Go 1.18

Generics

Generics Basics

- Generic types
- Generic functions
- Generic constraints
- Type approximation

Generic types

Generic types allow us to declare one or more `type param` where these params can be used to define the type of an argument, slice, map or another generic type.

This means that we can now avoid code duplication!

We also no longer need `interface{}` where we lose valuable type information.

Now we can define a generic struct for paginated results of `any` type:

```go
type Collection[T any] []T

type PaginatedResults[T any] struct {
    Page int
    Total int
    Items Collection[T]
}
```

```go
result := PaginatedResults[string]{
    Page:  1,
    Total: 10,
    Items: []string{"1", "2"},
}
```

Generic Parameters

We can specify type parameters for a function.

For example we can implement a function than takes a slice of any type

and returns a copy of that slice:

```go
func Copy[T any](s []T) []T {
    results = make([]T, 0, len(s))
    for _, elem := range s {
        results = append(results, elem)
    }
    return results
}
```

```go
strs := []string{"a", "b", "c"}
cpStrs := Copy(strs)
```

```go
ints := []int{"a", "b", "c"}
cpInts := Copy(ints)
```

Any number of type parameters can be set specified,

In this example we use one generic type for the input slice “T”

and another for the output “P”

(You can give them any name) convention is single character uppercase

```
func Map[T any, P any](items []T, fn func(T) P) []P {
    results := make([]P, 0, len(items))
    for _, elem := range items {
        results = append(results, fn(elem))
    }
    return results
}
```

Generic functions can make use of type inference,

In this example we use the previous ‘Map’ function to convert ints to strings.

The type of `strs` can be inferred (`[]string`) based on the return type of `IntToString` being a `string`

```
func IntToString(i int) string { return fmt.Sprint(i) }

func main() {
    ints := []int{1, 2, 3, 4, 5}
    strs := Map(ints, IntToString)
}
```

Limitations

Generic Parameters can only be used in functions, you cannot define generic methods, only functions with no receiver.

```
func (r Receiver) [T any]NotAllowed(t T) {...
func [T any]Allowed(t T) {...
```

Generic Constraints

Generic constraints allow you to use `==`, `>`, `<` operators in your functions.

There are also constraints that allow arithmetic or bitwise operators:

- any (does not work with any operators)
- comparable (works with comparison operator: ==)
- constraints.Ordered (works with order operators: >, <, >= , etc...)
- constraints.Integer (works with bitwise/bitshift operators: &, |, ^, <<, >>)
- constraints.Float (works with arithmetic operators: +, -, *, /, %)
- constraints.Unsigned
- constraints.Signed
- constraints.Complex

constraints.Ordered allows us to restrict ‘T’ to any type that works with the ‘<’ operator. So we can define a Sort function that works with any of these types:

int, int64, float32, string, etc… or any derived types eg: “type MyInt int”

```
func SortOrdered[T constraints.Ordered](list []T) {
    sort.Slice(
        list,
        func(i, j int) bool {
            return list[i] < list[j]
        },
    )
}
```

Custom constraints

A constraint can be defined as any interface with a particular set of functions:

```
type Stringer interface {
    String() string
}

func ToString[T Stringer](items T[]) string {
    ...
}
```

You can define a custom constraint as a union of multiple types or constraints

```
type Decimal interface {
    float32 | float64
}

type Integer interface {
    int64 | int32
}

type Number interface {
    Decimal | Integer
}
```

Type approximation

`~int` matches type int but also types derived from int eg. `type Integer int`.

Note that types from the constraints package all use approximate types!

```
type Integer interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}

type Float interface {
    ~float32 | ~float64
}
```

We can define a generic Sum function that can return the sum of any slice. So long as the elements of the slice support the '+=' operator.

"constraints.Float" and "constraints.Integer" both support '+=':

```
type Number interface {
    constraints.Float | constraints.Integer
}

func Sum[T Number](list []T) T {
    var sum T
    for _, elem := range list {
        sum += elem
    }
    return sum
}
```

More Examples!