Go 2 Generics Feedback

This page is meant to collect and organize feedback about the Go 2 contracts (generics) draft design.

A prototype implementation of the syntax can be found in https://go.dev/cl/149638 which may be patched on tip of the Go repo.

Please post feedback on your blog, Medium, GitHub Gists, mailing lists, Google Docs, etc. And then please link it here.

As the amount of feedback grows, please feel free to organize or reorganize this page by specific kind of feedback.

Supporting

- Roger Peppe, "Go contracts use case: generic mgo", September 2018
- Richard Fliam, "Go2 Generics Let You Construct the Natural Numbers", August 2018
- Roger Peppe, "Go generics at runtime", Part 1, Part 2, September 2018

Supplemental (supporting with modifications)

- Matt McCullough, "Towards Clarity: Syntax Changes for Contracts in Go" and "Angle Brace Delimiters for Go Contracts", May 2020
- Gert Cuykens, "Complete separation of generic syntax form regular Go code", Januari 2020
- Court Fowler, "Thoughts from a lazy programmer on the updated design", September 2019
- Andrew Phillips, "Example Types as Contracts", August 2019
- Alexey Nezhdanov, "A syntax simplification proposal", August 2019
- Bryan Ford, "Are Only Type Parameters Generic Enough for Go 2 Generics?", July 2019
- Tom Levy, "Go 2 Generics Feedback", June 2019
- Ole Bulbuk, "Why Go Contracts Are A Bad Idea In The Light Of A Changing Go Community", April 2019
- Tony Mottaz, "Go generic types and import injection", March 2019
- Gustavo Bittencourt, "Contracts only for Generic Types", March 2019
- David Heuschmann, "Problems With Using Parantheses for Type Argument Lists", February 2019
- Gustavo Bittencourt, "Contract with methods", February 2019
- Chris Siebenmann, "Go 2 Generics: Contracts are too clever", November 2018
- Chris Siebenmann, "Go 2 Generics: A way to make contracts more readable for people", November 2018
- Chris Siebenmann, "Go 2 Generics: Interfaces are not the right model for type constraints", November 2018
- alanfo, "Proposed changes to the Go draft generics design in the light of feedback received", October 2018
- Andy Balholm "Enumerated and structural contracts", October 2018
- Burak Serdar "Types are contracts", October 2018

- Patrick Smith, "Go generics for built-in and user-defined type parameters", September 2018
- Jacob Carlborg, "Go 2 draft D corrections", September 2018
- alanfo, "A simplified generics constraint system", September 2018
- Paul Borman, "Simplifying syntax", September 2018
- mrwhythat, "Go 2 generics draft notes", September 2018
- Roger Peppe, "Operator overloading", September 2018
- Peter McKenzie, "Alternative generics syntax", September 2018
- Ted Singer, "The design goal for syntax is to help humans read", September 2018
- alanfo, "Suggested amendment to Go 2 generics draft design", September 2018
- Dean Bassett, "If we're going to use contracts, allow unary + on string, September 2018"
- Kevin Gillette, "Regarding the Go 2 Generics Draft", September 2018
- jimmy frasche, "Embedding of type parameters should not be allowed", August 2018
- Javier Zunzunegui, "Compiling Generics", August 2018
- Liam Breck, "Please Don't Mangle the Function Signature", August 2018
- DeedleFake, "Feedback for Go 2 Design Drafts", August 2018
- Roberto (empijei) Clapis, "Hard to read syntax", August 2018
- Dominik Honnef, "My thoughts on the Go Generics Draft", August 2018

Counterproposals

- dotaheor, "Declare generics as mini-packages with generic parameters", August 2020
- Beoran, "Hygienic Macros", June 2019
- Randy O'Reilly, "Generic Native Types", June 2019
- Michal Štrba, "Giving up restricting types", May 2019
- Eric Miller, "Simple generics using const struct fields", March 2019
- dotaheor, "A solution to unify Go builtin and custom generics", February 2019
- Quentin Quaadgras, No syntax changes, 1 new type, 1 new builtin, December 2018
- Andy Balholm, "Contracts and Adaptors", November 2018
- Dean Bassett, "Contract embedding", October 2018
- Patrick Smith, "Go Generics with Adaptors", October 2018
- Ian Denhardt, "Go Generics: A Concrete Proposal Re: Using Interfaces Instead Of Contracts.", October 2018
- Arendtio "Generics in Go inspired by Interfaces", September 2018
- Scott Cotton, "Draft Proposal Modification for Unifying Contracts and Interfaces" (diff), September 2018

- ohir, "CGG, Craftsman Go Generics", September 2018
- ~~Dean Bassett, "Using interfaces instead of contracts", September 2018~~
  I have made a second proposal ("contract embedding") listed further down that solves the issues with this one
- dotaheor, "Combine contract and code together and view generic as compile-time calls with multiple outputs", September 2018. (Updated from time to time)
- Aleksei Pavliukov, "Extend type and func keywords", September 2018
- Han Tuo, "Generic as a kind of types -- type T generic {int, float64}", September 2018
- Nate Finch, "Go2 Contracts Go Too Far", September 2018
- Roger Peppe, "Go Contracts as type structs", September 2018
- Axel Wagner, "Scrapping contracts", September 2018
- Matt Sherman "Generics as built-in typeclasses", September 2018
- Roger Peppe, "Revised generics proposal", September 2018
- Steven Blenkinsop, "Response to the Go2 Contracts Draft Design – Auxiliary Types", September 2018
- Dave Cheney, "Maybe adding generics to Go IS about syntax after all", September 2018
- Christian Surlykke, "Constraints for Go Generics, September 2018"
- Some Gophers on go-nuts, "Unifying Interfaces and Contracts", August 2018
- Roger Peppe, "Go generics feedback, August 2018
- Ruan Kunliang, "Package level generics", August 2018
- Emily Maier, "Getting specific about generics", August 2018

Against

- Tokyo Gophers, "Comments from Go 2 draft design feedback event", October 2018
- Jason Moiron, "Notes on the Go2 Generics Draft", September 2018
- Yoshiki Shibukawa, "Feedback for generics/contract proposals, September 2018"

Adding Your Feedback

Please format all entries as below.

- *Your Name*, "*Title*", *month year*

To make it easier to see new feedback. Please *make a Gist*. And also help to keep the list sorted in reverse-chronological order by including your new entry at the *top* of the category list.

Quick comments

- Chester Gould: The only problem with this proposal is that explicit contracts seem to only make the code more verbose which is against the goal of simple readable code. Instead of writing explicit contracts, using the actual code we write as a kind of "implicit contract" would be much more simple and elegant. An

example of this is shown here. I acknowledge that this is addressed here, but I disagree that explicit contracts are the solution to problem. It seems to me that contracts are very close to what interfaces provide and so the behaviour of interfaces should be extended to allow behaviour closer to contracts rather than adding an entire new type of statement to the language.

- Izaak Weiss: A lot of the discussion has focused specifically on how to implement contracts, or something like that. However, most of the "useful examples" don't require contracts; they only require parametric polymorphism. Writing a typesafe `Merge` or `SortSlice` is possible without contracts. And for the simpler contracts, we can implement them via higher order functions. A generic hashmap can be parametric over a type with a `Hash` method, or it can take a `func(K) int64` when it is constructed, and use that to hash its keys. If more functions are required, structs holding these functions can be declared as pseudo-contracts, and then those can be passed to the generic function. This makes Go's polymorphism simple, explicit, and leaves room for further innovation regarding contracts or other mechanisms in the future, while allowing most of the benefits of generic types to be realized right now.

- Christoph Hack: I just watched Alexandrescu's last talk The next big Thing. He states "Concepts are a waste of time" and proposes a completely different, far more powerful, direction (even in comparison to everything possible in C++ today). Go already has most required features, like reflection and testing if a type implements an optional interface. The only thing missing is code generation. For example, `json.Marshal` works fine by using reflection, but if it could also (optionally) generate code by implementing a Go function that gets called by the compiler automatically and runs regular Go code, we would have everything. It might sound crazy at first and toy examples might look verbose, but I think Alexandrescu has a point there. Just think about gqlgen vs. the other reflection based graphql-libs for example. Please watch his talk!

- Bodie Solomon: I find the generics design a bit confusing and opaque. Please consider integrating some concepts from Zig's beautiful comptime functions! The design of Go 2 generics is clever, but I feel it goes against Go's traditional tight coupling between simple runtime semantics and simple syntax. Additionally, one of the biggest problems of Go, which prevents it from being a viable competitor everywhere I might like to use it, is that I cannot be rid of the GC and runtime. It was my strong hope that Go 2 would introduce compile-time-only generics such that I could reliably avoid the use of dynamic interfaces where I don't want them, without relying on codegen. Unfortunately it looks like that will be decided by the compiler without my input. Please, at least, consider giving users the ability to constrain generics to compile-time-only resolution, perhaps as a property of a Contract, rejecting compilation of dynamic types to satisfy the contract.

- Dag Sverre Seljebotn: C++ has a huge problem with people abusing metaprogramming ("generics") to do compile-time metaprogramming. I really wished Go had gone down the path of Julia, which offers hygienic macros. Even if it is kept strictly at a compile-time barrier and no run-time code generation, this would at least avoid all the bad tendencies we see in the C++ world that comes from their templating system. Things you can do with generics you can usually pull off with macros too (e.g., `SortSliceOfInts = MakeSliceSorterFunctionMacro!(int)` could generate a new function to sort a slice of integers). Link: https://docs.julialang.org/en/v0.6.1/manual/metaprogramming/

- Maxwell Corbin: The issues raised in the Discussion and Open Questions section all could be avoided by defining generics at the package rather than the function or type level. The reason for this is simple: types can reference themselves, but packages can't import themselves, and while there are many ways to algorithmically generate more type signatures, you cannot do the same with import statements. A quick example of such syntax might be:

```
\\ list
package list[T]
```

```
type T interface{}

type List struct {
    Val T
    Next *List
}

// main
package main

import (
    il "list"[int]
    sl "list"[string]
)

var iList = il.List{3}
var sList = sl.List{"hello"}

// etc...
```

The syntax in the example is probably needlessly verbose, but the point is that none of the unfortunate code examples from the blog post are even legal constructions. Package level generics avoids the most abusive problems of meta-programming while retaining the bulk of its usefulness.

- Andrew Gwozdziewycz: The use of the word `contract` gives me pause due to it overloading "contract" as in Design by Contract. While the generics use case has some similarities with the "contracts" in DbC if you squint a bit, the concepts are quite different. Since "contracts" are an established concept in Computer Science, I think it would be far less confusing to use a different name like `behavior` or `trait`. The design document also suggests reasons why using `interface` is not ideal, though, Go's contract mechanism seems too obvious an extension of interfaces to disregard so quickly... If it can be done `interface setter(x T) { x.Set(string) error }` and `interface addable(x T, y U) { x + y }` seem quite natural to read and understand.
    - Russell Johnston: Agreed that it would be great to merge contracts and interfaces. Another way around the operator-naming problem might be to provide some standard interfaces for the operators, with bodies inexpressible in normal Go code. For example, a standard `Multipliable` interface would allow the `*` and `*=` operators, while a standard `Comparable` interface would allow `==`, `!=`, `<`, `<=`, `>=`, and `>`. To express operators with multiple types, these interfaces would presumably need type parameters themselves, for example: `type Multipliable(s Self /* this exists implicitly on all interfaces */, t Other) interface { /* provided by the language */ }`. Then user-written interfaces/contracts could use these standard identifier-based names, neatly sidestepping the issues mentioned in the design document around syntax and types.
    - Roberto (empijei) Clapis: I agree on this and on the fact that it should be clearer where to use interfaces and where to use contracts. Unifying the two would be great, as they try to address overlapping issues.
    - Kurnia D Win: I think `constraint` is better keyword than `contract`. Personally i like `type addable constraint(x T, y U) { x + y }` instead of merging with interface.

- Hajime Hoshi: I feel like the supposed proposal is too huge to the problems we want to solve listed at https://go.googlesource.com/proposal/+/master/design/go2draft-generics-overview.md . I'm worried this feature would be abused and degrade readability of code. Sorry if I am missing, but the proposal doesn't say anything about `go generate`. Wouldn't `go generate` be enough to the problems?

- Stephen Rowles: I find the method syntax hard to parse, as a human reading it, it might be clearer to use a different type of enclosing brackets for the type section, e.g. : Me too 👍 +1. Yet another 👍 +1(Pasha Osipyants).

```
func Sum<type T Addable>(x []T) T {
    var total T
    for _, v := range x {
        total += v
    }
    return total
}
```

- yesuu: In this example, think of `T` as the parameter name and `type` as the parameter type. Obviously it is more reasonable to put the `type` behind, and contract is followed by `type`, like `chan int`.

```
func Sum(T type Addable)(x []T) T
```

  - Roberto Clapis: Please read this section
    - Seems like a bit of a cop-out tbh. It says "in general" which means there must already be exceptions. Go has a nice clear syntax making code simple to read and easy for teams to collaborate. I think it would be worth making the parser more complicated for the sake of making the code readability better. For large scale and long running project readability of the code, and hence maintainability, is king
    - What about this

- Seebs: Feedback a bit long to inline, August 2018. Summary basically "I would like a way to specify one contract for each of two types rather than one contract for both types", and "I would prefer `map[T1]T2` to `t1var == t1var` as a canonical form of "T1 must be an allowable map key".

- Seebs: What if contracts *were* just the type-parametric functions?. (Sep 1, 2018)

- Sean Quinlan: I find the contract syntax quite confusing. For something that is supposed to defined exactly what is needed and will be part of the documentation of an api, it can contain all sorts of cruft that does not impact the contract. Moreover, to quote from the design: "We don't need to explain the meaning of every statement that can appear in a contract body". That seems like the opposite of what I would want from a contract. The fact one can copy the body of a function into a contract and have it work seems like a bug to me, not a feature. Personally, I would much prefer a model that unifies interfaces and contracts. Interfaces feel much closer to what I would like a contract to look like and there is a lot of overlap. It seems probable that many contracts will also be interfaces?

- Nodir Turakulov: Please elaborate

> *Packages like container/list and container/ring, and types like sync.Map, will be updated to be compile-time type-safe.*

and

> *The math package will be extended to provide a set of simple standard algorithms for all numeric types, such as the ever popular Min and Max functions.*

or ideally add a section about transition/migration of existing types/funcs to use type polymorphism. FWIU adding type parameter(s) to an existing type/func most likely breaks an existing program that uses the type/func. How exactly will `math.Max` be changed? Is the intention to make backward-incompatible changes and write tools to automatically convert code to Go2? What is the general recommendation for authors of other libraries that provide funcs and types that currently operate with `interface{}`? Were default values for type parameters considered? e.g. type parameter for `math.Max` would default to `float64` and type parameter for `"container/list".List` would default to `interface{}`

- Ward Harold: If only for the sake of completeness the Modula-3 generics design should be incorporated into the Designs in Other Languages section. Modula-3 was a beautiful language that sadly got introduced at the wrong time.
    - Matt Holiday: Ditto mentioning the Alphard language, which was developed about the same time as CLU and also influenced the Ada design. See Alphard: Form and Content, Mary Shaw, ed., Springer 1991 for the various papers collected with some glue material. Alphard & Ada were my introductions to generic programming. Could Go beat C++ for finally delivering contracts after 40 years of waiting?
- Ole Begemann: You write on the Generics Overview page: "Swift added generics in Swift 4, released in 2017." This is not true. Swift has had generics since its first public release in 2014. Evidence (just one example of many): a transcript of an Apple developer talk on Swift from WWDC 2014 that talks at length about Swift's generics features.

  This is also incorrect: "`Equatable` appears to be a built-in in Swift, not possible to define otherwise." The `Equatable` protocol is defined in the Swift standard library, but there's nothing special about it. It's totally possible to define the same thing in "normal" code.
- Kevin Gillette: correction for "Contracts" Draft, as of 30 August 2018

  The one instance of `check.Convert(int, interface{})(0, 0)` should instead be `check.Convert(int, interface{})(0)` or provide an explanation as to why it the function should take two zeros instead of one.
- Adam Ierymenko: I have an idea for doing limited operator overloading in Go that might make this proposal more useful for numeric code. It's big so I stuck it in a Gist here.
    - DeedleFake: I completely agree with the arguments against operator overloading, and I'm quite glad overall that Go doesn't have it, but I also think that the inability to resolve the difference between `a == b` and `a.Equals(b)` via a contract is the biggest problem with the draft design as it currently stands. It means that you'd still wind up writing multiple functions for a fair number of things. Try writing a binary tree, for example. Should you use a contract with `t < t` or `t.Less(t)`? For a sum function, should you use `t + t` or `t.Plus(t)`? I definitely want a solution that doesn't involve operator overloading, though. Maybe there could be a way to specify an adapter that basically says `if a type T, which satisfies contract A but not B, is used for a parameter constrained by contract B, apply this to it in order to get it to satisfy contract B`. Contract B could require a `Plus()` method, for example, while contract A requires the use of `+`, so the adapter automatically attaches a user-specified `Plus()` method to `T` for the duration of its use under that contract.

    - Something that might work with this proposal is an `equal(a, b)` builtin that uses `a.Equals(b)` if it exists and `a == b` otherwise, failing to compile if the type is incomparable (and likewise for other operators). It's too weird to seriously consider but it would work with contracts and allow dodging the asymmetry between types that have operators and those that cannot without introducing operator overloading — jimmyfrasche
    - Another idea would be explicitly overloadable operators: `a + b` is not overloadable, but `a [+] b` can be overloaded. It will use normal + for primitive types, but will use `Operator+()` etc. for objects if those are present. I really do think that generics without some sane form of operator overloading or something like it are a lot less useful to the point that you might as well not even do it. -Adam Ierymenko (original poster)
  - Ian Denhardt: DeedleFake outlines the problems with not having operator overloading well I. I think proposals involving making the overloading "loud" are the wrong idea; instead, we should limit which operators can be overloaded to operators which satisfy these critera:
    1. The operator's semantics can be understood as a method call. Most of the operators on numbers pass this test; `big.Add` is still addition in the sense that we know it from int32, uint64 etc. Examples of operators that fail this test are `&&` and `||`; these are short circuting, which no function or method can replicate. They are fundamentally not methods, no matter how you look at them, and should not be overridable by methods. I think operator overloading gets a bad rap in part because C++ allows you to override *everything*, including crazy stuff like the *comma operator*.
    2. There should be clear use cases for overriding them. Again, arithmetic operators pass this test, along with `<` and friends. Pointer dereferencing passes the first test, but I'm having a hard time coming up with uses for "other types of pointers" that actually seem like a good idea. They are a bit more justifiable in C++, but garbage-collected pointers have basically got you covered.
    3. The normal meaning of the operator should be something that is easy to reason about. For example, pointers are a gnarly source of bugs, and having the possibility that `*foo` is doing something other than reading from a memory address makes an already difficult debugging session that much harder. On the other hand, the possibility that `+` may be a call to `big.Add` is relatively self-contained, and unlikely to cause great confusion.
    4. Finally, the standard library has to set a good example; methods overriding `+` should be conceptually addition, for example. C++ gets off on an utterly wrong foot here by defining what is morally `os.Stdout.ShiftLeft("Hello, World!")`.

- Eltjon Metko: How about specifying the contract after the type identifier inside the function Parameters? This way it can be inferred what T is and we can eliminate the first group of parenthesis.

```
func Sum(x []T:Addable) T {
    var total T
    for _, v := range x {
        total += v
    }
    return total
}
```

- Tristan Colgate-McFarlane: After going back and forward for a while, I've come down in favour of the proposal largely as is. A limited syntax for contracts might be preferable, but I believe it should allow referencing specific fields (not just methods as some have proposed). If anything can be done to make compatible interface and contracts inter-use easier, that would also be nice (though I think maybe no

additional specifications are needed. Lastly, I think it is worth considering deprecating interface types. Whilst drastic, contracts essentially also allow specifying behaviour. Any contract restrictions that limit that (such as refering to other types within the package), should probably be lifted. contracts appear to be a strict superset of interfaces, and I am generally against having two overlapping features. A tool to aide in writing interaces should also be considered.

- Patrick Smith: We might consider requiring the type keyword when defining methods on generic types. This makes the code a little more verbose, but clearer and more consistent (now type parameters are always preceded by the type keyword).

```
func (x Foo(type T)) Bar()
```

- Patrick Smith: In this example, is `Foo(T)` embedded in `Bar(T)`, or does `Bar(T)` have a method named `Foo`?

```
type Foo(type T) interface {}
type Bar(type T) interface {
        Foo(T)
}
```

- Xingtao Zhao: There are too many round brackets in the proposal. In the proposal, it is said that "[]" is ambiguous in some cases. While if we use [type T, S contract], there are no ambiguities any more.

- Dave Cheney: The earlier Type Functions proposal showed that a type declaration can support a parameter. If this is correct, then the proposed contract declaration could be rewritten from

```
contract stringer(x T) {
    var s string = x.String()
}
```

to

```
type stringer(x T) contract {
    var s string = x.String()
}
```

This supports Roger's observation that a contract is a superset of an interface. `type stringer(x T) contract { ... }` introduces a new contract type in the same way `type stringer interface { ... }` introduces a new interface type.

  - jimmyfrasche: A contract is not a type, though. You can't have a value that is a `stringer`. You can have a value of a type that is a `stringer`. It's a metatype. A type is a kind of predicate on values. You ask the compiler "is this value a `string`" and it answers yes (allowing compilation to continue) or no (stopping to tell you what went wrong). A contract is a predicate on a vector of types. You ask the compiler two questions. Do these types satisfy this contract? Then: do these values satisfy these types? Interfaces kind of blur these lines by storing a `(type, value)` pair, provided that the type has the appropriate methods. It's simultaneously a type and a metatype. Any generics system that does not use interfaces as the metatypes will unavoidably contain a superset of interfaces. While it is entirely possible to define a generics system that exclusively uses interfaces as the metatypes, that does mean losing the ability to write generic functions that use things that interfaces can't talk about, like operators. You have to limit the questions you can ask about the types to their method sets. (I'm fine with this).

- btj: Two very important entries are missing in the draft design document's Designs in Other Languages section: Haskell, with its typeclasses, and Scala, with its implicit arguments.
- iamgoroot: Wouldn't it be natural to make better type aliasing support and let user have generics as option? And you dont need much syntax for that

```
type Key _
type Value _

type IntStringHolder Holder<Key:int, Value:string>

type Holder struct {
    K Key
    V Value
}

func (h *Holder) Set(k Key, v Value) {
    h.K = k
    h.V = v
}

func main() {
    v:= IntStringHolder{}
    v.Set(7,"Lucky")
}
```

- antoniomo: While the draft clearly explains why `F<T>`, `F[T]` and non-ASCII (unable to type it here) `F<<T>>` where discarded, feels like `F{T}` would be more human readable than the sometimes three in a row set of `()`, while not complicating the parser with unbounded lookahead as you can't open a block in those circumstances.
- aprice2704: I really dislike the idea of using regular parentheses `(`, would a two-character sequence cause compiler overhead from unbounded lookahead? How about <| and |> ? would they work? They have the advantages of being quite distinct from `(`, making some visual sense in ascii, and in the 'Fira Code' font (highly recommended) which I use in VSCode there are ligatures to render these as little right or left pointing triangles.
- leaxoy: First I'm sorry for editing the page footer, however I can't remove footer content. This is my opinion: lots of `(` & `)` made go looks so messy, `<` & `>` like other language is better, and is more kindly for those come from other languages.
- Hajime Hoshi: I completely agree on aprice2704's syntax concern. Wouldn't `[[` / `]]` work, for example?