpleSum, would look as follows:

```
1    function sum{INumeric :: T}(self,x) result(s)
2       type(SimpleSum), intent(in) :: self
3       type(T),         intent(in) :: x(:)
4       type(T)                     :: s
5       integer :: i
6       s = T(0)
7       do i = 1, size(x)
8          s = s + x(i)
9       end do
10    end function sum
```

Whereas the next example illustrates how the same procedure would look as a stand-alone (i.e. non-encapsulated) generic function:

```
1    function sum{INumeric :: T}(x) result(s)
2       type(T), intent(in) :: x(:)
3       type(T)             :: s
4       integer :: i
5       s = T(0)
6       do i = 1, size(x)
7          s = s + x(i)
8       end do
9    end function sum
```

6.6 Generic type parameters for derived types

In addition to procedures, generic type parameter lists must be allowed also for derived types, as in the following example, in which the interface ISum from above is implemented by a parameterized derived-type named PairwiseSum:

```
1    type, implements(ISum) :: PairwiseSum{ISum :: U}
2       private
```

30

```
3        type(U) :: other
4     contains
5        procedure :: sum
6     end type PairwiseSum
```

PairwiseSum depends on a generic type parameter U, that is used within 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.

6.7 Generic type parameters for structure constructors

If a derived type is parameterized with a generic type, then its structure constructor must also be assumed to be parameterized with the same generic type. Hence, calls of structure constructors that are instantiated with particular argument types replacing the generic type parameters of their derived types, like e.g.

```
1     Averager{SimpleSum}()
2     Averager{PairwiseSum{SimpleSum}}()
```

must be legal. Here, SimpleSum would be a derived type that implements the ISum interface, but (in contrast to the PairwiseSum and Averager types) is not parameterized by any generic type parameters itself.

6.8 Extensibility to rank genericity

Fortran's special role, as a language that caters to numeric 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 for the purpose of a very first prototype implementation of the generics features discussed here this is not essential, and we thus prefer to focus on such an extension in a future document.

Chapter 7

Proposed Fortran versions of the test example

7.1 Functional version

(What to do about this one, given that Fortran doesn't have Go's functional programming capabilities? In this case, I think it should be sufficient to demonstrate only the decoupling of the argument types through generics, and leave all the other coupling in, as it is the case in the coupled functional Go version.)

7.2 Encapsulated version

Listing 7.1 gives our Fortran version of the encapsulated form of the test example that corresponds to the code versions that were presented in Chapter 3 for all the other languages.

- We employ here the Go borrowed syntax `integer | real(real64)` to implement the interface `INumeric`, that is used in order to express type genericity for the array `x` and the result of the summation `s` in our different implementations of method `sum`.

- As in the corresponding Go version, `INumeric` is defined by the user himself as a type set consisting of the set of intersecting operations defined in Fortran for the `integer` and `real(real64)` types. There is thus no need for an external dependency.

- The remaining interfaces `ISum` and `IAverager` make use of generic methods that are declared in terms of `INumeric`. However, in contrast to the Go version, none of these interfaces is parameterized itself, since we followed Swift's model of generics.

- Interface inheritance is expressed through the presence of the `implements(...)` specifier in a derived-type definition (equivalent to Swift).

- Conversions to generic types are done as in Go. Notice, how the compiler will have to do the necessary replacements of, e.g., `T(0)` in function `sum` of class `SimpleSum` by calls to Fortran's correct conversion functions for integer and real types of the right kinds.

- The example code makes use, in the main program, of the new structure constructors, with their enhancements that were discussed in Sect. 6.7, for the classes `Averager`, `SimpleSum`, and `PairwiseSum`.

- The 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.

Listing 7.1: Proposed encapsulated Fortran version of the array averaging example.

```fortran
1  module interfaces
2
3      use, intrinsic :: iso_fortran_env, only: real64
4
5      public :: INumeric, ISum, IAverager
6
7      abstract interface :: INumeric
8          integer | real(real64)
9      end interface
10
11     abstract interface :: ISum
12         function sum{INumeric :: T}(self,x) result(s)
13             type(ISum), intent(in) :: self
14             type(T),    intent(in) :: x(:)
15             type(T)                :: s
16         end function sum
17     end interface
18
19     abstract interface :: IAverager
20         function average{INumeric :: T}(self,x) result(a)
21             type(IAverager), intent(in) :: self
22             type(T),         intent(in) :: x(:)
23             type(T)                     :: a
24         end function average
25     end interface
26
27 end module interfaces
28
29
30 module simple_library
31
32     use interfaces, only: ISum, INumeric
33
34     public :: SimpleSum
35
36     type, implements(ISum) :: SimpleSum
37     contains
38         procedure :: sum
39     end type SimpleSum
40
41 contains
42
43     function sum{INumeric :: T}(self,x) result(s)
```

```
44        type(SimpleSum), intent(in) :: self
45        type(T),         intent(in) :: x(:)
46        type(T)                     :: s
47        integer :: i
48        s = T(0)
49        do i = 1, size(x)
50           s = s + x(i)
51        end do
52     end function sum
53
54 end module simple_library
55
56
57 module pairwise_library
58
59     use interfaces, only: ISum, INumeric
60
61     public :: PairwiseSum
62
63     type, implements(ISum) :: PairwiseSum
64        private
65        type(ISum), allocatable :: other
66     contains
67        procedure :: sum
68     end type PairwiseSum
69
70 contains
71
72     function sum{INumeric :: T}(self,x) result(s)
73        type(PairwiseSum), intent(in) :: self
74        type(T),           intent(in) :: x(:)
75        type(T)                       :: s
76        integer, parameter :: N = 2
77        integer :: m
78        if (size(x) <= N) then
79           s = self%other%sum(x)
80        else
81           m = size(x) / 2
82           s = self%sum(x(:m)) + self%sum(x(m+1:))
83        end if
84     end function sum
85
86 end module pairwise_library
87
88
89 module averager_library
90
91     use interfaces, only: IAverager, ISum, INumeric
```

```fortran
92
93     public :: Averager
94
95     type, implements(IAverager) :: Averager
96        private
97        type(ISum), allocatable :: drv
98     contains
99        procedure :: average
100    end type Averager
101
102 contains
103
104    function average{INumeric :: T}(self,x) result(a)
105       type(Averager), intent(in) :: self
106       type(T),        intent(in) :: x(:)
107       type(T)                    :: a
108       a = self%drv%sum(x) / T(size(x))
109    end function average
110
111 end module averager_library
112
113
114 program main
115
116    ! dependencies on abstractions
117    use interfaces,       only: IAverager
118
119    ! dependencies on implementations
120    use simple_library,   only: SimpleSum
121    use pairwise_library, only: PairwiseSum
122    use averager_library, only: Averager
123
124    ! declarations
125    integer :: key
126    type(IAverager), allocatable :: avs, avp, av
127
128    ! use of enhanced structure constructors
129    avs = Averager(SimpleSum())
130    avp = Averager(PairwiseSum(SimpleSum()))
131
132    write(*,'(a)') 'Simple   sum average: 1'
133    write(*,'(a)') 'Pairwise sum average: 2'
134    write(*,'(a)',advance='no') 'Choose an averaging method: '
135    read(*,*) key
136
137    select case (key)
138    case (1)
139       ! simple sum case
```

```
140        av = avs
141    case (2)
142        ! pairwise sum case
143        av = avp
144    case default
145        stop 'Case_not_implemented!'
146    end select
147
148    print '(f8.5)', av%average([1.d0, 2.d0, 3.d0, 4.d0, 5.d0])
149    print '(i8)',   av%average([1, 2, 3, 4, 5])
150
151 end program main
```

The most important point to notice in Listing 7.1 is how the main program is the only part of the code that (necessarily) depends on implementations. The *entire* rest of the code depends merely on abstract interfaces (see the use statements in the above modules). The Fortran version described here is therefore as clean as the Go implementation with respect to dependency management, and as easy to use as the Swift implementation.

The features described in this document have enabled us to avoid rigidity in the program, by both decoupling it and making it operate on multiple data types, thus allowing for a maximum of code reuse.

7.3   Encapsulated (mostly) static version

To finally demonstrate how methods can be made to use static, as opposed to dynamic dispatch, Listing 7.2 gives a version of the encapsulated Fortran code that contains all the required changes, as compared to Listing 7.1, to effect static dispatch of the sum methods.

The changes are confined to a parameterization of the PairwiseSum and Averager derived types, by generic type parameters that are named U. These type parameters are then used in order to statically declare the field objects "other" and "drv" of these derived types, respectively, that were previously declared as allocatable variables that were to be initialized dynamically at run-time. Correspondingly, there is no longer any such dynamic instantiation necessary for these objects. The only things that are required are instantiations of the parameterized derived types in which these objects are contained. Notice how these instantiations are carried out from the main program within the calls of Averager's constructor, which is provided with the types SimpleSum, and PairwiseSum.

Listing 7.2: Encapsulated Fortran version of the array averaging example with static method dispatch.

```
1 module interfaces
2
3    use, intrinsic :: iso_fortran_env, only: real64
4
5    public :: INumeric, ISum, IAverager
6
7    abstract interface :: INumeric
8        integer | real(real64)
9    end interface
```

```fortran
10
11    abstract interface :: ISum
12       function sum{INumeric :: T}(self,x) result(s)
13          type(ISum), intent(in) :: self
14          type(T),    intent(in) :: x(:)
15          type(T)                :: s
16       end function sum
17    end interface
18
19    abstract interface :: IAverager
20       function average{INumeric :: T}(self,x) result(a)
21          type(IAverager), intent(in) :: self
22          type(T),         intent(in) :: x(:)
23          type(T)                     :: a
24       end function average
25    end interface
26
27 end module interfaces
28
29
30 module simple_library
31
32    use interfaces, only: ISum, INumeric
33
34    public :: SimpleSum
35
36    type, implements(ISum) :: SimpleSum
37    contains
38       procedure :: sum
39    end type SimpleSum
40
41 contains
42
43    function sum{INumeric :: T}(self,x) result(s)
44       type(SimpleSum), intent(in) :: self
45       type(T),         intent(in) :: x(:)
46       type(T)                     :: s
47       integer :: i
48       s = T(0)
49       do i = 1, size(x)
50          s = s + x(i)
51       end do
52    end function sum
53
54 end module simple_library
55
56
57 module pairwise_library
```

```fortran
58
59     use interfaces, only: ISum, INumeric
60
61     public :: PairwiseSum
62
63     type, implements(ISum) :: PairwiseSum{ISum :: U}
64        private
65        type(U) :: other
66     contains
67        procedure :: sum
68     end type PairwiseSum
69
70  contains
71
72     function sum{INumeric :: T}(self,x) result(s)
73        type(PairwiseSum{U}), intent(in) :: self
74        type(T),              intent(in) :: x(:)
75        type(T)                          :: s
76        integer, parameter :: N = 2
77        integer :: m
78        if (size(x) <= N) then
79           s = self%other%sum(x)
80        else
81           m = size(x) / 2
82           s = self%sum(x(:m)) + self%sum(x(m+1:))
83        end if
84     end function sum
85
86  end module pairwise_library
87
88
89  module averager_library
90
91     use interfaces, only: IAverager, ISum, INumeric
92
93     public :: Averager
94
95     type, implements(IAverager) :: Averager{ISum :: U}
96        private
97        type(U) :: drv
98     contains
99        procedure :: average
100    end type Averager
101
102 contains
103
104    function average{INumeric :: T}(self,x) result(a)
105       type(Averager{U}), intent(in) :: self
```

```fortran
106        type(T),              intent(in) :: x(:)
107        type(T)                          :: a
108        a = self%drv%sum(x) / T(size(x))
109      end function average
110
111 end module averager_library
112
113
114 program main
115
116    ! dependencies on abstractions
117    use interfaces,       only: IAverager
118
119    ! dependencies on implementations
120    use simple_library,   only: SimpleSum
121    use pairwise_library, only: PairwiseSum
122    use averager_library, only: Averager
123
124    ! declarations
125    integer :: key
126    type(IAverager), allocatable :: avs, avp, av
127
128    ! use of parametrized structure constructors
129    avs = Averager{SimpleSum}()
130    avp = Averager{PairwiseSum{SimpleSum}}()
131
132    write(*,'(a)') 'Simple   sum average: 1'
133    write(*,'(a)') 'Pairwise sum average: 2'
134    write(*,'(a)',advance='no') 'Choose an averaging method: '
135    read(*,*) key
136
137    select case (key)
138    case (1)
139       ! simple sum case
140       av = avs
141    case (2)
142       ! pairwise sum case
143       av = avp
144    case default
145       stop 'Case not implemented!'
146    end select
147
148    print '(f8.5)', av%average([1.d0, 2.d0, 3.d0, 4.d0, 5.d0])
149    print '(i8)',   av%average([1, 2, 3, 4, 5])
150
151 end program main
```

Notice also, that in Listing 7.2, the av object of IAverager type that is initialized in the select

case statement of the main program still needs to make use of run-time polymorphism. This object cannot be made to employ compile-time polymorphism, as it is used within a statement that performs a run-time decision.

As a final remark we'd like to emphasize that a coding style as that given in Listing 7.1 should generally be preferred over that of Listing 7.2, as it leads to more readable code. The use of numerous generic parameters can quickly make code unreadable. We'd therefore recommend the default use of run-time polymorphism for managing dependencies on user implementation code, and the employment of generics with static dispatch for this latter task only 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.

Chapter 8

Comparison to J3's generics proposal for Fortran 202y

Feel free to add a corresponding code version here, since I am not sufficiently familiar with their approach.

Bibliography

[1] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–523, 12 1985.

[2] Ole-Johan Dahl. The birth of object orientation: the simula languages. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2004. Springer.

[3] M. Haveraaen, J. Järvi, and D. Rouson. Reflecting on generics in fortran. July 2019.

[4] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017.

[5] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.