Understanding Go 1.21 generics type inference – Encore Blog

Go 1.21 is out, and with it comes a whole slew of improvements like better generic type inference (what this article is about); the new builtin functions `min`, `max` and `clear`; and several new packages in the standard library (`maps`, `slices`, `cmp`, `log/slog` and `testing/slogtest`). Read the full release notes here.

Of particular interest, at least to us here at Encore, are the improvements to generics type inference, as it affects how Encore's static analysis works. However we felt the release notes were quite hard to understand, so this article breaks down the changes with more explanations and examples.

We've also just released Encore v1.24 which supports Go 1.21, so it's a good way to take these new changes for a spin.

Partially instantiated generic functions

From the Go 1.21 release notes:

> A (possibly partially instantiated generic) function may now be called with arguments that are themselves (possibly partially instantiated) generic functions. The compiler will attempt to infer the missing type arguments of the callee (as before) and, for each argument that is a generic function that is not fully instantiated, its missing type arguments (new). Typical use cases are calls to generic functions operating on containers (such as slices.IndexFunc) where a function argument may also be generic, and where the type argument of the called function and its arguments are inferred from the container type.

What does this mean?

Consider a function that takes a value and reports whether it's the zero value:
`func IsZero[T comparable](a T) bool`.

Prior to Go 1.21, passing such a function as an argument to another (generic or non-generic) function required you to explicitly specify the type arguments, even if it was obvious from context.

For example, consider:

```
// Any reports whether fn returns true for any number in the slice.
func Any(numbers []int, fn func(int) bool) bool {
	for i, v := range numbers {
		if fn(v) {
			return true
		}
	}
```

```
        return false
}
```

As a result, you can now write:

```
numbers := []int{1, 2, 3}

// This works in Go 1.21
anyZeroes := Any(numbers, IsZero)

// Prior to Go 1.21 you had to write:
anyZeroes := Any(numbers, IsZero[int])
```

As the release notes suggest, this also works with generic functions. So you can also write:

```
// This works in Go 1.21
firstZeroIndex := slices.IndexFunc(numbers, IsZero)

// Prior to Go 1.21 you had to write:
firstZeroIndex := slices.IndexFunc(numbers, IsZero[int])
```

Go 1.21 infers that `IsZero` means `IsZero[int]` from the type of `numbers`, even though `slices.IndexFunc` itself is a generic function.

Continuing in the same paragraph from the release notes:

> More generally, a generic function may now be used without explicit instantiation when it is assigned to a variable or returned as a result value if the type arguments can be inferred from the assignment.

This means the following is now valid (prior to Go 1.21 you would need to explicitly specify the type arguments)

```
// IsZero[string] is inferred from the assignment to
// a variable of type func(a string) bool.
var isZeroString func(a string) bool = IsZero

func IsNilPointerFactory[T any] func() func(val *T) bool {
  // IsZero[*T] is inferred from the return type, even though
  // this is a generic function.
  return IsZero
}
```

Interface assignment inference

Go 1.21 also improves type inference for generic interface types. From the release notes:

> Type inference now also considers methods when a value is assigned to an interface: type arguments for type parameters used in method signatures may be inferred from the corresponding parameter types of matching methods.

So what does that mean? Consider the following generic interface:

```
type RandomElementer[T any] interface {
	// RandomElement returns a random element from a collection.
	// If the collection is empty it returns (zero, false).
	RandomElement() (T, bool)
}
```

Then consider a generic helper function that calls `RandomElement` on some collection but panics if the collection is empty:

```
// MustRandom returns a random element from t collection.
// If the collection is empty it panics.
func MustRandom[T any](collection RandomElementer[T]) T {
	val, ok := collection.RandomElement()
	if !ok {
		panic("collection is empty")
	}
	return val
}
```

In Go 1.21 it's now possible to call `MustRandom` and the interface type will be inferred from the argument. Consider these two types, one generic and one not:

```
// MyList is a generic, custom list type.
type MyList[T any] []T
func (l MyList[T]) RandomElement() (T, bool) { /* ... */ }

// IPSet represents a set of IP addresses.
type IPSet map[netip.Addr]bool
func (s IPSet) RandomElement() (netip.Addr, bool) { /* ... */ }
```

These can now be used like so:

```
randomInt := MustRandom(MyList[int]{})
randomIP := MustRandom(IPSet{})
```

```
// In Go 1.20 you would have to do:
randomInt := MustRandom[int](MyList[int]{})
randomIP := MustRandom[netip.Addr](IPSet{})
```

Wonderful! Moving on:

> Similarly, since a type argument must implement all the methods of its corresponding constraint, the methods of the type argument and constraint are matched which may lead to the inference of additional type arguments.

This basically means the above type inference is also extended to functions that take additional types. For example, the `MustRandom` signature could be rewritten as:

```
func MustRandom[R RandomElementer[T], T any](collection R) T
```

Go 1.21 correctly infers both `R` and `T` at the same time when calling `MustRandom(IPSet{})`.

Type inference for untyped constants

> If multiple untyped constant arguments of different kinds (such as an untyped int and an untyped floating-point constant) are passed to parameters with the same (not otherwise specified) type parameter type, instead of an error, now type inference determines the type using the same approach as an operator with untyped constant operands. This change brings the types inferred from untyped constant arguments in line with the types of constant expressions.

Go has the notion of untyped constants, and the type of an expression like `a + b` where `a` and `b` are number literals is inferred from the constant values. For example:

```
var x = 1 + 2   // x is of type int
var y = 1 + 2.5 // y is of type float64 (because 2.5 can't be represented
as int)
```

Now consider writing a generic `Add` function that should behave like the built-in + operator:

```
func Add[T int | float6](a, b T) T {
  return a + b
}
```

In Go 1.21, this works exactly like you would expect:

```
var x = Add(1, 2)   // x is of type int
var y = Add(1, 2.5) // y is of type float64
```

In Go 1.20 the same code fails to compile, because the type inference would consider each argument separately. It would infer the type of `1` as `int` and then fail to unify `2.5` with `int`, resulting in the error "default type float64 of 2.5 does not match inferred type int for T".

Summary

That's the extent of the changes to generic type inference in Go 1.21. Hopefully the examples help making the changes more understandable; they certainly did for me.