A Proposal for Generic Types in Go

by: Kyle Lemons

November 12, 2011

Contents

Listings

1 Introduction

I have been using Go now for over a year. I think that through this time, I have gotten a pretty good feel for the language as a whole and the principles that guide it. There have been very few times where I have even been tempted to look for generic types while writing actual programs, but for some things (like the math package), there is clearly no good solution yet within the confines of the language. The creators of Go have acknowledged that they will consider adding generic types to the language if they find the right way to go about it, so here is my humble attempt at defining what generic types could look like within Go.

1.1 Philosophy

In designing this proposal for generic types, there are a few fundamental principles which have guided my hand. I have discussed various generic type proposals with other members of the community who have other design goals in mind, so I think it's probably a good idea to state these up front.

- Generic types should be simple.
- Generic types should not add any new syntax to the language.
- Generic types should not incur much, if any, run-time overhead.
- It should be possible to use a generic value as a map key or slice index.
- Generic types should not require the programmer to explicitly parameterize the Generic types (as templates in C++).
- Generic types should not require importing a package multiple times for multiple, different instances of the generic construct in question.

1.2 Overview

This proposal for generic types adds a single keyword to the language: `generic`. The keyword can be used in a type definition, in place of an interface or structure type. For a basic example of a package with generic types, consider a fictitious package `gen`, shown in Listing 1.

To demonstrate the simplicity of the generic types in this proposal, consider the short (but contrived) program in Listing 2 to sum up command-line arguments.

This code could be entirely permissable with generic types, because the compiler specializes `gen.Tern` and `gen.Check` for `Value` as type `string` and specializes `gen.ReduceFunc` for `Value, Result` as `string, int` during code generation. This could be done in a similar fashion to the way in which the C++ compiler would create a `string` version of the function for `gen::Tern<string>`, except with inferences for each generic type involved.

Compare this example to a scenario in which `Value` was an `interface` of some kind: no type assertions or run-time overhead is incurred, and the type safety of the code is guaranteed at compile time. If `Value` was an interface, the compiler would have no way of ensuring that the two types given to `gen.Tern` were the same type, and would not be able to know that adding a `string` to the result of the function call was acceptable. With the `interface`, these checks would either have the potential to `panic` or require extra boilerplate to check at run-time.

Listing 1: Basic example of generic types

```go
// Package gen provides some example generic functionality.
package gen

// Generic types for this package
type Value generic
type Result generic
type ReduceFunc func(Result, Value) Result

// Tern is the ternary operator: Tern(cond,t,f) == (cond?t:f)
func Tern(cond bool, t, f Value) Value {
  if cond {
    return t
  }
  return f
}

// Reduce returns the result of running the ReduceFunc on each successive value
// in list along with the last result.
func (red ReduceFunc) Reduce(r0 Result, list ...Value) Result {
  for _, v := range list {
    r0 = red(r0, v)
  }
  return r0
}

// Check strips out the os.Error return from a function call for use inline.
func Check(val Value, err os.Error) Value {
  if err != nil {
    panic(err)
  }
  return val
}
```

Listing 2: Usage of basic generic types

```go
package main

import (
  "fmt"
  "gen"
  "strconv"
)

func main() {
  add := gen.ReduceFunc(func(x0 int, x1 string) int {
    return x0 + gen.Check(strconv.Atoi(x1))
  })

  nums := os.Args[1:]
  fmt.Printf("Sum of your %s: %v\n",
    "number" + gen.Tern(len(nums)==1, "", "s"),
    add.Reduce(0, os.Args))
}
```

2 Generic Type Usage

In each of the following sections, I will attempt to do the following:

- Give an example of the usage of generic types in that context
- Explain when generic types can and cannot be used in that context
- Give a possible method for compiler/linker implementation of that application

2.1 Pure Generic Types

We have already seen an example of pure generic types: on Listing 1:5-6. Any named type whose underlying type has been specified with the `generic` keyword is a "generic type."

When the compiler finds a pure generic type, it will track this type in a much different way from other types. When the type is used as a part of another definition, it will not fully compile that entity and will instead store something similar to the parse tree of that construct, along with a set of restrictions about how the value was used (e.g. whether the value of the generic type was used as a slice index or map key, whether the value was casted, whether the value had any basic operators used and the type of the other operand, etc). This parse tree could be processed to some degree without specializing the types, but would probably need to be stored in the object files with very little obfuscation. Because of this restriction, it would be difficult to hide the implementation of a generic construction from the consumer of a library.

When the compiler is performing type checks and encounters a generic type (e.g. on Listing 2:11 when type-checking `gen.Check`), it will validate the inferred type against these restrictions before specializing a new instance of the function, interface, etc. It will collect the apparent types of all values necessary to specialize the type, and then ensure that all instances of a particular generic type have identical concrete types. Like the rest of Go, no implicit type conversion will happen.

2.2 Hybrid Generic Types

As with pure generic types, we have already seen an example of hybrid generic types: on Listing 1 line 7. In my (admittedly arbitrary) terminology, a hybrid generic type is a normal Go type (an interface, a function, a struct) which cannot be completely described without filling in the concrete type of one or more generic type.

At a very basic level, the code for an entire hybrid type (including all of its methods and conversions) can be specialized at the time in which a value of that type is declared. Values and variables specialized from a generic type should probably not maintain an annotation about the generic type from which they were specialized, as passing a value specialized with one generic type as a value of a different generic type with the same requirements seems sensible. This may weaken the strict typing of Go, though, especially when reading through code and eyeballing variable types, and so may not be a good idea if generic types and user-defined types are intermixed.

2.3 Generic Functions

Generic functions would behave the way I think we would all expect them to. At the call-site, the concrete types of each generic argument with the same type must match, and then the return types must match. I suspect that there will have to be a requirement that any generic type used in a return value must also be used in an argument, but it would be really cool if this wasn't the case.

One really nice aspect of Go is that every type has a well-defined zero value (though it may not be properly initialized), so it is possible to instantiate values of a generic type within the value of a function. I suspect that there will also have to be a requirement that all generic types in the function will have to also be used in the argument list.

Calling a function with generic arguments within a generic function should be possible. In general, it will cascade any requirements of the inner function's generic type outward. In some cases it may be possible to identify when one generic type cannot be passed as a value of another—for instance, if a generic value is indexed in the inner function and operated on with an arithmetic operator in the outer function or another inner function.

2.4 Generic Interfaces

Generic interfaces behave the way I think would be expected. It will probably incur a runtime penalty higher than that of a normal interface, however, for things like type switches and type assertions, due to the extra requirements (assignability, convertability, indexing, mathematical operations) attached to the various generic types in addition to requiring that the concrete types all match. It should, however, enable things like the coveted Equaler interface as in Listing 3.

Listing 3: Equaler interface.

```go
type Value generic
type Equaler interface {
  Equals(other Value) bool
}

// Search returns the index of the first element for which list[idx].Equals(val)
// returns true, or -1 if no matches are found.
func Search(list []Equaler, val Value) int {
  for i, v := range list {
    if v.Equals(val) {
      return i
    }
  }
  return -1
}

// SliceEqual returns true if all elements of a and b are Equal according to the
// Equals() method of the slice value type.
func SliceEqual(a, b []Equaler) bool {
  if len(a) != len(b) {
    return false
  }
  for i, v := range a {
    if !v.Equals(b[i]) {
      return false
    }
  }
  return true
}

// Integer implements Equaler.
type Integer int
func (i Integer) Equals(j Integer) bool {
  return i == j
}

// Compile-time check.
var _ Equaler = Integer(0)
```

3 Advantages

3.1 Simplicity and Orthogonality

I think this is one of the simpler generic type proposals possible. It does not change the semantics of any code, and only adds a single keyword. I believe that it is orthogonal enough to the type system that it wouldn't require much, if any, discussion in the specifications beyond a section describing generic types.

In other words, I don't think detailed discussions about the interaction with functions, interfaces, etc. would need to be discussed in *their* respective sections.

3.2 Implementations

This proposal has the advantage that the implementation suggested above isn't the only possible implementation. In rereading it, it could also be implemented by replacing all generics with interfaces and surrounding them with runtime type checks, or at runtime in an interpreter. The semantics are simple enough that the implementation itself could be done in any number of ways.

4 Drawbacks

4.1 Compilation

This proposal definitely incurs a compile-time penalty, and will increase object sizes. It also has the potential to create many specializations of generic functions without the programmer noticing, which will increase binary sizes. It will probably be quite difficult to combine identical specializations of the same generic entity when they are included in a binary through separate objects. As mentioned earlier, it also reduces the opacity of object files in general, as generic entities would be relatively easy to scrutinize and replicate.

4.2 Execution

The execution penalty of the specializations would be no higher than the equivalent non-generic entity, but compared to a reflection- or run-time-specialized generic approach could potentially perform worse (due to caching) if the number of specializations of an entity is high.