Deconstructing Type Parameters - The Go Programming Language

Ian Lance Taylor 26 September 2023

The Go Blog

slices package function signatures

The `slices.Clone` function is pretty simple: it makes a copy of a slice of any type.

```
func Clone[S ~[]E, E any](s S) S {
    return append(s[:0:0], s...)
}
```

This works because appending to a slice with zero capacity will allocate a new backing array. The function body winds up being shorter than the function signature, which is in part because the body is short, but also because the signature is long. In this blog post we'll explain why the signature is written the way that it is.

Simple Clone

We'll start by writing a simple generic `Clone` function. This is not the one in the `slices` package. We want to take a slice of any element type, and return a new slice.

```
func Clone1[E any](s []E) []E {
    // body omitted
}
```

The generic function `Clone1` has a single type parameter `E`. It takes a single argument `s` which is a slice of type `E`, and it returns a slice of the same type. This signature is straightforward for anybody familiar with generics in Go.

However, there is a problem. Named slice types are not common in Go, but people do use them.

```
// MySlice is a slice of strings with a special String method.
type MySlice []string

// String returns the printable version of a MySlice value.
func (s MySlice) String() string {
    return strings.Join(s, "+")
}
```

Let’s say that we want to make a copy of a `MySlice` and then get the printable version, but with the strings in sorted order.

```
func PrintSorted(ms MySlice) string {
    c := Clone1(ms)
    slices.Sort(c)
    return c.String() // FAILS TO COMPILE
}
```

Unfortunately, this doesn’t work. The compiler reports an error:

```
c.String undefined (type []string has no field or method String)
```

We can see the problem if we manually instantiate `Clone1` by replacing the type parameter with the type argument.

```
func InstantiatedClone1(s []string) []string
```

The Go assignment rules allow us to pass a value of type `MySlice` to a parameter of type `[]string`, so calling `Clone1` is fine. But `Clone1` will return a value of type `[]string`, not a value of type `MySlice`. The type `[]string` doesn’t have a `String` method, so the compiler reports an error.

Flexible Clone

To fix this problem, we have to write a version of `Clone` that returns the same type as its argument. If we can do that, then when we call `Clone` with a value of type `MySlice`, it will return a result of type `MySlice`.

We know that it has to look something like this.

```
func Clone2[S ?](s S) S // INVALID
```

This `Clone2` function returns a value that is the same type as its argument.

Here I’ve written the constraint as `?`, but that’s just a placeholder. To make this work we need to write a constraint that will let us write the body of the function. For `Clone1` we could just use a constraint of `any` for the element type. For `Clone2` that won’t work: we want to require that `s` be a slice type.

Since we know we want a slice, the constraint of `S` has to be a slice. We don’t care what the slice element type is, so let’s just call it `E`, as we did with `Clone1`.

```
func Clone3[S []E](s S) S // INVALID
```

This is still invalid, because we haven’t declared `E`. The type argument for `E` can be any type, which means it also has to be a type parameter itself. Since it can be any type, its constraint is `any`.

```
func Clone4[S []E, E any](s S) S
```

This is getting close, and at least it will compile, but we're not quite there yet. If we compile this version, we get an error when we call `Clone4(ms)`.

```
MySlice does not satisfy []string (possibly missing ~ for []string in
[]string)
```

The compiler is telling us that we can't use the type argument `MySlice` for the type parameter `S`, because `MySlice` does not satisfy the constraint `[]E`. That's because `[]E` as a constraint only permits a slice type literal, like `[]string`. It doesn't permit a named type like `MySlice`.

Underlying type constraints

As the error message hints, the answer is to add a `~`.

```
func Clone5[S ~[]E, E any](s S) S
```

To repeat, writing type parameters and constraints `[S []E, E any]` means that the type argument for `S` can be any unnamed slice type, but it can't be a named type defined as a slice literal. Writing `[S ~[]E, E any]`, with a `~`, means that the type argument for `S` can be any type whose underlying type is a slice type.

For any named type `type T1 T2` the underlying type of `T1` is the underlying type of `T2`. The underlying type of a predeclared type like `int` or a type literal like `[]string` is just the type itself. For the exact details, see the language spec. In our example, the underlying type of `MySlice` is `[]string`.

Since the underlying type of `MySlice` is a slice, we can pass an argument of type `MySlice` to `Clone5`. As you may have noticed, the signature of `Clone5` is the same as the signature of `slices.Clone`. We've finally gotten to where we want to be.

Before we move on, let's discuss why the Go syntax requires a `~`. It might seem that we would always want to permit passing `MySlice`, so why not make that the default? Or, if we need to support exact matching, why not flip things around, so that a constraint of `[]E` permits a named type while a constraint of, say, `=[]E`, only permits slice type literals?

To explain this, let's first observe that a type parameter list like `[T ~MySlice]` doesn't make sense. That's because `MySlice` is not the underlying type of any other type. For instance, if we have a definition like `type MySlice2 MySlice`, the underlying type of `MySlice2` is `[]string`, not `MySlice`. So either `[T ~MySlice]` would permit no types at all, or it would be the same as `[T MySlice]` and only match `MySlice`. Either way, `[T ~MySlice]` isn't useful. To avoid this confusion, the language prohibits `[T ~MySlice]`, and the compiler produces an error like

```
invalid use of ~ (underlying type of MySlice is []string)
```

If Go didn't require the tilde, so that `[S []E]` would match any type whose underlying type is `[]E`, then we would have to define the meaning of `[S MySlice]`.

We could prohibit `[S MySlice]`, or we could say that `[S MySlice]` only matches `MySlice`, but either approach runs into trouble with predeclared types. A predeclared type, like `int` is its own underlying type. We want to permit people to be able to write constraints that accept any type argument whose underlying type is `int`. In the language today, they can do that by writing `[T ~int]`. If we don't require the tilde we would still need a way to say "any type whose underlying type is `int`". The natural way to say that would be `[T int]`. That would mean that `[T MySlice]` and `[T int]` would behave differently, although they look very similar.

We could perhaps say that `[S MySlice]` matches any type whose underlying type is the underlying type of `MySlice`, but that makes `[S MySlice]` unnecessary and confusing.

We think it's better to require the ~ and be very clear about when we are matching the underlying type rather than the type itself.

Type inference

Now that we've explained the signature of `slices.Clone`, let's see how actually using `slices.Clone` is simplified by type inference. Remember, the signature of `Clone` is

```
func Clone[S ~[]E, E any](s S) S
```

A call of `slices.Clone` will pass a slice to the parameter `s`. Simple type inference will let the compiler infer that the type argument for the type parameter `S` is the type of the slice being passed to `Clone`. Type inference is then powerful enough to see that the type argument for `E` is the element type of the type argument passed to `S`.

This means that we can write

```
    c := Clone(ms)
```

without having to write

```
    c := Clone[MySlice, string](ms)
```

If we refer to `Clone` without calling it, we do have to specify a type argument for `S`, as the compiler has nothing it can use to infer it. Fortunately, in that case, type inference is able to infer the type argument for `E` from the argument for `S`, and we don't have to specify it separately.

That is, we can write

```
    myClone := Clone[MySlice]
```

without having to write

```
    myClone := Clone[MySlice, string]
```

Deconstructing type parameters

The general technique we've used here, in which we define one type parameter `S` using another type parameter `E`, is a way to deconstruct types in generic function signatures. By deconstructing a type, we can name, and constrain, all aspects of the type.

For example, here is the signature for `maps.Clone`.

```
func Clone[M ~map[K]V, K comparable, V any](m M) M
```

Just as with `slices.Clone`, we use a type parameter for the type of the parameter `m`, and then deconstruct the type using two other type parameters `K` and `V`.

In `maps.Clone` we constrain `K` to be comparable, as is required for a map key type. We can constrain the component types any way we like.

```
func WithStrings[S ~[]E, E interface { String() string }](s S) (S,
[]string)
```

This says that the argument of `WithStrings` must be a slice type for which the element type has a `String` method.

Since all Go types can be built up from component types, we can always use type parameters to deconstruct those types and constrain them as we like.