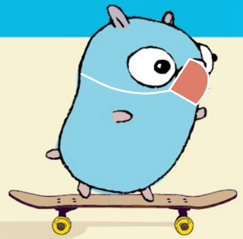
GopherCon/GoVirCon 2020

Typing [Generic] Go



Robert Griesemer
Google
gri@golang.org

Ian Lance Taylor

Featherweight Go

Philip Wadler et al.

https://arxiv.org/abs/2005.11710 (paper)

https://zenodo.org/record/4048298 (artifacts)

(or google "featherweight go paper")

Go community

Ayke van Laethem◆Bill Kennedy◆Chris Hines◆Daniel Martí
Dave Cheney◆Elena Morozova◆Jaana Dogan◆Jon Bodner
Josh Bleecher-Snyder◆Kevin Gillette◆Mitchell Hashimoto
Roger Peppe◆Ronna Steinberg

and all the Gophers that have provided feedback over the years.

Type parameters draft design

Primary difference from last year's design:

Instead of contracts we now use interfaces to express constraints.

https://blog.golang.org/generics-next-step
https://go.googlesource.com/proposal/+/refs/heads/master/design/go2draft-type-parameters.md
(or google "type parameters draft design")

Type parameters draft design

Primary new language features:

1. **Type parameters**
   Mechanism to parameterize a type or function by types.

2. **Constraints**
   Mechanism to express requirements on type parameters.

3. **Type inference** (optional)

An ordinary parameter list

parameter names
(variables)

```
(x, y aType, z anotherType)
```

parameter types
(types)

A type parameter list

type parameter names
(types)

```
[P, Q aConstraint, R anotherConstraint]
```

Convention: Type parameter
names are capitalized.

constraints
(meta-types)

Sorting in Go

```
func Sort(data Interface)

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

What we really want

```
func Sort(list []Elem)
```

// use

```
Sort(myList)
```

Type parameters to the rescue

```
func Sort[Elem ?](list []Elem)
```

Type parameter list

Constraints

- A constraint specifies the requirements which a type argument must satisfy.

- In generic Go, constraints are interfaces.

A type argument is valid if it implements its constraint.

Generic Sort

```
func Sort[Elem interface{ Less(y Elem) bool }](list []Elem)
```

The constraint is an interface, but the actual type argument can be any type that implements that interface.

Generic Sort

```
func Sort[Elem interface{ Less(y Elem) bool }](list []Elem)
```

constraint refers to type parameter

Declaration and scope of type parameters

type parameter declaration

```
func Sort[Elem interface{ Less(y Elem) bool }](list []Elem) {
    ...
}
```

type parameter scope

type parameter uses

The scope of a type parameter starts at the opening "[" and ends at the end of the generic type or function declaration.

Using generic Sort

```
func Sort[Elem interface{ Less(y Elem) bool }](list []Elem)

type book struct{…}
func (x book) Less(y book) bool {…}

var bookshelf []book
…
Sort[book](bookshelf) // generic function call
```

Type-checking the Sort call: Instantiation

```
Sort[book]                                        (bookshelf)
```

pass type argument

```
Sort[Elem interface{ Less(y Elem) bool }] (list []Elem)
```

substitute book for `Elem`

```
Sort[book interface{ Less(y book) bool }] (list []book)
```

verify that book satisfies the type parameter constraint

```
#Sort[book]                                       (list []book)
```

# indicates **instantiated** function "`Sort[book]`"

Type-checking the Sort call: Invocation

```
(bookshelf)
```

pass ordinary argument

```
#Sort[book]                (list []book)
```

verify that bookshelf can be assigned to []book

```
#Sort[book]                (bookshelf)
```

Type-checking a generic call

1) Instantiation (new)

- replace type parameters with type arguments in entire signature
- verify that each type argument satisfies its constraint

Then, using the instantiated signature:

2) Invocation (as usual)

- verify that each ordinary argument can be assigned to its parameter

Separating instantiation from invocation

```
booksort := Sort[book] // == #Sort[book]
booksort(bookshelf)
```

Types can be generic, too

```
… interface{ Less(y Elem) bool } …

type Lesser[T any] interface{
    Less(y T) bool
}
```

any stands for "no constraint"
(same as "interface{}")

Declaration and scope of type parameters

type parameter declaration

```
type Lesser[T any] interface{
    Less(y T) bool
}
```

type parameter use

Sort, decomposed

```
type Lesser[T any] interface{
    Less(y T) bool
}

func Sort[Elem Lesser[Elem]](list []Elem)
```

instantiation of Lesser

A generic function or type must be instantiated before it can be used.

Sort internals

```
func Sort[Elem interface{ Less(y Elem) bool }](list []Elem)
{
    …
    var i, j int
    …
    if list[i].Less(List[j]) {
        …
    }
    …
}
```

- type of `list[i]`, `list[j]` is `Elem`
- `Elem` is NOT an interface type!

A type parameter is a real type.
It is not an interface type.

(But it may be instantiated with an interface type.)

Are we there yet?

```
Sort[book](bookshelf)
```

$$\Big\downarrow ?$$

```
Sort(bookshelf)
```

Argument type inference

```
                        bookshelf
                            | underlying type
                            v
    type unification    []book

func Sort[Elem ...]([]Elem)
                            | inference
                            v
                       => Elem == book
```

Type-checking a generic call (refined)

1. If no type arguments are provided:
   Use argument type inference to infer type arguments.

2. Type-check generic function instantiation.

3. Type-check instantiated function invocation.

With argument type inference most generic calls look like regular calls.

What is missing?

So far, constraints can only describe method requirements.

For instance, `Sort([]int{1, 2, 3})` won't work:

`int` does not implement the `Elem` constraint (no `Less` method). Could do:

```
type myInt int

func (x myInt) Less(y myInt) bool { return x < y }
```

but that is cumbersome.

Type lists

A constraint interface may have a list of types (besides methods):

```
type Float interface {
    type float32, float64
}

// Sin computes sin(x) for x of type float32 or float64.
func Sin[T Float](x T) T
```

(For a generalization beyond generics, see issue #41716.)

Satisfying a type list

An argument type satisfies a constraint with a type list if

1) The argument type implements the methods of the constraint

2) The argument type or its underlying type is found in the type list.

As usual, the satisfaction check happens **after** substitution.

A generic min function

```
func min[T Ordered](x, y T) T …

type Ordered interface {
    type int, int8, int16, …, uint, uint8, uint16, …,
    float32, float64, string
}
```

min internals

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

- type of x, y is T, constrained by Ordered
- "<" is permitted because each type in the type list of Ordered supports "<"

Different type parameters are different types

```
func invalid[Tx, Ty Ordered](x Tx, y Ty) Tx {
    …
    if x < y { … // INVALID
    …
}
```

- x is of type Tx, y is of type Ty
- Tx and Ty are different types
- "<" requires that both operands have the same type

Example: Combining []byte and string operations

```
type Bytes interface {
    type []byte, string
}

// Index returns the index of the first instance of sep
// in s, or -1 if sep is not present in s.
func Index[bytes Bytes](s, sep bytes) int
```

Example: Relationships between type parameters

```
type Pointer[T any] interface {
    type *T
}
```

The type argument for PT must be a pointer to the type argument for T.

```
func f[T any, PT Pointer[T]](x T)
```

or with inlined constraint:

```
func foo[T any, PT interface{type *T}](x T)
```

Arrived (mostly)

```
func BasicSort[Elem Ordered](list []Elem)

func Sort[Elem Lesser[Elem]](list []Elem)

type Lesser[Elem any] interface {
    Less(Elem) Elem
}
```

Summary

Declarations

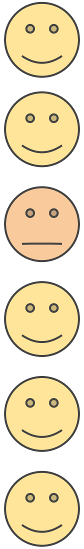
- Type parameter lists are like ordinary parameter lists with "[" "]".
- Function and type declarations may have type parameter lists.
- Type parameters are constrained by interfaces.

Use

- Generic functions and types must be instantiated when used.
- Type inference (if applicable) makes function instantiation implicit.
- Instantiation is valid if the type arguments satisfy their constraints.

How happy are we with this design?

- Type parameters
- Interfaces as constraints
- Type lists in interfaces
- Syntax ("[" "]" brackets)
- Type inference
- Fit with rest of Go



(Design as of early Oct, 2020)

Closing thoughts

With great power comes great responsibility

- Type parameters ("generics") are a new tool in the toolset of Go.
- Orthogonal to the rest of the language.
- Orthogonality opens a new dimension of coding styles.

Genericity introduces abstraction, and needless abstraction introduces complexity. Move cautiously!

Examples (1)

```
// ReadAll reads from r until an error or EOF and
// returns the data it read.
func ReadAll(r io.Reader) ([]byte, error)
```

vs

```
func ReadAll[reader io.Reader](r reader) ([]byte, error)
```

=> Generic version doesn't solve a real problem.

Examples (2)

```
// Drain drains any elements remaining on the channel.
func Drain[T any](c <-chan T)

// Merge merges two channels of some element type into
// a single channel.
func Merge[T any](c1, c2 <-chan T) <-chan T
```

=> Type parameters enable code that is not possible otherwise.

When to use generics

1. Improved static type safety.

2. More efficient memory use.

3. (Significantly) better performance.

type-checked

Generics are ~~glorified~~ macros.

Think twice before using a macro.

Next steps

The Go team is actively pursuing a real implementation (in a branch) so we can iron out any outstanding open problems.

We continue to look for feedback:

- Can you write the code you expect to write with generics?
- Do you run into unforeseen problems?

How to play:

https://go2goplay.golang.org/ (playground)

`git checkout dev.go2go` (go2go command)

Thank you!