`go/types`: The Go Type Checker

This document is maintained by Alan Donovan `adonovan@google.com`.

October 2015 GothamGo talk on go/types

Contents

Changes in Go 1.18

Go 1.18 introduces generics, and several corresponding new APIs for `go/types`. This document is not yet up-to-date for these changes, but a guide to the new changes

exists at `x/exp/typeparams/example`.

Introduction

The `go/types` package is a type-checker for Go programs, designed by Robert Griesemer. It became part of Go's standard library in Go 1.5. Measured by lines of code and by API surface area, it is one of the most complex packages in Go's standard library, and using it requires a firm grasp of the structure of Go programs. This tutorial will help you find your bearings. It comes with several example programs that you can obtain with `go get` and play with. We assume you are a proficient Go programmer who wants to build tools to analyze or manipulate Go programs and that you have some knowledge of how a typical compiler works.

The type checker complements several existing standard packages for analyzing Go programs. We've listed them below.

```
→   go/types
    go/constant
    go/parser
    go/ast
    go/scanner
    go/token
```

Starting at the bottom, the `go/token` package defines the lexical tokens of Go. The `go/scanner` package tokenizes an input stream and records file position information for use in diagnostics or for file surgery in a refactoring tool. The `go/ast` package defines the data types of the abstract syntax tree (AST). The `go/parser` package provides a robust recursive-descent parser that constructs the AST. And `go/constant` provides representations and arithmetic operations for the values of compile-time constant expressions, as we'll see in Constants.

The `golang.org/x/tools/go/packages` package from the `x/tools` repository is a client of the type checker that loads, parses, and type-checks a complete Go program from source code. We use it in some of our examples and you may find it useful too.

The Go type checker does three main things. First, for every name in the program, it determines which declaration the name refers to; this is known as *identifier resolution*. Second, for every expression in the program, it determines what type that expression has, or reports an error if the expression has no type, or has an inappropriate type for its context; this is known as *type deduction*. Third, for every constant expression in the program, it determines the value of that constant; this is known as *constant evaluation*. Superficially, it appears that these three processes could be done sequentially, in the order above, but perhaps surprisingly, they must be done together. For example, the value of a constant may depend on the type of an expression due to operators like `unsafe.Sizeof`. Conversely, the type of an expression may depend on the value of a constant, since array types contain constants. As a result, type deduction and constant evaluation must be done together. As another example, we cannot resolve the identifier `k` in the composite literal `T{k: 0}` until we know whether

`T` is a struct type. If it is, then `k` must be found among `T`'s fields. If not, then `k` is an ordinary reference to a constant or variable in the lexical environment. Consequently, identifier resolution and type deduction are also inseparable in the general case.

Nonetheless, the three processes of identifier resolution, type deduction, and constant evaluation can be separated for the purpose of explanation.

An Example

The code below shows the most basic use of the type checker to check the *hello, world* program, supplied as a string. Later examples will be variations on this one, and we'll often omit boilerplate details such as parsing. To check out and build the examples, run `go get golang.org/x/example/gotypes/...`.

```
// go get golang.org/x/example/gotypes/pkginfo
```

```
package main

import (
    "fmt"
    "go/ast"
    "go/importer"
    "go/parser"
    "go/token"
    "go/types"
    "log"
)

const hello = `package main

import "fmt"

func main() {
        fmt.Println("Hello, world")
}`

func main() {
    fset := token.NewFileSet()

    // Parse the input string, []byte, or io.Reader,
    // recording position information in fset.
    // ParseFile returns an *ast.File, a syntax tree.
    f, err := parser.ParseFile(fset, "hello.go", hello, 0)
    if err != nil {
        log.Fatal(err) // parse error
    }

    // A Config controls various options of the type checker.
    // The defaults work fine except for one setting:
    // we must specify how to deal with imports.
    conf := types.Config{Importer: importer.Default()}
```

```
    // Type-check the package containing only file f.
    // Check returns a *types.Package.
    pkg, err := conf.Check("cmd/hello", fset, []*ast.File{f}, nil)
    if err != nil {
        log.Fatal(err) // type error
    }

    fmt.Printf("Package  %q\n", pkg.Path())
    fmt.Printf("Name:    %s\n", pkg.Name())
    fmt.Printf("Imports: %s\n", pkg.Imports())
    fmt.Printf("Scope:   %s\n", pkg.Scope())
}
```

First, the program creates a `token.FileSet`. To avoid the need to store file names and line and column numbers in every node of the syntax tree, the `go/token` package provides `FileSet`, a data structure that stores this information compactly for a sequence of files. A `FileSet` records each file name only once, and records only the byte offsets of each newline, allowing a position within any file to be identified using a small integer called a `token.Pos`. Many tools create a single `FileSet` at startup. Any part of the program that needs to convert a `token.Pos` into an intelligible location—as part of an error message, for instance—must have access to the `FileSet`.

Second, the program parses the input string. More realistic packages contain several source files, so the parsing step must be repeated for each one, or better, done in parallel. Third, it creates a `Config` that specifies type-checking options. Since the *hello, world* program uses imports, we must indicate how to locate the imported packages. Here we use `importer.Default()`, which loads compiler-generated export data, but we'll explore alternatives in Imports.

Fourth, the program calls `Check`. This creates a `Package` whose path is `"cmd/hello"`, and type-checks each of the specified files—just one in this example. The final (nil) argument is a pointer to an optional `Info` struct that returns additional deductions from the type checker; more on that later. `Check` returns a `Package` even when it also returns an error. The type checker is robust to ill-formed input, and goes to great lengths to report accurate partial information even in the vicinity of syntax or type errors. `Package` has this definition:

```
type Package struct{ ... }
func (*Package) Path() string
func (*Package) Name() string
func (*Package) Scope() *Scope
func (*Package) Imports() []*Package
```

Finally, the program prints the attributes of the package, shown below. (The hexadecimal number may vary from one run to the next.)

```
$ go build golang.org/x/example/gotypes/pkginfo
$ ./pkginfo
Package  "cmd/hello"
Name:    main
Imports: [package fmt ("fmt")]
```

```
Scope:   package "cmd/hello" scope 0x820533590 {
.  func cmd/hello.main()
}
```

A package's `Path`, such as `"encoding/json"`, is the string by which import declarations identify it. It is unique within a workspace, and for published packages it must be globally unique.

A package's `Name` is the identifier in the `package` declaration of each source file within the package, such as `json`. The type checker reports an error if not all the package declarations in the package agree. The package name determines how the package is known when it is imported into a file (unless a renaming import is used), but is otherwise not visible to a program.

`Scope` returns the package's *lexical block*, which provides access to all the named entities or *objects* declared at package level. `Imports` returns the set of packages directly imported by this one, and may be useful for computing dependencies (see Initialization Order).

Objects

The task of identifier resolution is to map every identifier in the syntax tree, that is, every `ast.Ident`, to an object. For our purposes, an *object* is a named entity created by a declaration, such as a `var`, `type`, or `func` declaration. (This is different from the everyday meaning of object in object-oriented programming.)

Objects are represented by the `Object` interface:

```
type Object interface {
    Name() string   // package-local object name
    Exported() bool // reports whether the name starts with a capital letter
    Type() Type     // object type
    Pos() token.Pos // position of object identifier in declaration

    Parent() *Scope // scope in which this object is declared
    Pkg() *Package  // nil for objects in the Universe scope and labels
    Id() string     // object id (see Ids section below)
}
```

The first four methods are straightforward; we'll explain the other three later. `Name` returns the object's name—an identifier. `Exported` is a convenience method that reports whether the first letter of `Name` is a capital, indicating that the object may be visible from outside the package. It's a shorthand for `ast.IsExported(obj.Name())`. `Type` returns the object's type; we'll come back to that in Types.

`Pos` returns the source position of the object's declaring identifier. To make sense of a `token.Pos`, we need to call the `(*token.FileSet).Position` method, which returns a struct with individual fields for the file name, line number, column, and byte offset, though usually we just call its `String` method:

```
fmt.Println(fset.Position(obj.Pos())) // "hello.go:10:6"
```

Objects for predeclared functions and types such as `len` and `int` do not have a valid (non-zero) position: `!obj.Pos().IsValid()`.

There are eight kinds of objects in the Go type checker. Most familiar are the kinds that can be declared at package level: constants, variables, functions, and types. Less familiar are statement labels, imported package names (such as `json` in a file containing an `import "encoding/json"` declaration), built-in functions (such as `append` and `len`), and the pre-declared `nil`. The eight types shown below are the only concrete types that satisfy the `Object` interface. In other words, `Object` is a *discriminated union* of 8 possible types, and we commonly use a type switch to distinguish them.

```
Object = *Func          // function, concrete method, or abstract method
       | *Var           // variable, parameter, result, or struct field
       | *Const         // constant
       | *TypeName      // type name
       | *Label         // statement label
       | *PkgName       // package name, e.g. json after import "encoding/json"
       | *Builtin       // predeclared function such as append or len
       | *Nil           // predeclared nil
```

Objects are canonical. That is, two Objects `x` and `y` denote the same entity if and only if `x==y`. (This is generally true but beware that parameterized types complicate matters; see https://github.com/golang/exp/tree/master/typeparams/example for details.)

Object identity is significant, and objects are routinely compared by the addresses of the underlying pointers. A package-level object (func/var/const/type) can be uniquely identified by its name and enclosing package. The `golang.org/x/tools/go/types/objectpath` package defines a naming scheme for objects that are exported from their package or are unexported but form part of the type of an exported object. But for most objects, including all function-local objects, there is no simple way to obtain a string that uniquely identifies it.

The `Parent` method returns the `Scope` (lexical block) in which the object was declared; we'll come back to this in Scopes. Fields and methods are not found in the lexical environment, so their objects have no `Parent`.

The `Pkg` method returns the `Package` to which this object belongs, even for objects not declared at package level. Only predeclared objects have no package. The `Id` method will be explained in Ids.

Not all methods make sense for each kind of object. For instance, the last four kinds above have no meaningful `Type` method. And some kinds of objects have methods in addition to those required by the `Object` interface:

```
func (*Func) Scope() *Scope
```

```
func (*Var) Anonymous() bool
func (*Var) IsField() bool
func (*Const) Val() constant.Value
func (*TypeName) IsAlias() bool
func (*PkgName) Imported() *Package
```

`(*Func).Scope` returns the lexical block containing the function's type parameters, parameters, results, and other local declarations. `(*Var).IsField` distinguishes struct fields from ordinary variables, and `(*Var).Anonymous` discriminates named fields like the one in `struct{T T}` from anonymous fields like the one in `struct{T}`. `(*Const).Val` returns the value of a named constant.

`(*TypeName).IsAlias` reports whether the type name declares an alias for an existing type (as in `type I = int`), as opposed to defining a new `Named` type, as in `type Celsius float64`. (Most `TypeName`s for which `IsAlias()` is true have a `Type()` of type `*types.Alias`, but `IsAlias()` is also true for the predeclared `byte` and `rune` types, which are aliases for `uint8` and `int32`.)

`(*PkgName).Imported` returns the package (for instance, `encoding/json`) denoted by a given import name such as `json`. Each time a package is imported, a new `PkgName` object is created, usually with the same name as the `Package` it denotes, but not always, as in the case of a renaming import. `PkgName`s are objects, but `Package`s are not. We'll look more closely at this in Imports.

All relationships between the syntax trees (`ast.Node`s) and type checker data structures such as `Object`s and `Type`s are stored in mappings outside the syntax tree itself. Be aware that the `go/ast` package also defines an older deprecated type called `Object` that resembles—and predates—the type checker's `Object`, and that `ast.Object`s are held directly by identifiers in the AST. They are created by the parser, which has a necessarily limited view of the package, so the information they represent is at best partial and in some cases wrong, as in the `T{k: 0}` example mentioned above. If you are using the type checker, there is no reason to use the older `ast.Object` mechanism.

Identifier Resolution

Identifier resolution computes the relationship between identifiers and objects. Its results are recorded in the `Info` struct optionally passed to `Check`. The fields related to identifier resolution are shown below.

```
type Info struct {
    Defs       map[*ast.Ident]Object
    Uses       map[*ast.Ident]Object
    Implicits  map[ast.Node]Object
    Selections map[*ast.SelectorExpr]*Selection
    Scopes     map[ast.Node]*Scope
    ...
}
```

Since not all facts computed by the type checker are needed by every client, the API lets clients control which components of the result should be recorded and which discarded: only fields that hold a non-nil map will be populated during the call to `Check`.

The two fields of type `map[*ast.Ident]Object` are the most important: `Defs` records *declaring* identifiers and `Uses` records *referring* identifiers. In the example below, the comments indicate which identifiers are of which kind.

```
var x int        // def of x, use of int
fmt.Println(x)   // uses of fmt, Println, and x
type T struct{U} // def of T, use of U (type), def of U (field)
```

The final line above illustrates why we don't combine `Defs` and `Uses` into one map. In the anonymous field declaration `struct{U}`, the identifier `U` is both a use of the type `U` (a `TypeName`) and a definition of the anonymous field (a `Var`).

The function below prints the location of each referring and defining identifier in the input program, and the object it refers to.

```
// go get golang.org/x/example/gotypes/defsuses
```

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

    for id, obj := range info.Defs {
        fmt.Printf("%s: %q defines %v\n",
            fset.Position(id.Pos()), id.Name, obj)
    }
    for id, obj := range info.Uses {
        fmt.Printf("%s: %q uses %v\n",
            fset.Position(id.Pos()), id.Name, obj)
    }
    return nil
}
```

Let's use the *hello, world* program again as the input:

```
// go get golang.org/x/example/gotypes/hello
```

```
package main
```

```
import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

This is what it prints:

```
$ go build golang.org/x/example/gotypes/defsuses
$ ./defsuses
hello.go:1:9: "main" defines <nil>
hello.go:5:6: "main" defines func hello.main()
hello.go:6:9: "fmt" uses package fmt
hello.go:6:13: "Println" uses func fmt.Println(a ...interface{}) (n int, err
        error)
```

Notice that the `Defs` mapping may contain nil entries in a few cases. The first line of output reports that the package identifier `main` is present in the `Defs` mapping, but has no associated object.

The `Implicits` mapping handles two cases of the syntax in which an `Object` is declared without an `ast.Ident`, namely type switches and import declarations. In the type switch below, which declares a local variable `y`, the type of `y` is different in each case of the switch:

```
switch y := x.(type) {
case int:
    fmt.Printf("%d", y)
case string:
    fmt.Printf("%q", y)
default:
    fmt.Print(y)
}
```

To represent this, for each single-type case, the type checker creates a separate `Var` object for `y` with the appropriate type, and `Implicits` maps each `ast.CaseClause` to the `Var` for that case. The `default` case, the `nil` case, and cases with more than one type all use the regular `Var` object that is associated with the identifier `y`, which is found in the `Defs` mapping.

The import declaration below defines the name `json` without an `ast.Ident`:

```
import "encoding/json"
```

`Implicits` maps this `ast.ImportSpec` to the `PkgName` object named `json` that it implicitly declares.

The `Selections` mapping, of type `map[*ast.SelectorExpr]*Selection`, records the meaning of each expression of the form *`expr`*`.f`, where *`expr`* is an expression or

type and `f` is the name of a field or method. These expressions, called *selections*, are represented by `ast.SelectorExpr` nodes in the AST. We'll talk more about the `Selection` type in Selections.

Not all `ast.SelectorExpr` nodes represent selections. Expressions like `fmt.Println`, in which a package name precedes the dot, are *qualified identifiers*. They do not appear in the `Selections` mapping, but their constituent identifiers (such as `fmt` and `Println`) both appear in `Uses`.

Referring identifiers that are not part of an `ast.SelectorExpr` are *lexical references*. That is, they are resolved to an object by searching for the innermost enclosing lexical declaration of that name. We'll see how that search works in the next section.

Scopes

The `Scope` type is a mapping from names to objects.

```
type Scope struct{ ... }

func (s *Scope) Names() []string
func (s *Scope) Lookup(name string) Object
```

`Names` returns the set of names in the mapping, in sorted order. (It is not a simple accessor though, so call it sparingly.) The `Lookup` method returns the object for a given name, so we can print all the entries or *bindings* in a scope like this:

```
for _, name := range scope.Names() {
    fmt.Println(scope.Lookup(name))
}
```

The *scope* of a declaration of a name is the region of program source in which a reference to the name resolves to that declaration. That is, scope is a property of a declaration. However, in the `go/types` API, the `Scope` type represents a *lexical block*, which is one component of the lexical environment. Consider the *hello, world* program again:

```
package main

import "fmt"

func main() {
    const message = "hello, world"
    fmt.Println(message)
}
```

There are four lexical blocks in this program. The outermost one is the *universe block*, which maps the pre-declared names like `int`, `true`, and `append` to their objects—a `TypeName`, a `Const`, and a `Builtin`, respectively. The universe block is represented by

the global variable `Universe`, of type `*Scope`, although it's logically a constant so you shouldn't modify it.

Next is the *package block*, which maps `"main"` to the `main` function. Following that is the *file block*, which maps `"fmt"` to the `PkgName` object for this import of the `fmt` package. And finally, the innermost block is that of function `main`, a local block, which contains the declaration of `message`, a `Const`. The `main` function is trivial, but many functions contain several blocks since each `if`, `for`, `switch`, `case`, or `select` statement creates at least one additional block. Local blocks nest to arbitrary depths.

The structure of the lexical environment thus forms a tree, with the universe block at the root, the package blocks beneath it, the file blocks beneath them, and then any number of local blocks beneath the files. We can access and navigate this tree structure with the following methods of `Scope`:

```
func (s *Scope) Parent() *Scope
func (s *Scope) NumChildren() int
func (s *Scope) Child(i int) *Scope
```

`Parent` lets us walk up the tree, and `Child` lets us walk down it. Note that although the `Parent` of every package `Scope` is `Universe`, `Universe` has no children. This asymmetry is a consequence of using a global variable to hold `Universe`.

To obtain the universe block, we use the `Universe` global variable. To obtain the lexical block of a `Package`, we call its `Scope` method. To obtain the scope of a file (`*ast.File`), or any smaller piece of syntax such as an `*ast.IfStmt`, we consult the `Scopes` mapping in the `Info` struct, which maps each block-creating syntax node to its block. The lexical block of a named function or method can also be obtained by calling its `(*Func).Scope` method.

To look up a name in the lexical environment, we must search the tree of lexical blocks, starting at a particular `Scope` and walking up to the root until a declaration of the name is found. For convenience, the `LookupParent` method does this, returning not just the object, if found, but also the `Scope` in which it was declared, which may be an ancestor of the initial one:

```
func (s *Scope) LookupParent(name string, pos token.Pos) (*Scope, Object)
```

The `pos` parameter determines the position in the source code at which the name should be resolved. The effective lexical environment is different at each point in the block because it depends on which local declarations appear before or after that point. (We'll see an illustration in a moment.)

`Scope` has several other methods relating to source positions:

```
func (s *Scope) Pos() token.Pos
func (s *Scope) End() token.Pos
func (s *Scope) Contains(pos token.Pos) bool
func (s *Scope) Innermost(pos token.Pos) *Scope
```

`Pos` and `End` report the `Scope`'s start and end position which, for explicit blocks, coincide with its curly braces. `Contains` is a convenience method that reports whether a position lies in this interval. `Innermost` returns the innermost scope containing the specified position, which may be a child or other descendent of the initial scope.

These features are useful for tools that wish to resolve names or evaluate constant expressions as if they had appeared at a particular point within the program. The next example program finds all the comments in the input, treating the contents of each one as a name. It looks up each name in the environment at the position of the comment, and prints what it finds. Observe that the `ParseComments` flag directs the parser to preserve comments in the input.

```
// go get golang.org/x/example/gotypes/lookup
```

```
func main() {
    fset := token.NewFileSet()
    f, err := parser.ParseFile(fset, "hello.go", hello, parser.ParseComments)
    if err != nil {
        log.Fatal(err) // parse error
    }

    conf := types.Config{Importer: importer.Default()}
    pkg, err := conf.Check("cmd/hello", fset, []*ast.File{f}, nil)
    if err != nil {
        log.Fatal(err) // type error
    }

    // Each comment contains a name.
    // Look up that name in the innermost scope enclosing the comment.
    for _, comment := range f.Comments {
        pos := comment.Pos()
        name := strings.TrimSpace(comment.Text())
        fmt.Printf("At %s,\t%q = ", fset.Position(pos), name)
        inner := pkg.Scope().Innermost(pos)
        if _, obj := inner.LookupParent(name, pos); obj != nil {
            fmt.Println(obj)
        } else {
            fmt.Println("not found")
        }
    }
}
```

The expression `pkg.Scope().Innermost(pos)` finds the innermost `Scope` that encloses the comment, and `LookupParent(name, pos)` does a name lookup at a specific position in that lexical block.

A typical input is shown below. The first comment causes a lookup of `"append"` in the file block. The second comment looks up `"fmt"` in the `main` function's block, and so on.

```
const hello = `
package main

import "fmt"

// append
func main() {
        // fmt
        fmt.Println("Hello, world")
        // main
        main, x := 1, 2
        // main
        print(main, x)
        // x
}
// x
`
```

Here's the output:

```
$ go build golang.org/x/example/gotypes/lookup
$ ./lookup
At hello.go:6:1,        "append" = builtin append
At hello.go:8:9,        "fmt" = package fmt
At hello.go:10:9,       "main" = func cmd/hello.main()
At hello.go:12:9,       "main" = var main int
At hello.go:14:9,       "x" = var x int
At hello.go:16:1,       "x" = not found
```

Notice how the two lookups of `main` return different results, even though they occur in the same block, because one precedes the declaration of the local variable named `main` and the other follows it. Also notice that there are two lookups of the name `x` but only the first one, in the function block, succeeds.

Download the program and modify both the input program and the set of comments to get a better feel for how name resolution works.

The table below summarizes which kinds of objects may be declared at each level of the tree of lexical blocks.

```
            Universe File Package Local
Builtin        ✔
Nil            ✔
Const          ✔            ✔      ✔
TypeName       ✔            ✔      ✔
Func                        ✔
Var                         ✔      ✔
PkgName              ✔
Label                              ✔
```

Initialization Order

In the course of identifier resolution, the type checker constructs a graph of references among declarations of package-level variables and functions. The type checker reports an error if the initializer expression for a variable refers to that variable, whether directly or indirectly.

The reference graph determines the initialization order of the package-level variables, as required by the Go spec, using a breadth-first algorithm. First, variables in the graph with no successors are removed, sorted into the order in which they appear in the source code, then added to a list. This creates more variables that have no successors. The process repeats until they have all been removed.

The result is available in the `InitOrder` field of the `Info` struct, whose type is `[]Initializer`.

```
type Info struct {
    ...
    InitOrder []Initializer
    ...
}

type Initializer struct {
    Lhs []*Var // var Lhs = Rhs
    Rhs ast.Expr
}
```

Each element of the list represents a single initializer expression that must be executed, and the variables to which it is assigned. The variables may number zero, one, or more, as in these examples:

```
var _ io.Writer = new(bytes.Buffer)
var rx = regexp.MustCompile("^b(an)*a$")
var cwd, cwdErr = os.Getwd()
```

This process governs the initialization order of variables within a package. Across packages, dependencies must be initialized first, although the order among them is not specified. That is, any topological order of the import graph will do. The `(*Package).Imports` method returns the set of direct dependencies of a package.

Types

The main job of the type checker is, of course, to deduce the type of each expression and to report type errors. Like `Object`, `Type` is an interface type used as a discriminated union of several concrete types but, unlike `Object`, `Type` has very few methods because types have little in common with each other. Here is the interface:

```
type Type interface {
    Underlying() Type
}
```

And here are the 14 concrete types that satisfy it:

```
Type = *Basic
     | *Pointer
     | *Array
     | *Slice
     | *Map
     | *Chan
     | *Struct
     | *Tuple
     | *Signature
     | *Alias
     | *Named
     | *Interface
     | *Union
     | *TypeParam
```

With the exception of `Named` types, instances of `Type` are not canonical. (Even for `Named` types, parameterized types complicate matters; see https://github.com/golang/exp/tree/master/typeparams/example.) That is, it is usually a mistake to compare types using `t1==t2` since this equivalence is not the same as the type identity relation defined by the Go spec. Use this function instead:

```
func Identical(t1, t2 Type) bool
```

For the same reason, you should not use a `Type` as a key in a map. The `golang.org/x/tools/go/types/typeutil` package provides a map keyed by types that uses the correct equivalence relation.

The Go spec defines three relations over types. *Assignability* governs which pairs of types may appear on the left- and right-hand side of an assignment, including implicit assignments such as function calls, map and channel operations, and so on. *Comparability* determines which types may appear in a comparison `x==y` or a switch case or may be used as a map key. *Convertibility* governs which pairs of types are allowed in a conversion operation `T(v)`. You can query these relations with the following predicate functions:

```
func AssignableTo(V, T Type) bool
func Comparable(T Type) bool
func ConvertibleTo(V, T Type) bool
```

Let’s take a look at each kind of type.

Basic types

`Basic` represents all types that are not composed from simpler types. This is essentially the set of underlying types that a constant expression is permitted to have–

strings, booleans, and numbers—but it also includes `unsafe.Pointer` and untyped nil.

```
type Basic struct{...}
func (*Basic) Kind() BasicKind
func (*Basic) Name() string
func (*Basic) Info() BasicInfo
```

The `Kind` method returns an “enum” value that indicates which basic type this is. The kinds `Bool`, `String`, `Int16`, and so on, represent the corresponding predeclared boolean, string, or numeric types. There are two aliases: `Byte` is an alias for `Uint8` and `Rune` is an alias for `Int32`. The kind `UnsafePointer` represents `unsafe.Pointer`. The kinds `UntypedBool`, `UntypedInt` and so on represent the six kinds of “untyped” constant types: boolean, integer, rune, float, complex, and string. The kind `UntypedNil` represents the type of the predeclared `nil` value. And the kind `Invalid` indicates the invalid type, which is used for expressions containing errors, or for objects without types, like `Label`, `Builtin`, or `PkgName`.

The `Name` method returns the name of the type, such as `"float64"`, and the `Info` method returns a bitfield that encodes information about the type, such as whether it is signed or unsigned, integer or floating point, or real or complex.

`Typ` is a table of canonical basic types, indexed by kind, so `Typ[String]` returns the `*Basic` that represents `string`, for instance. Like `Universe`, `Typ` is logically a constant, so don’t modify it.

A few minor subtleties:

- According to the Go spec, pre-declared types such as `int` are named types for the purposes of assignability, even though the type checker does not represent them using `Named`.
- `unsafe.Pointer` is a pointer type for the purpose of determining whether the receiver type of a method is legal, even though the type checker does not represent it using `Pointer`.
- The “untyped” types are usually only ascribed to constant expressions, but there is one exception. A comparison `x==y` has type “untyped bool”, so the result of this expression may be assigned to a variable of type `bool` or any other named boolean type.

Simple Composite Types

The types `Pointer`, `Array`, `Slice`, `Map`, and `Chan` are pretty self-explanatory. All have an `Elem` method that returns the element type `T` for a pointer `*T`, an array `[n]T`, a slice `[]T`, a map `map[K]T`, or a channel `chan T`. This should feel familiar if you’ve used the `reflect.Value` API.

In addition, the `*Map`, `*Chan`, and `*Array` types have accessor methods that return their key type, direction, and length, respectively:

```
func (*Map) Key() Type
func (*Chan) Dir() ChanDir      // = Send | Recv | SendRecv
func (*Array) Len() int64
```

Struct Types

A struct type has an ordered list of fields and a corresponding ordered list of field tags.

```
type Struct struct{ ... }
func (*Struct) NumFields() int
func (*Struct) Field(i int) *Var
func (*Struct) Tag(i int) string
```

Each field is a `Var` object whose `IsField` method returns true. Field objects have no `Parent` scope, because they are resolved through selections, not through the lexical environment.

Thanks to embedding, the expression `new(S).f` may be a shorthand for a longer expression such as `new(S).d.e.f`, but in the representation of `Struct` types, these field selection operations are explicit. That is, the set of fields of struct type `S` does not include `f`. An anonymous field is represented like a regular field, but its `Anonymous` method returns true.

One subtlety is relevant to tools that generate documentation. When analyzing a declaration such as this,

```
type T struct{x int}
```

it may be tempting to consider the `Var` object for field `x` as if it had the name `"T.x"`, but beware: field objects do not have canonical names and there is no way to obtain the name `"T"` from the `Var` for `x`. That's because several types may have the same underlying struct type, as in this code:

```
type T struct{x int}
type U T
```

Here, the `Var` for field `x` belongs equally to `T` and to `U`, and short of inspecting source positions or walking the AST—neither of which is possible for objects loaded from compiler export data—it is not possible to ascertain that `x` was declared as part of `T`. The type checker builds the exact same data structures given this input:

```
type T U
type U struct{x int}
```

A similar issue applies to the methods of named interface types.

Tuple Types

Like a struct, a tuple type has an ordered list of fields, and fields may be named.

```
type Tuple struct{ ... }
func (*Tuple) Len() int
func (*Tuple) At(i int) *Var
```

Although tuples are not the type of any variable in Go, they are the type of some expressions, such as the right-hand sides of these assignments:

```
v, ok = m[key]
v, ok = <-ch
v, ok = x.(T)
f, err = os.Open(filename)
```

Tuples also represent the types of the parameter list and the result list of a function, as we will see.

Since empty tuples are common, the nil `*Tuple` pointer is a valid empty tuple.

Function and Method Types

The types of functions and methods are represented by a `Signature`, which has a tuple of parameter types and a tuple of result types.

```
type Signature struct{ ... }
func (*Signature) Recv() *Var
func (*Signature) Params() *Tuple
func (*Signature) Results() *Tuple
func (*Signature) Variadic() bool
```

Variadic functions such as `fmt.Println` have the `Variadic` flag set. The final parameter of such functions is always a slice, or in the special case of certain calls to `append`, a string.

A `Signature` for a method, whether concrete or abstract, has a non-nil receiver parameter, `Recv`. The type of the receiver is usually a named type or a pointer to a named type, but it may be an unnamed struct or interface type in some cases. Method types are rather second-class: they are only used for the `Func` objects created by method declarations, and no Go expression has a method type. When printing a method type, the receiver does not appear, and the `Identical` predicate ignores the receiver.

The types of `Builtin` objects like `append` cannot be expressed as a `Signature` since those types require parametric polymorphism. `Builtin` objects are thus ascribed the `Invalid` basic type. However, the type of each *call* to a built-in function has a specific and expressible Go type. These types are recorded during type checking for later use

(TypeAndValue).

Alias Types

Type declarations come in two forms, aliases and defined types.

Aliases, though introduced only in Go 1.9 and not very common, are simplest, so we'll present them first and explain defined types in the next section ("Named Types").

An alias type declaration declares an alternative name for an existing type. For example, this declaration lets you use the type `Dictionary` as a synonym for `map[string]string`:

```
type Dictionary = map[string]string
```

The declaration creates a `TypeName` object for `Dictionary`. The object's `IsAlias` method returns true, and its `Type` method returns an `Alias`:

```
    type Alias struct{ ... }
func (a *Alias) Obj() *TypeName
func (a *Alias) Origin() *Alias
func (a *Alias) Rhs() Type
func (a *Alias) SetTypeParams(tparams []*TypeParam)
func (a *Alias) TypeArgs() *TypeList
func (a *Alias) TypeParams() *TypeParamList
```

The type on the right-hand side of an alias declaration, such as `map[string]string` in the example above, can be accessed using the `Rhs()` method. The `types.Unalias(t)` helper function recursively applies `Rhs`, removing all `Alias` types from the operand t and returning the outermost non-alias type.

The `Obj` method returns the declaring `TypeName` object, such as `Dictionary`; it provides the name, position, and other properties of the declaration. Conversely, the `TypeName` object's `Type` method returns the `Alias` type.

Starting with Go 1.24, alias types may have type parameters. For example, this declaration creates an Alias type with a type parameter:

```
type Set[T comparable] = map[T]bool
```

Each instantiation such as `Set[string]` is identical to the corresponding instantiation of the alias' right-hand side type, such as `map[string]bool`.

The remaining methods–Origin, SetTypeParams, TypeArgs, TypeParams–are all concerned with type parameters. For now, see https://github.com/golang/exp/tree/master/typeparams/example.

Prior to Go 1.22, aliases were not materialized as `Alias` types: each reference to an alias type such as `Dictionary` would be immediately replaced by its right-hand side

type, leaving no indication in the output of the type checker that an alias was present. By materializing alias types, optionally in Go 1.22 and by default starting in Go 1.23, we can more faithfully record the structure of the program, which improves the quality of diagnostic messages and enables certain analyses and code transformations. And, crucially, it enabled the addition of parameterized aliases in Go 1.24.)

Named Types

The second form of type declaration, and the only kind prior to Go 1.9, does not use an equals sign:

```
type Celsius float64
```

This declaration does more than just give a name to a type. It first defines a new `Named` type whose underlying type is `float64`; this `Named` type is different from any other type, including `float64`. The declaration binds the `TypeName` object to the `Named` type.

Since Go 1.9, the Go language specification has used the term *defined types* instead of named types; the essential property of a defined type is not that it has a name (aliases and type parameters also have names) but that it is a distinct type with its own method set. However, the type checker API predates that change and instead calls defined types `Named` types.

```
type Named struct{ ... }
func (*Named) NumMethods() int
func (*Named) Method(i int) *Func
func (*Named) Obj() *TypeName
func (*Named) Underlying() Type
```

The `Named` type's `Obj` method returns the `TypeName` object, which provides the name, position, and other properties of the declaration. Conversely, the `TypeName` object's `Type` method returns the `Named` type.

A `Named` type may appear as the receiver type in a method declaration. Methods are associated with the `Named` type, not the name (the `TypeName` object); it's possible—though cryptic—to declare a method on a `Named` type using one of its aliases. The `NumMethods` and `Method` methods enumerate the declared methods associated with this `Named` type (or a pointer to it), in the order they were declared. However, due to the subtleties of anonymous fields and the difference between value and pointer receivers, a named type may have more or fewer methods than this list. We'll return to this in Method Sets.

Every `Type` has an `Underlying` method, but for all of them except `*Named` and `*Alias`, it is simply the identity function. For a named or alias type, `Underlying` returns its underlying type, which is always an unnamed type. Thus `Underlying` returns `int` for both `T` and `U` below.

```
type T int
type U T
```

Clients of the type checker often use type assertions or type switches with a `Type` operand. When doing so, it is often necessary to switch on the type that underlies the type of interest, and failure to do so may be a bug.

This is a common pattern:

```
// handle types of composite literal
switch u := t.Underlying().(type) {  // remove any *Named and *Alias types
case *Struct:        // ...
case *Map:           // ...
case *Array, *Slice: // ...
default:
    panic("impossible")
}
```

Interface Types

Interface types are represented by `Interface`.

```
type Interface struct{ ... }
func (*Interface) Empty() bool
func (*Interface) NumMethods() int
func (*Interface) Method(i int) *Func
func (*Interface) NumEmbeddeds() int
func (*Interface) Embedded(i int) *Named
func (*Interface) NumExplicitMethods() int
func (*Interface) ExplicitMethod(i int) *Func
```

Syntactically, an interface type has a list of explicitly declared methods (`ExplicitMethod`), and a list of embedded named interface types (`Embedded`), but many clients care only about the complete set of methods, which can be enumerated via `Method`. All three lists are ordered by name. Since the empty interface is an important special case, the `Empty` predicate provides a shorthand for `NumMethods() == 0`.

As with the fields of structs (see above), the methods of interfaces may belong equally to more than one interface type. The `Func` object for method `f` in the code below is shared by `I` and `J`:

```
type I interface { f() }
type J I
```

Because the difference between interface (abstract) and non-interface (concrete) types is so important in Go, the `IsInterface` predicate is provided for convenience.

```
func IsInterface(Type) bool
```

The type checker provides three utility methods relating to interface satisfaction:

```
func Implements(V Type, T *Interface) bool
func AssertableTo(V *Interface, T Type) bool
func MissingMethod(V Type, T *Interface, static bool) (method *Func, wrongType boo
```

The `Implements` predicate reports whether a type satisfies an interface type. `MissingMethod` is like `Implements`, but instead of returning false, it explains why a type does not satisfy the interface, for use in diagnostics.

`AssertableTo` reports whether a type assertion `v.(T)` is legal. If `T` is a concrete type that doesn't have all the methods of interface `v`, then the type assertion is not legal, as in this example:

```
// error: io.Writer is not assertible to int
func f(w io.Writer) int { return w.(int) }
```

TypeParam types

A `TypeParam` is the type of a type parameter. For example, the type of the variable `x` in the `identity` function below is a `TypeParam`:

```
func identity[T any](x T) T { return x }
```

As with `Alias` and `Named` types, each `TypeParam` has an associated `TypeName` object that provides its name, position, and other properties of the declaration.

See https://github.com/golang/exp/tree/master/typeparams/example for a more thorough treatment of parameterized types.

Union types

A `Union` is the type of type-parameter constraint of the form `func f[T int | string]`.

See https://github.com/golang/exp/tree/master/typeparams/example for a more thorough treatment of parameterized types.

TypeAndValue

The type checker records the type of each expression in another field of the `Info` struct, namely `Types`:

```
type Info struct {
```

```
    ...
    Types map[ast.Expr]TypeAndValue
}
```

No entries are recorded for identifiers since the `Defs` and `Uses` maps provide more information about them. Also, no entries are recorded for pseudo-expressions like `*ast.KeyValuePair` or `*ast.Ellipsis`.

The value of the `Types` map is a `TypeAndValue`, which (unsurprisingly) holds the type and value of the expression, and in addition, its *mode*. The mode is opaque, but has predicates to answer questions such as: Does this expression denote a value or a type? Does this value have an address? Does this expression appear on the left-hand side of an assignment? Does this expression appear in a context that expects two results? The comments in the code below give examples of expressions that satisfy each predicate.

```
type TypeAndValue struct {
    Type  Type
    Value constant.Value // for constant expressions only
    ...
}

func (TypeAndValue) IsVoid() bool      // e.g. "main()"
func (TypeAndValue) IsType() bool      // e.g. "*os.File"
func (TypeAndValue) IsBuiltin() bool   // e.g. "len(x)"
func (TypeAndValue) IsValue() bool     // e.g. "*os.Stdout"
func (TypeAndValue) IsNil() bool       // e.g. "nil"
func (TypeAndValue) Addressable() bool // e.g. "a[i]" but not "f()", "m[key]"
func (TypeAndValue) Assignable() bool  // e.g. "a[i]", "m[key]"
func (TypeAndValue) HasOk() bool       // e.g. "<-ch", "m[key]"
```

The statement below inspects every expression within the AST of a single type-checked file and prints its type, value, and mode:

```
// go get golang.org/x/example/gotypes/typeandvalue
```

```
// f is a parsed, type-checked *ast.File.
ast.Inspect(f, func(n ast.Node) bool {
    if expr, ok := n.(ast.Expr); ok {
        if tv, ok := info.Types[expr]; ok {
            fmt.Printf("%-24s\tmode:  %s\n", nodeString(expr), mode(tv))
            fmt.Printf("\t\t\t\ttype:  %v\n", tv.Type)
            if tv.Value != nil {
                fmt.Printf("\t\t\t\tvalue: %v\n", tv.Value)
            }
        }
    }
    return true
})
```

It makes use of these two helper functions, which are not shown:

```
// nodeString formats a syntax tree in the style of gofmt.
func nodeString(n ast.Node) string

// mode returns a string describing the mode of an expression.
func mode(tv types.TypeAndValue) string
```

Given this input:

```
const input = `
package main

var m = make(map[string]int)

func main() {
    v, ok := m["hello, " + "world"]
    print(rune(v), ok)
}
`
```

the program prints:

```
$ go build golang.org/x/example/gotypes/typeandvalue
$ ./typeandvalue
make(map[string]int)            mode:  value
                                type:  map[string]int

make                            mode:  builtin
                                type:  func(map[string]int) map[string]int

map[string]int                  mode:  type
                                type:  map[string]int

string                          mode:  type
                                type:  string

int                             mode:  type
                                type:  int

m["hello, "+"world"]            mode:  value,assignable,ok
                                type:  (int, bool)

m                               mode:  value,addressable,assignable
                                type:  map[string]int

"hello, " + "world"             mode:  value
                                type:  string
                                value: "hello, world"

"hello, "                       mode:  value
                                type:  untyped string
                                value: "hello, "

"world"                         mode:  value
                                type:  untyped string
                                value: "world"

print(rune(v), ok)              mode:  void
                                type:  ()

print                           mode:  builtin
                                type:  func(rune, bool)

rune(v)                         mode:  value
                                type:  rune

rune                            mode:  type
                                type:  rune
```

```
...more not shown...
```

Notice that the identifiers for the built-ins `make` and `print` have types that are specific to the particular calls in which they appear. Also notice `m["hello"]` has a 2-tuple type `(int, bool)` and that it is assignable, but unlike the variable `m`, it is not addressable.

Download the example and vary the inputs and see what the program prints.

Here's another example, adapted from the `govet` static checking tool. It checks for accidental uses of a method value `x.f` when a call `x.f()` was intended; comparing a method `x.f` against nil is a common mistake.

```
// go get golang.org/x/example/gotypes/nilfunc
```

```
// CheckNilFuncComparison reports unintended comparisons
// of functions against nil, e.g., "if x.Method == nil {".
func CheckNilFuncComparison(info *types.Info, n ast.Node) {
    e, ok := n.(*ast.BinaryExpr)
    if !ok {
        return // not a binary operation
    }
    if e.Op != token.EQL && e.Op != token.NEQ {
        return // not a comparison
    }

    // If this is a comparison against nil, find the other operand.
    var other ast.Expr
    if info.Types[e.X].IsNil() {
        other = e.Y
    } else if info.Types[e.Y].IsNil() {
        other = e.X
    } else {
        return // not a comparison against nil
    }

    // Find the object.
    var obj types.Object
    switch v := other.(type) {
    case *ast.Ident:
        obj = info.Uses[v]
    case *ast.SelectorExpr:
        obj = info.Uses[v.Sel]
    default:
        return // not an identifier or selection
    }

    if _, ok := obj.(*types.Func); !ok {
        return // not a function or method
    }

    fmt.Printf("%s: comparison of function %v %v nil is always %v\n",
        fset.Position(e.Pos()), obj.Name(), e.Op, e.Op == token.NEQ)
}
```

Given this input,

```
const input = `package main

import "bytes"

func main() {
    var buf bytes.Buffer
    if buf.Bytes == nil && bytes.Repeat != nil && main == nil {
        // ...
    }
}
`
```

the program reports these errors:

```
$ go build golang.org/x/example/gotypes/nilfunc
$ ./nilfunc
input.go:7:5: comparison of function Bytes == nil is always false
input.go:7:25: comparison of function Repeat != nil is always true
input.go:7:48: comparison of function main == nil is always false
```

Selections

A *selection* is an expression `expr.f` in which `f` denotes either a struct field or a method. A selection is resolved not by looking for a name in the lexical environment, but by looking within a *type*. The type checker ascertains the meaning of each selection in the package—a surprisingly tricky business—and records it in the `Selections` mapping of the `Info` struct, whose values are of type `Selection`:

```
type Selection struct{ ... }
func (s *Selection) Kind() SelectionKind // = FieldVal | MethodVal | MethodExpr
func (s *Selection) Recv() Type
func (s *Selection) Obj() Object
func (s *Selection) Type() Type
func (s *Selection) Index() []int
func (s *Selection) Indirect() bool
```

The `Kind` method discriminates between the three (legal) kinds of selections, as indicated by the comments below.

```
type T struct{Field int}
func (T) Method() {}
var v T

                    // Kind            Type
    var _ = v.Field  // FieldVal        int
    var _ = v.Method // MethodVal       func()
    var _ = T.Method // MethodExpr      func(T)
```

Because of embedding, a selection may denote more than one field or method, in which case it is ambiguous, and no `Selection` is recorded for it.

The `Obj` method returns the `Object` for the selected field (`*Var`) or method (`*Func`). Due to embedding, the object may belong to a different type than that of the receiver expression *`expr`*. The `Type` method returns the type of the selection. For a field selection, this is the type of the field, but for method selections, the result is a function type that is not the same as the type of the method. For a `MethodVal`, the receiver parameter is dropped, and for a `MethodExpr`, the receiver parameter becomes a regular parameter, as shown in the example above.

The `Index` and `Indirect` methods report information about implicit operations occurring during the selection that a compiler would need to know about. Because of embedding, a selection *`expr`*`.f` may be shorthand for a sequence containing several implicit field selections, *`expr`*`.d.e.f`, and `Index` reports the complete sequence. And because of automatic pointer dereferencing during struct field accesses and method calls, a selection may imply one or more indirect loads from memory; `Indirect` reports whether this occurs.

Clients of the type checker can call `LookupFieldOrMethod` to look up a name within a type, as if by a selection. This function has an intimidating signature, but conceptually it accepts just a `Type` and a name, and returns a `Selection`:

```
func LookupFieldOrMethod(T Type, addressable bool, pkg *Package, name string) \
    (obj Object, index []int, indirect bool)
```

The result is not actually a `Selection`, but it contains the three main components of one: `Obj`, `Index`, and `Indirect`.

The `addressable` flag should be set if the receiver is a *variable* of type `T`, since in a method selection on a variable, an implicit address-of operation (`&`) may occur. The flag indicates whether the methods of type `*T` should be considered during the lookup. (You may wonder why this parameter is necessary. Couldn't clients instead call `LookupFieldOrMethod` on the pointer type `*T` if the receiver is a `T` variable? The answer is that if `T` is an interface type, the type `*T` has no methods at all.)

The final two parameters of `LookupFieldOrMethod` are `(pkg *Package, name string)`. Together they specify the name of the field or method to look up. This brings us to `Id`s.

Ids

`LookupFieldOrMethod`'s need for a `Package` parameter is a subtle consequence of the *Uniqueness of identifiers* section in the Go spec: "Two identifiers are different if they are spelled differently, or if they appear in different packages and are not exported." In practical terms, this means that a type may have two methods (or two fields, or one of each) both named `f` so long as those methods are defined in different packages, as in this example:

```
package a
type A int
func (A) f()

package b
type B int
func (B) f()

package c
import ( "a"; "b" )
type C struct{a.A; b.B} // C has two methods called f
```

The type `c.C` has two methods named `f`, but there is no ambiguity because the two `f`s are distinct identifiers—think of them as f^a and f^b. For an exported method, this situation *would* be ambiguous because there is no distinction between F^a and F^b; there is only `F`.

Despite having two methods called `f`, neither of them can be called from within package `c` because `c` has no way to identify them. Within `c`, `f` is the identifier f^c, and type `C` has no method of that name. But if we pass an instance of `C` to code in package `a` and call its `f` method via an interface, f^a is called.

The practical consequence for tool builders is that any time you need to look up a field or method by name, or construct a map of fields and/or methods keyed by name, it is not sufficient to use the object's name as a key. Instead, you must call the `Object.Id` method, which returns a string that incorporates the object name, and for unexported objects, the package path too. There is also a standalone function `Id` that combines a name and the package path in the same way:

```
func Id(pkg *Package, name string) string
```

This distinction applies to selections *`expr`*`.f`, but not to lexical references `x` because for unexported identifiers, declarations and references always appear in the same package.

Fun fact: the `reflect.StructField` type records both the `Name` and the `PkgPath` strings for the same reason. The `FieldByName` methods of `reflect.Value` and `reflect.Type` match field names without regard to the package. If there is more than one match, they return an invalid value.

Method Sets

The *method set* of a type is the set of methods that can be called on any value of that type. (A variable of type `T` has access to all the methods of type `*T` as well, due to the implicit address-of operation during method calls, but those extra methods are not part of the method set of `T`.)

Clients can request the method set of a type `T` by calling `NewMethodSet(T)`:

```
type MethodSet struct{ ... }
func NewMethodSet(T Type) *MethodSet
func (s *MethodSet) Len() int
func (s *MethodSet) At(i int) *Selection
func (s *MethodSet) Lookup(pkg *Package, name string) *Selection
```

The `Len` and `At` methods access a list of `Selections`, all of kind `MethodVal`, ordered by `Id`. The `Lookup` function allows lookup of a single method by name (and package path, as explained in the previous section).

`NewMethodSet` can be expensive, so for applications that compute method sets repeatedly, `golang.org/x/tools/go/types/typeutil` provides a `MethodSetCache` type that records previous results. If you only need a single method, don't construct the `MethodSet` at all; it's cheaper to use `LookupFieldOrMethod`.

The next program generates a boilerplate declaration of a new concrete type that satisfies an existing interface. Here's an example:

```
$ ./skeleton io ReadWriteCloser buffer
// *buffer implements io.ReadWriteCloser.
type buffer struct{}
func (b *buffer) Close() error {
    panic("unimplemented")
}
func (b *buffer) Read(p []byte) (n int, err error) {
    panic("unimplemented")
}
func (b *buffer) Write(p []byte) (n int, err error) {
    panic("unimplemented")
}
```

The three arguments are the package and the name of the existing interface type, and the name of the new concrete type. The `main` function (not shown) loads the specified package and calls `PrintSkeleton` with the remaining two arguments:

```
// go get golang.org/x/example/gotypes/skeleton
```

```
func PrintSkeleton(pkg *types.Package, ifacename, concname string) error {
    obj := pkg.Scope().Lookup(ifacename)
    if obj == nil {
        return fmt.Errorf("%s.%s not found", pkg.Path(), ifacename)
    }
    if _, ok := obj.(*types.TypeName); !ok {
        return fmt.Errorf("%v is not a named type", obj)
    }
    iface, ok := obj.Type().Underlying().(*types.Interface)
    if !ok {
        return fmt.Errorf("type %v is a %T, not an interface",
            obj, obj.Type().Underlying())
    }
    // Use first letter of type name as receiver parameter.
```

```
    if !isValidIdentifier(concname) {
        return fmt.Errorf("invalid concrete type name: %q", concname)
    }
    r, _ := utf8.DecodeRuneInString(concname)

    fmt.Printf("// *%s implements %s.%s.\n", concname, pkg.Path(), ifacename)
    fmt.Printf("type %s struct{}\n", concname)
    mset := types.NewMethodSet(iface)
    for i := 0; i < mset.Len(); i++ {
        meth := mset.At(i).Obj()
        sig := types.TypeString(meth.Type(), (*types.Package).Name)
        fmt.Printf("func (%c *%s) %s%s {\n\tpanic(\"unimplemented\")\n}\n",
            r, concname, meth.Name(),
            strings.TrimPrefix(sig, "func"))
    }
    return nil
}
```

First, `PrintSkeleton` locates the package-level named interface type, handling various error cases. Then it chooses the name for the receiver of the new methods: the first letter of the concrete type. Finally, it iterates over the method set of the interface, printing the corresponding concrete method declarations.

There's a subtlety in the declaration of `sig`, which is the string form of the method signature. We could have obtained this string from `meth.Type().String()`, but this would cause any named types within it to be formatted with the complete package path, for instance `net/http.ResponseWriter`, which is informative in diagnostics but not legal Go syntax. The `TypeString` function (explained in Formatting Values) allows the caller to control how packages are printed. Passing `(*types.Package).Name` causes only the package name `http` to be printed, not the complete path. Here's another example that illustrates it:

```
$ ./skeleton net/http Handler myHandler
// *myHandler implements net/http.Handler.
type myHandler struct{}
func (m *myHandler) ServeHTTP(http.ResponseWriter, *http.Request) {
    panic("unimplemented")
}
```

The following program inspects all pairs of package-level named types in `pkg`, and reports the types that satisfy each interface type.

```
// go get golang.org/x/example/gotypes/implements
```

```
// Find all named types at package level.
var allNamed []*types.Named
for _, name := range pkg.Scope().Names() {
    if obj, ok := pkg.Scope().Lookup(name).(*types.TypeName); ok {
        allNamed = append(allNamed, obj.Type().(*types.Named))
    }
}
```

```
// Test assignability of all distinct pairs of
// named types (T, U) where U is an interface.
for _, T := range allNamed {
    for _, U := range allNamed {
        if T == U || !types.IsInterface(U) {
            continue
        }
        if types.AssignableTo(T, U) {
            fmt.Printf("%s satisfies %s\n", T, U)
        } else if !types.IsInterface(T) &&
            types.AssignableTo(types.NewPointer(T), U) {
            fmt.Printf("%s satisfies %s\n", types.NewPointer(T), U)
        }
    }
}
```

Given this input,

```
// go get golang.org/x/example/gotypes/implements
```

```
const input = `package main

type A struct{}
func (*A) f()

type B int
func (B) f()
func (*B) g()

type I interface { f() }
type J interface { g() }
`
```

the program prints:

```
$ go build golang.org/x/example/gotypes/implements
$ ./implements
*hello.A satisfies hello.I
hello.B satisfies hello.I
*hello.B satisfies hello.J
```

Notice that the method set of `B` does not include `g`, but the method set of `*B` does. That's why we needed the second assignability check, using the pointer type `types.NewPointer(T)`.

Constants

A constant expression is one whose value is guaranteed to be computed at compile time. Constant expressions may appear in types, specifically as the length of an array

type such as `[16]byte`, so one of the jobs of the type checker is to compute the value of each constant expression.

As we saw in the `typeandvalue` example, the type checker records the value of each constant expression like `"Hello, " + "world"`, storing it in the `Value` field of the `TypeAndValue` struct. Constants are represented using the `Value` interface from the `go/constant` package.

```
package constant // go/constant

type Value interface {
    Kind() Kind
}

type Kind int // one of Unknown, Bool, String, Int, Float, Complex
```

The interface has only one method, for discriminating the various kinds of constants, but the package provides many functions for inspecting a value of a known kind,

```
// Accessors
func BoolVal(x Value) bool
func Float32Val(x Value) (float32, bool)
func Float64Val(x Value) (float64, bool)
func Int64Val(x Value) (int64, bool)
func StringVal(x Value) string
func Uint64Val(x Value) (uint64, bool)
func Bytes(x Value) []byte
func BitLen(x Value) int
func Sign(x Value) int
```

for performing arithmetic on values,

```
// Operations
func Compare(x Value, op token.Token, y Value) bool
func UnaryOp(op token.Token, y Value, prec uint) Value
func BinaryOp(x Value, op token.Token, y Value) Value
func Shift(x Value, op token.Token, s uint) Value
func Denom(x Value) Value
func Num(x Value) Value
func Real(x Value) Value
func Imag(x Value) Value
```

and for constructing new values:

```
// Constructors
func MakeBool(b bool) Value
func MakeFloat64(x float64) Value
func MakeFromBytes(bytes []byte) Value
func MakeFromLiteral(lit string, tok token.Token, prec uint) Value
func MakeImag(x Value) Value
func MakeInt64(x int64) Value
```

```
func MakeString(s string) Value
func MakeUint64(x uint64) Value
func MakeUnknown() Value
```

All numeric `Value`s, whether integer or floating-point, signed or unsigned, or real or complex, are represented more precisely than ordinary Go types like `int64` and `float64`. Internally, the `go/constant` package uses multi-precision data types like `Int`, `Rat`, and `Float` from the `math/big` package so that `Values` and their arithmetic operations are accurate to at least 256 bits, as required by the Go specification.

Size and Alignment

Because the calls `unsafe.Sizeof(v)`, `unsafe.Alignof(v)`, and `unsafe.Offsetof(v.f)` are all constant expressions, the type checker must be able to compute the memory layout of any value `v`.

By default, the type checker uses the same layout algorithm as the Go 1.5 `gc` compiler targeting `amd64`. Clients can configure the type checker to use a different algorithm by providing an instance of the `types.Sizes` interface in the `types.Config` struct:

```
package types

type Sizes interface {
    Alignof(T Type) int64
    Offsetsof(fields []*Var) []int64
    Sizeof(T Type) int64
}
```

For common changes, like reducing the word size to 32 bits, clients can use an instance of `StdSizes`:

```
type StdSizes struct {
    WordSize int64
    MaxAlign int64
}
```

This type has two basic size and alignment parameters from which it derives all the other values using common assumptions. For example, pointers, functions, maps, and channels fit in one word, strings and interfaces require two words, and slices need three. The default behaviour is equivalent to `StdSizes{8, 8}`. For more esoteric layout changes, you'll need to write a new implementation of the `Sizes` interface.

The `hugeparam` program below prints all function parameters and results whose size exceeds a threshold. By default, the threshold is 48 bytes, but you can set it via the `-bytes` command-line flag. Such a tool could help identify inefficient parameter passing in your programs.

```
// go get golang.org/x/example/gotypes/hugeparam
```

```
var bytesFlag = flag.Int("bytes", 48, "maximum parameter size in bytes")

func PrintHugeParams(fset *token.FileSet, info *types.Info, sizes types.Sizes,
        files []*ast.File) {
    checkTuple := func(descr string, tuple *types.Tuple) {
        for i := 0; i < tuple.Len(); i++ {
            v := tuple.At(i)
            if sz := sizes.Sizeof(v.Type()); sz > int64(*bytesFlag) {
                fmt.Printf("%s: %q %s: %s = %d bytes\n",
                    fset.Position(v.Pos()),
                    v.Name(), descr, v.Type(), sz)
            }
        }
    }
    checkSig := func(sig *types.Signature) {
        checkTuple("parameter", sig.Params())
        checkTuple("result", sig.Results())
    }
    for _, file := range files {
        ast.Inspect(file, func(n ast.Node) bool {
            switch n := n.(type) {
            case *ast.FuncDecl:
                checkSig(info.Defs[n.Name].Type().(*types.Signature))
            case *ast.FuncLit:
                checkSig(info.Types[n.Type].Type.(*types.Signature))
            }
            return true
        })
    }
}
```

As before, `Inspect` applies a function to every node in the AST. The function cares about two kinds of nodes: declarations of named functions and methods (`*ast.FuncDecl`) and function literals (`*ast.FuncLit`). Observe the two cases' different logic to obtain the type of each function.

Here's a typical invocation on the standard `encoding/xml` package. It reports a number of places where the 7-word `StartElement` type is copied.

```
% ./hugeparam encoding/xml
/go/src/encoding/xml/marshal.go:167:50: "start" parameter:
        encoding/xml.StartElement = 56 bytes
/go/src/encoding/xml/marshal.go:734:97: "" result: encoding/xml.StartElement =
        56 bytes
/go/src/encoding/xml/marshal.go:761:51: "start" parameter:
        encoding/xml.StartElement = 56 bytes
/go/src/encoding/xml/marshal.go:781:68: "start" parameter:
        encoding/xml.StartElement = 56 bytes
/go/src/encoding/xml/xml.go:72:30: "" result: encoding/xml.StartElement = 56
        bytes
```

Imports

The type checker's `Check` function processes a slice of parsed files (`[]*ast.File`) that make up one package. When the type checker encounters an import declaration, it needs the type information for the objects in the imported package. It gets it by calling the `Import` method of the `Importer` interface shown below, an instance of which must be provided by the `Config` . This separation of concerns relieves the type checker from having to know any of the details of Go workspace organization, `GOPATH` , compiler file formats, and so on.

```
type Importer interface {
    Import(path string) (*Package, error)
}
```

Most of our examples used the trivial `Importer` implementation, `importer.Default()` , provided by the `go/importer` package. This importer looks for `.a` files written by the compiler that was used to build the program. In addition to object code, these files contain *export data*, that is, a description of all the objects declared by the package, and also of any objects from other packages that were referred to indirectly. Because export data includes information about dependencies, the type checker need load at most one file per import, instead of one per transitive dependency.

Compiler export data is compact and efficient to locate, load, and parse, but it has some shortcomings. First, it does not contain complete syntax trees nor semantic information about the bodies of all functions, so it is not suitable for interprocedural analyses. Second, compiler object data may be stale. Nothing detects or ensures that the object files are more recent than the source files from which they were derived. Generally, object data for standard packages is likely to be up-to-date, but for user packages, it depends on how recently the user ran a `go install` or `go build -i` command.

The `golang.org/tools/x/go/packages` package provides a comprehensive means of loading packages from source. It runs `go list` to query the project metadata, performs `cgo` preprocessing if necessary, reads and parses the source files, and optionally type-checks each package. It can load a whole program from source, or load just the initial packages from source and load all their dependencies from export data. It loads independent packages in parallel to hide I/O latency, and detects and reports import cycles. For each package, it provides the `types.Package` containing the package's lexical environment, the list of `ast.File` syntax trees for each file in the package, the `types.Info` containing type information for each syntax node, a list of type errors associated with that package, and other information too. Since some of this information is more costly to compute, the API allows you to select which parts you need, but since this is a tutorial we'll generally request complete information so that it is easier to explore.

The `doc` program below demonstrates a simple use of `go/packages` . It is a rudimentary implementation of `go doc` that prints the type, methods, and documentation of the package-level object specified on the command line. Here's an

example:

```
$ ./doc net/http File
type net/http.File interface{Readdir(count int) ([]os.FileInfo, error);
        Seek(offset int64, whence int) (int64, error); Stat() (os.FileInfo,
        error); io.Closer; io.Reader}
$GOROOT/src/io/io.go:92:2: method (net/http.File) Close() error
$GOROOT/src/io/io.go:71:2: method (net/http.File) Read(p []byte) (n int, err
        error)
/go/src/net/http/fs.go:65:2: method (net/http.File) Readdir(count int)
        ([]os.FileInfo, error)
$GOROOT/src/net/http/fs.go:66:2: method (net/http.File) Seek(offset int64,
        whence int) (int64, error)
/go/src/net/http/fs.go:67:2: method (net/http.File) Stat() (os.FileInfo, error)

 A File is returned by a FileSystem's Open method and can be
served by the FileServer implementation.

The methods should behave the same as those on an *os.File.
```

Observe that it prints the correct location of each method declaration, even though, due to embedding, some of `http.File`'s methods were declared in another package. Here's the first part of the program, showing how to load complete type information including typed syntax, for a single package `pkgpath`, plus exported type information for its dependencies.

```
// go get golang.org/x/example/gotypes/doc
```

```
pkgpath, name := os.Args[1], os.Args[2]

// Load complete type information for the specified packages,
// along with type-annotated syntax.
// Types for dependencies are loaded from export data.
conf := &packages.Config{Mode: packages.LoadSyntax}
pkgs, err := packages.Load(conf, pkgpath)
if err != nil {
    log.Fatal(err) // failed to load anything
}
if packages.PrintErrors(pkgs) > 0 {
    os.Exit(1) // some packages contained errors
}

// Find the package and package-level object.
pkg := pkgs[0]
obj := pkg.Types.Scope().Lookup(name)
if obj == nil {
    log.Fatalf("%s.%s not found", pkg.Types.Path(), name)
}
```

By default, `go/packages`, instructs the parser to retain comments during parsing. The rest of the program prints the output:

```
// go get golang.org/x/example/gotypes/doc
```

```
// Print the object and its methods (incl. location of definition).
fmt.Println(obj)
for _, sel := range typeutil.IntuitiveMethodSet(obj.Type(), nil) {
    fmt.Printf("%s: %s\n", pkg.Fset.Position(sel.Obj().Pos()), sel)
}

// Find the path from the root of the AST to the object's position.
// Walk up to the enclosing ast.Decl for the doc comment.
for _, file := range pkg.Syntax {
    pos := obj.Pos()
    if !(file.FileStart <= pos && pos < file.FileEnd) {
        continue // not in this file
    }
    path, _ := astutil.PathEnclosingInterval(file, pos, pos)
    for _, n := range path {
        switch n := n.(type) {
        case *ast.GenDecl:
            fmt.Println("\n", n.Doc.Text())
            return
        case *ast.FuncDecl:
            fmt.Println("\n", n.Doc.Text())
            return
        }
    }
}
```

We used `IntuitiveMethodSet` to compute the method set, instead of `NewMethodSet`. The result of this convenience function, which is intended for use in user interfaces, includes methods of `*T` as well as those of `T`, since that matches most users' intuition about the method set of a type. (Our example, `http.File`, didn't illustrate the difference, but try running it on a type with both value and pointer methods.)

Also notice `PathEnclosingInterval`, which finds the set of AST nodes that enclose a particular point, in this case, the object's declaring identifier. By walking up with path, we find the enclosing declaration, to which the documentation is attached.

Formatting support

All types that satisfy `Type` or `Object` define a `String` method that formats the type or object in a readable notation. `Selection` also provides a `String` method. All package-level objects within these data structures are printed with the complete package path, as in these examples:

```
[]encoding/json.Marshaler                                    // a *Slice type
encoding/json.Marshal                                        // a *Func object
(*encoding/json.Encoder).Encode                              // a *Func object (m
func (enc *encoding/json.Encoder) Encode(v interface{}) error // a method *Signatu
func NewEncoder(w io.Writer) *encoding/json.Encoder          // a function *Signa
```

This notation is unambiguous, but it is not legal Go syntax. Also, package paths may be long, and the same package path may appear many times in a single string, for instance, when formatting a function of several parameters. Because these strings often form part of a tool's user interface—as with the diagnostic messages of `hugeparam` or the code generated by `skeleton`—many clients want more control over the formatting of package names.

The `go/types` package provides these alternatives to the `String` methods:

```
func ObjectString(obj Object, qf Qualifier) string
func TypeString(typ Type, qf Qualifier) string
func SelectionString(s *Selection, qf Qualifier) string

type Qualifier func(*Package) string
```

The `TypeString`, `ObjectString`, and `SelectionString` functions are like the `String` methods of the respective types, but they accept an additional argument, a `Qualifier`.

A `Qualifier` is a client-provided function that determines how a package name is rendered as a string. If it is nil, the default behavior is to print the package's path, just like the `String` methods do. If a caller passes `(*Package).Name` as the qualifier, that is, a function that accepts a package and returns its `Name`, then objects are qualified only by the package name. The above examples would look like this:

```
[]json.Marshaler
json.Marshal
(*json.Encoder).Encode
func (enc *json.Encoder) Encode(v interface{}) error
func NewEncoder(w io.Writer) *json.Encoder
```

Often when a tool prints some output, it is implicitly in the context of a particular package, perhaps one specified by the command line or HTTP request. In that case, it is more natural to omit the package qualification altogether for objects belonging to that package, but to qualify all other objects by their package's path. That's what the `RelativeTo(pkg)` qualifier does:

```
func RelativeTo(pkg *Package) Qualifier
```

The examples below show how `json.NewEncoder` would be printed using three qualifiers, each relative to a different package:

```
// RelativeTo "encoding/json":
func NewEncoder(w io.Writer) *Encoder

// RelativeTo "io":
func NewEncoder(w Writer) *encoding/json.Encoder
```

```
// RelativeTo any other package:
func NewEncoder(w io.Writer) *encoding/json.Encoder
```

Another qualifier that may be relevant to refactoring tools (but is not currently provided by the type checker) is one that renders each package name using the locally appropriate name within a given source file. Its behavior would depend on the set of import declarations, including renaming imports, within that source file.

Getting from A to B

The type checker and its related packages represent many aspects of a Go program in many different ways, and analysis tools must often map between them. For instance, a named entity may be identified by its `Object`; by its declaring identifier (`ast.Ident`) or by any referring identifier; by its declaring `ast.Node`; by the position (`token.Pos`) of any those nodes; or by the filename and line/column number (or byte offset) of those `token.Pos` values.

In this section, we'll list solutions to a number of common problems of the form "I have an A; I need the corresponding B".

To map **from a `token.Pos` to an `ast.Node`**, call the helper function `astutil.PathEnclosingInterval`. It returns the enclosing `ast.Node`, and all its ancestors up to the root of the file. If you don't know which file `*ast.File` the `token.Pos` belongs to, you can iterate over the parsed files of the package and quickly test whether its position falls within the file's range, from `File.FileStart` to `File.FileEnd`.

To map **from an `Object` to its declaring syntax**, call `Pos` to get its position, then use `PathEnclosingInterval` as before. This approach is suitable for a one-off query. For repeated use, it may be more efficient to visit the syntax tree and construct the mapping between declarations and objects.

To map **from an `ast.Ident` to the `Object`** it refers to (or declares), consult the `Uses` or `Defs` map for the package, as shown in Identifier Resolution.

To map **from an `Object` to its documentation**, find the object's declaration, and look at the attached `Doc` field. You must have set the parser's `ParseComments` flag. See the `doc` example in Imports.