Summary of Go Generics Discussions
[Work in Progress]

Note:
If you notice some error, please create a suggestion in the doc.

Things to look for: typos, unclear arguments, highly biased statements, structure problems, missing sections, additional real-world examples (especially in the problems section), missing generics approaches.

Motivation

The generics discussion keeps turning up on the forum with the same points being argued over and over and over and over again. This document tries to summarize all the details, arguments for and against generics.

This is not a proposal nor is it here to argue for either side of the debate, it is designed to collect all the arguments. This is here so that people can go over the relevant points without digging through hundreds of forum posts and repeating the same statements, questions and answers.

Guidelines

To keep the arguments and examples to the point there are few helpful rules:

- No abstract examples/arguments. These cause the discussion to lose focus and make examples harder to follow. The example/argument must be traceable to a real-world problem - Go is intended to solve real problems not imaginary ones.
- Examples must show the full complexity of the problem domain. Simplified examples trivialize the problems and the solutions intended to solve those simplified examples may not work for the complex problems.
- Examples of problematic Go code must be the “best way” of writing the code in Go - if it can be improved then the improved version should be used instead.
- Arguments must be straight to the point and as concise as possible.

Disclaimer

***This document is a distillation of convoluted generics discussions and it is not an official opinion of the Go team, although it does incorporate their opinions*.**

Overview

At the heart of the problem with generics are these points (from http://research.swtch.com/generic):

1. (The C approach.) Leave them out. This slows programmers. But it adds no complexity to the language.
2. (The C++ approach.) Compile-time specialization or macro expansion. This slows compilation. It generates a lot of code, much of it redundant, and needs a good linker to eliminate duplicate copies. The individual specializations may be efficient but the program as a whole can suffer due to poor use of the instruction cache. I have heard of simple libraries having text segments shrink from megabytes to tens of kilobytes by revising or eliminating the use of templates.
3. (The Java approach.) Box everything implicitly. This slows execution. Compared to the implementations the C programmer would have written or the C++ compiler would have generated, the Java code is smaller but less efficient in both time and space, because of all the implicit boxing and unboxing. A vector of bytes uses significantly more than one byte per byte. Trying to hide the boxing and unboxing may also complicate the type system. On the other hand, it probably makes better use of the instruction cache, and a vector of bytes can be written separately.

The generic dilemma is this: *do you want slow programmers*, *slow compilers and bloated binaries*, or *slow execution times*?

And from the Go FAQ (http://golang.org/doc/faq#generics):

> *Generics may well be added at some point.* ***We don't feel an urgency for them****, although we understand some programmers do.*
>
> *Generics are convenient but they come at a cost in complexity in the type system and run-time. We haven't yet found a design that gives value proportionate to the complexity, although we continue to think about it. Meanwhile, Go's built-in maps and slices, plus the ability to use the empty interface to construct containers (with explicit unboxing) mean in many cases it is possible to write code that does what generics would enable, if less smoothly.*
>
> ***This remains an open issue.***

Problems

As with any feature it should solve a set of problems, if there are better ways of solving the problems, those approaches should be used instead. This section tries to give an overview of problems that “generics” intends to solve and alternative approaches to solving them. It tries to also show pros/cons for using generics to solve those problems.

The examples for alternative solutions are marked with **e.g.**

Generic Data Structures

This is the usual problem of needing to re-implement data structure for a concrete type.

Cases:

- Set
- Tree
- Matrix
- Graphs

Pros for using generics:

- Faster to use an already existing package
- Don’t need to re-implement hard-to-get-right structures.
- Type-safety

Cons for using generics:

- The generic structures are less useful than a concrete one. The interface can be tailored to a particular need e.g. having *EditorBuffer* instead of *GapBuffer* and *EditorBuffer.MoveCursor* instead of *GapBuffer.MoveGap*.
    - Obviously with using generics the generic structure can be embedded to a structure and reduce the amount of things person has to reinvent.
        - The amount of reinvention depends on how many third-party packages exist.
        - *Of course given that people take easier way out, it is likely that won’t happen and they use directly the generic structure instead.*
- Generic structures can be less optimized than a concrete solution. A package for a concrete problem can be more optimized than a generic/abstract package. (e.g. optimizing *EditorBuffer* vs *GapBuffer*)

Alternative solutions:

- use simpler structures instead of complicated structures
    - map/slice suffice for most of the cases encountered in programming. *(based on some community opinions* https://groups.google.com/d/msg/golang-nuts/smT_0BhHfBs/MWwGIB-n40kJ*)*

    - Fancy algorithms are slow when n is small, and n is usually small. - *Rob Pike*
    - **e.g.** use *map[int]struct{}* instead of *Set*
    - **e.g.** use *[]Item* instead of *List*
- code generation
- misc
    - **e.g.** https://github.com/golang-collections uses []byte to store data internally

Generic Algorithms

This is the need to implement code where the underlying algorithm is the same across multiple types.

Cases:

- sort
- best / min / max
- image processing
    - convolution
- signal processing
    - FFT / DFT
    - filters
    - Z transform, Laplace transform
    - correlations
- map intersect
- mathematical operations on classes of numbers (https://groups.google.com/d/msg/golang-nuts/PYJayE50JZg/toCYpU0qMAIJ)

Pros for using generics:

- Faster to use an already existing package
- Don't need to re-implement hard-to-get-right algorithms.
- Type-safety

Cons for using generics:

- Generic algorithms can be harder to follow than their non-generic counterparts.
- Generic algorithms can be less optimized than a concrete solution. A package for a concrete problem can be more optimized than a generic/abstract package.

Alternative solutions:

- re-implement the code
- use simpler algorithms
    - Fancy algorithms are slow when n is small, and n is usually small. - *Rob Pike*
- use interfaces to solve the abstraction problems
    - **e.g.** sort (http://golang.org/pkg/sort/) shows how to implement sorting for slices
    -

        - it needs to implement two additional lines (Len/Swap) compared to the non-generic solution
        - it is slower than a solution without interfaces
    - **e.g.** image (http://golang.org/pkg/image/), shows how one interface can be used for multiple underlying image formats
    - **e.g.** A* (https://gist.github.com/egonelbre/10578266)
        - contains typecasts *(code generation could be used to remove the typing problem)*
- use reflect
    - **e.g.** slice https://github.com/bradfitz/slice for sorting with a single function
- code generation
    - **e.g.** gob (http://golang.org/pkg/encoding/gob/) package contains generated code for encoding/decoding
        - http://golang.org/src/encoding/gob/decgen.go
        - http://golang.org/src/encoding/gob/dec_helpers.go
    - **e.g.** slice (modified https://github.com/egonelbre/slice) generated code for specific struct sizes

Functional Code

These are the usual higher-order functions such as map, reduce (fold), filter, zip etc.

Cases:

- typesafe data transformations: map, fold, zip
-

Pros for using generics:

- Concise way to express data transformations.

Cons for using generics:

- The fastest solution needs to take into account when and which order to apply those transformations, and how much data is generated at each step.
- It is harder to read for beginners.

Alternative solutions:

- use for loops and usual language constructs

Language Extensions

These are the problems of implementing things such as Rx (see http://reactive-extensions.github.io/learnrx/), LINQ (see http://msdn.microsoft.com/en-us/library/bb397926.aspx):

Pros for using generics:

- Adds a lot of expressivity to the language.
- Allows more type-checking during runtime.

Cons for using generics:

- Implementing such things using generics adds a lot of complexity
- Code won't look idiomatic.
- Language extensions need to make different trade-offs compared to the actual language and adjusting such trade-offs via generics is difficult.

Alternative solutions:

- DSLs provide a clearer way of implementing different paradigms, this also means that it can much more easily express the intent of the solution or optimize it based on the case.
    - SQL, Datalog vs LINQ - SQL and Datalog provide a clearer way to query and combine data compared to LINQ; of course there is a language bridge that must be crossed when using a DSL.
    - **e.g.** ivy language - http://godoc.org/robpike.io/ivy implements a custom language designed for APL style computations
- Code generation - converting a DSL into Go can be an easy way to bridge the gap between language.
- Use interface{}
    - Using interface{} doesn't guarantee type safety.
    - **e.g.** http://godoc.org/github.com/coocood/qbs

Alternatives

There are of course alternatives to using generics. This section tries to cover alternatives to using generics - with their own pros/cons.

Re-implement the code / copy-paste

This adds overhead for the programmers and it can be error-prone to implement a single thing multiple times. (See code-generation to avoid such problems.) Of course the re-implemented and copy-pasted code needs to be maintained.

If the code cannot be easily generated then specialized linting and vetting tools can be helpful.

Use interfaces

Sometimes interfaces can be used to support the genericness of a data-structure/package.

Of course this can introduce type-casts that could cause errors - however there is no data to show how rare/frequent these types of errors are. Using interfaces adds overhead in data-size and the code is slower than the specialized version.

Examples:

- sort - http://golang.org/pkg/sort/
- heap - http://golang.org/pkg/container/heap/
- A* - https://gist.github.com/egonelbre/10578266

Use reflect

Reflection can be used in some cases to provide a generic implementation with type-safety. The reflection adds overhead and some type-casts may still be necessary.

Reflection adds overhead and the code is slower than the specialized version. Also using reflection is not trivial.

Examples:

- sorting over arbitrary types: https://github.com/bradfitz/slice

Code generation

Code generation can provide a lot of similar features as generics - although it is less convenient. But also, code generation offers more possibilities and uses than generics.

Tooling:

- go generate
  - https://golang.org/cmd/go/#hdr-Generate_Go_files_by_processing_source
  - http://blog.golang.org/generate
- gen
  - https://clipperhouse.github.io/gen/
- gotemplate
  - https://github.com/ncw/gotemplate

Generics approaches

This lists the possible ways of implementing generics - and their pros/cons with a link to a proposal or prototype. The word “proposal” is used very loosely here, it can also be a simple link to a thread discussing that approach. The word “prototype” is also loosely used here, it can even be some code that transpiles to Go. There isn’t one generic that fits all needs, so figuring out which kind of generics is ideal we also need to know the different ways of implementing generics.

“Generics” must keep in mind that it must nicely integrate with all of existing Go codebase. I.e. interfaces, packages, reflection, structures etc.

There are of course constraints that a good “generics” must adhere to, to be integrated to Go:

Constraints:

- it must take into account interfaces, structs, func types
- must compile fast
- must not generate code at runtime - Go needs to work in environments where run-time code generation is not allowed (mentioned in https://docs.google.com/document/pub?id=1IXHI5Jr9k4zDdmUhcZImH59bOUK0G325J1FY6hdelcM)

General

Statements that apply to all the generics approaches:

Pros:

-

Cons:

- Problems with cyclic dependencies

Go

Built-in generic types, i.e. the current Go approach (*generic types map/chan/slice*).

Pros:

- standardized types complex types
- smaller language/compiler (if there aren’t many generic types)
- language constructs can be optimized for these types
- the code is more concrete (because users can build less abstractions)

Cons:

- each generic type adds complication to compiler
- each generic type makes the language more complicated

- the generic types must perform well in lots of cases
- the language is less flexible (because users can build less abstractions)

C++

Type/function level specialization or macro expansion.

Pros:
- The individual specializations can be efficient.

Cons:
- This slows compilation.
- It generates a lot of code, much of it redundant, and needs a good linker to eliminate duplicate copies.
- Program as a whole can suffer due to poor use of the instruction cache.

Unknowns:
-

Proposals:
- http://thwd.github.io/ts/

Prototypes:
-

Java

Box everything implicitly.

Pros:
- It probably makes better use of the instruction cache, and a vector of bytes can be written separately.

Cons:
- Slows execution.
- Compared to the implementations the C programmer would have written or the C++ compiler would have generated, the Java code is smaller but less efficient in both time and space, because of all the implicit boxing and unboxing.
- A vector of bytes uses significantly more than one byte per byte. Trying to hide the boxing and unboxing may also complicate the type system.

Unknowns:
-

Proposals:
-

Prototypes:
-

Modula-3

Package level specialization.

Pros:

- Easier to understand.
- Very little syntax needed to support
- Harder to abuse
    - packages provide a larger unit of reuse, which means that the reused part has to have some significant importance, such as “tree”, “set”, “astar”
- Could be integrated to the std library without breaking compatibility.

Cons:

- While functional idioms like map or filter are possible, doesn’t allow for clean chaining due to need to qualify each specialized function call with a package name.

Unknowns:

- compilation speed
- runtime speed
-

Proposals:

- https://groups.google.com/d/msg/golang-nuts/JThDpFJftCY/1MqzfeBjvT4J
- https://groups.google.com/d/msg/golang-nuts/y3LqthbBiuY/tJZw232o7ggJ
-

Prototypes:

- https://github.com/champioj/geno
- http://bouk.co/blog/idiomatic-generics-in-go/
-

C# / .NET

Reified generics. *(Specializes for primitives, boxes for large structs and references.)*

Blocker:

- Uses runtime code generation. (http://msdn.microsoft.com/en-us/library/f4a6ta2h.aspx)

Java/C++ mix

Specialize for primitives, box for larger values.

Ada

D

Haskell / Rust

Haskell's parametrically polymorphic types

Pros:

-

Cons:

- hard to fit into simpler language (https://groups.google.com/d/msg/golang-nuts/smT_0BhHfBs/MWwGlB-n40kJ)

Unknowns:

-

Proposals:

-

Prototypes:

MLton / ML / OCaml

Parametric polymorphism and generic modules (functors).

Sather

BETA

CLU

Automatically inferred primitives

Automatically inferred typing for functions such as slice by using a untyped slice in function definition.

Pros:

- Relatively easy to understand.
- Little syntax needed to support
- Could be integrated to the std library without breaking compatibility.

Cons:

- Only supports the primitive specialization
- Overuse may lead to harder to understand code
- Complexity of compiler / type inference engine
- New code won't be accepted by older tools

Unknowns:

- compilation speed (due to type inference)
- code bloat

Proposals:

- https://docs.google.com/document/d/1sOjHJY1uAN2plaxEgHUNeT7C1xDsfLmxCVcWpmOG2CQ

Prototypes:

- https://github.com/anlhord/go - contains a working parser

Generics Syntax

The discussion about which syntax to use is omitted at this moment. Which exact syntax to use adds little value to the main problem of which generics approach to use. This problem can also introduce a lot of unnecessary bike-shedding. The discussion of this subject is suspended until other sections have sufficient quality.

Additional Reading

[todo: organize]

https://groups.google.com/forum/#!msg/golang-nuts/smT_0BhHfBs/MWwGlB-n40kJ
https://groups.google.com/d/topic/golang-nuts/PYJayE50JZg/discussion