Updating tools to support type parameters.

This guide is maintained by Rob Findley (`rfindley@google.com`).

status: this document is currently a rough-draft. See golang/go#50447 for more details.

1. Who should read this guide
2. Introduction
3. Summary of new language features and their APIs
4. Examples
    1. Generic types: type parameters
    2. Constraint Interfaces
    3. Instantiation
    4. Generic types continued: method sets and predicates
5. Updating tools while building at older Go versions
6. Further help

Who should read this guide

Read this guide if you are a tool author seeking to update your tools to support generics Go code. Generics introduce significant new complexity to the Go type system, because types can now be *parameterized*. While the fundamentals of the `go/types` APIs remain the same, some previously valid assumptions no longer hold. For example:

- Type declarations need not correspond 1:1 with the types they define.
- Interfaces are no longer determined entirely by their method set.
- The set of concrete types implementing `types.Type` has grown to include `types.TypeParam` and `types.Union`.

Introduction

With Go 1.18, Go now supports generic programming via type parameters. This document is a guide for tool authors that want to update their tools to support the new language constructs.

This guide assumes knowledge of the language changes to support generics. See the following references for more information:

- The original proposal for type parameters.
- The addendum for type sets.
- The latest language specification (still in-progress as of 2021-01-11).
- The proposals for new APIs in go/token and go/ast, and in go/types.

It also assumes knowledge of `go/ast` and `go/types`. If you're just getting started, x/example/gotypes is a great introduction (and was the inspiration for this guide).

Summary of new language features and their APIs

The introduction of of generic features appears as a large change to the language, but

a high level introduces only a few new concepts. We can break down our discussion into the following three broad categories: generic types, constraint interfaces, and instantiation. In each category below, the relevant new APIs are listed (some constructors and getters/setters may be elided where they are trivial):

Generic types. Types and functions may be *generic*, meaning their declaration may have a non-empty *type parameter list*, as in `type List[T any] ...` or `func f[T1, T2 any]() { ... }`. Type parameter lists define placeholder types (*type parameters*), scoped to the declaration, which may be substituted by any type satisfying their corresponding *constraint interface* to *instantiate* a new type or function.

Generic types may have methods, which declare `receiver type parameters` via their receiver type expression: `func (r T[P1, ..., PN]) method(...) (...) {...}`.

New APIs: - The field `ast.TypeSpec.TypeParams` holds the type parameter list syntax for type declarations. - The field `ast.FuncType.TypeParams` holds the type parameter list syntax for function declarations. - The type `types.TypeParam` is a `types.Type` representing a type parameter. On this type, the `Constraint` and `SetConstraint` methods allow getting/setting the constraint, the `Index` method returns the numeric index of the type parameter in the type parameter list that declares it, and the `Obj` method returns the object in the scope a for the type parameter (a `types.TypeName`). Generic type declarations have a new `*types.Scope` for type parameter declarations. - The type `types.TypeParamList` holds a list of type parameters. - The method `types.Named.TypeParams` returns the type parameters for a type declaration. - The method `types.Named.SetTypeParams` sets type parameters on a defined type. - The function `types.NewSignatureType` creates a new (possibly generic) signature type. - The method `types.Signature.RecvTypeParams` returns the receiver type parameters for a method. - The method `types.Signature.TypeParams` returns the type parameters for a function.

Constraint Interfaces: type parameter constraints are interfaces, expressed by an interface type expression. Interfaces that are only used in constraint position are permitted new embedded elements composed of tilde expressions (`~T`) and unions (`A | B | ~C`). The new builtin interface type `comparable` is implemented by types for which `==` and `!=` are valid (note that interfaces must be statically comparable in this case, i.e., each type in the interface's type set must be comparable). As a special case, the `interface` keyword may be omitted from constraint expressions if it may be implied (in which case we say the interface is *implicit*).

New APIs: - The constant `token.TILDE` is used to represent tilde expressions as an `ast.UnaryExpr`. - Union expressions are represented as an `ast.BinaryExpr` using `|`. This means that `ast.BinaryExpr` may now be both a type and value expression. - The method `types.Interface.IsImplicit` reports whether the `interface` keyword was elided from this interface. - The method `types.Interface.MarkImplicit` marks an interface as being implicit. - The method `types.Interface.IsComparable` reports whether every type in an interface's type set is comparable. - The method `types.Interface.IsMethodSet` reports whether an interface is defined entirely by its methods (has no *specific types*). - The type `types.Union` is a type that represents an embedded union expression in an interface. May only appear as an embedded element in interfaces. - The type `types.Term` represents a (possibly tilde) term of a union.

Instantiation: generic types and functions may be *instantiated* to create non-generic types and functions by providing *type arguments* (`var x T[int]`). Function type arguments may be *inferred* via function arguments, or via type parameter constraints.

New APIs: - The type `ast.IndexListExpr` holds index expressions with multiple indices, as in instantiation expressions with multiple type arguments or in receivers declaring multiple type parameters. - The function `types.Instantiate` instantiates a generic type with type arguments. - The type `types.Context` is an opaque instantiation context that may be shared to reduce duplicate instances. - The field `types.Config.Context` holds a shared `Context` to use for instantiation while type-checking. - The type `types.TypeList`

holds a list of types. - The type `types.ArgumentError` holds an error associated with a specific type argument index. Used to represent instantiation errors. - The field `types.Info.Instances` maps instantiated identifiers to information about the resulting type instance. - The type `types.Instance` holds information about a type or function instance. - The method `types.Named.TypeArgs` reports the type arguments used to instantiate a named type.

Examples

The following examples demonstrate the new APIs, and discuss their properties. All examples are runnable, contained in subdirectories of the directory holding this README.

Generic types: type parameters

We say that a type is *generic* if it has type parameters but no type arguments. This section explains how we can inspect generic types with the new `go/types` APIs.

Type parameter lists

Suppose we want to understand the generic library below, which defines a generic `Pair`, a constraint interface `Constraint`, and a generic function `MakePair`.

```
package main

type Constraint interface {
    Value() any
}

type Pair[L, R any] struct {
    left  L
    right R
}

func MakePair[L, R Constraint](l L, r R) Pair[L, R] {
    return Pair[L, R]{l, r}
}
```

We can use the new `TypeParams` fields in `ast.TypeSpec` and `ast.FuncType` to access the type parameter list. From there, we can access type parameter types in at least three ways: - by looking up type parameter definitions in `types.Info` - by calling `TypeParams()` on `types.Named` or `types.Signature` - by looking up type parameter objects in the declaration scope. Note that there now may be a scope associated with an `ast.TypeSpec` node.

```
func PrintTypeParams(fset *token.FileSet, file *ast.File) error {
    conf := types.Config{Importer: importer.Default()}
    info := &types.Info{
        Scopes: make(map[ast.Node]*types.Scope),
        Defs:   make(map[*ast.Ident]types.Object),
    }
    _, err := conf.Check("hello", fset, []*ast.File{file}, info)
    if err != nil {
        return err
    }

    // For convenience, we can use ast.Inspect to find the nodes we want to
    // investigate.
```

```
    ast.Inspect(file, func(n ast.Node) bool {
        var name *ast.Ident              // the name of the generic object, or nil
        var tparamFields *ast.FieldList  // the list of type parameter fields
        var tparams *types.TypeParamList // the list of type parameter types
        var scopeNode ast.Node           // the node associated with the declaration
scope

        switch n := n.(type) {
        case *ast.TypeSpec:
            name = n.Name
            tparamFields = n.TypeParams
            tparams = info.Defs[name].Type().(*types.Named).TypeParams()
            scopeNode = n
        case *ast.FuncDecl:
            name = n.Name
            tparamFields = n.Type.TypeParams
            tparams = info.Defs[name].Type().(*types.Signature).TypeParams()
            scopeNode = n.Type
        default:
            // Not a type or function declaration.
            return true
        }

        // Option 1: find type parameters by looking at their declaring field list.
        if tparamFields != nil {
            fmt.Printf("%s has a type parameter field list with %d fields\n",
name.Name, tparamFields.NumFields())
            for _, field := range tparamFields.List {
                for _, name := range field.Names {
                    tparam := info.Defs[name]
                    fmt.Printf("  field %s defines an object %q\n", name.Name,
tparam)
                }
            }
        } else {
            fmt.Printf("%s does not have a type parameter list\n", name.Name)
        }

        // Option 2: find type parameters via the TypeParams() method on the
        // generic type.
        if tparams.Len() > 0 {
            fmt.Printf("%s has %d type parameters:\n", name.Name, tparams.Len())
            for i := 0; i < tparams.Len(); i++ {
                tparam := tparams.At(i)
                fmt.Printf("  %s has constraint %s\n", tparam, tparam.Constraint())
            }
        } else {
            fmt.Printf("%s does not have type parameters\n", name.Name)
        }

        // Option 3: find type parameters by looking in the declaration scope.
        scope, ok := info.Scopes[scopeNode]
        if ok {
            fmt.Printf("%s has a scope with %d objects:\n", name.Name, scope.Len())
            for _, name := range scope.Names() {
                fmt.Printf("  %s is a %T\n", name, scope.Lookup(name))
            }
        } else {
            fmt.Printf("%s does not have a scope\n", name.Name)
        }

        return true
    })
    return nil
```

```
}
```

This program produces the following output. Note that not every type spec has a scope.

```
> go run golang.org/x/tools/internal/typeparams/example/findtypeparams
Constraint does not have a type parameter list
Constraint does not have type parameters
Constraint does not have a scope
Pair has a type parameter field list with 2 fields
  field L defines an object "type parameter L any"
  field R defines an object "type parameter R any"
Pair has 2 type parameters:
  L has constraint any
  R has constraint any
Pair has a scope with 2 objects:
  L is a *types.TypeName
  R is a *types.TypeName
MakePair has a type parameter field list with 2 fields
  field L defines an object "type parameter L hello.Constraint"
  field R defines an object "type parameter R hello.Constraint"
MakePair has 2 type parameters:
  L has constraint hello.Constraint
  R has constraint hello.Constraint
MakePair has a scope with 4 objects:
  L is a *types.TypeName
  R is a *types.TypeName
  l is a *types.Var
  r is a *types.Var
```

Constraint Interfaces

In order to allow operations on type parameters, Go 1.18 introduces the notion of *type sets*, which is abstractly the set of types that implement an interface. This section discusses the new syntax for restrictions on interface type sets, and the APIs we can use to understand them.

New interface elements

Consider the generic library below:

```
package p

type Numeric interface{
    ~int | ~float64 // etc...
}

func Square[N Numeric](n N) N {
    return n*n
}

type Findable interface {
    comparable
}

func Find[T Findable](s []T, v T) int {
    for i, v2 := range s {
        if v2 == v {
            return i
```

```
        }
    }
    return -1
}
```

In this library, we can see a few new features added in Go 1.18. The first is the new syntax in the `Numeric` type: unions of tilde-terms, specifying that the numeric type may only be satisfied by types whose underlying type is `int` or `float64`.

The `go/ast` package parses this new syntax as a combination of unary and binary expressions, which we can see using the following program:

```
func PrintNumericSyntax(fset *token.FileSet, file *ast.File) {
    // node is the AST node corresponding to the declaration for "Numeric."
    node := file.Scope.Lookup("Numeric").Decl.(*ast.TypeSpec)
    // Find the embedded syntax node.
    embedded := node.Type.(*ast.InterfaceType).Methods.List[0].Type
    // Use go/ast's built-in Print function to inspect the parsed syntax.
    ast.Print(fset, embedded)
}
```

Output:

```
     0  *ast.BinaryExpr {
     1  .  X: *ast.UnaryExpr {
     2  .  .  OpPos: p.go:6:2
     3  .  .  Op: ~
     4  .  .  X: *ast.Ident {
     5  .  .  .  NamePos: p.go:6:3
     6  .  .  .  Name: "int"
     7  .  .  }
     8  .  }
     9  .  OpPos: p.go:6:7
    10  .  Op: |
    11  .  Y: *ast.UnaryExpr {
    12  .  .  OpPos: p.go:6:9
    13  .  .  Op: ~
    14  .  .  X: *ast.Ident {
    15  .  .  .  NamePos: p.go:6:10
    16  .  .  .  Name: "float64"
    17  .  .  }
    18  .  }
    19  }
```

Once type-checked, these embedded expressions are represented using the new `types.Union` type, which flattens the expression into a list of `*types.Term`. We can also investigate two new methods of interface: `types.Interface.IsComparable`, which reports whether the type set of an interface is comparable, and `types.Interface.IsMethodSet`, which reports whether an interface is expressible using methods alone.

```
func PrintInterfaceTypes(fset *token.FileSet, file *ast.File) error {
    conf := types.Config{}
    pkg, err := conf.Check("hello", fset, []*ast.File{file}, nil)
    if err != nil {
        return err
    }

    PrintIface(pkg, "Numeric")
    PrintIface(pkg, "Findable")
```

```
    return nil
}

func PrintIface(pkg *types.Package, name string) {
    obj := pkg.Scope().Lookup(name)
    fmt.Println(obj)
    iface := obj.Type().Underlying().(*types.Interface)
    for i := 0; i < iface.NumEmbeddeds(); i++ {
        fmt.Printf("\tembeded: %s\n", iface.EmbeddedType(i))
    }
    for i := 0; i < iface.NumMethods(); i++ {
        fmt.Printf("\tembeded: %s\n", iface.EmbeddedType(i))
    }
    fmt.Printf("\tIsComparable(): %t\n", iface.IsComparable())
    fmt.Printf("\tIsMethodSet(): %t\n", iface.IsMethodSet())
}
```

Output:

```
type hello.Numeric interface{~int|~float64}
        embeded: ~int|~float64
        IsComparable(): true
        IsMethodSet(): false
type hello.Findable interface{comparable}
        embeded: comparable
        IsComparable(): true
        IsMethodSet(): false
```

The `Findable` type demonstrates another new feature of Go 1.18: the comparable built-in. Comparable is a special interface type, not expressible using ordinary Go syntax, whose type-set consists of all comparable types.

Implicit interfaces

For interfaces that do not have methods, we can inline them in constraints and elide the `interface` keyword. In the example above, we could have done this for the `Square` function:

```
package p

func Square[N ~int|~float64](n N) N {
    return n*n
}
```

In such cases, the `types.Interface.IsImplicit` method reports whether the interface type was implicit. This does not affect the behavior of the interface, but is captured for more accurate type strings:

```
func ShowImplicit(pkg *types.Package) {
    Square := pkg.Scope().Lookup("Square").Type().(*types.Signature)
    N := Square.TypeParams().At(0)
    constraint := N.Constraint().(*types.Interface)
    fmt.Println(constraint)
    fmt.Println("IsImplicit:", constraint.IsImplicit())
}
```

Output:

```
~int|~float64
IsImplicit: true
```

The `types.Interface.MarkImplicit` method is used to mark interfaces as implicit by the importer.

Type sets

The examples above demonstrate the new APIs for *accessing* information about the new interface elements, but how do we understand *type sets*, the new abstraction that these elements help define? Type sets may be arbitrarily complex, as in the following example:

```
package complex

type A interface{ ~string|~[]byte }

type B interface{ int|string }

type C interface { ~string|~int }

type D interface{ A|B; C }
```

Here, the type set of `D` simplifies to `~string|int`, but the current `go/types` APIs do not expose this information. This will likely be added to `go/types` in future versions of Go, but in the meantime we can use the `typeparams.NormalTerms` helper:

```
func PrintNormalTerms(pkg *types.Package) error {
    D := pkg.Scope().Lookup("D").Type()
    terms, err := typeparams.NormalTerms(D)
    if err != nil {
        return err
    }
    for i, term := range terms {
        if i > 0 {
            fmt.Print("|")
        }
        fmt.Print(term)
    }
    fmt.Println()
    return nil
}
```

which outputs:

```
~string|int
```

See the documentation for `typeparams.NormalTerms` for more information on how this calculation proceeds.

Instantiation

We say that a type is *instantiated* if it is created from a generic type by substituting type arguments for type parameters. Instantiation can occur via explicitly provided type arguments, as in the expression `T[A_1, ..., A_n]`, or implicitly, through type inference.. This section describes how to find and understand instantiated types.

Finding instantiated types

Certain applications may find it useful to locate all instantiated types in a package. For this purpose, `go/types` provides a new `types.Info.Instances` field that maps instantiated identifiers to information about their instance.

For example, consider the following code:

```
package p

type Pair[L, R comparable] struct {
    left  L
    right R
}

func (p Pair[L, _]) Left() L {
    return p.left
}

func Equal[L, R comparable](x, y Pair[L, R]) bool {
    return x.left == y.left && x.right == y.right
}

var X Pair[int, string]
var Y Pair[string, int]

var E = Equal[int, string]
```

We can find instances by type-checking with the `types.Info.Instances` map initialized:

```
func CheckInstances(fset *token.FileSet, file *ast.File) (*types.Package, error) {
    conf := types.Config{}
    info := &types.Info{
        Instances: make(map[*ast.Ident]types.Instance),
    }
    pkg, err := conf.Check("p", fset, []*ast.File{file}, info)
    for id, inst := range info.Instances {
        posn := fset.Position(id.Pos())
        fmt.Printf("%s: %s instantiated with %s: %s\n", posn, id.Name,
FormatTypeList(inst.TypeArgs), inst.Type)
    }
    return pkg, err
}
```

Output:

```
hello.go:21:9: Equal instantiated with [int, string]: func(x p.Pair[int, string], y
p.Pair[int, string]) bool
hello.go:10:9: Pair instantiated with [L, _]: p.Pair[L, _]
hello.go:14:34: Pair instantiated with [L, R]: p.Pair[L, R]
hello.go:18:7: Pair instantiated with [int, string]: p.Pair[int, string]
hello.go:19:7: Pair instantiated with [string, int]: p.Pair[string, int]
```

The `types.Instance` type provides information about the (possibly inferred) type arguments that were used to instantiate the generic type, and the resulting type. Notably, it does not include the *generic* type that was instantiated, because this type can be found using `types.Info.Uses[id].Type()` (where `id` is the identifier node being instantiated).

Note that the receiver type of method `Left` also appears in the `Instances` map. This may be counterintuitive – more on this below.

Creating new instantiated types

`go/types` also provides an API for creating type instances: `types.Instantiate`. This function accepts a generic type and type arguments, and returns an instantiated type (or an error). The resulting instance may be a newly constructed type, or a previously created instance with the same type identity. To facilitate the reuse of frequently used instances, `types.Instantiate` accepts a `types.Context` as its first argument, which records instances.

If the final `validate` argument to `types.Instantiate` is set, the provided type arguments will be verified against their corresponding type parameter constraint; i.e., `types.Instantiate` will check that each type arguments implements the corresponding type parameter constraint. If a type arguments does not implement the respective constraint, the resulting error will wrap a new `ArgumentError` type indicating which type argument index was bad.

```
func Instantiate(pkg *types.Package) error {
    Pair := pkg.Scope().Lookup("Pair").Type()
    X := pkg.Scope().Lookup("X").Type()
    Y := pkg.Scope().Lookup("Y").Type()

    // X and Y have different types, because their type arguments are different.
    Compare("X", "Y", X, Y)

    // Create a shared context for the subsequent instantiations.
    ctxt := types.NewContext()

    // Instantiating with [int, string] yields an instance that is identical (but
    // not equal) to X.
    Int, String := types.Typ[types.Int], types.Typ[types.String]
    inst1, _ := types.Instantiate(ctxt, Pair, []types.Type{Int, String}, true)
    Compare("X", "inst1", X, inst1)

    // Instantiating again returns the same exact instance, because of the shared
    // Context.
    inst2, _ := types.Instantiate(ctxt, Pair, []types.Type{Int, String}, true)
    Compare("inst1", "inst2", inst1, inst2)

    // Instantiating with 'any' is an error, because any is not comparable.
    Any := types.Universe.Lookup("any").Type()
    _, err := types.Instantiate(ctxt, Pair, []types.Type{Int, Any}, true)
    var argErr *types.ArgumentError
    if errors.As(err, &argErr) {
        fmt.Printf("Argument %d: %v\n", argErr.Index, argErr.Err)
    }

    return nil
}

func Compare(leftName, rightName string, left, right types.Type) {
    fmt.Printf("Identical(%s, %s) : %t\n", leftName, rightName, types.Identical(left,
right))
    fmt.Printf("%s == %s : %t\n\n", leftName, rightName, left == right)
}
```

Output:

```
Identical(p.Pair[int, string], p.Pair[string, int]) : false
p.Pair[int, string] == p.Pair[string, int] : false

Identical(p.Pair[int, string], p.Pair[int, string]) : true
p.Pair[int, string] == p.Pair[int, string] : false

Identical(p.Pair[string, int], p.Pair[int, string]) : false
p.Pair[string, int] == p.Pair[int, string] : false

Identical(p.Pair[int, string], p.Pair[int, string]) : true
p.Pair[int, string] == p.Pair[int, string] : true

Argument 1: any does not implement comparable
```

Using a shared context while type checking

To share a common `types.Context` argument with a type-checking pass, set the new `types.Config.Context` field.

Generic types continued: method sets and predicates

Generic types are fundamentally different from ordinary types, in that they may not be used without instantiation. In some senses they are not really types: the go spec defines types as "a set of values, together with operations and methods", but uninstantiated generic types do not define a set of values. Rather, they define a set of *types*. In that sense, they are a "meta type", or a "type template" (disclaimer: I am using these terms imprecisely).

However, for the purposes of `go/types` it is convenient to treat generic types as a `types.Type`. This section explains how generic types behave in existing `go/types` APIs.

Method Sets

Methods on uninstantiated generic types are different from methods on an ordinary type. Consider that for an ordinary type `T`, the receiver base type of each method in its method set is `T`. However, this can't be the case for a generic type: generic types cannot be used without instantiation, and neither can the type of the receiver variable. Instead, the receiver base type is an *instantiated* type, instantiated with the method's receiver type parameters.

This has some surprising consequences, which we observed in the section on instantiation above: for a generic type `G`, each of its methods will define a unique instantiation of `G`, as each method has distinct receiver type parameters.

To see this, consider the following example:

```
package p

type Pair[L, R any] struct {
    left  L
    right R
}

func (p Pair[L, _]) Left() L {
    return p.left
}
```

```
func (p Pair[_, R]) Right() R {
    return p.right
}

var IntPair Pair[int, int]
```

Let’s inspect the method sets of the types in this library:

```
func PrintMethods(pkg *types.Package) {
    // Look up *Named types in the package scope.
    lookup := func(name string) *types.Named {
        return pkg.Scope().Lookup(name).Type().(*types.Named)
    }

    Pair := lookup("Pair")
    IntPair := lookup("IntPair")
    PrintMethodSet("Pair", Pair)
    PrintMethodSet("Pair[int, int]", IntPair)
    LeftObj, _, _ := types.LookupFieldOrMethod(Pair, false, pkg, "Left")
    LeftRecvType := LeftObj.Type().(*types.Signature).Recv().Type()
    PrintMethodSet("Pair[L, _]", LeftRecvType)
}

func PrintMethodSet(name string, typ types.Type) {
    fmt.Println(name + ":")
    methodSet := types.NewMethodSet(typ)
    for i := 0; i < methodSet.Len(); i++ {
        method := methodSet.At(i).Obj()
        fmt.Println(method)
    }
    fmt.Println()
}
```

Output:

```
Pair:
func (p.Pair[L, _]).Left() L
func (p.Pair[_, R]).Right() R

Pair[int, int]:
func (p.Pair[int, int]).Left() int
func (p.Pair[int, int]).Right() int

Pair[L, _]:
func (p.Pair[L, _]).Left() L
func (p.Pair[L, _]).Right() _
```

In this example, we can see that all of `Pair`, `Pair[int, int]`, and `Pair[L, _]` have distinct method sets, though the method set of `Pair` and `Pair[L, _]` intersect in the `Left` method.

Only the objects in `Pair`’s method set are recorded in `types.Info.Defs`. To get back to this “canonical” method object, the `typeparams` package provides the `OriginMethod` helper:

```
func CompareOrigins(pkg *types.Package) {
    Pair := pkg.Scope().Lookup("Pair").Type().(*types.Named)
    IntPair := pkg.Scope().Lookup("IntPair").Type().(*types.Named)
    Left, _, _ := types.LookupFieldOrMethod(Pair, false, pkg, "Left")
```

```
    LeftInt, _, _ := types.LookupFieldOrMethod(IntPair, false, pkg, "Left")

    fmt.Println("Pair.Left == Pair[int, int].Left:", Left == LeftInt)
    origin := typeparams.OriginMethod(LeftInt.(*types.Func))
    fmt.Println("Pair.Left == OriginMethod(Pair[int, int].Left):", Left == origin)
}
```

Output:

```
Pair.Left == Pair[int, int].Left: false
Pair.Left == OriginMethod(Pair[int, int].Left): true
```

Predicates

Predicates on generic types are not defined by the spec. As a consequence, using e.g. `types.AssignableTo` with operands of generic types leads to an undefined result.

The behavior of predicates on generic `*types.Named` types may generally be derived from the fact that type parameters bound to different names are different types. This means that most predicates involving generic types will return `false`.

`*types.Signature` types are treated differently. Two signatures are considered identical if they are identical after substituting one's set of type parameters for the other's, including having identical type parameter constraints. This is analogous to the treatment of ordinary value parameters, whose names do not affect type identity.

Consider the following code:

```
func OrdinaryPredicates(pkg *types.Package) {
    var (
        Pair        = pkg.Scope().Lookup("Pair").Type()
        LeftRighter = pkg.Scope().Lookup("LeftRighter").Type()
        Mer         = pkg.Scope().Lookup("Mer").Type()
        F           = pkg.Scope().Lookup("F").Type()
        G           = pkg.Scope().Lookup("G").Type()
        H           = pkg.Scope().Lookup("H").Type()
    )

    fmt.Println("AssignableTo(Pair, LeftRighter)", types.AssignableTo(Pair,
LeftRighter))
    fmt.Println("AssignableTo(Pair, Mer): ", types.AssignableTo(Pair, Mer))
    fmt.Println("Identical(F, G)", types.Identical(F, G))
    fmt.Println("Identical(F, H)", types.Identical(F, H))
}
```

Output:

```
AssignableTo(Pair, LeftRighter) false
AssignableTo(Pair, Mer):  true
Identical(F, G) true
Identical(F, H) false
```

In this example, we see that despite their similarity the generic `Pair` type is not assignable to the generic `LeftRighter` type. We also see the rules for signature identity in practice.

This begs the question: how does one ask questions about the relationship between generic types? In order to phrase such questions we need more information: how does

one relate the type parameters of `Pair` to the type parameters of `LeftRighter`? Does it suffice for the predicate to hold for one element of the type sets, or must it hold for all elements of the type sets?

We can use instantiation to answer some of these questions. In particular, by instantiating both `Pair` and `LeftRighter` with the type parameters of `Pair`, we can determine if, for all type arguments `[X, Y]` that are valid for `Pair`, `[X, Y]` are also valid type arguments of `LeftRighter`, and `Pair[X, Y]` is assignable to `LeftRighter[X, Y]`. The `typeparams.GenericAssignableTo` function implements exactly this predicate:

```
func GenericPredicates(pkg *types.Package) {
    var (
        Pair        = pkg.Scope().Lookup("Pair").Type()
        LeftRighter = pkg.Scope().Lookup("LeftRighter").Type()
    )
    fmt.Println("GenericAssignableTo(Pair, LeftRighter)",
typeparams.GenericAssignableTo(nil, Pair, LeftRighter))
}
```

Output:

```
GenericAssignableTo(Pair, LeftRighter) true
```

Updating tools while building at older Go versions

In the examples above, we can see how a lot of the new APIs integrate with existing usage of `go/ast` or `go/types`. However, most tools still need to build at older Go versions, and handling the new language constructs in-line will break builds at older Go versions.

For this purpose, the `x/exp/typeparams` package provides functions and types that proxy the new APIs (with stub implementations at older Go versions).

Further help

If you’re working on updating a tool to support generics, and need help, please feel free to reach out for help in any of the following ways: - By mailing the golang-tools mailing list. - Directly to me via email (`rfindley@google.com`). - For bugs, you can file an issue.