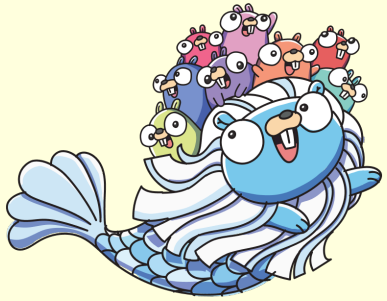
Why we can’t have nice things:

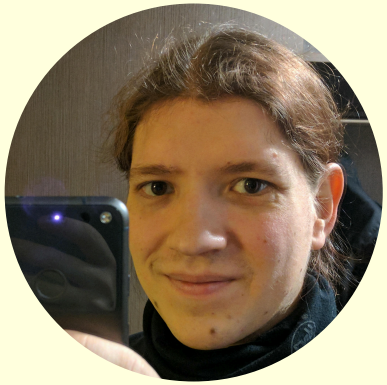
Generic methods



Axel Wagner

https://blog.merovius.de/

@Merovius@chaos.social



2025-01-24

Examples

math/rand/v2

func N

```
func N[Int intType](n Int) Int
```

N returns a pseudo-random number in the half-open interval [0,n) from the default Source. The type parameter Int can be any integer type. It panics if n <= 0.

math/rand/v2

func N

```
func N[Int intType](n Int) Int
```

N returns a pseudo-random number in the half-open interval [0,n) from the default Source. The type parameter Int can be any integer type. It panics if n <= 0.

```
// Sleep for a random duration up to 1s.
func RandomSleep() {
  time.Sleep(rand.N(time.Second))
}
```

math/rand/v2

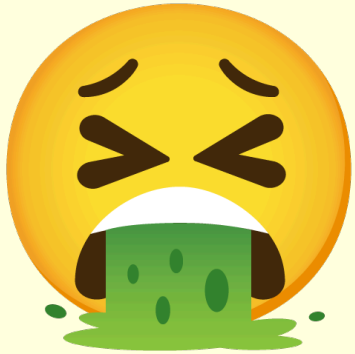
```
type Rand
    func New(src Source) *Rand
    func (r *Rand) ExpFloat64() float64
    func (r *Rand) Float32() float32
    func (r *Rand) Float64() float64
    func (r *Rand) Int() int
    func (r *Rand) Int32() int32
    func (r *Rand) Int32N(n int32) int32
    func (r *Rand) Int64() int64
    func (r *Rand) Int64N(n int64) int64
    func (r *Rand) IntN(n int) int
    func (r *Rand) NormFloat64() float64
    func (r *Rand) Perm(n int) []int
    func (r *Rand) Shuffle(n int, swap func(i, j int))
    func (r *Rand) Uint() uint
    func (r *Rand) Uint32() uint32
    func (r *Rand) Uint32N(n uint32) uint32
    func (r *Rand) Uint64() uint64
    func (r *Rand) Uint64N(n uint64) uint64
    func (r *Rand) UintN(n uint) uint
```

math/rand/v2

```
// Sleep for a random duration up to 1s.
func RandomSleep(r *rand.Rand) {
  time.Sleep(time.Duration(r.IntN(int(time.Second))))
}
```

math/rand/v2

```go
// Sleep for a random duration up to 1s.
func RandomSleep(r *rand.Rand) {
  time.Sleep(time.Duration(r.IntN(int(time.Second))))
}
```



Cache

```
type Cache interface {
  // Get the value associated with key. On a cache miss, calls
  // make to populate the cache.
  Get(key string, make func(key string) any) any
}

func DoThing(c Cache, …) Stuff {
  // …
  v := c.Get("foo", calculate).(internalState)
  // …
}

func calculate(key string) any { /* … */ }
```

Cache

```
type Cache[Key, Value any] interface {
  // Get the value associated with k. On a cache miss, calls
  // make to populate the cache.
  Get(k Key, make func(Key) Value) Value
}

func DoThing(c Cache[string, internalState], …) Stuff {
  // …
  v := c.Get("foo", calculate)
  // …
}

func calculate(key string) internalState { /* … */ }
```

Cache

```
type Cache interface {
  // Get the value associated with k. On a cache miss, calls
  // make to populate the cache.
  Get[Key, Value any](k Key, make func(Key) Value) Value
}

func DoThing(c Cache, …) Stuff {
  // …
  v := c.Get("foo", calculate)
  // …
}

func calculate(key string) internalState { /* … */ }
```

iter.Seq[E]

```
package xiter

func Map[A, B any](s iter.Seq[A], f func(A) B) iter.Seq[B]
func Filter[A any](s iter.Seq[A], f func(A) bool) iter.Seq[A]
func Reduce[A any](s iter.Seq[A], f func(A, A) A) A
// …
```

iter.Seq[E]

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := xiter.Reduce(
    xiter.Map(
      ListShapes(),
      Shape.Area,
    ),
    math.Max,
  )
  fmt.Printf("The largest shape has area %v", m)
}
```

iter.Seq[E]

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := xiter.Reduce(
    math.Max,
    xiter.Map(
      Shape.Area,
      ListShapes(),
    ),
  )
  fmt.Printf("The largest shape has area %v", m)
}
```

iter.Seq[E]

```
package iter

type Seq[A any] func(yield func(A) bool)

func (s Seq[A]) Map[B any](f func(A) B) Seq[B]
func (s Seq[A]) Filter(f func(A) bool) Seq[A]
func (s Seq[A]) Reduce(f func(A, A) A) A
```

iter.Seq[E]

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := ListShapes().
    Map(Shape.Area).
    Reduce(math.Max)
  fmt.Printf("The largest shape has area %v", m)
}
```

iter.Seq[E]

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := ListShapes().
    Map(Shape.Area).
    Reduce(math.Max)
  fmt.Printf("The largest shape has area %v", m)
}
```

iter.Seq[E]

```
package iter

type Seq[A any] func(yield func(A) bool)

// iter.go:5:21: method must have no type parameters
func (s Seq[A]) Map[B any](f func(A) B) Seq[B]
func (s Seq[A]) Filter(f func(A) bool) Seq[A]
func (s Seq[A]) Reduce(f func(A, A) A) A
```

proposal: spec: allow type parameters in methods #49085

Open **mariomac** opened this issue on Oct 20, 2021 · 333 comments

mariomac commented on Oct 20, 2021 • edited by ianlancetaylor

According to the Type parameters proposal, it is not allowed to define type parameters in methods.

This limitation prevents to define functional-like stream processing primitives, e.g.:

```
func (si *stream[IN]) Map[OUT any](f func(IN) OUT) stream[OUT]
```

While I agree that these functional streams might be unefficient and Go is not designed to cover this kind of use cases, I would like to emphasize that Go adoption in stream processing pipelines (e.g. Kafka) is a fact. Allowing type parameters in methods would allow constructing DSLs that would greatly simplify some existing use cases.

Other potential use cases that would benefit from type paremeters in methods:

- DSLs for testing: `Assert(actual).ToBe(expected)`
- DSLs for mocking: `On(obj.Sum).WithArgs(7, 8).ThenReturn(15)`

Edited by **@ianlancetaylor** to add: for a summary of why this has not been approved, please see https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#no-parameterized-methods .

👍 713 👎 7 🎉 6 ❤️ 14 🚀 13 👀 20

proposal: spec: allow type parameters in methods #49085

Open **mariomac** opened this issue on Oct 20, 2021 · 333 comments

mariomac commented on Oct 20, 2021 • edited by ianlancetaylor

According to the Type parameters proposal, it is not allowed to define type parameters in methods.

This limitation prevents to define functional-like stream processing primitives, e.g.:

```
func (si *stream[IN]) Map[OUT any](f func(IN) OUT) stream[OUT]
```

While I agree that these functional streams might be unefficient and Go is not designed to cover this kind of use cases, I would like to emphasize that Go adoption in stream processing pipelines (e.g. Kafka) is a fact. Allowing type parameters in methods would allow constructing DSLs that would greatly simplify some existing use cases.

Other potential use cases that would benefit from type paremeters in methods:

- DSLs for testing: `Assert(actual).ToBe(expected)`
- DSLs for mocking: `On(obj.Sum).WithArgs(7, 8).ThenReturn(15)`

Edi…**lancetaylor** to add: for a summary of why this has not been approved, please see https://go.g…rce.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#no-parameterized-methods .

713 7 6 14 13 20

proposal: spec: allow type parameters in methods #49085

Open **mariomac** opened this issue on Oct 20, 2021 · 333 comments

mariomac commented on Oct 20, 2021 • edited by ianlancetaylor

According to the Type paran posal, it is not allowed to define type parameters in methods.

This limitation prevents to de unctional-like stream processing primitives, e.g.:

```
func (si *stream[IN]) Map[OUT any](f func(IN) OUT) stream[OUT]
```

While I agree that these functional streams might be unefficient and Go is not designed to cover this kind of use cases, I would like to emphasize that Go adoption in stream processing pipelines (e.g. Kafka) is a fact. Allowing type parameters in methods would allow constructing DSLs that would greatly simplify some existing use cases.

Other potential use cases that would benefit from type paremeters in methods:

- DSLs for testing: `Assert(actual).ToBe(expected)`
- DSLs for mocking: `On(obj.Sum).WithArgs(7, 8).ThenReturn(15)`

Edi nlancetaylor to add: for a summary of why this has not been approved, please see https://go.g rce.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#no-parameterized-methods .

713 7 6 14 13 20

proposal: spec: allow type parameters in methods #49085

Open **mariomac** opened this issue on Oct 20, 2021 · 333 comments

mariomac commented on Oct 20, 2021 • edited by ianlancetaylor

According to the Type paramet[illegible]posal, it is not allowed to define type parameters in methods.

This limitation prevents to de[illegible]unctional-like stream processing primitives, e.g.:

```
func (si *stream[IN]) Map[OUT any](f func(IN) OUT) stream[OUT]
```

While I agree that these functional streams might be unefficient and Go is not designed to cover this kind of use cases, I would like to emphasize that Go adoption in stream processing pipelines (e.g. Kafka) is a fact. Allowing type parameters in methods would allow constructing DSLs that would greatly simplify some existing use cases.

Other potential use cases that would benefit from type paremeters in methods:

- DSLs for testing: `Assert(actual).ToBe(expected)`
- DSLs for mocking: `On(obj.Sum).WithArgs(7, 8).ThenReturn(15)`

Edi[illegible]**lancetaylor** to add: for a summary of why this has not been approved, please see https://go.g[illegible]rce.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#no-parameterized-methods .

713 7 6 14 13 20

The generic dilemma

The generic dilemma

> “The generic dilemma is this: do you want slow programmers, slow compilers and bloated binaries, or slow execution times?”
>
> — Russ Cox, 2009-12-03

The generic dilemma

> “The generic dilemma is this: do you want slow programmers, slow compilers and bloated binaries, or slow execution times?”
>
> — Russ Cox, 2009-12-03

- Slow programmers: No generics
- Slow compilers, bloated binaries: Compile-time expansion (e.g. C++)
- Slow execution time: Runtime boxing (e.g. Java)

Go’s answer

> “**We believe that this design permits different implementation choices.** […] In other words, this design permits people to stop choosing slow programmers, and permits the implementation to decide between slow compilers […] or slow execution times […].”
>
> — Type Parameters Proposal

Go's answer

> "**We believe that this design permits different implementation choices.** […] In other words, this design permits people to stop choosing slow programmers, and permits the implementation to decide between slow compilers […] or slow execution times […]."
>
> — Type Parameters Proposal

Implementation strategy is an optimization choice.

Go's answer

> "**We believe that this design permits different implementation choices.** […] In other words, this design permits people to stop choosing slow programmers, and permits the implementation to decide between slow compilers […] or slow execution times […]."
>
> — Type Parameters Proposal

Implementation strategy is an optimization choice.

Importantly: "Different implementations" can mean "different versions of the same compiler".

First class generic functions

First class generic functions

```
var f func[A any](A) int
```

First class generic functions

```
func DoThing(f func[A any](A) int) int {


}
```

First class generic functions

```
func DoThing(f func[A any](A) int) int {
  x := f[string]("Hello, world")


}
```

First class generic functions

```
func DoThing(f func[A any](A) int) int {
  x := f[string]("Hello, world")
  y := f[int](42)
  return x+y
}
```

First class generic functions

```
// Syntax error: function type must have no type parameters
func DoThing(f func[A any](A) int) int {
  x := f[string]("Hello, world")
  y := f[int](42)
  return x+y
}
```

Implementation

Implementation

```
type _table struct {
  _string func(string) int
  _int    func(int) int
}
func DoThing(f _table) int {
  x := f._string("Hello, world")
  y := f._int(42)
  return x+y
}
```

Implementation

```
type _table struct {
  _string func(string) int
  _int    func(int) int
}
func DoThing(f _table) int {
  x := f._string("Hello, world")
  y := f._int(42)
  return x+y
}

func F[T any](x T) int { /* … */ }
func main() {
  fmt.Println(DoThing(F))
}
```

Implementation

```
type _table struct {
  _string func(string) int
  _int    func(int) int
}
func DoThing(f _table) int {
  x := f._string("Hello, world")
  y := f._int(42)
  return x+y
}

func F[T any](x T) int { /* … */ }
func main() {
  fmt.Println(DoThing(_table{F[string], F[int]}))
}
```

Implementation

```
var ch = make(chan func[T any](T), 1)

func Thing1() { f := <-ch; f[string]("Hello, world") }
func Thing2() { f := <-ch; f[int](42) }

func F[T any](v T) { /* … */ }
func main() {
  ch <- F
  switch rand.N(2) {
  case 0: Thing1()
  case 1: Thing2()
  }
}
```

Implementation

```
package pkgA

var Chan = make(chan func[T any](T), 1)

package pkgB

func Thing1() { f := <-pkgA.Chan; f[string]("Hello, world") }

package pkgC

func Thing2() { f := <-pkgA.Chan; f[int](42) }
```

Implementation

```
func Add[T intType](a, b T) T { return a + b }
func Sub[T intType](a, b T) T { return a - b }
func Mul[T intType](a, b T) T { return a * b }
func Div[T intType](a, b T) T { return a % b }
func Max[T intType](a, b T) T { return max(a, b) }
func Min[T intType](a, b T) T { return min(a, b) }
func UseInt(fs ...func[T intType](T, T) T) {
  for _, f := range fs { f[int](1,2) }
}
func UseUint(fs ...func[T intType](T, T) T) {
  for _, f := range fs { f[uint](1,2) }
}
func main() {
  UseInt(Add, Mul, Max)
  UseUint(Sub, Div, Min)
}
```

Implementation

Why is this not a problem right now?

Implementation

Why is this not a problem right now?

```
func DoThing() {
  // generic functions must be fully instantiated before use!
  sompkg.SomeFunction[int, string](42, "Hello, world!")
}
```

Implementation

Why is this not a problem right now?

```
func DoThing() {
  // generic functions must be fully instantiated before use!
  sompkg.SomeFunction[int, string](42, "Hello, world!")
}
```

Statically couples the type arguments with a reference to the body!

Methods

Generic methods in interfaces

If we had generic methods, they should be usable in interfaces:

```
type Caller interface{
  Call[T any](T) int
}
```

Generic methods in interfaces

If we had generic methods, they should be usable in interfaces:

```
type Caller interface{
  Call[T any](T) int
}
```

But:

```
func F(c Caller) int {
  x := c.Call[string]("Hello, world")
  y := c.Call[int](42)
  return x+y
}
```

Generic methods in interfaces

If we had generic methods, they should be usable in interfaces:

```
type Caller interface{
  Call[T any](T) int
}
```

But:

```
func F(c Caller) int {
  x := c.Call[string]("Hello, world")
  y := c.Call[int](42)
  return x+y
}
```

```
func F(f func[A any](A) int) int {
  x := f[string]("Hello, world")
  y := f[int](42)
  return x+y
}
```

Generic methods in interfaces

If we had generic methods, they should be usable in interfaces:

```
type Caller interface{
  Call[T any](T) int
}
```

But:

```
func F(c Caller) int {
  x := c.Call[string]("Hello, world")
  y := c.Call[int](42)
  return x+y
}
```

```
func F(f func[A any](A) int) int {
  x := f[string]("Hello, world")
  y := f[int](42)
  return x+y
}
```

This is just first class generic functions with extra steps!

Generic methods in interfaces

If we had generic methods, they should be usable in interfaces:

```
type Caller interface{
  Call[T any](T) int
}
```

But:

```
func F(c Caller) int {
  x := c.Call[string]("Hello, world")
  y := c.Call[int](42)
  return x+y
}
```

```
func F(f func[A any](A) int) int {
  x := f[string]("Hello, world")
  y := f[int](42)
  return x+y
}
```

This is just first class generic functions with extra steps!

Corollary: No generic methods in interfaces.

Satisfying interfaces

If we had generic methods, we probably want them to implement regular interfaces:

```
type Writer …
// Write can write any string or []byte type, generalizing io.Writer.
func (w *Writer) Write[S ~string|~[]byte](s S) (int, error) { … }

// We probably would like this to be allowed.
var _ io.Writer = new(Writer)
```

Satisfying interfaces

If we had generic methods, we probably want them to implement regular interfaces:

```
type Writer …
// Write can write any string or []byte type, generalizing io.Writer.
func (w *Writer) Write[S ~string|~[]byte](s S) (int, error) { … }

// We probably would like this to be allowed.
var _ io.Writer = new(Writer)
```

Type assertions:

```
func F(v any) {
  v.(int)       // "Normal" type assertion

}
```

Satisfying interfaces

If we had generic methods, we probably want them to implement regular interfaces:

```
type Writer …
// Write can write any string or []byte type, generalizing io.Writer.
func (w *Writer) Write[S ~string|~[]byte](s S) (int, error) { … }

// We probably would like this to be allowed.
var _ io.Writer = new(Writer)
```

Type assertions:

```
func F(v any) {
  v.(int)       // "Normal" type assertion
  v.(io.Reader) // Interface type assertion
}
```

Satisfying interfaces

Thus:

```
type intCaller interface{ Call(int) int }
type stringCaller interface{ Call(string) int }
```

Satisfying interfaces

Thus:

```
type intCaller interface{ Call(int) int }
type stringCaller interface{ Call(string) int }
func F(v any) int {
  x := v.(stringCaller).Call("Hello, world")
  y := v.(intCaller).Call(42)
  return x+y
}
```

Satisfying interfaces

Thus:

```
type intCaller interface{ Call(int) int }
type stringCaller interface{ Call(string) int }
func F(v any) int {
  x := v.(stringCaller).Call("Hello, world")
  y := v.(intCaller).Call(42)
  return x+y
}
```

This is just first class generic functions with extra steps!

Satisfying interfaces

Thus:

```
type intCaller interface{ Call(int) int }
type stringCaller interface{ Call(string) int }
func F(v any) int {
  x := v.(stringCaller).Call("Hello, world")
  y := v.(intCaller).Call(42)
  return x+y
}
```

This is just first class generic functions with extra steps!

Corollary: Generic methods don’t implement **any** interfaces.

What's left

Call chaining:

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := ListShapes().
    Map(Shape.Area).
    Reduce(math.Max)
  fmt.Printf("The largest shape has area %v", m)
}
```

What's left

Call chaining:

```
type Shape interface{ Area() float64 }
func ListShapes() iter.Seq[Shape] { … }

func main() {
  m := ListShapes().
    Map(Shape.Area).
    Reduce(math.Max)
  fmt.Printf("The largest shape has area %v", m)
}
```

Has the static association between body and type argument.

What's left

"In Go, one of the main roles of methods is to permit types to implement interfaces. [...] we could decide that parameterized methods do not, in fact, implement interfaces, but then it's much less clear why we need methods at all."

— Type Parameters Proposal

"You are proposing large and significant language changes, which make generic methods behave significantly different from non–generic methods. The benefit of these changes appears to be to permit call chaining. I don't think the benefit outweighs the cost."

— Ian Lance Taylor

Summary

Summary

1. Everybody agrees this is a useful feature and wants it.

Summary

1. Everybody agrees this is a useful feature and wants it.
2. Implementing it requires runtime boxing or excluding interfaces.

Summary

1. Everybody agrees this is a useful feature and wants it.
2. Implementing it requires runtime boxing or excluding interfaces.
3. **Requiring** runtime boxing is an unacceptable cost.

Summary

1. Everybody agrees this is a useful feature and wants it.
2. Implementing it requires runtime boxing or excluding interfaces.
3. **Requiring** runtime boxing is an unacceptable cost.
4. The cost of excluding interfaces isn't justified by the leftover benefits.

Thank you