Go 2 Draft Designs

Hello everyone, I’m here to talk about the draft designs that the Go team recently released for possible future additions to Go, specifically about two areas: improved error handling and parametric polymorphism (AKA generics), which have consistently been the two most requested features by the Go community over the past few years.

Go 2: actually a thing?

Now, you might be thinking “haha, Go 2, suuure”. And I’ll grant you, even I have sometimes felt like that. But I think that releasing these drafts shows that the Go team really *is* trying to move the language forward and solve these problems.

Generics

Requirements:

- Don't allow arbitrary compile–time metaprogramming
- Be implementable at compile time or run time
- Explicit type constraints
- No boxing

Before we start, there are some pitfalls of generics in other languages - particularly C++ and Java - that we'd like to avoid in Go.

C++'s templates are famously Turing-complete. This means that one can do arbitrary meta-programming at compile time; but the result tends to be nigh-unreadable code (ever tried debugging code that uses Boost?). In general, a good syntax for polymorphism isn't necessarily a good syntax for metaprogramming. They should be separate.

Another problem in C++ is that each use of a template generates a new copy of its code. This usually gives maximum runtime performance, but also a lot of bloat. Java has the exact opposite problem; every instantiation of a generic uses the same code. Instead, the generics design draft proposes to impose some constraints so that any generic function (including methods on generic types) can be implemented either way, letting the compiler decide which is best on a case-by-case basis.

In (current) C++, the constraints on the type parameters of a template are defined implicitly by their usage, and checked when the template is expanded. This frequently leads to incomprehensible errors from deep within nested templates, when an inappropriate type is substituted. Recording the type constraints explicitly allows the compiler to complain immediately when a type isn't implemented.

We also want to avoid boxing - particularly of primitive types - as happens in Java, or in "generic" code we write in Go today; boxing requires heap allocations, which makes a lot of useful, simple generic functions (like Min/Max) unusably slow.

Generics

```
type Apple struct { Weight float64 }

// ...
j := -1
for i, a := range apples {
  if a.Weight > 0.2 {
    j = i
  }
}
if j == -1 {
  // ...
}
bigApple := apples[j]
```

So, what to we want generics for? Here's a bit of code that looks up an apple in a slice of apples. Notice anything?

Generics

```
type Apple struct { Weight float64 }

// ...
j := -1
for i, a := range apples {
  if a.Weight > 0.2 {
    j = i
    break
  }
}
if j == -1 {
  // ...
}
bigApple := apples[j]
```

I forgot to break out of the loop after finding the apple. If there were a function to let me just say "find an apple whose weight > 0.2", I wouldn't have made this mistake.

Generics

```
type Apple struct { Weight float64 }
type Pear struct { Mass float64 }

func Find(xs []Apple, pred func(Apple) bool) (Apple, bool) {
  for i, x := range xs {
    if pred(x) {
      return x, true
    }
  }
  return Apple{}, false
}

// works
bigApple, ok := Find(apples, func(a Apple) bool { return a.Weight > 0.2 })
// doesn't work
weirdPear, ok := Find(pears, func(p Pear) bool { return p.Mass < 0 })
```

Now, we can at least clarify our intent by wrapping the loop in a function. But while we can make the predicate a parameter, we still must write the function in terms of Apples. If we want to use the same pattern again with a different type, like Pear, we have to write another version of this function, or copy it. Either option is a good way to end up with bugs, as you just saw.

Generics

```
func Find(xs, pred interface{}) (interface{}, bool) {
  rxs := reflect.ValueOf(xs)
  predV := reflect.ValueOf(pred)
  for i := 0; i < rxs.Len(); i++ {
    x := rxs.Index(i)
    if predV.Call([]reflect.Value{x})[0].Bool() {
      return x.Interface(), true
    }
  }
  return reflect.Zero(rxs.Type().Elem()), false
}

// works
bigApple, ok := Find(apples, func(a Apple) bool { return a.Weight > 0.2 })
// also works
weirdPear, ok := Find(pears, func(p Pear) bool { return p.Weight < 0 })
// doesn't work, but compiles
nonsense, ok := Find("potato", make(chan bool))
```

... unless we use reflection to implement Find, but then we take a huge performance penalty (usually an order of magnitude), get WAY uglier code, and lose compile-time type safety.

Generics

```
type Apple struct { Weight float64 }
type Pear struct { Mass float64 }

func Find(type T)(xs []T, pred func(T) bool) (T, bool) {
  for i, x := range xs {
    if pred(x) {
      return x, true
    }
  }
  var x T // generic way to get the zero value of T
  return x, false
}

// works
bigApple, ok := Find(apples, func(a Apple) bool { return a.Weight > 0.2 })
// also works
weirdPear, ok := Find(pears, func(p Pear) bool { return p.Weight < 0 })
// doesn't even compile
nonsense, ok := Find("potato", make(chan bool))
```

Here's how we make a generic Find with the draft's syntax. We add a type parameter after the function name but before the regular parameters, and refer to it throughout.

Generics

```
func Bogosort(type T)(xs []T) {
  for !IsSorted(xs) {
    Shuffle(xs)
  }
}

func IsSorted(type T)(xs []T) bool {
  for i, x := range xs {
    if i > 0 && x < xs[i - 1] {
      return false
    }
  }
  return true
}
```

Let's look at another example. Suppose we want to make a generic sort function. This looks like it might work. But it doesn't!

Generics

```
func Bogosort(type T)(xs []T) {
  for !IsSorted(xs) {
    Shuffle(xs)
  }
}

func IsSorted(type T)(xs []T) bool {
  for i, x := range xs {
    if i > 0 && x < xs[i - 1] {
      return false
    }
  }
  return true
}
```

The problem is that T is allowed to be any type. There's only a few operations available for every type, and comparison isn't one of them.

Generics

```
contract Comparable(x T) { x < x }

func Bogosort(type T)(xs []T) {
  for !IsSorted(xs) {
    Shuffle(xs)
  }
}

func IsSorted(type T)(xs []T) bool {
  for i, x := range xs {
    if i > 0 && x < xs[i - 1] {
      return false
    }
  }
  return true
}
```

So, we need to declare that T is comparable. We do that by using something called a contract. Within the contract, we exemplify, using code - notice how the syntax is very similar to regular function syntax -, the operations that values of type T need to support - in this case, just comparison.

This is most similar to the idea of "concepts" in C++, but somewhat less verbose.

Generics

```
package bytes

func Contains(b, subslice []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
func Split(s []byte, sep []byte) [][]byte


package strings

func Contains(b, subslice []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
func Split(s []byte, sep []byte) [][]byte
```

Here's a third example. As you might know, Go has two sets of string functions. One operates on []bytes, which are mutable, and the other on strings, which are immutable. But the logic is exactly the same.

Generics

```
package strings

contract String(s T) {
  var b byte = s[0]
  s[0:0]
  len(s)
}

func Contains(type S String)(b, subslice S) bool
func Index(type S String)(s, sep S) int
func Join(type S String)(s []S, sep S) []S
func Split(type S String)(s S, sep S) []S
```

With a contract describing the operations common to string and []byte - indexing to get a byte, slicing and getting the length - we can define a set of functions that operate on both.

(We don't need to specify the result type of slicing; it's always the type of the thing being sliced.)

Generics

```
package sync

type Pool struct {
  New func() interface{}
}

func (p *Pool) Get() interface{}
func (p *Pool) Put(x interface{})
```

Generic functions aren't the only thing we can do, of course. We can also make generic types. Here's the standard library's sync.Pool type, which has to use interface{} to allow making pools of any kind of item.

Generics

```
package sync

type Pool(type T) struct {
  New func() T
}

func (p *Pool(T)) Get() T
func (p *Pool(T)) Put(x T)
```

Making the Pool generic is simple. Note that you have to define the receiver type as Pool(T) in each method. The methods can refer to T, but can't introduce any type parameters of their own. (This is one of the constraints needed to allow the compiler to choose freely how to implement the code.)

Error handling

```
func cp(src, dest string) error {
  srcF, err := os.Open(src)
  if err != nil {
    return err
  }
  defer srcF.Close()
  destF, err := os.Create(dest)
  if err != nil {
    return err
  }
  if _, err := io.Copy(destF, srcF); err != nil {
    os.Remove(dest)
    destF.Close()
    return err
  }
  err = destF.Close()
  if err != nil {
    os.Remove(dest)
  }
  return err
}
```

Now, about error handling. If you've ever worked with Go, you've probably had to write something like this. It's annoying to write `if err != nil { return err }` so many times, but the worse problem is that you have to *read* it so many times. It's hard to tell at a glance what this function does, because the important bits are buried in boilerplate.

Error handling

Why does it work this way? Two main reasons.

- Make error checks explicit
  - Makes it harder to write incorrect code by accident
- Handle errors with ordinary control flow constructs
  - Makes the language simpler, and easier to learn

Now, it’s important to remember that Go’s error handling works this way for 2 main reasons:

Error handling

```
func cp(src, dest string) error {
  srcF, err := os.Open(src)
  if err != nil {
    return err
  }
  defer srcF.Close()
  destF, err := os.Create(dest)
  if err != nil {
    return err
  }
  if _, err := io.Copy(destF, srcF); err != nil {
    os.Remove(dest)
    destF.Close()
    return err
  }
  err = destF.Close()
  if err != nil {
    os.Remove(dest)
  }
  return err
}
```

Looking back at the code, I’ve highlighted the important bits, which makes it a lot easier to scan. As you can see, about half of it is error handling boilerplate, and the rest is scattered.

Error handling

```
func cp(src, dest string) error {
  srcF := check os.Open(src)
  defer srcF.Close()
  destF := check os.Create(dest)
  handle err {
    destF.Close()
    os.Remove(dest)
  }
  check io.Copy(destF, srcF)
  check destF.Close()
  return nil
}
```

And now, we have the equivalent code using the error handling draft design. All the highlighted bits are there, but there’s a lot less noise between them.

Error handling

```
func cp(src, dest string) error {
  srcF := check os.Open(src)
  defer srcF.Close()
  destF := check os.Create(dest)
  handle err {
    destF.Close()
    os.Remove(dest)
  }
  check io.Copy(destF, srcF)
  check destF.Close()
  return nil
}
```

Because of that, we don’t need the highlights anymore.

What the “check” keyword does is, essentially, add in the if statements that were in the original version. It also invokes any “handle” block that comes before it.
Let’s suppose we want to wrap the error before returning:

Error handling

```
func cp(src, dest string) error {
  handle err {
    return fmt.Errorf("error copying %s to %s: %v",
      src, dest, err)
  }
  srcF := check os.Open(src)
  defer srcF.Close()
  destF := check os.Create(dest)
  handle err {
    destF.Close()
    os.Remove(dest)
  }
  check io.Copy(destF, srcF)
  check destF.Close()
  return nil
}
```

We only need to write the wrapping code *once*. (The first “handle” block overrides the default return that “check” would otherwise do.) With current Go, we would have to either repeat it several times, or write a wrapper function.

“Check” does another interesting thing for us. Notice how we’re not declaring an “err” variable to store the errors? That’s because “check” doesn’t just add the error handling path; it also removes the error from the call’s returns. So, for example, os.Open returns a os.File and an error, but “check os.Open” returns just a os.File.

Error handling

```
type Point struct { X, Y, Z float64 }

func ParsePoint(s string) (Point, error) {
  pieces := strings.SplitN(s, ",", 3)
  var p Point
  var err error
  if p.X, err = strconv.ParseFloat(pieces[0], 64);
err != nil {
    return Point{}, error
  }
  if p.Y, err = strconv.ParseFloat(pieces[1], 64);
err != nil {
    return Point{}, error
  }
  if p.Z, err = strconv.ParseFloat(pieces[2], 64);
err != nil {
    return Point{}, error
  }
  return p, nil
}
```

Why is that helpful? Because it means we can more easily compose things that return errors. Here's an example. This function calculates a Point from a string, but because ParseFloat can return an error, we must write it in an imperative style, as a sequence of steps. It's pretty ugly.
Now let's rewrite it…

Error handling

```
type Point struct { X, Y, Z float64 }

func ParsePoint(s string) (Point, error) {
  handle err {
    return Point{}, fmt.Errorf("invalid point %q: %v", s, err)
  }
  pieces := strings.SplitN(s, ",", 3)
  return Point{
    check strconv.ParseFloat(pieces[0], 64),
    check strconv.ParseFloat(pieces[1], 64),
    check strconv.ParseFloat(pieces[2], 64)}
}
```

Not only do we cut on the repetition, we are now able to express ParsePoint much more smoothly - almost like we had exceptions. And we even added a better error message.

Error inspection

```
func cp(src, dest string) error {
  handle err {
    return fmt.Errorf("error copying %s to %s: %v",
      src, dest, err)
  }
  srcF := check os.Open(src)
  ...
```

On another note, look at our error wrapping pattern. It seems fine, right?

Error inspection

```
err := cp("/etc/candy", "/mount/floppy0/icecream")
if os.IsNotExist(err) {
  fmt.Println("Your floppy doesn't contain ice cream.")
}
check err
```

```
err := cp("/etc/candy", "/mount/floppy0/icecream")
if err == os.ErrDiskFull {
  fmt.Println("Your stomach is full.")
}
check err
```

However, it prevents us from checking for specific errors like this, because "err" here isn't the original error from, say, os.Open; it's a wrapper that IsNotExist doesn't recognise and therefore cannot pick apart.

The same problem happens if you compare with a sentinel error - worse, in fact, since == can't be extended to understand any wrappers, like a function can.

Error inspection

```
type CopyError struct {
  From, To string
  Err error
}

func (ce CopyError) Error() string {
  return fmt.Sprintf("copy %s to %s: %v", ce.From, ce.To, err)
}

func cp(dest, src string) error {
  handle err {
    return CopyError{src, dest, err}
  }
  // ...
}

err := cp("/etc/candy", "/mount/floppy0/icecream")
if err.(CopyError).Err == os.ErrDiskFull {
  fmt.Println("Your stomach is full.")
}
check err
```

We can solve this with a more sophisticated wrapper. And so can everyone else - each in a different, incompatible way. Our code won't be able to see through other people's wrappers, and vice-versa.

Error inspection

```
package errors

type Wrapper interface {
  Unwrap() error
}

func Is(err, target error) bool
func As(type E)(err error) (E, bool)

type Formatter interface {
  Format(p Printer) (next error)
}
```

To solve this, the Go team proposes to introduce a standard interface for error wrappers, as well as two functions that use that interface to look for either a specific error value, or a specific type anywhere in an error's wrapper chain.

There's also an error printing interface, Formatter, to help standardise the display format and support internationalisation of the messages.

Notably, this can all be done in current Go. (We could express As without the generic type, although a bit more awkwardly.)

Error inspection

```
type CopyError struct {
  From, To string
  Err error
}

func (ce CopyError) Error() string { return "..." }
func (err CopyError) Unwrap() error { return err.Err }

func cp(dest, src string) error { /* ... */ }

err := cp("/etc/candy", "/mount/floppy0/icecream")
if errors.Is(err, os.ErrDiskFull) {
  fmt.Println("Your stomach is full.")
}
check err

err := eatAllCandy()
if pe, ok := errors.As(*os.PathError)(err); ok {
  fmt.Println("failed to eat candy: ", ce.Path)
}
```

All we have to do is give our wrapper an Unwrap method, and the Is and As function will be able to see through it.

Of course, for this to work properly, all error wrappers have to implement this interface. But there's precedent for this in the community: consider how widely-adopted interfaces like io.Reader, io.Writer or http.Handler are. And this one is, in most cases, trivial to implement.

More info at
golang.org/s/go2designs