Introduction to Generic Programming in Go

Paul Bivrell
12 May 2021

What will you learn?

Minimal Agenda

- Discuss what types of problems generics solves
- Explore the accepted proposal for generics in Go
- Run example code in a demo implementation

My Goal:

Make you excited for the Generics in Go

What do you mean Generics in Go?

Generics and Go

Loose Definition:

Generics is a style of programming which allow programmers to write operations where the type of the data is defined by the user of the code rather than the writer.

Issue: This style of programming does not currently exist in Go

Solution:

On Feb 20, 2021 the proposal (https://github.com/golang/go/issues/43651#issuecomment-782705489) to add generic programming to Go has been accepted.

As early as 1.18 (Feb 2022) we could see generics in Go 🎉

The problem

Assertion

In Go we can not write one block of code that works with different types

List Search

```
package main

import "fmt"

func main() {
    fmt.Println(Search("a", []string{"a", "b", "c"}))
}

func Search(find string, list []string) (item string) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

Run

List Search (now with ints)

```
package main

import "fmt"

func main() {
    fmt.Println(Search(2, []int{1, 2, 3}))
}

func Search(find int, list []int) (item int) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

Run

Generic search?

But what about interface{}?

```
package main

import "fmt"

func main() {
    fmt.Println(Search("a", []interface{}{"a", "b", "c"}))
    fmt.Println(Search(2, []interface{}{1, 2, 3}))
}

func Search(find interface{}, list []interface{}) (item interface{}) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

Run

Static typing?

The previous code misses the mark for a good generic implementation because it subtraverts the purpose of static typing.

```
var myString string
myString = Search("b", []interface{}{"a", "b", "c"})
fmt.Println(myString)
```

Run

The interface{} implementation returns an interface{} not a string. We required a type cast.

```
var myString string
myString = Search("b", []interface{}{"a", "b", "c"}).(string)
fmt.Println(myString)
```

Run

This method avoids all of the benifits and compile time safety we get from using a staticly typed language.

Other alternatives

Reflect pkg (https://golang.org/pkg/reflect/)

- Difficult to read and write
- Potentially slow

Code generation

- Complicates build process
- Undermines go tool chain
- Uncompilable Artifact
- Not standardized

The (proposed) solution

Copy example

Let's look at a very simple example. How can we use generics to write the following functions a single time.

```
// CopyString takes the value from src and stores it in the destination
func CopyString(dst *string, src string) {
    *dst = src
}

// CopyInt takes the value from src and stores it in the destination
func CopyInt(dst *int, src int) {
    *dst = src
}

type AStruct struct{}

// CopyAStruct takes the value from src and store it in the destintation
func CopyAStruct(dst *AStruct, src AStruct) {
    *dst = src
}
```

Generic copy

```
func Copy[T any](dst *T, src T) {
    *dst = src
}
```

What's new here?

- Type parameter list
- Type constriant

Runnable Link (https://go2goplay.golang.org/p/CbcKpfcRf0E)

Making Search Generic

Working Generic Search?

```
package main

import "fmt"

func main() {
    fmt.Println(Search("a", []string{"a", "b", "c"}))
    fmt.Println(Search(2, []int{1, 2, 3}))
}

func Search[T any](find T, list []T) (item T) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

Runnable link (https://go2goplay.golang.org/p/LMcNNUX_lGh)

- This code does not work :(

Comparable type constraint

Operator == does not work on **any** type

New Type Constraint:

We need to change our type constriant from **any** to a constraint that compares things

```
func Search[T any](find T, list []T) (item T) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

- **comparable** like **any** is a predefined type constraint. It matches any type that can be compared with == and !=

Generic search

```
package main

import "fmt"

func main() {
    fmt.Println(Search("a", []string{"a", "b", "c"}))
    fmt.Println(Search(2, []int{1, 2, 3}))
}

func Search[T any](find T, list []T) (item T) {
    for _, v := range list {
        if v == find {
            return v
        }
    }
    return item
}
```

Runnable link (https://go2goplay.golang.org/p/g_JOUlzJ4k0).

Custom type constraints

Max function

Let's reduce these max functions in to a single generic function

```
func FloatMax(floats ...float64) float64 {
    var max float64
    for _, v := range floats {
        if v > max {
            max = v
        }
    }
    return max
}

func IntMax(ints ...int64) int64 {
    var max int64
    for _, v := range ints {
        if v > max {
            max = v
        }
    }
    return max
}
```

Generic Max

```
func Max[T ?](things ...T) T {
    var max T
    for _, v := range things  {
        if v > max {
            max = v
        }
    }
    return max
}
```

Runnable link (https://go2goplay.golang.org/p/OMwYimlv-ea)

Defining our own type constraint

Operator > does not wok on every type. We must define a type constraint

```
type Ordered interface {
    type int, float64
}

func Max[T Ordered](things ...T) T {
    var max T
    for _, v := range things {
        if v > max {
            max = v
        }
    }
    return max
}
```

Runnable link (https://go2goplay.golang.org/p/n-cXKd-RUCJ)

Note that **any** is just an alias for **interface{}**

More type constraints

Multiple type constraints

The type parameter list as it implies can take multiple values for a type parameter

```
type integer interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr
}

func Convert[To, From integer](from From) To {
    to := To(from)
    if From(to) != from {
        panic("conversion out of range")
    }
    return to
}
```

Runnable link (https://go2goplay.golang.org/p/02nd1sCKzvU)

Type constraints with methods

Interfaces as Interfaces

We can't define operators on user defined types. We instead use methods on types. Unsurprisingly we can use the standard interface syntax to constrain a generic type.

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

    for i, v1 := range list {
        for j, v2 := range list {
            if v1.Less(v2) {
                list[i], list[j] = list[j], list[i]
            }
        }

    }
}
```

Runnable link (https://go2goplay.golang.org/p/E4AfKBBS4Tl)

Closing Words

Tip of the iceberg

- There is a lot more to explore in the current generics proposal (https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md)

Subject to Change

- The implementation can change (https://github.com/golang/go/issues/45346)

So far so Good

This generic implementation is relatively simple, solves problems with minimal additional complexity, and feels like Go.

Resources

- Generics Playground (https://go2goplay.golang.org/)
- Accepted type parameter proposal (https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md)
- Proposal discussion on github (https://github.com/golang/go/issues/43651)
- GopherCon 2019 Generics Talk - Ian Lance Taylor (https://www.youtube.com/watch?v=WzgLqE-3IhY&t=296s)
- GopherCon 2020 Generics Talk - Robert Griesemer (https://www.youtube.com/watch?v=TborQFPY2IM)
- Indepth proposal review - Bill Kennedy (https://www.youtube.com/watch?v=gIEPspmbMHM)

Thank you

Paul Bivrell
12 May 2021
Slack: @pbj (#ZgotmplZ)