Summary of Go Generics Discussions

living document

Note: If you notice any errors, please request access and 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.

Due to griefing, this document has been marked as view-only, if you wish to contribute, please request for access.

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, as well as the arguments for and against generics. Things will be added as new comments or suggestions arise.

For an introduction to what generics are, please see Generic Programming in Wikipedia.

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. If you notice something missing, please add it.

Guidelines

To keep the arguments and examples to the point there are a 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.**
- **Arguments should take the point of view of an average programmer** - not the über-programmer who doesn't make design mistakes.

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, with minor changes):

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.
3. (The Java approach.) Use reference types for everything. 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 the additional memory overhead for primitives. A vector of bytes uses significantly more space than one byte per byte. Trying to hide it 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*?

Obviously, these points do not cover the full design space of generics. For the full list see “Generics Approaches” chapter.

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.***

For more information:
- https://github.com/golang/go/issues/15292.
- https://go.googlesource.com/proposal/+/master/design/go2draft-generics-overview.md

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 show pros/cons for using generics to solve those problems. Pros/Cons also try to go a step further in asking whether this problem should be solved (with Generics) in the first place.

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:

- Sets
- Trees
- Matrix
- Graphs
- Hash tables
- Thread-safe variants of data-structures

Pros:

- Faster to use an already existing package
- Don’t need to re-implement hard-to-get-right structures.
- Type-safety
- Easy way to reuse and share code for different types.
- Easy way to enforce consistent API across different types.
- There’s more incentive to optimize generic parts as they can have a larger impact.
- You need to write code only to your specific needs and not the generic part.

Cons:

- Generic structures tend to accumulate features from all uses, resulting in increased compile times or code bloat or needing a smarter linker.
- Generic structures and the APIs that operate on them tend to be more abstract than purpose-built APIs, which can impose cognitive burden on callers. e.g. having *EditorBuffer* instead of *GapBuffer* and *EditorBuffer.MoveCursor* instead of *GapBuffer.MoveGap*.
    - Of course, many almost similar API-s is not better.
- In-depth optimizations are very non-generic and context specific, hence it’s harder to optimize them in a generic algorithm. (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/MWwGlB-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
- consistent API
    - can also be enforced via linters and checkers
        - **e.g.** http://play.golang.org/p/4bCOV3tYz7
    - generated code can be used to create a consistent API

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

Cases:
- sort / reverse
- best / min / max
- image processing
    - convolution
- signal processing
    - FFT / DFT
    - filters
    - Z transform, Laplace transform
    - correlations
- map intersect
- set operations
- mathematical operations on classes of numbers (https://groups.google.com/d/msg/golang-nuts/PYJayE50JZg/toCYpU0qMAIJ)
- In general, algorithms that work on "generic data structures".

Pros:
- Faster to use an already existing package
- Don't need to re-implement hard-to-get-right algorithms.
- Type-safety
- Easy way to reuse and share code for different algorithms.
- Easy way to enforce consistent API across different algorithms.
- There's more incentive to optimize generic parts as they can have a larger impact.
- You need to write code only to your specific needs and not the generic part.

Cons:

- Generic algorithms tend to accumulate features from all uses, resulting in increased compile times or code bloat or needing more advanced linker to counter it.
    - **e.g**. boost Dijkstra shortest path
- In-depth optimizations are very non-generic and context specific, hence it's harder to optimize them in a generic algorithm.

Alternative solutions:

- re-implement the code
- use simpler algorithms
    - Fancy algorithms are slow when n is small, and n is usually small. - *Rob Pike*
        - In some areas, n is usually very large – examples include image and video processing and computational science, as well as big data.
- use interfaces to solve the abstraction problems
    - **e.g.** sort (http://golang.org/pkg/sort/) shows how to implement sorting for slices -
        - interfaces cannot be inlined and hence some performance optimizations aren't available
        - interface method calls are a few cycles slower than regular struct calls
        - Interface based solution is more verbose *(sort.Slice however avoids it partially)*
    - **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 and/or unsafe
    - **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
    - **e.g.** gen (https://github.com/clipperhouse/gen), where types are marked up in comments, and method code is generated via templates
- consistent API
    - can also be enforced via linters and vetters
    - generated code can be used to create a consistent API

Functional Code

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

Cases:

- type safe data transformations: map, fold, zip
  - Including parallel versions
- type generic error handling
- optional types
- result types
- continuation monad
- pure functions

Pros:

- Allows concise ways to express data transformations.
- Can allow to transform a given algorithm from sequential to concurrent with fewer changes in code
- Mutation of state by mistake in the input structs is much less likely to happen in functional code
- Provides greater reuse of composable functions

Cons:

- 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.
- Functional code is harder to debug with a step-by-step debugger

Alternative solutions:

- use for-loops and usual language constructs
- use interfaces:
  - **e.g.** https://github.com/robpike/filter
- error handling
  - **e.g.** http://blog.golang.org/errors-are-values

Domain Modeling

Generics can be used to specify domain constraints and structures.

Cases:

- Web frameworks *[example needed]*
- Application frameworks *[example needed]*
- Financial contracts (Simon Peyton Jones' paper)

Pros:

- Aids domain modeling *[example needed]*
- Type-safety
- Reduces boilerplate

Cons:

-

Alternative solutions:

-

Language Extensions

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

Pros:

- Adds a lot of expressivity to the language.
- Allows more type-checking during compile-time, compared to using *interface{}*.
- Easier and safer to convert some algorithms from sequential to concurrent.

Cons:

- Language extensions make the code less consistent, requiring learning different dialects.
- Language extensions need to make different trade-offs compared to the actual language and adjusting such trade-offs via generics is difficult.
- It is easy to introduce different ways of doing the same thing which will require more learning. For example, if there are N different ways of iterating things in M code bases, then you need to understand and keep in mind all the differences between N approaches.

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.
    - Difficulty of implementing a DSL prevents proliferation of poorly designed languages, but also restricts people from improving DSL-s.
    - **e.g.** go-linq - https://github.com/ahmetb/go-linq
    - **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 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. Similarly to code-generation this suffers from potential code-bloat.

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 can be less convenient. But, also, code generation has more uses than generics. Code generation may result in large binaries when a lot of code is generated. Code generation requires support in each build system one happens to use Go in. In addition code generation may require learning a separate templating language of the code generation tool.

Examples:

- gob.Encoder, gob.Decoder helpers:
  - https://golang.org/src/encoding/gob/decgen.go -> https://golang.org/src/encoding/gob/dec_helpers.go
  - https://golang.org/src/encoding/gob/encgen.go -> https://golang.org/src/encoding/gob/enc_helpers.go
- Faster swap in bradfitz/slice package:
  - https://github.com/egonelbre/slice/blob/master/genstruct.go -> https://github.com/egonelbre/slice/blob/master/struct.go
- BLAS float32 generation from float64 implementation
  - https://github.com/gonum/blas/blob/master/native/level1single.go

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)
- go-inline (https://github.com/sasha-s/go-inline)
- goast (https://github.com/go-goast/goast)
- gomacro (https://github.com/gomacro/)
  - code generation + unsafe to break type system
- genny (https://github.com/cheekybits/genny)
  - code generation - generic code is valid, testable Go code
- Generic (https://github.com/taylorchu/generic)
  - Generate testable and type-checked go code at package-level
- Gengo (https://github.com/kubernetes/gengo)
  - Code parsers + template optimizations for code generation.
- ppgo (https://github.com/gobwas/ppgo)
  - using C preprocessor for go code generation

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, to figure out which kind of generics is ideal, we need to know the different ways of implementing generics.

“Generics” must keep in mind that it must nicely integrate with all of the 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:

- the type parameter must work with interfaces, structs, func types, chans, maps
- must compile fast
- must not generate code at runtime - Go needs to work in environments (e.g. GCP) where run-time code generation is not allowed (mentioned in https://docs.google.com/document/pub?id=1IXHI5Jr9k4zDdmUhcZImH59bOUK0G325J1FY6hdelcM)

The more detailed explanation can be found here https://github.com/golang/proposal/blob/master/design/15292-generics.md#what-we-want-from-generics-in-go

General
Statements that apply to all the generics approaches:

Pros:

-

Cons:

- Problems with cyclic dependencies

Unclassified

- If you wish to help out, please figure out how these languages should be classified or do they use something unique: Sather, BETA, CLU.

Built-in generics

This is the current Go approach. Instead of having user-defined types you have a set of common generic functions and types. (map/chan/slice)

Pros:

- standardized 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 fewer abstractions, such as Rx and LINQ

Type specialization

This is the approach that C++ and Rust use. It is also a mechanism (along with functors) by which generics are implemented in MLton. You specify a parameterized type or function and it can be specialized by another type.

Pros:

- The individual specializations can be efficient.

Cons:

- Slows compilation.
- It generates a lot of code, much of it redundant, and needs a good linker to eliminate duplicate copies.
    - *I have heard of simple libraries having text segments shrink from megabytes to tens of kilobytes by revising or eliminating the use of templates. (from https://research.swtch.com/generic)*
- Program as a whole can suffer due to poor use of the instruction cache.
- Interacts badly with dynamic polymorphism. For instance, virtual methods cannot have more type parameters than the enclosing class, since it would not be possible to compute the vtable size.

Unknowns:

-

Proposals:

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

Prototypes/Tools:

-

Implicit boxing

This the approach that Java uses. SML implementations (other than MLton), Haskell, and OCaml also use it. Similarly to "type level templates" you specify a parameterized type or function, but instead of specializing, you box any value that is used as the parameterization type.

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 implicit boxing and unboxing.
- A vector of bytes uses significantly more space than one byte per byte. Trying to hide the boxing and unboxing may also complicate the type system.
- The abstraction is often leaky because generic types are erased during compilation. E.g. runtime introspection does not return the expected types and type-safety can be broken easily at runtime (where a List<String> and a List<Integer> are both List'<Object>').

Unknowns:

-

Proposals:

-

Prototypes/Tools:

-

Package templates

Instead of specifying an individual type for specialization, the whole package is generic. You specialize the package by fixing the type parameters when importing. Similar scoping has been used in Modula-3, OCaml, SML (so-called “functors”), and Ada.

Pros:

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

Cons:

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

Notes:

- compilation speed in the simplest case would be the same or faster than including a package
- code-bloat reduction between specialized package can increase compile time

Unknowns:

- code-bloat handling
- init handling
- how to do generic specialization for a particular type
    - set<bool> and set<int> could use widely different internal implementations.
    - could be done with build flags to include/exclude files based on type parameters

Proposals:

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

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

Prototypes/Tools:

- https://github.com/champioj/geno
- http://bouk.co/blog/idiomatic-generics-in-go/
- https://github.com/sasha-s/go-inline
- https://github.com/ncw/gotemplate

Type alias rebinding

This is somewhat similar to Package Specialization, except you rebind type aliases to create specific types and that can be specified at type level.

Pros:

- Easier to understand.
- Very little syntax needed to support

Cons:

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

Notes:

- compilation speed in the simplest case would be the same or faster than including a package
- code-bloat reduction between specialized package can increase compile time

Unknowns:

- code-bloat handling
- init handling
- how to do generic specialization for a particular type
    - set<bool> and set<int> could use widely different internal implementations.
    - could be done with build flags to include/exclude files based on type parameters

Proposals:

- https://twitter.com/__dc0d__/status/925754379684081665

Prototypes/Tools:

- https://github.com/dc0d/goreuse

Mix boxing and specialization

This is a compromise between boxing and specialization. For small types you specialize and for large types you box.

Parameterized template scopes

This is an approach used by D and Ada. Instead of specifying a single type, you specify a parameterized scope. It can be seen as an approach between "parameterized type specialization" and "package specialization".

Pros:

- D and Ada show that this approach can be performant

Cons:

-

Unknowns:

-

Proposals:

-

Prototypes/Tools:

Qualified types

This is an approach demonstrated by Jai. Instead of having free-floating type names use the specialized value name as a qualifier for parametric types.

Pros:

- Gets rid of repetition and naming issues associated with scoped type names.

Cons:

-

Unknowns:

-

Proposals:

- https://play.golang.org/p/5rPlhbD-9Q

Prototypes/Tools:

Sum Types

Sum types or variant types allow specifying constraints what can be passed to a function, solving some of the type-checking issues. (https://golang.org/doc/faq#variant_types)

Pros:

-

Cons:

- Problems defining zero value of a sum type.
- Very similar to how interfaces already work.
- Solve a limited subset of generics problems.

Proposals:

- Reddit discussion
- https://github.com/golang/go/issues/19412
- https://manishearth.github.io/blog/2018/02/01/a-rough-proposal-for-sum-types-in-go/

Compile-time function evaluation

This approach is used by Zig, Jai, D, Nim and by most Lisp dialects. Instead of providing complicated generics approaches, have the possibility to evaluate a function that generates code at compile time. This can work together with the other “generics” approaches.

Pros:

- Compile time function evaluation makes clearer how code gets generated for different template instantiations (as compared to templating tricks).
- Jai shows that this approach can be performant and powerful
- Zig shows that this approach can be relatively readable

Cons:

- Needs an interpreter that can run at compile-time
- Impact on build time and binary sizes similar to type specialization
- Has security implications
    - Can be executed in a restricted environment with limited resources or bounds.

Unknowns:

-

Proposals:

- https://github.com/golang/proposal/blob/master/design/15292/2016-09-compile-time-functions.md

Prototypes/Tools:

- gomacro - https://github.com/cosmos72/gomacro

Polymorphic function specialization

Polymorphic functions where the specialization is determined by the call-sites. Used by all ML-family languages, C++, Rust, D, and many others.

Pros:

- Relatively easy to understand.
- Little syntax needed to support

Cons:

- Doesn't handle generic data-structures
- restrictions in how it can be used
    - type manipulation (e.g. Mul(Matrix[4,2]{}, Matrix[2,3,4]{})
-

Unknowns:

- compilation speed (due to type inference)

Proposals:

- https://docs.google.com/document/d/1zhZfI46cDTuj9KvVCjWDKe84oNfiJKRV2svfVwgegUc

Out-of-bounds boxing

Each generic function will have a stub for each specialization and one additional generic invocation. The generic invocation contains a type parameter. For any place requiring variance there is a type switch to different behavior. Any specialized call will invoke the stub, any generic call will call the generic version.

Pros:

- avoids code-bloat

- avoids boxing of values

Cons:
- reflection and GC may need more meta-information than this scheme provides, which either can make it impossible to use or can significantly complicate their implementation

Unknowns:
- compilation speed
- implementation difficulty
- performance

Discussions:
- https://groups.google.com/forum/#!topic/golang-nuts/wSu7VjLdVXs

Witness Tables

This is the approach used by Swift and is explained in https://www.youtube.com/watch?v=ctS8FzqcRug. It uses reified type metadata to allow generic code to abstract over the memory layout of data types and avoid boxing. Generic data is packed the same as it would be without generics. Generic functions are compiled once and separate compilation is possible, so the caller does not need to have the implementation available. At runtime, zero or more "witness tables" are passed to the generic function to allow generic code to manipulate the concrete data indirectly using function pointers.

Pros:
- avoids code-bloat
- avoids boxing of values
- does not require multiple compilations of generic code

Cons:
- reflection and GC may need more meta-information than this scheme provides

Unknowns:
- compilation speed
- implementation difficulty
- performance

Discussions:
-

Inferred primitive specialization

Automatically inferred typing for functions such as slice by using an 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
- https://github.com/golang/go/issues/21132

Prototypes/Tools:

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

Constrained specialization

Include explicit way to specify constraints on types either through interfaces or special constraint declarations.

Pros:

- Better error messages
- Better code documentation
-

Cons:

- Needs extra syntax (unless interfaces are used)
-

Unknowns:

-

Proposals:

- https://github.com/surlykke/Go-Generics-with-constraints

Prototypes/Tools:

-

Profile-time specialization

Instead of runtime specialization as in C# and .NET. You do a separate profiling (when needed) to specialize particular cases, this profiling creates hints for the compiler. Something similar can be found in StrongTalk as an inlining-database.

Pros:

- Could be integrated without changing the language
- Allows to select appropriate trade-offs based on profiling
- Since profiling is a separate pass, it does not cause significant compilation overhead, unless requested.

Cons:

- To use, you must do a separate profiling pass. However, only necessary for optimal code.
- The profile/hints must be created somehow. *One could imagine, running the application in a special mode; or a separate benchmark must be created; or a separate tool analyses the code and produces the profile.*

Hindley-Milner Type System

This is the approach that Haskell and Rust take.

Pros:

- Allows more constraints and abstract definitions

Cons:

- hard to fit into simpler language (https://groups.google.com/d/msg/golang-nuts/smT_0BhHfBs/MWwGIB-n40kJ)
- Producing good error messages with Haskell style type-system is still an open problem.
- Type checking Hindley-Milner type systems is worse case exponential unless the care is made to prevent pathological inputs.

Runtime specialization

This is the approach used by C# and .NET. During runtime you specialize the types that are often used allowing better decision making with regards to optimizations.

Blocker:

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

Generics Research

Research on generics programming

1. http://www.crest.iu.edu/publications/prints/2003/comparing_generic_programming03.pdf
2. https://arxiv.org/pdf/0708.2255.pdf
3. http://www.cs.cornell.edu/~yizhou/papers/genus-pldi2015.pdf
4. https://arxiv.org/pdf/0708.2255.pdf
5. http://www.cs.cornell.edu/andru/papers/familia/familia.pdf
6. https://www.cs.cornell.edu/~ross/publications/shapes/shapes-pldi14.pdf
7. https://www.cs.cmu.edu/~rwh/theses/morrisett.pdf

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 bikeshedding. The discussion of this subject is suspended until other sections have sufficient quality.

Additional Comments

Russ Cox comment in a Hacker News thread:
Source https://news.ycombinator.com/item?id=9622417

> I'm sorry, but no: Generics *are* a technical issue and *are not* a political one. The Go team is not against generics per se, only against doing things that are not well understood and/or don't work well with Go.
>
> There are deep technical issues that must be solved to fit the idea of generics into Go in a way that works well with the rest of the system, and we don't have solutions to those. I wrote on my blog about one issue years ago (http://research.swtch.com/generic), but there are others too. Even supposing you get past the problem on that page, the next thing you would run into is how to allow programmers to omit type annotations in a useful, easy-to-explain way. As an example, C++ lets you write make_pair(1, "foo") instead of make_pair<int, string>(1, "foo"), but the logic behind inferring the annotations takes pages and pages of specification, which doesn't make for a particularly understandable programming model, nor something the compiler can easily explain when things go wrong. And then there's a princess in another castle after that one I am sure.
>
> We have spoken to a few true experts in Java generics and each of them has said roughly the same thing: be very careful, it's not as easy as it looks, and you're stuck with all the mistakes you make. As a demonstration, skim through most of http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.ht... and see how long before you start to think "was this really the best way to do this?" (For example, http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypePa..., but note that the latter page is only one part of the FAQ, not the entire FAQ.)
>
> To be very clear, we acknowledge this fact: there are definite disadvantages to not having generics. You either use interface{} and give up compile-time checking or you write code generators and complicate your build process. But there are also definite disadvantages to generics as implemented in existing languages, and there is a very large advantage to not compromising today: it makes adopting a better solution tomorrow that much easier.
>
> As I said in the interview at http://www.pl-enthusiast.net/2015/03/25/interview-with-gos-r...:
>
> > Go is more an engineering project than a pure research project. Like most engineering, it is fundamentally conservative, using ideas that are proven and well understood and will work well together. The research literature's influence comes mainly through experience with its application in earlier languages. For example, the experience with CSP applied in a handful of earlier languages—Promela, Squeak, Newsqueak, Alef, Limbo, even Concurrent ML—was just as crucial as Hoare's original paper to bringing that idea to practice in Go. Programming language researchers are sometimes disappointed that Go hasn't picked up more of the recent ideas from the literature, but those ideas simply haven't had time to pass through the filter of practical experience.
>
> I believe generics is one of those ideas. It certainly needs at least one more iteration, possibly a few more than that.