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

Konstantinos Kifonidis, Ondrej Certik, Derick Carnazzola

July 23, 2023

Abstract

.

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 **?**. Their work introduced into programming what is nowadays known as "virtual method" (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 (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, 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 not just a program's data, i.e. state, but also its procedures, 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 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 types has so far been the task of OO programming and OO design patterns. Martin 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.

Both these developments taken together have recently opened up the possibility to treat polymorphism 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 efficiency and flexibility.

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 cryptical compiler error messages. Surprisingly, Java, the language that truly popularized polymorphic interfaces in OOP, 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. 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. In the survey of programming languages presented in Sect. **??**, 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 in more complex situations that require OO techniques in order to hide state.

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 the 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 forms of assignment. One for "variables" that are intended to be immutable, and one for regular mutable variables.
- Go has array slices that most closely resemble those of Python's Numpy (which exclude the last element of an array slice).

Our algorithm for caclulating 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 some hard-wired dependencies on specific user-defined implementations. Namely, function `pairwise_sum` depends on the implementation of function `simple_sum`. While functions `simple_average` and `pairwise_average` depend on the implementations of functions `simple_sum`, and `pairwise_sum`, respectively. In addition to these dependencies on user-defined implementations, the entire program depends rigidly on the `int` data type to declare the arrays that it is operating on, as well as the results of the summations. Hence, this code cannot be used on arrays of any other data type than `int`s.

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
7     s = 0
8     for i := 0; i < len(x); i++ {
9       s += x[i]
10    }
11    return s
12 }
13
```

```
func pairwise_sum(x []int) int {
   N := 2
   if ( len(x) <= N ) {
     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

To be inserted here ...

3 Survey of modern languages

In the present section we will give implementations, in various modern languages, of encapsulated (i.e. OO) code versions of the test example. We will employ run-time polymorphism to manage the dependencies on user-defined abstract data types, and generics in order to manage the dependencies on language intrinsic abstract data types. This will serve to illustrate how both run-time and compile-time polymorphism can be typically used for dependency management in these modern languages. The survey will also serve 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 generally advocate to code problems, that are as simple as this test case, in such an encapsulated manner. However, in more complex cases, where state would be involved that would have to be hidden, an encapsulated programming style will be difficult to avoid. Hence the test problem will also have to stand in, in this section, for emulating such a more complex problem, that would benefit from an encapsulated coding style.

3.1 Go

Go supported run-time polymorphism through (polymorphic) "interfaces" since its inception. Since version 1.18, Go supports also compile-time polymorphism through generics that are bounded by generics constraints. Such constraints in Go are expressed through interfaces. Go therefore supports "strong concepts".

Go 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. Encapsulation in Go 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). Go allows the programmer to make such ADTs conform to interfaces, by implementing all the functions whose signatures are contained in an interface. There' no explicit statement to implement interfaces. Instead an ADT is implicitly assumed to implement an interface whenever it provides implementations of all the interface's functions.

3.1.1 Encapsulated version coded in Go

Listing 2 gives an encapsulated version of the test example coded in Go. The various versions of the `sum` functions have been encapsulated in ADTs as described above. The crucial elements for managing any dependencies on these user-defined ADTs in this encapsulated code version are the (polymorphic) interfaces `ISum` and `IAverager`. These are required in order to express polymorphism for the various objects of the Sum and Averager ADTs.

Listing 2: Encapsulated Go version of the array averaging example.

```
1  package main
2
3  import ("fmt")
4
5  // ...........
6  // Interfaces
7  // ...........
8
9  type INumeric interface {
10    int | float64
11 }
12
13 type ISum[T INumeric] interface {
14    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 {
   N := 2
   if ( len(x) <= N ) {
     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))
}
```

3.2 Rust

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 3: 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,
            }
        }
    }

    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 somewhat 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 the other modern languages, Swift supports both 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 has generics capabilities that are on par with those of Go and Rust. For instance, all these languages allow for the parametrization of structures (and in Swift's case also for classes). Where Swift differs from both Go an Rust, is that it allows for parameterized *methods*. Whereas both Go and Rust have their users parameterize entire interfaces/traits instead, in order to achieve the same functionality.

The Swift way of directly parametrizing methods but *not* protocols has some advantages from the user's perspective, because it almost completely obviates the need for manual instantiations of generics. In particular, there is no need (compared to the other languages) to always manually instantiate a different protocol for every different generic type parameter that the user wishes to employ with a generic method.

The Swift compiler is able to instantiate Swift generics largely automatically, through inspection of the argument types that are passed to functions and methods. 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 also 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.

3.3.1 Encapsulated version coded in Swift

Listing 4 gives an example of how the encapsulated version of the array averaging test problem can be programmed in Swift. See the following remarks in order to understand this code:

- 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 therefore supports "strong concepts", like the other languages considered here.
- 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.
- Interface (i.e. protocol) inheritance in Swift is signified with the `:` operator appearing behind a 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.
- The Swift version makes use of language supplied structure constructors.
- Array slices are not arrays themselves. So an explicit conversion using an `Array()` constructor is required in such cases.
- Function and method calls in Swift make use of (mandatory) 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).
- Notice, how in contrast to both Go and Rust, the Swift code doesn't instantiate any interfaces by making use of type parameters for them. E.g. there is only a single version of the `IAverager` interface, and hence there's only a need in the main program to declare a single variable, `av`, of that interface (`av` being polymorphic over different `Averager` types). This single object is then straightforwardly used with data of *both* the `Int` and `Float64` types.

Listing 4: 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 {
        let N = 2
        if ( x.count <= N ) {
            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) )
```

3.4 Conclusions from the different implementations

Since the use of run-time polymorphism by means of interfaces is very similar in all 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 for parametrization of structures, interfaces, and ordinary functions, but not methods. Nevertheless, generic Go code is quite easy to read and to understand. What sets Go apart from the other languages is its brilliant new notion to interpret interfaces as type sets, and 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 very pleasant.

3.4.2 Conclusions on Rust

Rust's basic model to implement generics is similar to Go's in that it allows for parametrization of structures, interfaces, and ordinary functions. Being the oldest of the "modern" languages, Rust has a few quirks which render its use for the management of all types of dependencies through polymorphism somewhat suboptimal when compared to the other languages considered here. Its basic deficiency is its C++-like philosophy to copiously rely on external dependencies, even for rather basic tasks, like initializing a generic type. The Rust version of our test case is therefore marred by some avoidable dependencies on external libraries (called "crates" in Rust), which partly defeat the purpose of programming in a polymorphic fashion, in order 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.

3.4.3 Conclusions on Swift

We consider Swift's basic model of implementing generics by allowing parametrized structures and methods (but not parametrized interfaces) to be both the easiest to read, and the easiest to use from a programmer's perspective. We hence consider it to be the most attractive model to base Fortran's basic generic capabilities on, provided that it can be implemented sufficiently easily, and that it be enhanced by Go's idea to interpret interfaces as type sets.

4 New additions to Fortran: Subtyping

4.1 Named abstract interfaces (traits)

4.2 Multiple interface inheritance

4.3 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.

4.4 Improved structure constructors

4.5 File extension with predefined new defaults

5 New additions to Fortran: 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 those additional new features that are exclusively required in order to further support compile-time polymorphism, i.e. generics.

5.1 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 metatype, 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 metatype, 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 metatypes 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.

5.2 Interfaces as type sets

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

For simple use cases, it should be optionally possible for the programmer to employ a shorthand notation like in the following modification of the example of an abstract interface declaration given previously in Sect. 5.1:

```
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
```

This would enable one to declare a type constraint for a generic type, without having to explicitly declare an abstract interface for it beforehand. The above notation would then define an abstract interface implicitly, to be used as a type constraint for type `T`. In this particular example, to admit only the default `integer`, or `real(real64)` types, for `T`.

5.3 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. The thus defined, language provided, interface `INumeric` could then be used from user code through a `use` statement like in the following example

```
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`.

5.4 Conversions to generic types

5.5 Generic type parameters for methods and procedures

As already mentioned above, LFortran's design of generics should follow Swift's, if possible, and allow generic type parameters to be used in both type-bound (i.e. method) and ordinary procedures. An implementation of the generic method `sum` of Sect. 5.1, 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
```

While 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
```

5.6 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 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 in order to declare a field variable of `type(U)` within `PairwiseSum`, that 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.

5.7 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` type) is not parameterized by any generic type parameters itself.

6 Proposed Fortran versions of the test example

6.1 Functional version

To be inserted here ...

6.2 Encapsulated version

Listing 5 gives our Fortran version of the encapsulated form of the test example that corresponds to the code versions that were presented in Sect. **??** 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 (i.e. class) 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. **??**, 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 5: Proposed encapsulated Fortran version of the array averaging example.

```
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)
```

```
         type(IAverager), intent(in) :: self
         type(T),         intent(in) :: x(:)
         type(T)                     :: a
      end function average
   end interface

end module interfaces


module simple_library

   use interfaces, only: ISum, INumeric

   public :: SimpleSum

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

contains

   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

end module simple_library


module pairwise_library

   use interfaces, only: ISum, INumeric

   public :: PairwiseSum

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

contains

   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
   type(IAverager), allocatable :: avs, avp, av

   ! use of enhanced structure constructors
```

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

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

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

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

end program main
```

The most important point to notice in Listing 5 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. Notice the use of run-time polymorphism in Listing 5 by the `av` object of `IAverager` type that is initialized in the `select case` statement. This object necessarily cannot employ compile-time polymorphism, as it is employed within a statement that performs a run-time decision.

6.3 Encapsulated (mostly) static version

Do we need to have this one as a further example?

7 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.