

Go Generics

Michel Casabianca
casa@sweetohm.net

Generics are the most important new feature of the *1.18* version of Go that was just released. I offer you a quick tour of this new feature in this article.

Before Go 1.18

It has always been possible to produce generic code with Go using **interface{}** type. For instance, you write a function that prints n times given value with:

```
package main

import "fmt"

func Repeat(something interface{}, times int) {
    for i := 0; i < times; i++ {
        fmt.Println(something)
    }
}

func main() {
    Repeat("Hello World!", 3)
    Repeat(42, 3)
}
```

On the Playground

This example is very simple because function `fmt.Println()` accepts any type. Before Go *1.18*, its signature was `func Println(a ...interface{}) (n int, err error)`.

Furthermore, one can define an argument type with a specific interface. For instance:

```
package main

import (
    "errors"
    "strconv"
)

type Failure int

func (t Failure) Error() string {
    return strconv.Itoa(int(t))
}

func PrintError(err error) {
```

```

        println("error: " + err.Error())
    }

    func main() {
        PrintError(errors.New("This is a test!"))
        PrintError(Failure(42))
    }

```

On the Playground

Type `error` is an interface that defines a single method `Error() string`. Thus you can send anything to function `PrintError()` provided it implements method `Error()`.

The beginning of troubles

Let's suppose we want to write a function that returns the maximum of given values. We could write, for integers, following code:

```

package main

func Max(x, y int) int {
    if x > y {
        return x
    }
    return y
}

func main() {
    println(Max(1, 2))
}

```

On the Playground

If we want to generalize this function for other types, interfaces are not of any help because no function can define comparison operators. Thus we have to **write this function for all types**! It would be possible to accept type `interface{}`, but we would have to do **type assertions** and this would not simplify things.

Generics to the rescue

Go 1.18 implements *Generics*. We can now add *type parameters* in function signature. To make our `Max()` function generic, we could write:

```

package main

func Max[N int | float64](x, y N) N {
    if x > y {
        return x
    }
}

```

```

        return y
    }

    func main() {
        println(Max(1, 2))
        println(Max(1.2, 2.1))
    }

```

On the Playground

This way, with type parameter `[N int | float64]`, we indicate that function parameters may be of type `int` or `float64`. Note that we can't mix types, thus call `Max(1, 2.0)` would not compile.

Interfaces strike back

With Go *1.18*, we can now define interfaces as a list of types. We could write example above as follows:

```

package main

type Number interface {
    int | int16 | int32 | int64 | float32 | float64
}

func Max[N Number](x, y N) N {
    if x > y {
        return x
    }
    return y
}

func main() {
    println(Max(1, 2))
    println(Max(1.2, 2.1))
}

```

On the Playground

Type aliases

If we define an alias for given type, we can include it in an interface with `~` character, as follows:

```

package main

type Number interface {
    ~int | ~int16 | ~int32 | ~int64 | ~float32 | ~float64
}

type Num int

```

```

func Max[N Number](x, y N) N {
    if x > y {
        return x
    }
    return y
}

func main() {
    println(Max(Num(1), Num(2)))
}

```

On the Playground

Thus `~int` includes type `int` but also all its aliases, and thus also `Num`.

Constraints

It can be very tedious to define your own interfaces with type lists. Package golang.org/x/exp/constraints provides following interfaces:

- **Signed** : `~int | ~int8 | ~int16 | ~int32 | ~int64`
- **Unsigned** : `~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr`
- **Integer** : `Signed | Unsigned`
- **Float** : `~float32 | ~float64`
- **Ordered** : `Integer | Float | ~string`
- **Complex** : `~complex64 | ~complex128`

We could now use constraint `constraints.Ordered` as follows:

```

package main

import "golang.org/x/exp/constraints"

func Max[N constraints.Ordered](x, y N) N {
    if x > y { return x }
    return y
}

func main() {
    println(Max("abc", "def"))
}

```

On the Playground

Furthermore, Go 1.18 defines two other constraints:

- **any** which is a new name for `interface{}`
- **comparable** for types that can be compared with `==` and `!=` operators

Instantiation

It is possible to pass type arguments while calling a generic function. For instance:

```
m := Max[int](1, 2)
```

Expression `Max[int]` is an *instantiation* of generic function `Max`. It defines types for parameters. We could write:

```
MaxFloat := Max[float64]  
m := MaxFloat(1.0, 2.0)
```

Function `MaxFloat` is now a non generic function that accepts only `float` arguments.

Types with type parameters

Let's say we want to compute the sum of all elements in a given list. With standard Go linked lists, we could write:

```
package main  
  
import "container/list"  
  
func main() {  
    list := &list.List{}  
    list.PushBack(1)  
    list.PushBack(2)  
    list.PushBack(3)  
    sum := 0  
    for e := list.Front(); e != nil; e = e.Next() {  
        sum += e.Value  
    }  
    println(sum)  
}
```

On the Playground

This doesn't compile because we can't add *interface{}* types, which is type for list elements value: `src/list.go:12:3: invalid operation: sum += e.Value (mismatched types int and any)`.

Using type `interface{}` or `any` is boring because we must cast values to use them. Of course there is a Generics based solution. Here is a minimalist implementation of linked list with Generics:

```
package main  
  
type Element[T any] struct {
```

```

        Next  *Element[T]
        Value T
    }

    type List[T any] struct {
        Front *Element[T]
        Last  *Element[T]
    }

    func (l *List[T]) PushBack(value T) {
        node := &Element[T]{
            Next: nil,
            Value: value,
        }
        if l.Front == nil {
            l.Front = node
            l.Last = node
        } else {
            l.Last.Next = node
            l.Last = node
        }
    }

    func main() {
        list := &List[int]{}
        list.PushBack(1)
        list.PushBack(2)
        list.PushBack(3)
        sum := 0
        for n := list.Front; n != nil; n = n.Next {
            sum += n.Value
        }
        println(sum)
    }

```

On the Playground

In this code we added type parameters to type definitions, as in `Element[T any]`. This notation indicates that we define type `Element` that contains type `T` that may be anything. We don't have to cast values to use them.

It is important to note that we set list type on instantiation:

```
list := &List[int]{}

```

This way we tell compiler that our list contains `int` and we now can use them as integers.

Type inference

We saw that we can set type parameters while calling a generic function with:

```
m := Max[int](1, 2)
```

In this case, compiler knows parameters types because we tell it. But when we write:

```
m := Max(1, 2)
```

In this case compiler **infers parameters type** of the generic function from argument's type while performing call. This inference type is called *function argument type inference*. Nevertheless, it is sometimes impossible to infer types for return values, as in this example:

```
func NewT[T any]() *T {  
    ...  
}
```

We must then help compiler *instantiating* function before calling it:

```
t := NewT[int]()
```

When to use generics?

First of all, **don't define constraints before writing code**. This might sound a good idea to anticipate writing constraints before your code, but this is useless.

Use case for Generics is when you have **duplicated code with many types**. In this case, Generics are a better alternative than using `interface{}` type for performance, memory usage and code simplicity. This is the case for data structures (such as *linked lists* or *binary trees* for instance).

Conclusion

Generics are the new big thing in Go 1.18 which is the most important release since Go was Open Sourced. Nevertheless, this feature was not heavily tested on production and thus **should be used with care**, and of course widely tested.



Generics Gopher