GopherCon, December 8, 2021

Generics!

Robert Griesemer

Google

gri@golang.org

Ian Lance Taylor

Google

iant@golang.org

New language features in Go 1.18

1. Type parameters for functions and types
2. Type sets defined by interfaces
3. Type inference

Type parameters

Type parameter lists

$$[P,\ Q\ \mathtt{constraint}_1,\ R\ \mathtt{constraint}_2]$$

Type parameter lists look like ordinary parameter lists with square brackets. It is customary to start type parameters with upper-case letters to emphasize that they are types.

min

```
func min(x, y float64) float64 {
    if x < y {
        return x
    }
    return y
}
```

A basic min function for float64 arguments.

Generic min

```
func min[T constraints.Ordered](x, y T) T {
    if x < y {
        return x
    }
    return y
}
```

The type parameter T, declared in a type parameter list, takes the place of float64.

Calling generic min

```
func min[T constraints.Ordered](x, y T) T {
    if x < y {
        return x
    }
    return y
}

m := min[int](2, 3)
```

To call the generic min function, we provide type and ordinary arguments.

Instantiation

1. Substitute type arguments for type parameters.

2. Check that type arguments implement their constraints.

Instantiation fails if step 2 fails.

Instantiating generic min

```
fmin := min[float64]

m := fmin(2.71, 3.14)
```

min[float64](2.71, 3.14) is evaluated as (min[float64])(2.71, 3.14).
Instantiation produces a non-generic function.

A generic binary tree

```
type Tree[T interface{}] struct{
    left, right *Tree[T]
    data        T
}

func (t *Tree[T]) Lookup(x T) *Tree[T]

var stringTree Tree[string]
```

Types can have type parameter lists, too.
Methods declare the respective type parameters with the receiver.

Type sets

The types of value parameters

```
func min(x, y float64) float64
```

Ordinary parameter lists have a type for each value parameter.
This type defines a set of values.

The types of type parameters

```
func min[T constraints.Ordered](x, y T) T
```

Type parameter lists also have a type for each type parameter.
This type defines a set of types. It is called the type constraint.

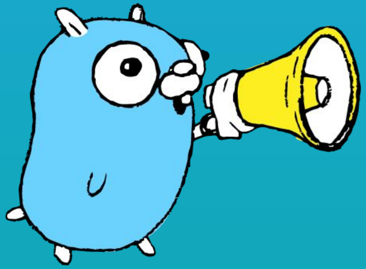
Type constraints

```
func min[T constraints.Ordered](x, y T) T {
    if x < y {
        return x
    }
    return y
}
```

constraints.Ordered has two functions:

1) Only types with orderable values can be passed as type arguments to T.

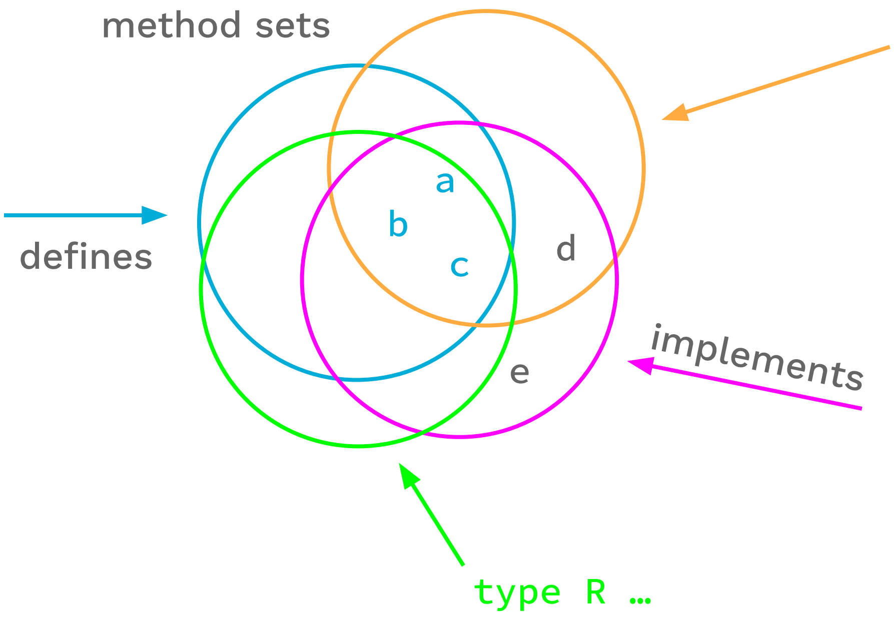
2) Values of type T can be used as operands for < in the function body.



Type constraints are interfaces.

Interfaces define method sets

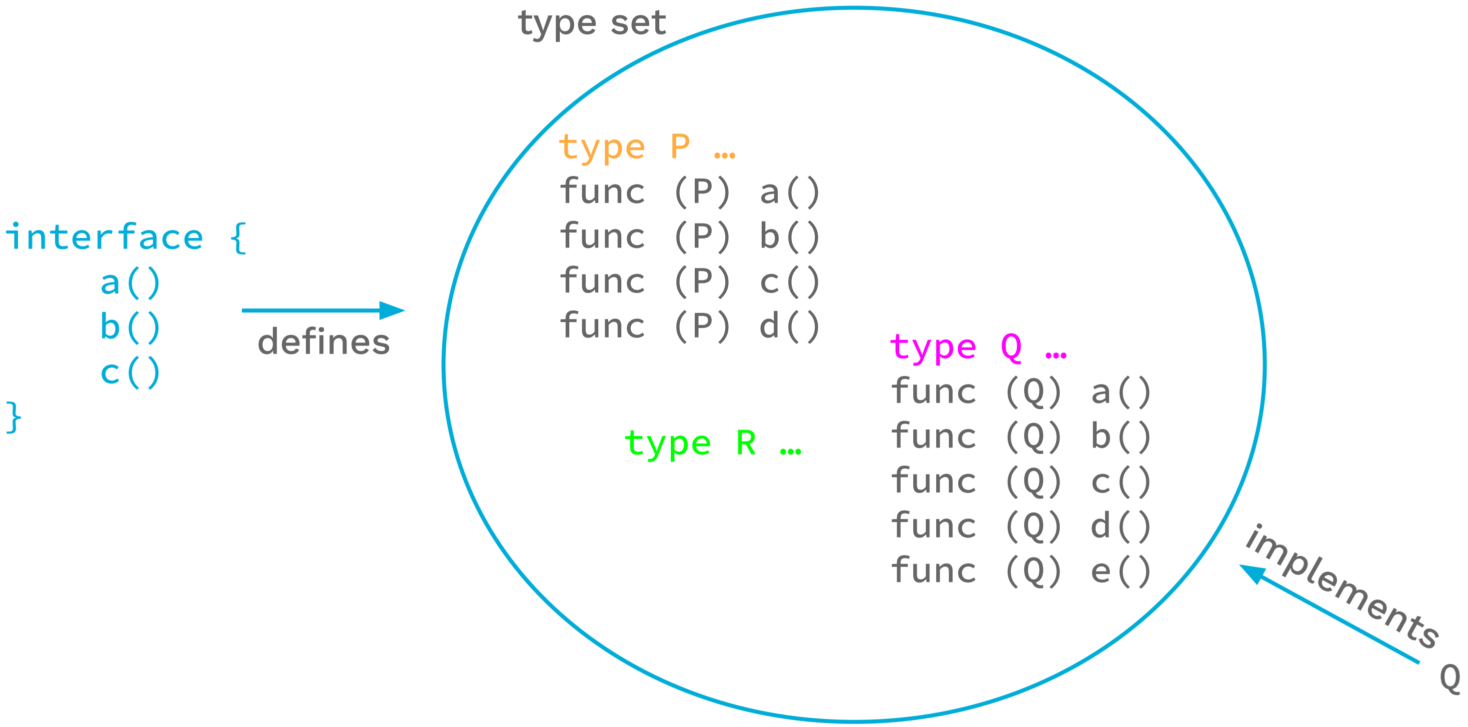
```
interface {
    a()
    b()
    c()
}
```




```
type P …
func (P) a()
func (P) b()
func (P) c()
func (P) d()
```

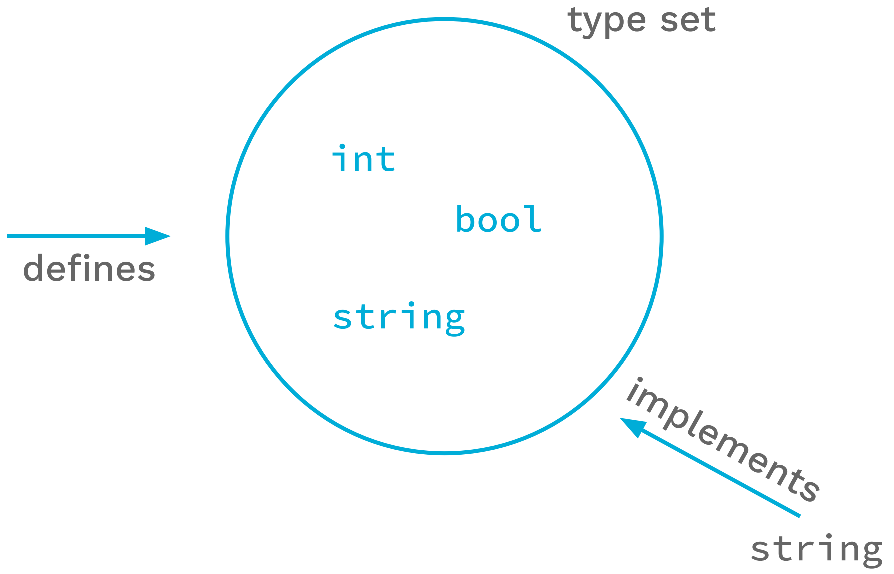
```
type Q …
func (Q) a()
func (Q) b()
func (Q) c()
func (Q) d()
func (Q) e()
```

Interfaces also define type sets



A type set with three types

```
interface {
    int|string|bool
}
```



constraints.Ordered

```
package constraints

type Ordered interface {
   Integer|Float|~string
}
```

Ordered defines the set of all integer, floating-point, and string types.
The < operator is supported by every type in this type set.

The ~ token

```
package constraints

type Ordered interface {
    Integer|Float|~string
}
```

~ is a new token added to Go.
~T means the set of all types with underlying type T.

The two functions of a type constraint

1. The type set of a constraint is the set of valid type arguments.

2. If all types in the constraint support an operation, that operation may be used with the respective type parameter.*

   (* some restrictions apply)

Constraint literals

```
[S interface{~[]E}, E interface{}]
```

It is common to write constraint literals "in line".

Constraint literals

```
[S interface{~[]E}, E interface{}]

[S ~[]E, E interface{}]
```

In constraint position, `interface{E}` may be written as E for type elements E.

Constraint literals

```
[S interface{~[]E}, E interface{}]

[S ~[]E, E interface{}]

[S ~[]E, E any]
```

The new predeclared identifier any is an alias for `interface{}`.

Type inference

Calling min without type inference

```
func min[T constraints.Ordered](x, y T) T

var a, b, m float64

m = min[float64](a, b)
```

Passing type arguments leads to more verbose code.

Calling min with type inference

```
func min[T constraints.Ordered](x, y T) T

var a, b, m float64

m = min[float64](a, b)

m = min(a, b)
```

The type argument float64 is inferred from the arguments a and b.

The details of type inference are complicated but using it is easy.

Constraint type inference

Scale a slice of any integer type

```
// This implementation has a problem,
// as we will see.
func Scale[E constraints.Integer](s []E, c E) []E {
    r := make([]E, len(s))
    for i, v := range s {
        r[i] = v * c
    }
    return r
}
```

A simple multi-dimensional point type

```
type Point []int32
```

A string representation of Point

```
func (p Point) String() string {
    // details not important…
}
```

Scale a Point

```
func ScaleAndPrint(p Point) {
    r := Scale(p, 2)
    fmt.Println(r.String()) // DOES NOT COMPILE
}

// Compiler error:
// r.String undefined
// (type []int32 has no field or method String)
```

Scale a slice of any integer type

```
// Now we see the problem with this implementation.
func Scale[E constraints.Integer](s []E, c E) []E {
    r := make([]E, len(s))
    for i, v := range s {
        r[i] = v * c
    }
    return r
}
```

Scale a slice of any integer type

```
// All changed code is highlighted.
func Scale[S ~[]E, E constraints.Integer](s S, sc E) S {
    r := make(S, len(s))
    for i, v := range s {
        r[i] = v * c
    }
    return r
}
```

Scale a Point

```
func ScaleAndPrint(p Point) {
    r := Scale(p, 2)
    fmt.Println(r.String())
}

// Why don't we have to write
//    r := Scale[Point, int32](p, 2)
// ?
```

Details at https://go.dev/s/generics-proposal

Constraint Type Inference

Deduce type arguments from type parameter constraints

Scale a slice of any integer type

```
// Same definition as before.
func Scale[S ~[]E, E constraints.Integer](s S, sc E) S {
    r := make(S, len(s))
    for i, v := range s {
        r[i] = v * c
    }
    return r
}
```

Infer E from S

```
type Point []int32

func ScaleAndPrint(p Point) {
    r := Scale(p, 2) // calls Scale[Point, int32]
    …
}

func Scale[S ~[]E, E constraints.Integer](s S, sc E) S {
```

“

IGOR STRAVINSKY

Whatever diminishes constraint diminishes strength.

When to use generics

When to use generics

Writing Go

Write code, don't design types.

When are type parameters useful?

When are type parameters useful?

- Functions that work on slices, maps, and channels of any element type.

When are type parameters useful?

- Functions that work on slices, maps, and channels of any element type.
- General purpose data structures.

Tree is a generic binary tree.

```
type Tree[T any] struct {
    cmp  func(T, T) int
    root *node[T]
}

type node[T any] struct {
    left, right *node[T]
    data        T
}
```

The value is stored directly in each leaf as a T, not an interface{}.

The find method returns where val goes in bt.

```
func (bt *Tree[T]) find(val T) **node[T] {
    pl := &bt.root
    for *pl != nil {
        switch cmp := bt.cmp(val, (*pl).data); {
        case cmp < 0: pl = &(*pl).left
        case cmp > 0: pl = &(*pl).right
        default: return pl
        } }
    return pl
}
```

When are type parameters useful?

- Functions that work on slices, maps, and channels of any element type.
- General purpose data structures.
  - When operating on type parameters, prefer functions to methods.

When are type parameters useful?

- Functions that work on slices, maps, and channels of any element type.
- General purpose data structures.
  - When operating on type parameters, prefer functions to methods.
- When a method looks the same for all types.

SliceFn implements sort.Interface for any slice type.

```go
type SliceFn[T any] struct {
    s   []T
    cmp func(T, T) bool
}

func (s SliceFn[T]) Len() int { return len(s.s) }
func (s SliceFn[T]) Swap(i, j int) {
    s.s[i], s.s[j] = s.s[j], s.s[i] }
func (s SliceFn[T]) Less(i, j int) bool {
    return s.cmp(s.s[i], s.s[j]) }
```

SortFn uses SliceFn to sort a slice using a function.

```
func SortFn[T any](s []T, cmp func(T, T) bool) {
    sort.Sort(SliceFn[T]{s, cmp})
}

// This is similar to sort.Slice, but the comparison
// function uses values rather than indexes.
```

When not to use generics

When are type parameters not useful?

When are type parameters not useful?

- When just calling a method on the type argument.

Do not write code like this!

```
// good
func ReadFour(r io.Reader) ([]byte, error)

// bad
func ReadFour[T io.Reader](r T) ([]byte, error)
```

The function should be written without the type parameter.

When are type parameters not useful?

- When just calling a method on the type argument.
- When the implementation of a common method is different for each type.

When are type parameters not useful?

- When just calling a method on the type argument.
- When the implementation of a common method is different for each type.
- When the operation is different for each type, even without a method.

One simple guideline

When are type parameters useful?

- Avoid boilerplate.

When are type parameters useful?

- Avoid boilerplate.
  - Corollary: don’t use type parameters prematurely; wait until you are about to write boilerplate code.

“

WALLACE D. WATTLES

It is essential to have good tools, but it is also essential that the tools should be used in the right way.

Thanks