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

Konstantinos Kifonidis, Ondrej Certik, Derick Carnazzola

July 29, 2023

Abstract

TBD.

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. [3]), 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 [3] 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/traits, and the mechanism of multiple interface inheritance that was introduced to express run-time polymorphism in Objective C. 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 [4]. 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 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 such 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.

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 section 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 Sect. 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 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 1: Monomorphic functional version of the array averaging example in Go.

```
1 package main
2
3 import "fmt"
4
5 func simple_sum(x []int) int {
6    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
// ..............

func main() {

   xi := []int{1,2,3,4,5}

   var key int

   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!")
   }

}
```

2.2 Polymorphic functional implementation

Listing 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 Sect. 3, even this dependency is eliminated.

A few remarks are in order for a better understanding of Listing 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 uses exclusively 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 intersecting 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: Polymorphic functional version of the array averaging example in Go.

```
package main

import "fmt"

type INumeric interface{ int | 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))
}
```

```
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 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 section, with objects replacing the closures (both being merely slightly different realizations of the same idea).

3 Survey of modern languages

In the present section we give implementations, in various modern languages, of encapsulated (i.e. OO) code versions of the test example. As in the functional example 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 mod-

ern languages. The survey also serves 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 section, 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 an ADT (i.e. to use multiple interface inheritance), even though it offers no explicit language statement for this purpose.

Instead, an ADT 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 a reference to an ADT object to be passed to its methods (by means of a separate parameter list, in front of a method's actual name). It is otherwise completely decoupled from the ADT's (i.e. the actual `struct`'s) definition. Hence, existing ADTs (even if their source code itself is inaccessible) can be made to retroactively implement new interfaces and methods, and hence to be used in new settings. Go, finally, makes it explicit in its syntax that interfaces 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 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. To allow methods to make use of such parameters one has to parameterize the structures and interfaces to which these methods or, respectively, their signatures belong.

3.1.1 Encapsulated version coded in Go

Listing 3 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: Encapsulated Go version of the array averaging example.

```go
package main

import "fmt"

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

type INumeric interface {
   int | float64
}

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

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

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

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() {

   var avi IAverager[int]
   var avf IAverager[float64]

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

   var key int

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

   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 `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. unparam-

eterized) `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".

3.2.1 Encapsulated version coded in Rust

Listing 4: Encapsulated Rust version of the array averaging example.

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

```

```rust
    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!");
        }
    }

    println!("{}", avi.average(&xi, xi.len() as i32));
    println!("{}", avf.average(&xf, xf.len() as f64));
}
```

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 (also) supports run-time 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. We will demonstrate this in the next section.

Swift generics support "strong concepts", and are thus fully type-checked, and their capabilities are on par with those of Go and Rust. For instance, all these languages allow for the parameterization of functions and structures (and in Swift's case also classes). Where Swift differs from both Go an Rust, is that it also allows for parameterized *methods*. Whereas both Go and Rust have their users parameterize entire interfaces/traits instead, in order to achieve the same functionality. This has some interesting consequences for the programmer, that will be discussed in detail below.

3.3.1 Encapsulated version coded in Swift

Listing 5 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.
- Implementation of a protocol (i.e. interface inheritance) is signified in Swift with the `:` operator appearing behind a structure's (or a class's) name. There is no separate `impl` block as in Rust, or implicit conformance to a protocol by implementing all its methods, as in Go. Functionality for a `struct` or `class` to implement protocols retroactively, is provided through an "extension" block, instead.
- 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 5: 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 program
```

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

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

var av : IAverager = avs

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

var key: Int?

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

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

print( av.average(x: xi) / xi.count )
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 OO 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 from the different implementations

Since the use of run-time polymorphism by means of interfaces is very similar in all the languages considered here, we will only make some final comments on the languages' compile-time, i.e. generics features.

3.4.1 Conclusions on 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 can make some code duplication in client code unavoidable. Nevertheless, generic Go code is quite easy to read and to understand. What sets Go apart from some of 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.

3.4.2 Conclusions on 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 (called "crates" in Rust), 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.

3.4.3 Conclusions on 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.

4 Fortran additions I: File extension with new defaults

5 Fortran additions II: Subtyping

5.1 Named abstract interfaces (traits)

5.2 Multiple interface inheritance

5.3 Enhanced type declarations

Named abstract interfaces 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 extended to accept named abstract interfaces.

5.4 Improved structure constructors

6 Fortran additions III: Generics

The new features that were discussed in the previous section 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 additional new features 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, there are some significant differences to types that are specified in the standard parameter list. For instance, the specification of a rank, or an `intent`, for meta-types like `T`, makes no sense. This is because the latter are always scalar input parameters. The syntax used above, that deviates slightly from how Fortran's usual function arguments are declared, therefore 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
```

```
         type(T),    intent(in) :: x(:)
         type(T)                :: s
      end function sum
   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:

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

```
module user_code

   use, intrinsic :: generic_constraints, only: INumeric

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

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:

```
   function sum{INumeric :: T}(self,x) result(s)
      type(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
```

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

```
   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
```

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:

```
   type, implements(ISum) :: PairwiseSum{ISum :: U}
      private
      type(U) :: other
   contains
      procedure :: sum
   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.

```
   Averager{SimpleSum}()
   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 we prefer to focus on such an extension in a future document, because it is an issue that is completely orthogonal to the genericity of types (although it would be handled in much the same way).

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 6 gives our Fortran version of the encapsulated form of the test example that corresponds to the code versions that were presented in Sect. 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 6: Proposed encapsulated Fortran version of the array averaging example.

```
module interfaces

   use, intrinsic :: iso_fortran_env, only: real64

   public :: INumeric, ISum, IAverager

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

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

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

   public :: Averager

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

contains

   function average{INumeric :: T}(self,x) result(a)
      type(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

   ! declarations
   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 6 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 gives a version of the encapsulated Fortran code that contains all the required changes, as compared to Listing 6, 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: 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
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{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
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, 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 6 should generally be preferred over that of Listing 7, 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.

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.

References

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

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