

Constraining Complexity in the Generics Design

January 5, 2024

I gave a talk at [GopherConAU 2023](#) about a particular problem we encountered when designing generics for Go and what we might do about it.

This blog post is meant as a supplement to that talk. It mostly reproduces its content, while giving some supplementary information and more detailed explanations where necessary.

So if you prefer to ingest your information from text, then this blog post should serve you well. If you prefer a talk, [you can watch the recording](#) and use it to get some additional details in the relevant sections.

The talk (and hence this post) is also a follow-up to [a previous blog post of mine](#). But I believe the particular explanation I give here should be a bit more approachable and is also more general. If you have read that post and are just interested in the differences, feel free to skip to [the Type Parameter Problem](#).

With all that out of the way, let us get into it.

The Problem

If you are using Go generics, you are probably aware that it's possible to constrain type parameters. This makes sure that a type argument has all the operations that your generic function expects available to it.

One particular way to constrain a type parameter is using *union elements*, which allow you to say that a type has to be from some list of types. The most common use of this

is to allow you to use Go's operators on a generic parameter:

```
// Allows any type argument that has underlying type int, uint or string.
type Ordered interface {
    ~int | ~uint | ~string
}

func Max[T Ordered](a, b T) T {
    // As all int, uint and string types support the > operator, our generic
    // function can use it:
    if a > b {
        return a
    }
    return b
}
```

Another case this would be very useful for would be to allow us to call a method as a fallback:

```
type Stringish interface {
    fmt.Stringer | ~string
}

func Stringify[T Stringish](v T) string {
    if s, ok := any(v).(fmt.Stringer); ok {
        return s.String()
    }
    return reflect.ValueOf(v).String()
}
```

However, if we try this, the compiler will complain:

| cannot use fmt.Stringer in union (fmt.Stringer contains methods)

And if we check the spec, we find a specific exception for this:

| Implementation restriction: A union (with more than one term) cannot contain the predeclared identifier `comparable` or interfaces that specify methods, or embed `comparable` or interfaces that specify methods.

To explain why this restriction is in place, we will dive into a bit of theory.

Some Theory

You have probably heard about the **P** versus **NP** problem. It concerns two particular classes of computational problems:

- **P** is the class of problems that can be *solved* efficiently¹. An example of this is multiplication of integers: If I give you two integers, you can write an algorithm that quickly multiplies them.
- **NP** is the class of problems that can be *verified* efficiently: If you have a candidate for a solution, you can write an efficient algorithm that verifies it. An example is factorization: If you give me an integer N and a prime p , you can efficiently check whether or not it is a factor of N . You just divide N by p and check whether there is any remainder.

Every problem in **P** is also in **NP**: If you can efficiently *solve* a problem, you can also easily *verify* a solution, by just doing it yourself and comparing the answers.

However, the opposite is not necessarily true. For example, if I give you an integer N and tell you to give me a non-trivial factor of it, the best you could probably do is try out all possible candidates until you find one. This is *exponential* in the size of the input (an integer with k digits has on the order of 10^k candidate factors).

We generally assume that there are in fact problems which are in **NP** but not in **P** - but we have not actually *proven* so. Doing that is the **P** versus **NP** problem.

While we have not proven that there *are* such “hard” problems, we *did* prove that there are some problems which are “at least as hard as any other problem in **NP**”. This means that if you can solve *them* efficiently, you can solve *any* problem in **NP** efficiently. These are called “**NP**-hard” or “**NP**-complete”².

One such problem is the Boolean Satisfiability Problem. It asks you to take in a boolean formula - a composition of some boolean variables, connected with “and”, “or” and “not” operators - and determine an assignment to the variables that makes the formula true.

So, for example, I could ask you to find me a *satisfying assignment* for this function:

```
func F(x, y, z bool) bool {
    return (!x || z) && (y || z) && (x || !z)
}
```

For example, $F(\text{true}, \text{true}, \text{false})$ is `false`, so it is not a satisfying assignment. But $F(\text{false}, \text{true}, \text{false})$ is `true`, so that is a satisfying assignment.

It is easy to verify whether any given assignment satisfies your formula - you just substitute all the variables and evaluate it. But to *find* one, you probably have to try out all possible inputs. And for n variables, you have 2^n different options, so this takes exponential time.

In practice, this means that if you can show that solving a particular problem would allow you to solve SAT, your problem is *itself* **NP**-hard: It would be at least as hard as solving SAT, which is at least as hard as solving any other **NP** problem. And as we assume that **NP** \neq **P**, this means your problem can probably not be solved efficiently.

The last thing we need to mention is **co-NP**, the class of *complements* of problems in **NP**. The complement of a (decision) problem is simply the same problem, with the answer is inverted: You have to answer “yes” instead of “no” and vice versa. And where with **NP**, a “yes” answer should have an efficiently verifiable proof, with **co-NP**, a “no” answer should have an efficiently verifiable proof.

Notably, the actual *difficulty* of solving the problem does not change. To decide between “yes” and “no” is just as hard, you just turn around the answer. So, in a way, this is a technicality.

A **co-NP** complete problem is simply a problem that is the complement of an **NP** complete problem and as you would expect, it is just as hard and it is at least as hard as any other problem in **co-NP**.

Now, with the theory out of the way, let's look at Go again.

The Type Parameter Problem

When building a Go program, the compiler has to solve a couple of computational problems as well. For example, it has to be able to answer “does a given type argument satisfy a given constraint”. This happens if you instantiate a generic function with a concrete type:

```
func F[T C]() {} // where C is some constraint
func G() {
    F[int]() // Allowed if and only if int satisfies C.
}
```

This problem is in **P**: The compiler can just evaluate the constraint as if it was a logical formula, with `|` being an “or” operator, multiple lines being an “and” operator and checking if the type argument has the right methods or underlying types on the way.

Another problem it has to be able to solve is whether a given constraint `c1` *implies* another constraint `c2` : Does every type satisfying `c1` also satisfy `c2` ? This comes up if you instantiate a generic function *with a type parameter*:

```
func F[T C1]() {
    G[T]() // Allowed if and only if C1 implies C2
}
func G[T C2]() {}
```

My claim now is that this problem (which I will call the “Type Parameter Problem” for the purposes of this post) is **co-NP** complete³.

To prove this claim, we *reduce* SAT to the (complement of the) Type Parameter Problem. We show that if we *had* a Go compiler which solves this problem, we can use it so solve the SAT problem as well. And we do that, by translating an arbitrary boolean formula into a Go program and then check whether it compiles.

On a technical note, we are going to assume that the fomula is in Conjunctive Normal Form (CNF): A list of terms connected with “and” operators, where each term is a list of (possibly negated) variables connected with “or” terms. The example I used above is in CNF and we use it as an example to demonstrate the translation:

```
func F(x, y, z bool) bool {
```

```
    return (!x || z) && (y || z) && (x || !z)
}
```

This assumption may seem like a cheat, but importantly, SAT is still **NP**-complete with it.

The first step in our reduction is to model our boolean variables. Every variable can be either true or false and it can appear negated or not negated. We encode that by defining two interfaces per variable³:

```
type X interface { X() }    // X is assigned "true"
type NotX interface{ NotX() } // X is assigned "false"
```

This allows us to translate our formula directly, using union elements for “or” and interface-embedding for “and”:

```
// Represents (!x || z) && (y || z) && (x || !z)
type Formula interface {
    NotX | Z
    Y | Z
    X | NotZ
}
```

There are, however, two issues with this:

1. A type could have *neither* of `X()` and `NotX()` .
2. A type could have *both* of `X()` and `NotX()` .

This breaks our representation, because a boolean variable always has to be *exactly* true or false - it can't be neither and it can't be both.

To address the first point, we define another interface:

```
type AtLeastOne interface {
    X | NotX
    Y | NotY
    Z | NotZ
}
```

Any type satisfying `AtLeastOne` has to assign at least one of true and false to each variable.

Similarly, we define an interface to address the second problem:

```
type Both_X interface { X; NotX }
type Both_Y interface { Y; NotY }
type Both_Z interface { Z; NotZ }
type Both interface {
    Both_X | Both_Y | Both_Z
}
```

Any type satisfying `Both` now assigns both true *and* false to at least one variable.

To represent a valid, satisfying assignment, a type thus has to

1. satisfy `Formula`
2. satisfy `AtLeastOne`
3. *not* satisfy `Both`

Now, we ask our compiler to type-check this Go program⁴:

```
func G[T Both]() {}
func F[T interface{ Formula; AtLeastOne }]() {
    G[T]() // Allowed if and only if (Formula && AtLeastOne) => Both
}
```

This program should compile, if and only if any type satisfying `Formula` and `AtLeastOne` also satisfies `Both`. Because we are looking at the *complement* of SAT, we invert this, to get our final answer:

```
!( (Formula && AtLeastOne) => Both )
<=> !( !(Formula && AtLeastOne) || Both ) // "A => B" is equivalent to "!A || B"
<=> !( !(Formula && AtLeastOne && !Both) ) // De Morgan's law
<=> Formula && AtLeastOne && !Both // Double negation
```

This finishes our reduction: The compiler should reject the program, if and only if the formula has a satisfying assignment. The Type Parameter Problem is at least as hard

as the complement of SAT.

Going forward

So the restriction on methods in union elements is in place, because we are concerned about type checking Go would become a very hard problem if we allowed them. But that is, of course, a deeply dissatisfying situation.

Our `stringish` example would clearly be a very useful constraint - so useful, in fact, that it was used an example in the original design doc. More generally, this restriction prevents us from having a good way to express operator constraints for generic functions and types. We currently end up writing multiple versions of the same functions, one that uses operators and one that takes functions to do the operations. This leads to boilerplate and extra API surface⁵.

The slices package contains a bunch of examples like that (look for the `Func` suffix to the name):

```
// Uses the == operator. Useful for predeclared types (int, string,...) and
// structs/arrays of those.
func Contains[S ~[]E, E comparable](s S, v E) bool
// Uses f. Needed for slices, maps, comparing by pointer-value or other notions
// of equality.
func ContainsFunc[S ~[]E, E any](s S, f func(E) bool) bool
```

So we should consider compromises, allowing us to get *some* of the power of removing this restriction at least.

Option 1: Ignore the problem

This might be a surprising option to consider after spending all these words on demonstrating that this problem is hard to solve, but we can at least consider it: We simply say that a Go compiler has to include *some form* of (possibly limited) SAT solver and is allowed to just give up after some time, if it can not find a proof that a program is safe.

C++ concepts do this. A C++ compiler has to determine if one constraint implies another one, when it has to decide which of multiple overloaded generic functions to invoke. And it does so using a simple SAT solver. In particular, if it wants to prove $P \Rightarrow Q$, it first converts P into Disjunctive Normal Form (DNF) and then convert Q into Conjunctive Normal Form (CNF).

With P in DNF and Q in DNF, $P \Rightarrow Q$ is easy to prove (and disprove). But this normalization into DNF or CNF *itself* requires exponential time in general. And you can indeed create C++ programs that crash C++ compilers.

Personally, I find all versions of this option very dissatisfying:

- Leaving the heuristic up to the implementation feels like too much wiggle-room for what makes a valid Go program.
- Describing an explicit heuristic in the spec takes up a lot of the complexity budget of the spec.
- Allowing the compiler to try and give up after some time feels antithetical to the pride Go takes in fast compilation.

Option 2: Limit the expressiveness of interfaces

For the interfaces as they exist today, we actually *can* solve the SAT problem: Any interface can ultimately be represented in the form (with some elements perhaps being empty):

```
interface {
    A | ... | C | ~X | ... | ~Z // for some concrete types
    comparable
    M1(...) (...)
    // ...
    Mn(...) (...)
}
```

And it is straight-forward to use this representation to do the kind of inference we need.

This tells us that there are *some* restrictions we can put on the kinds of interfaces we can write down, while still not running into the kinds of problems discussed in this post.

That's because every such kind of interfaces gives us a restricted *sub problem* of SAT, which only looks at formulas conforming to some extra restrictions.

One example of such a sub problem we actually used above, where we assumed that our formula is in Conjunctive Normal Form. Another important such sub problem is the one where the formulas are in Disjunctive Normal Form instead: Where we have a list of terms linked with "or" operators and each term is a list of (possibly negated) variables linked with "and" operators. For DNF, the SAT problem is efficiently solvable.

We could take advantage of that by allowing union elements to contain methods - but only if

1. There is exactly one union in the top-level interface.
2. The interfaces embedded in that union are "easy" interfaces, i.e. ones we allow today.

So, for example

```
type Stringish interface {
    // Allowed: fmt.Stringer and ~string are both allowed today
    fmt.Stringer | ~string
}
type A interface {
    // Not Allowed: Stringish is not allowed today, so we have more than one level
    Stringish | ~int
}
type B interface {
    // Allowed: Same as A, but we "flattened" it, so each element is an
    // "easy" interface.
    fmt.Stringer | ~string | ~int
}
type C interface {
    // Not Allowed: Can only have a single union (or must be an "easy" interface)
    fmt.Stringer | ~string
    comparable
}
```

This restriction makes our interfaces be in DNF, in a sense. It's just that every "variable" of our DNF is itself an "easy" interface. If we need to solve SAT for one of these, we first

solve it on the SAT formula to determine which “easy” interfaces need to be satisfied and then use our current algorithms to check which of those *can* be satisfied.

Of course, this restriction is somewhat hard to explain. But it would allow us to write at least some of the useful programs we want to use this feature for. And we might find another set of restrictions that are easier to explain but still allow that.

We should probably try to collect some useful programs that we would want to write with this feature and then see, for some restricted interface languages if they allow us to write them.

Option 3: Make the type-checker conservative

For our reduction, we assumed that the compiler should allow the program if *and only if* it can prove that every type satisfying c_1 also satisfies c_2 .

We could allow it to reject some programs that *would* be valid, though. We could describe an algorithm for determining if c_1 implies c_2 that can have false negatives: Rejecting a theoretically safe program, just because it cannot *prove* that it is safe with that algorithm, requiring you to re-write your program into something it can handle more easily.

Ultimately, this is kind of what a type system does: It gives you a somewhat limited language to write a proof to the compiler that your program is “safe”, in the sense that it satisfies certain invariants. And if you accidentally pass a variable of the wrong type - even if your program would still be perfectly valid - you might have to add a conversion or call some function that verifies its invariants, before being allowed to do so.

For this route, we still have to decide *which* false negatives we are willing to accept though: What is the algorithm the compiler should use?

For some cases, this is trivial. For example, this should obviously compile:

```
func StringifyAll[T Stringish](vals ...T) []string {
    out := make([]string, len(vals))
    for i, v := range vals {
        // Stringify as above. Should be allowed, as T uses the same constraint
```

```

    // as Stringify.
    out[i] = Stringify(v)
}
return out
}

```

But other cases are not as straight forward and require *some* elaboration:

```

func Marshal[T Stringish | ~bool | constraints.Integer](v T) string { /* ... */ }

// Stringish appears in the union of the target constraint.
func F[T Stringish](v T) string { return Marshal[T](v) }

// string has underlying type string and fmt.Stringer is the Stringish union.
func G[T string|fmt.Stringer](v T) string { return Marshal[T](v) }

// The method name is just a different representation of fmt.Stringer
func H[T interface{ String() string }](v T) string { return Marshal[T](v) }

```

These examples are still simple, but they are useful, so should probably be allowed. But they already show that there is *somewhat* complex inference needed: Some terms on the left might satisfy some terms on the right, but we can not simply compare them as a subset relation, we actually have to take into account the different cases.

And remember that converting to DNF or CNF takes exponential time, so the simple answer of “convert the left side into DNF and the right side into CNF, then check each term individually” does not solve our problem.

In practice, this option has a large intersection with the previous one: The algorithm would probably reject programs that use interfaces with too complex a structure on either side, to guarantee that it terminates quickly. But it would allow us, in principle, to use *different* restrictions for the left and the right hand side: Allow you to write *any* interface and only check the structure if you actually use them in a way that would make inference impossible.

We have to decide whether we would find that acceptable though, or whether it seems to confusing in practice. Describing the algorithm also would take quite a lot of space and complexity budget in the spec.

Future-proofing

Lastly, when we talk about this we should keep in mind possible future extensions to the generics design.

For example, there is a proposal by Rog Peppe to add a type-switch on type parameters. The proposal is to add a new type switch syntax for type parameters, where every case has a new constraint and in that branch, you could use the type parameter *as if it was further constrained by that*. So, for example, it would allow us to rewrite `Stringify` without `reflect` :

```
func Stringify[T Stringish](v T) string {
    switch type T {
    case fmt.Stringer:
        // T is constrained by Stringish *and* fmt.Stringer. So just fmt.Stringer
        // Calling String on a fmt.Stringer is allowed.
        return v.String()
    case ~string:
        // T is constrained by Stringish *and* ~string. So just ~string
        // Converting a ~string to string is allowed.
        return string(v)
    }
}
```

The crux here is, that this proposal allows us to create new, implicit interfaces out of old ones.

If we restrict the structure of our interfaces, these implicit interfaces might violate this structure. And if we make the type checker more conservative, a valid piece of code might no longer be valid if copied into a type parameter switch, if the implicit constraints would lead to a generic all the compiler can't prove to be safe.

Of course it is impossible to know what extension we *really* want to add in the future. But we should at least consider some likely candidates during the discussion.


Summary

I hope I convinced you that

1. Simply allowing methods in unions would make type-checking Go code **co-NP** hard.
2. But we might be able to find *some* compromise that still allows us to do *some* of the things we want to use this for.
3. The devil is in the details and we still have to think hard and carefully about those.


1. “efficient”, in this context, means “in polynomial time in the size of the input”.

In general, if an input to an algorithm gets larger, the time it needs to run grows. We can look at how fast this growth is, how long the algorithm takes by the size of the input. And if that growth is at most polynomial, we consider that “efficient”, in this context.

In practice, even many polynomial growth functions are too slow for our taste. But we still make this qualitative distinction in complexity theory. 


2. The difference between these two terms is that “**NP**-hard” means “at least as difficult than any problem in **NP**”. While “**NP**-complete” means “**NP**-hard and also *itself* in **NP**”.

So an **NP**-hard problem might indeed be *even harder* than other problems in **NP**, while an **NP**-complete problem is not.


For us, the difference does not really matter. All problems we talk about are in **NP**. 

3. If you have read [my previous post on the topic](#), you might notice a difference here.

Previously, I defined `NotX` as `interface{ X() int }` and relied on this being mutually exclusive with `x` : You can’t have two methods with the same name but different signatures.

This is one reason I think this proof is nicer than my previous one. It does not require “magical” knowledge like that, instead *only* requiring you to be able to define interfaces with arbitrary method names. Which is extremely open. 

4. The other reason I like this proof better than my previous one is that it no longer relies on the abstract problem of “proving that a type set is empty”. While the [principle of explosion](#) is familiar to Mathematicians, it is hard to take its implications seriously if you are not.

Needing to type-check a generic function call is far more obvious as a problem that *needs* solving and it is easier to find understandable examples. 

5. And inefficiencies, as calling a method on a type parameter can often be devirtualized and/or inlined. A `func` value sometimes can't. For example if it is stored in a field of a generic type, the compiler is usually unable to prove that it doesn't change at runtime. [↩](#)

[programming](#)[golang](#)