Everything You Always Wanted to Know About Type Inference - And a Little Bit More - The Go Programming Language

Robert Griesemer 9 October 2023

The Go Blog

This is the blog version of my talk on type inference at GopherCon 2023 in San Diego, slightly expanded and edited for clarity.

What is type inference?

Wikipedia defines type inference as follows:

> Type inference is the ability to automatically deduce, either partially or fully, the type of an expression at compile time. The compiler is often able to infer the type of a variable or the type signature of a function, without explicit type annotations having been given.

The key phrase here is "automatically deduce … the type of an expression". Go supported a basic form of type inference from the start:

```
const x = expr  // the type of x is the type of expr
var x = expr
x := expr
```

No explicit types are given in these declarations, and therefore the types of the constant and variables `x` on the left of = and `:=` are the types of the respective initialization expressions, on the right. We say that the types are *inferred* from (the types of) their initialization expressions. With the introduction of generics in Go 1.18, Go's type inference abilities were significantly expanded.

Why type inference?

In non-generic Go code, the effect of leaving away types is most pronounced in a short variable declaration. Such a declaration combines type inference and a little bit of syntactic sugar—the ability to leave away the `var` keyword—into one very compact statement. Consider the following map variable declaration:

```
var m map[string]int = map[string]int{}
```

vs

```
m := map[string]int{}
```

Omitting the type on the left of `:=` removes repetition and at the same time increases readability.

Generic Go code has the potential to significantly increase the number of types appearing in code: without type inference, each generic function and type instantiation requires type arguments. Being able to omit them becomes even more important. Consider using the following two functions from the new slices package:

```
package slices
func BinarySearch[S ~[]E, E cmp.Ordered](x S, target E) (int, bool)
func Sort[S ~[]E, E cmp.Ordered](x S)
```

Without type inference, calling `BinarySearch` and `Sort` requires explicit type arguments:

```
type List []int
var list List
slices.Sort[List, int](list)
index, found := slices.BinarySearch[List, int](list, 42)
```

We'd rather not repeat `[List, int]` with each such generic function call. With type inference the code simplifies to:

```
type List []int
var list List
slices.Sort(list)
index, found := slices.BinarySearch(list, 42)
```

This is both cleaner and more compact. In fact it looks exactly like non-generic code, and type inference makes this possible.

Importantly, type inference is an optional mechanism: if type arguments make code clearer, by all means, write them down.

Type inference is a form of type pattern matching

Inference compares type patterns, where a type pattern is a type containing type parameters. For reasons that will become obvious in a bit, type parameters are sometimes also called *type variables*. Type pattern matching allows us to infer the types that need to go into these type variables. Let's consider a short example:

```
// From the slices package
// func Sort[S ~[]E, E cmp.Ordered](x S)

type List []int
var list List
slices.Sort(list)
```

The `Sort` function call passes the `list` variable as function argument for the parameter `x` of `slices.Sort`. Therefore the type of `list`, which is `List`, must match the type of `x`, which is type parameter `S`. If `S` has the type `List`, this assignment becomes valid. In reality, the rules for assignments are complicated, but for now it's good enough to assume that the types must be identical.

Once we have inferred the type for `S`, we can look at the type constraint for `S`. It says—because of the tilde `~` symbol—that the *underlying type* of `S` must be the slice `[]E`. The underlying type of `S` is `[]int`, therefore `[]int` must match `[]E`, and with that we can conclude that `E` must be `int`. We've been able to find types for `S` and `E` such that corresponding types match. Inference has succeeded!

Here's a more complicated scenario where we have a lot of type parameters: `S1`, `S2`, `E1`, and `E2` from `slices.EqualFunc`, and `E1` and `E2` from the generic function `equal`. The local function `foo` calls `slices.EqualFunc` with the `equal` function as an argument:

```
// From the slices package
// func EqualFunc[S1 ~[]E1, S2 ~[]E2, E1, E2 any](s1 S1, s2 S2, eq
func(E1, E2) bool) bool

// Local code
func equal[E1, E2 comparable](E1, E2) bool { … }

func foo(list1 []int, list2 []float64) {
    …
    if slices.EqualFunc(list1, list2, equal) {
        …
    }
    …
}
```

This is an example where type inference really shines as we can potentially leave away six type arguments, one for each of the type parameters. The type pattern matching approach still works, but we can see how it may get complicated quickly because the number of type relationships is proliferating. We need a systematic approach to determine which type parameters and which types get involved with which patterns.

It helps to look at type inference in a slightly different way.

Type equations

We can reframe type inference as a problem of solving type equations. Solving equations is something that we are all familiar with from high school algebra. Luckily, solving type equations is a simpler problem as we will see shortly.

Let’s look again at our earlier example:

```
// From the slices package
// func Sort[S ~[]E, E cmp.Ordered](x S)

type List []int
var list List
slices.Sort(list)
```

Inference succeeds if the type equations below can be solved. Here ≡ stands for *is identical to*, and `under(S)` represents the underlying type of `S`:

```
S ≡ List        // find S such that S ≡ List is true
under(S) ≡ []E  // find E such that under(S) ≡ []E is true
```

The type parameters are the *variables* in the equations. Solving the equations means finding values (type arguments) for these variables (type parameters), such that the equations become true. This view makes the type inference problem more tractable because it gives us a formal framework that allows us to write down the information that flows into inference.

Being precise with type relations

Until now we have simply talked about types having to be identical. But for actual Go code that is too strong a requirement. In the previous example, `S` need not be identical to `List`, rather `List` must be assignable to `S`. Similarly, `S` must satisfy its corresponding type constraint. We can formulate our type equations more precisely by using specific operators that we write as `:≡` and `∈`:

```
S :≡ List         // List is assignable to S
S ∈ ~[]E          // S satisfies constraint ~[]E
E ∈ cmp.Ordered   // E satisfies constraint cmp.Ordered
```

Generally, we can say that type equations come in three forms: two types must be identical, one type must be assignable to the other type, or one type must satisfy a type constraint:

```
X ≡ Y             // X and Y must be identical
X :≡ Y            // Y is assignable to X
X ∈ Y             // X satisfies constraint Y
```

(Note: In the GopherCon talk we used the symbols $≡_A$ for `:≡` and $≡_C$ for `∈`. We believe `:≡` more clearly evokes an assignment relation; and `∈` directly expresses that the type represented by a type parameter must be an element of its constraint’s type set.)

Sources of type equations

In a generic function call we may have explicit type arguments, though most of the time we hope that they can be inferred. Typically we also have ordinary function arguments. Each explicit type argument contributes a (trivial) type equation: the type parameter must be identical to the type argument because the code says so. Each ordinary function argument contributes another type equation: the function argument must be assignable to its corresponding function parameter. And finally, each type constraint provides a type equation as well by constraining what types satisfy the constraint.

Altogether, this produces `n` type parameters and `m` type equations. In contrast to basic high school algebra, `n` and `m` don't have to be the same for type equations to be solvable. For instance, the single equation below allows us to infer the type arguments for two type parameters:

```
map[K]V ≡ map[int]string  // K ➞ int, V ➞ string (n = 2, m = 1)
```

Let's look at each of these sources of type equations in turn:

1. Type equations from type arguments

For each type parameter declaration

```
func f[…, P constraint, …]…
```

and explicitly provided type argument

```
f[…, A, …]…
```

we get the type equation

```
P ≡ A
```

We can trivially solve this for `P`: `P` must be `A` and we write `P ➞ A`. In other words, there is nothing to do here. We could still write down the respective type equation for completeness, but in this case, the Go compiler simply substitutes the type arguments for their type parameters throughout and then those type parameters are gone and we can forget about them.

2. Type equations from assignments

For each function argument `x` passed to a function parameter `p`

```
f(…, x, …)
```

where `p` or `x` contain type parameters, the type of `x` must be assignable to the type of the parameter `p`. We can express this with the equation

```
𝑻(p) :≡ 𝑻(x)
```

where 𝑻(`x`) means “the type of `x`”. If neither `p` nor `x` contains type parameters, there is no type variable to solve for: the equation is either true because the assignment is valid Go code, or false if the code is invalid. For this reason, type inference only considers types that contain type parameters of the involved function (or functions).

Starting with Go 1.21, an uninstantiated or partially instantiated function (but not a function call) may also be assigned to a function-typed variable, as in:

```
// From the slices package
// func Sort[S ~[]E, E cmp.Ordered](x S)

var intSort func([]int) = slices.Sort
```

Analogous to parameter passing, such assignments lead to a corresponding type equation. For this example it would be

```
𝑻(intSort) :≡ 𝑻(slices.Sort)
```

or simplified

```
func([]int) :≡ func(S)
```

together with equations for the constraints for `S` and `E` from `slices.Sort` (see below).

3. Type equations from constraints

Finally, for each type parameter `P` for which we want to infer a type argument, we can extract a type equation from its constraint because the type parameter must satisfy the constraint. Given the declaration

```
func f[…, P constraint, …]…
```

we can write down the equation

```
P ∈ constraint
```

Here, the `∈` means “must satisfy constraint” which is (almost) the same as being a type element of the constraint’s type set. We will see later that some constraints (such as `any`) are not useful or currently cannot be used due to limitations of the implementation. Inference simply ignores the respective equations in those cases.

Type parameters and equations may be from multiple functions

In Go 1.18, inferred type parameters had to all be from the same function. Specifically, it was not possible to pass a generic, uninstantiated or partially instantiated function as a function argument, or assign it to a (function-typed) variable.

As mentioned earlier, in Go 1.21 type inference also works in these cases. For instance, the generic function

```
func myEq[P comparable](x, y P) bool { return x == y }
```

can be assigned to a variable of function type

```
var strEq func(x, y string) bool = myEq  // same as using myEq[string]
```

without `myEq` being fully instantiated, and type inference will infer that the type argument for `P` must be `string`.

Furthermore, a generic function may be used uninstantiated or partially instantiated as an argument to another, possibly generic function:

```
// From the slices package
// func CompactFunc[S ~[]E, E any](s S, eq func(E, E) bool) S

type List []int
var list List
result := slices.CompactFunc(list, myEq)  // same as using
slices.CompactFunc[List, int](list, myEq[int])
```

In this last example, type inference determines the type arguments for `CompactFunc` and `myEq`. More generally, type parameters from arbitrarily many functions may need to be inferred. With multiple functions involved, type equations may also be from or involve multiple functions. In the `CompactFunc` example we end up with three type parameters and five type equations:

```
Type parameters and constraints:
    S ~[]E
    E any
    P comparable

Explicit type arguments:
    none

Type equations:
    S :≡ List
    func(E, E) bool :≡ func(P, P) bool
    S ∈ ~[]E
    E ∈ any

    P ∈ comparable

Solution:
    S ➞ List
    E ➞ int
    P ➞ int
```

Bound vs free type parameters

At this point we have a clearer understanding of the various source of type equations, but we have not been very precise about which type parameters to solve the equations for. Let’s consider another example. In the code below, the function body of `sortedPrint` calls `slices.Sort` for the sorting part. `sortedPrint` and `slices.Sort` are generic functions as both declare type parameters.

```
// From the slices package
// func Sort[S ~[]E, E cmp.Ordered](x S)

// sortedPrint prints the elements of the provided list in sorted order.
func sortedPrint[F any](list []F) {
    slices.Sort(list)  // 𝑻(list) is []F
    …                  // print list
}
```

We want to infer the type argument for the `slices.Sort` call. Passing `list` to parameter `x` of `slices.Sort` gives rise to the equation

```
𝑻(x) :≡ 𝑻(list)
```

which is the same as

```
S :≡ []F
```

In this equation we have two type parameters, `S` and `F`. Which one do we need to solve the type equation for? Because the invoked function is `Sort`, we care about its type parameter `S`, not the type parameter `F`. We say that `S` is *bound* to `Sort` because it is declared by `Sort`. `S` is the relevant type variable in this equation. By contrast, `F` is bound to (declared by) `sortedPrint`. We say that `F` is *free* with respect to `Sort`. It has its own, already given type. That type is `F`, whatever that is (determined at instantiation time). In this equation, `F` is already given, it is a *type constant*.

When solving type equations we always solve for the type parameters bound to the function we are calling (or assigning in case of a generic function assignment).

Solving type equations

The missing piece, now that we have established how to collect the relevant type parameters and type equations, is of course the algorithm that allows us to solve the equations. After the various examples, it probably has become obvious that solving `X ≡ Y` simply means comparing the types `X` and `Y` recursively against each other, and in the process determining suitable type arguments for type parameters that may occur in `X` and `Y`. The goal is to make the types `X` and `Y` *identical*. This matching process is called *unification*.

The rules for type identity tell us how to compare types. Since *bound* type parameters play the role of type variables, we need to specify how they are matched against other types. The rules are as follows:

If type parameter `P` has an inferred type, `P` stands for that type.

If type parameter `P` doesn't have an inferred type and is matched against another type `T`, `P` is set to that type: `P ➞ T`. We say that the type `T` was inferred for `P`.

If `P` matches against another type parameter `Q`, and neither `P` nor `Q` have an inferred type yet, `P` and `Q` are *unified*.

Unification of two type parameters means that they are joined together such that going forward they both denote the same type parameter value: if one of `P` or `Q` is matched against a type `T`, both `P` and `Q` are set to `T` simultaneously (in general, any number of type parameters may be unified this way).

Finally, if two types `X` and `Y` are different, the equation cannot be made true and solving it fails.

Unifying types for type identity

A few concrete examples should make this algorithm clear. Consider two types `X` and `Y` containing three bound type parameters `A`, `B`, and `C`, all appearing in the type equation `X ≡ Y`. The goal is to the solve this equation for the type parameters; i.e., find suitable type arguments for them such that `X` and `Y` become identical and thus the equation becomes true.

```
X: map[A]struct{i int; s []B}
Y: map[string]struct{i C; s []byte}
```

Unification proceeds by comparing the structure of `X` and `Y` recursively, starting at the top. Simply looking at the structure of the two types we have

```
map[…]… ≡ map[…]…
```

with the `…` representing the respective map key and value types that we're ignoring at this step. Since we have a map on both sides, the types are identical so far. Unification proceeds recursively, first with the key types which are `A` for the `X` map, and `string` for the `Y` map. Corresponding key types must be identical, and from that we can immediately infer that the type argument for `A` must be `string`:

```
A ≡ string => A → string
```

Continuing with the map element types, we arrive at

```
struct{i int; s []B} ≡ struct{i C; s []byte}
```

Both sides are structs so unification proceeds with the struct fields. They are identical if they are in the same order, with the same names, and identical types. The first field pair is `i int` and `i C`. The names match and because `int` must unify with `C`, thus

```
int ≡ C => C → int
```

This recursive type matching continues until the tree structure of the two types is fully traversed, or until a conflict appears. In this example, eventually we end up with

```
[]B ≡ []byte => B ≡ byte => B → byte
```

Everything works out fine and unification infers the type arguments

```
A → string
B → byte
C → int
```

Unifying types with different structures

Now, let's consider a slight variation of the previous example: here `X` and `Y` don't have the same type structure. When the type trees are compared recursively, unification still successfully infers the type argument for `A`. But the value types of the maps are different and unification fails.

```
X: map[A]struct{i int; s []B}
Y: map[string]bool
```

Both `X` and `Y` are map types, so unification proceeds recursively as before, starting with the key types. We arrive at

```
A ≡ string => A → string
```

also as before. But when we proceed with the map's value types we have

```
struct{…} ≡ bool
```

The `struct` type doesn't match `bool`; we have different types and unification (and thus type inference) fails.

Unifying types with conflicting type arguments

Another kind of conflict appears when different types match against the same type parameter. Here we have again a version of our initial example but now the type parameter `A` appears twice in `X`, and `C` appears twice in `Y`.

```
X: map[A]struct{i int; s []A}
Y: map[string]struct{i C; s []C}
```

The recursive type unification works out fine at first and we have the following pairings of type parameters and types:

```
A   ≡ string => A → string  // map key type
int ≡ C      => C → int     // first struct field type
```

When we get to the second struct field type we have

```
[]A ≡ []C => A ≡ C
```

Since both `A` and `C` have a type argument inferred for them, they stand for those type arguments, which are `string` and `int` respectively. These are different types, so `A` and `C` can't possibly match. Unification and thus type inference fails.

Other type relations

Unification solves type equations of the form `X ≡ Y` where the goal is *type identity*. But what about `X :≡ Y` or `X ∈ Y`?

A couple of observations help us out here: The job of type inference is solely to find the types of omitted type arguments. Type inference is always followed by type or function instantiation which checks that each type argument actually satisfies its respective type constraint. Finally, in case of a generic function call, the compiler also checks that function arguments are assignable to their corresponding function parameters. All of these steps must succeed for the code to be valid.

If type inference is not precise enough it may infer an (incorrect) type argument where no type may exist. If that is the case, either instantiation or argument passing will fail. Either way, the compiler will produce an error message. It's just that the error message may be slightly different.

This insight allows us to play a bit loose with the type relations `:≡` and `∈`. Specifically, it allows us to simplify them such that they can be treated almost the same as `≡`. The goal of the simplifications is to extract as much type information as possible from a type equation, and thus to infer type arguments where a precise implementation may fail, because we can.

Simplifying X :≡ Y

Go's assignability rules are pretty complicated, but most of the time we can actually get by with type identity, or a slight variation of it. As long as we find potential type arguments, we're happy, exactly because type inference is still followed by type instantiation and function invocation. If inference finds a type argument where it shouldn't, it'll be caught later. Thus, when matching for assignability, we make the following adjustments to the unfication algorithm:

When a named (defined) type is matched against a type literal, their underlying types are compared instead.

When comparing channel types, channel directions are ignored.

Furthermore, the assignment direction is ignored: `X :≡ Y` is treated like `Y :≡ X`.

These adjustments apply only at the top level of a type structure: for instance, per Go's assignability rules, a named map type may be assigned to an unnamed map type, but the key and element types must still be identical. With these changes, unification for assignability becomes a (minor) variation of unification for type identity. The following example illustrates this.

Let's assume we are passing a value of our earlier `List` type (defined as `type List []int`) to a function parameter of type `[]E` where `E` is a bound type parameter (i.e., `E` is declared by the generic function that is being called). This leads to the type equation `[]E :≡ List`. Attempting to unify these two types requires comparing `[]E` with `List` These two types are not identical, and without any changes to how unification works, it will fail. But because we are unifying for assignability, this initial match doesn't need to be exact. There's no harm in continuing with the underlying type of the named type `List`: in the worst case we may infer an incorrect type argument, but that will lead to an error later, when assignments are checked. In the best case, we find a useful and correct type argument. In our example, inexact unification succeeds and we correctly infer `int` for `E`.

Simplifying X ∈ Y

Being able to simplify the constraint satisfaction relation is even more important as constraints can be very complex.

Again, constraint satisfaction is checked at instantiation time, so the goal here is to help type inference where we can. These are typically situations where we know the structure of a type parameter; for instance we know that it must be a slice type and we care about the slice's element type. For example, a type parameter list of the form `[P ~[]E]` tells us that whatever `P` is, its underlying type must be of the form `[]E`. These are exactly the situations where the constraint has a core type.

Therefore, if we have an equation of the form

```
P ∈ constraint               // or
P ∈ ~constraint
```

and if `core(constraint)` (or `core(~constraint)`, respectively) exists, the equation can be simplified to

```
P        ≡ core(constraint)
under(P) ≡ core(~constraint)  // respectively
```

In all other cases, type equations involving constraints are ignored.

Expanding inferred types

If unification is successful it produces a mapping from type parameters to inferred type arguments. But unification alone doesn't ensure that the inferred types are free of bound type parameters. To see why this is the case, consider the generic function `g` below which is invoked with a single argument `x` of type `int`:

```
func g[A any, B []C, C *A](x A) { … }

var x int
g(x)
```

The type constraint for `A` is `any` which doesn't have a core type, so we ignore it. The remaining type constraints have core types and they are `[]C` and `*A` respectively. Together with the argument passed to `g`, after minor simplifications, the type equations are:

```
    A :≡ int
    B ≡ []C
    C ≡ *A
```

Since each equation pits a type parameter against a non-type parameter type, unification has little to do and immediately infers

```
    A ➞ int
    B ➞ []C
    C ➞ *A
```

But that leaves the type parameters `A` and `C` in the inferred types, which is not helpful. Like in high school algebra, once an equation is solved for a variable `x`, we need to substitute `x` with its value throughout the remaining equations. In our example, in a first step, the `C` in `[]C` is substituted with the inferred type (the "value") for `C`, which is `*A`, and we arrive at

```
    A ➞ int
    B ➞ []*A    // substituted *A for C
    C ➞ *A
```

In two more steps we replace the `A` in the inferred types `[]*A` and `*A` with the inferred type for `A`, which is `int`:

```
    A → int
    B → []*int  // substituted int for A
    C → *int    // substituted int for A
```

Only now inference is done. And like in high school algebra, sometimes this doesn't work. It's possible to arrive at a situation such as

```
    X → Y
    Y → *X
```

After one round of substitutions we have

```
    X → *X
```

If we keep going, the inferred type for `X` keeps growing:

```
    X → **X     // substituted *X for X
    X → ***X    // substituted *X for X
    etc.
```

Type inference detects such cycles during expansion and reports an error (and thus fails).

Untyped constants

By now we have seen how type inference works by solving type equations with unification, followed by expansion of the result. But what if there are no types? What if the function arguments are untyped constants?

Another example helps us shed light on this situation. Let's consider a function `foo` which takes an arbitrary number of arguments, all of which must have the same type. `foo` is called with a variety of untyped constant arguments, including a variable `x` of type `int`:

```
func foo[P any](...P) {}

var x int
foo(x)         // P → int, same as foo[int](x)
foo(x, 2.0)    // P → int, 2.0 converts to int without loss of precision
foo(x, 2.1)    // P → int, but parameter passing fails: 2.1 is not
assignable to int
```

For type inference, typed arguments take precedence over untyped arguments. An untyped constant is considered for inference only if the type parameter it's assigned to doesn't have an inferred type yet. In these first three calls to `foo`, the variable `x` determines the inferred type for `P`: it's the type of `x` which is `int`. Untyped constants are ignored for type inference in this case and the calls behave exactly as if `foo` was explicitly instantiated with `int`.

It gets more interesting if `foo` is called with untyped constant arguments only. In this case, type inference considers the default types of the untyped constants. As a quick reminder, here are the possible default types in Go:

```
Example     Constant kind              Default type    Order

true        boolean constant           bool
42          integer constant           int             earlier in list
'x'         rune constant              rune                |
3.1416      floating-point constant    float64             v
-1i         complex constant           complex128      later in list
"gopher"    string constant            string
```

With this information in hand, let's consider the function call

```
foo(1, 2)    // P ➞ int (default type for 1 and 2)
```

The untyped constant arguments `1` and `2` are both integer constants, their default type is `int` and thus it's `int` that is inferred for the type parameter `P` of `foo`.

If different constants—say untyped integer and floating-point constants—compete for the same type variable, we have different default types. Before Go 1.21, this was considered a conflict and led to an error:

```
foo(1, 2.0)    // Go 1.20: inference error: default types int, float64
don't match
```

This behavior was not very ergonomic in use and also different from the behavior of untyped constants in expressions. For instance, Go permits the constant expression `1 + 2.0`; the result is the floating-point constant `3.0` with default type `float64`.

In Go 1.21 the behavior was changed accordingly. Now, if multiple untyped numeric constants are matched against the same type parameter, the default type that appears later in the list of `int`, `rune`, `float64`, `complex` is selected, matching the rules for constant expressions:

```
foo(1, 2.0)    // Go 1.21: P ➞ float64 (larger default type of 1 and 2.0;
behavior like in 1 + 2.0)
```

Special situations

By now we've got the big picture about type inference. But there are a couple of important special situations that deserve some attention.

Parameter order dependencies

The first one has to do with parameter order dependencies. An important property we want from type inference is that the same types are inferred irrespective of the order of the function parameters (and corresponding argument order in each call of that function).

Let’s reconsider our variadic `foo` function: the type inferred for `P` should be the same irrespective of the order in which we pass the arguments `s` and `t` (playground).

```
func foo[P any](...P) (x P) {}

type T struct{}

func main() {
    var s struct{}
    var t T
    fmt.Printf("%T\n", foo(s, t))
    fmt.Printf("%T\n", foo(t, s)) // expect same result independent of
parameter order
}
```

From the calls to `foo` we can extract the relevant type equations:

```
𝑻(x) :≡ 𝑻(s) => P :≡ struct{}    // equation 1
𝑻(x) :≡ 𝑻(t) => P :≡ T           // equation 2
```

Sadly, the simplified implementation for `:≡` produces an order dependency:

If unification starts with equation 1, it matches `P` against `struct`; `P` doesn’t have a type inferred for it yet and thus unification infers `P ➞ struct{}`. When unification sees type `T` later in equation 2, it proceeds with the underlying type of `T` which is `struct{}`, `P` and `under(T)` unify, and unification and thus inference succeeds.

Vice versa, if unification starts with equation 2, it matches `P` against `T`; `P` doesn’t have a type inferred for it yet and thus unification infers `P ➞ T`. When unification sees `struct{}` later in equation 1, it proceeds with the underlying type of the type `T` inferred for `P`. That underlying type is `struct{}`, which matches `struct` in equation 1, and unification and thus inference succeeds.

As a consequence, depending on the order in which unification solves the two type equations, the inferred type is either `struct{}` or `T`. This is of course unsatisfying: a program may suddenly stop compiling simply because arguments may have been shuffled around during a code refactoring or cleanup.

Restoring order independence

Luckily, the remedy is fairly simple. All we need is a small correction in some situations.

Specifically, if unification is solving `P :≡ T` and

`P` is a type parameter which already has inferred a type `A`: `P ➞ A`

`A :≡ T` is true

`T` is a named type

then set the inferred type for `P` to `T`: `P ➞ T`

This ensures that `P` is the named type if there is choice, no matter at which point the named type appeared in a match against `P` (i.e., no matter in which order the type equations are solved). Note that if different named types match against the same type parameter, we always have a unfication failure because different named types are not identical by definition.

Because we made similar simplifications for channels and interfaces, they also need similar special handling. For instance, we ignore channel directions when unifying for assignability and as a result may infer a directed or bidirectional channel depending on argument order. Similar problems occur with interfaces. We're not going to discuss these here.

Going back to our example, if unification starts with equation 1, it infers `P ➞ struct{}` as before. When it proceeds with equation 2, as before, unification succeeds, but now we have exactly the condition that calls for a correction: `P` is a type parameter which already has a type (`struct{}`), `struct{}`, `struct{} :≡ T` is true (because `struct{} ≡ under(T)` is true), and `T` is a named type. Thus, unification makes the correction and sets `P ➞ T`. As a result, irrespective of the unification order, the result is the same (`T`) in both cases.

Self-recursive functions

Another scenario that causes problems in a naive implementation of inference is self-recursive functions. Let's consider a generic factorial function `fact`, defined such that it also works for floating-point arguments (playground). Note that this is not a mathematically correct implementation of the gamma function, it is simply a convenient example.

```
func fact[P ~int | ~float64](n P) P {
    if n <= 1 {
        return 1
    }
    return fact(n-1) * n
}
```

The point here is not the factorial function but rather that `fact` calls itself with the argument `n-1` which is of the same type `P` as the incoming parameter `n`. In this call, the type parameter `P` is simultaneously a bound and a free type parameter: it is bound because it is declared by `fact`, the function that we are calling recursively. But it is also free because it is declared by the function enclosing the call, which happens to also be `fact`.

The equation resulting from passing the argument `n-1` to parameter `n` pits `P` against itself:

```
𝑻(n) :≡ 𝑻(n-1) => P :≡ P
```

Unification sees the same `P` on either side of the equation. Unification succeeds since both types are identical but there's no information gained and `P` remains without an inferred type. As a consequence, type inference fails.

Luckily, the trick to address this is simple: Before type inference is invoked, and for (temporary) use by type inference only, the compiler renames the type parameters in the signatures (but not the bodies) of all functions involved in the respective call. This doesn't change the meaning of the function signatures: they denote the same generic functions irrespective of what the names of the type parameters are.

For the purpose of this example, let's assume the `P` in the signature of `fact` got renamed to `Q`. The effect is as if the recursive call was done indirectly through a `helper` function (playground):

```
func fact[P ~int | ~float64](n P) P {
    if n <= 1 {
        return 1
    }
    return helper(n-1) * n
}

func helper[Q ~int | ~float64](n Q) Q {
    return fact(n)
}
```

With the renaming, or with the `helper` function, the equation resulting from passing `n-1` to the recursive call of `fact` (or the `helper` function, respectively) changes to

```
𝑻(n) :≡ 𝑻(n-1) => Q :≡ P
```

This equation has two type parameters: the bound type parameter `Q`, declared by the function that is being called, and the free type parameter `P`, declared by the enclosing function. This type equation is trivially solved for `Q` and results in the inference `Q → P` which is of course what we'd expect, and which we can verify by explicitly instantiating the recursive call (playground):

```
func fact[P ~int | ~float64](n P) P {
    if n <= 1 {
        return 1
    }
    return fact[P](n-1) * n
}
```

What’s missing?

Conspicuously absent from our description is type inference for generic types: currently generic types must always be explicitly instantiated.

There are a couple of reasons for this. First of all, for type instantiation, type inference only has type arguments to work with; there are no other arguments as is the case for function calls. As a consequence, at least one type argument must always be provided (except for pathological cases where type constraints prescribe exactly one possible type argument for all type parameters). Thus, type inference for types is only useful to complete a partially instantiated type where all the omitted type arguments can be inferred from the equations resulting from type constraints; i.e., where there are at least two type parameters. We believe this is not a very common scenario.

Second, and more pertinent, type parameters allow an entirely new kind of recursive types. Consider the hypothetical type

```
type T[P T[P]] interface{ … }
```

where the constraint for `P` is the type being declared. Combined with the ablity to have multiple type parameters that may refer to each other in complex recursive fashion, type inference becomes much more complicated and we don’t fully understand all the implications of that at the moment. That said, we believe it shouldn’t be too hard to detect cycles and proceed with type inference where no such cycles exist.

Finally, there are situations where type inference is simply not strong enough to make an inference, typically because unification works with certain simplifying assumptions such as the ones described earlier in this post. The primary example here is constraints which have no core type, but where a more sophisticated approach might be able to infer type information anyway.

These are all areas where we may see incremental improvements in future Go releases. Importantly, we believe that cases where inference currently fails are either rare or unimportant in production code, and that our current implementation covers a large majority of all useful code scenarios.

That said, if you run into a situation where you believe type inference should work or went astray, please file an issue! As always, the Go team loves to hear from you, especially when it helps us making Go even better.