# Navigating unfamiliar code with the Go Guru

**Alan Donovan**
**Go team, Google NYC**

GopherCon 2016
Denver, Colorado
July 11, 2016

# guru

"An editor-integrated tool for code navigation and comprehension"

tool `golang.org/x/tools/cmd/guru`
docs `golang.org/s/using-guru`

(Until recently, known as `oracle`)

*Priestess of Delphi* (1891), John Collier

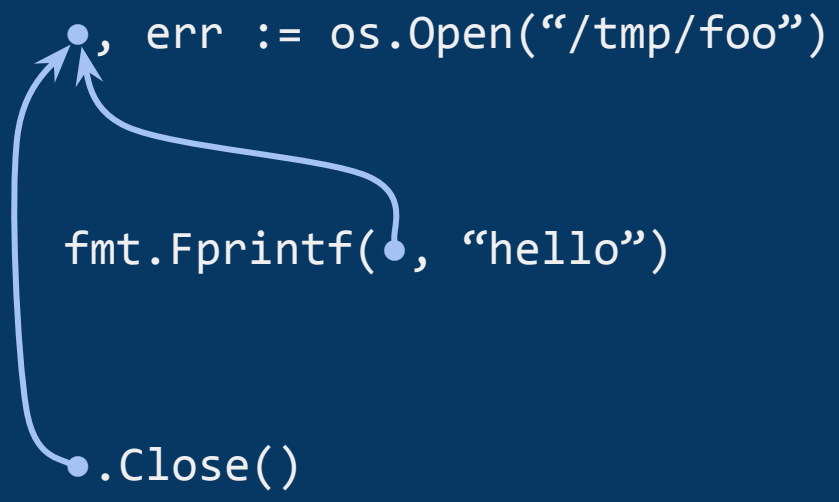
# Outline

1. Names
2. Types
3. Aliases
4. Future work

# Names

```
f, err := os.Open(“/tmp/foo”)

fmt.Fprintf(f, “hello”)

f.Close()
```

# Names



# Questions about names

Where is this name defined?
Who else refers to this name?

# Demo: name queries

1. Identifier highlighting (`what`)
2. Jump to definition (`definition`)
3. Find all references (`referrers`)

# Free names

```
func main() {
    x := 1
    if err := f(x); err != nil {
        log.Fatal(err)
    }
}
```

free names = {x}

```
func f(int) error { … }
```

# Behind the scenes

```
Usage: guru <mode> <position>
```

**mode** determines the query and **position** specifies the code of interest

```
$ guru referrers ~/go/src/fmt/print.go:#3669
~/go/src/fmt/print.go:132:6: references to func newPrinter() *pp
~/go/src/fmt/print.go:180.7-180.16:       p := newPrinter()
~/go/src/fmt/print.go:195.7-195.16:       p := newPrinter()
~/go/src/fmt/print.go:214.7-214.16:       p := newPrinter()
~/go/src/fmt/print.go:231.7-231.16:       p := newPrinter()
~/go/src/fmt/print.go:246.7-246.16:       p := newPrinter()
~/go/src/fmt/print.go:263.7-263.16:       p := newPrinter()
```

# Query modes

| | |
|---|---|
| definition | show declaration of selected identifier |
| referrers | show all refs to entity denoted by selected identifier |
| what | show basic information about the selected syntax node |
| freevars | show free names of selection |
| describe | describe selected syntax: definition, methods, etc |
| implements | show *implements* relation for selected type or method |
| callees | show possible targets of selected function call |
| callers | show possible callers of selected function |
| callstack | show path from callgraph root to selected function |
| peers | show send/receive corresponding to selected channel op |
| pointsto | show variables the selected pointer may point to |
| whicherrs | show possible values of the selected error variable |

Names

Types

Aliases

# Supported editors

| | |
|---|---|
| Acme | `https://github.com/davidrjenni/A` |
| | `https://github.com/mjibson/aw` |
| Atom.io | `https://atom.io/packages/go-oracle` |
| Eclipse | `https://github.com/GoClipse/goclipse` |
| Emacs | `https://github.com/dominikh/go-mode.el` |
| Sublime Text | `https://alvarolm.github.io/GoGuru` |
| Vim | `https://github.com/fatih/vim-go` |
| VSCode | `https://github.com/Microsoft/vscode-go` |

| | |
|---|---|
| Minimal support: | ability to run a compiler and display its diagnostics |
| Fancier: | modified buffer support, highlighting, jump to definition |

# JSON output

```
$ guru -json what ~/go/src/fmt/print.go:#5174
{
        "enclosing": [...],
        "srcdir": "~/go/src",
        "importpath": "fmt",
        "object": "Fprintf",
        "sameids": [
            "~/go/src/fmt/print.go:179:6",
            "~/go/src/fmt/print.go:190:9"
        ]
}
```

source locations
to highlight

```
// Fprintf formats according to a format
// It returns the number of bytes written
func Fprintf(w io.Writer, format string,
        p := newPrinter()
        p.doPrintf(format, a)
        n, err = w.Write(p.buf)
        p.free()
        return
}

// Printf formats according to a format s
// It returns the number of bytes written
func Printf(format string, a ...interface
        return Fprintf(os.Stdout, format,
}
```

# Types

# Questions about types

What is the value of this constant?
What is the type of this expression?
What are its operators, fields, and methods?
How much space does this variable occupy in memory?
Which interfaces does this type satisfy?

# Adaptations to unavailable type information

```
class C {
    char *m_pszfoo;
}
```

“Hungarian” notation

```
Observer observer = new Observer();
```

“stutter”

# Demo: type queries

1. Describe selected syntax (`describe`)
2. Show related concrete/interface types (`implements`)

# Aliases

# What is aliasing?

```
var x int
p := &x
print(x) // “0”
*p = 1
print(x) // “1”
```

An assignment to `*p` changes the value of `x`, and vice versa

`*p` and `x` are *aliases* for the same variable

Pointers are not the only reference type in Go:

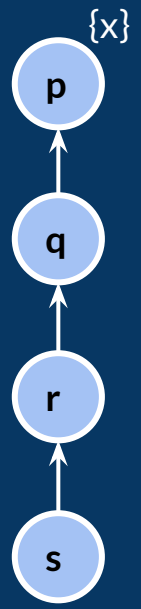
`*T`  `[]T`  `map[K]V`  `chan T`  `func`

# Pointer analysis

golang.org/x/tools/go/pointer

A whole-program static analysis that tells us:
*which variables might this pointer point to?*

```
var x int
p := &x
s := f(p)

func f(q *int) (r *int) { return q }
```



# Questions about aliases

Which variable does this pointer point to?
Which kinds of `error` might this `err` variable contain?

Where might this dynamic call dispatch to?
Where is this function called from?
By what path might this function be called from `main`?

Where is the corresponding receive for this channel send?

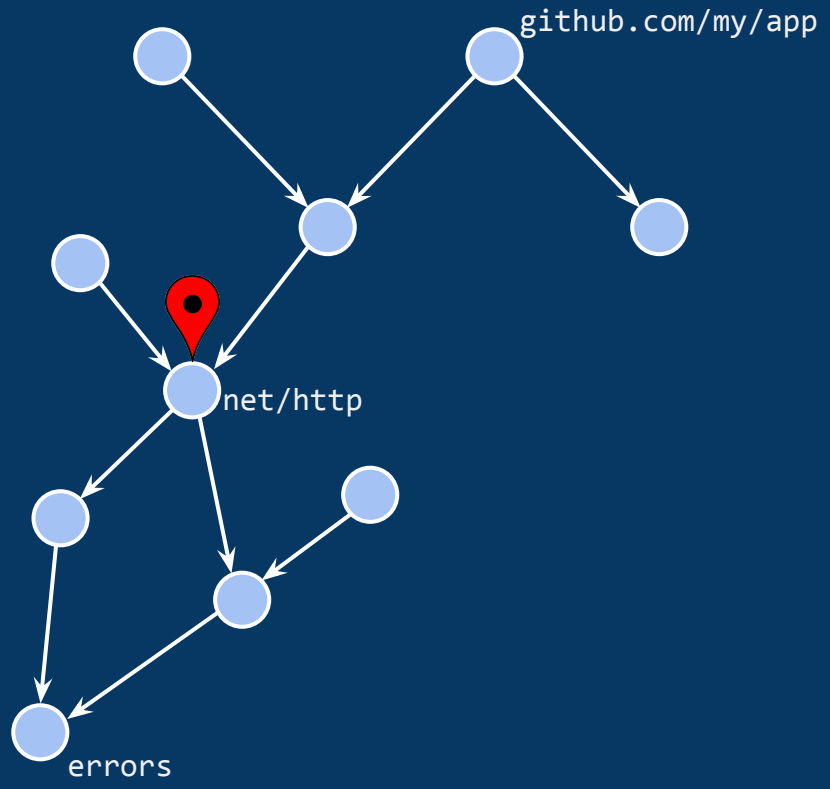
# Pointer analysis tells us channel peers too

```
M:  ch := make(chan int, 1)
    ch1 := ch

S:  ch1 <- 1
    ch2 := ch

R:  print(<-ch2)
```

Because these sets intersect,
that is, ch1 and ch2 may alias,
the operations S and R may communicate

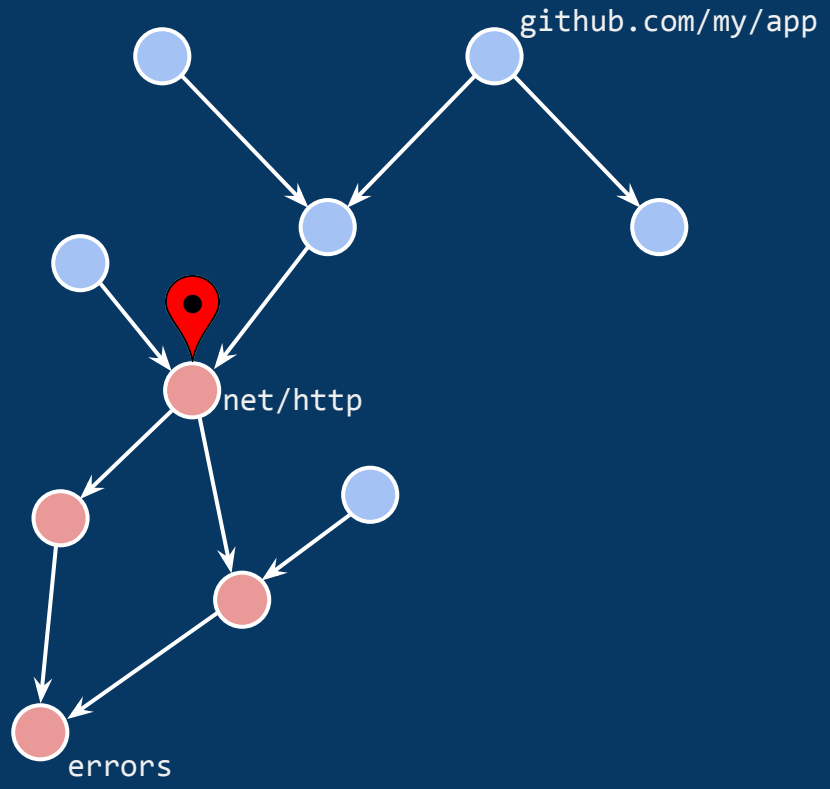
```
pointsto(ch1) = {M}
pointsto(ch2) = {M}
```

# Pointer analysis scope



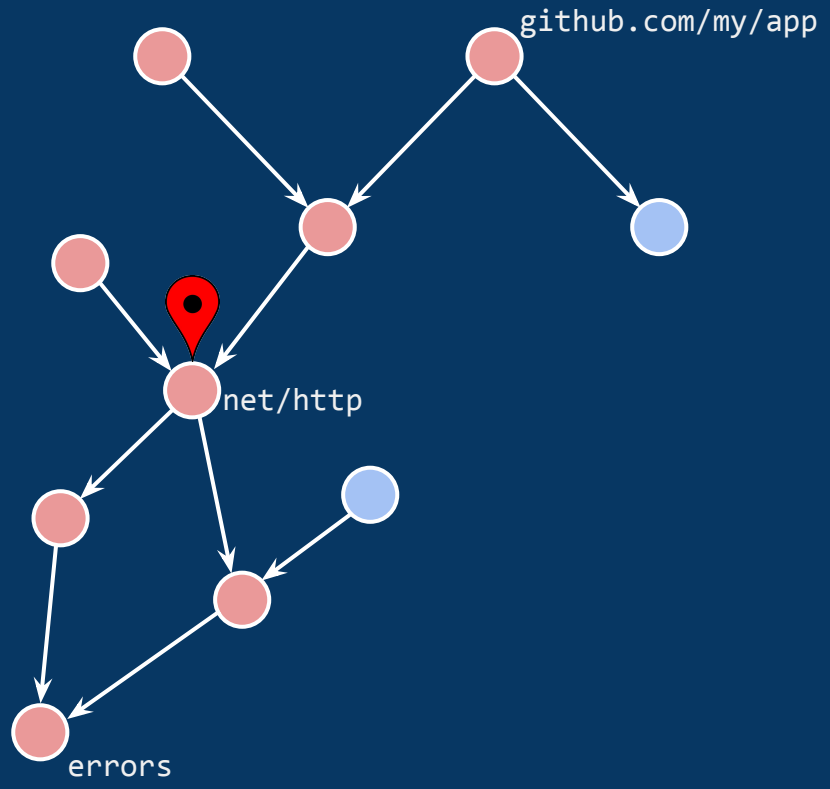

Consider a query about the `net/http` package

# Forward analysis




A forward analysis works
forwards from the query package
Example: `describe`, `definition`

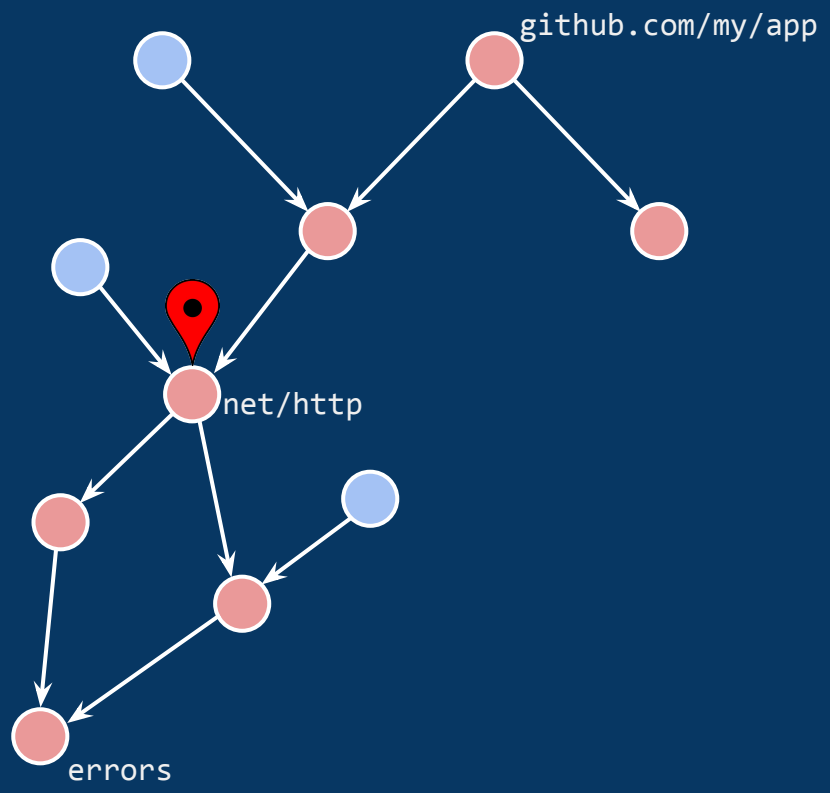
# Backward analysis




A backward analysis also works backwards from the query package
Example: `referrers`, `implements`

# Pointer analysis

with scope `{github.com/my/app}`




Pointer analysis works forwards from the *scope*, a set of main packages specified by the user

# Demo: alias queries

1. Points-to analysis (`pointsto`, `whicherrs`)
2. Channel send/receive peers (`peers`)
3. Call graph queries (`callers`, `callees`, `callstack`)

# Future work

# Challenge: stateless design

| | |
|---|---|
| Pro: | Simple implementation |
| | Results always fresh |
| Con: | Slower for larger code bases |
| Solution: | Disk-based cache of saved type information |
| | (Go 1.7 export data format contains position information) |
| | In-memory cache of workspace import graph |
| | Fsnotify-based daemon responds to file system operations |

Sacrifice some freshness for better performance

Proposed changes will not affect the command-line interface

# Challenge: pointer analysis

Pro: Impressively precise results

Con: Requires extra configuration (“scope”)
Requires input that compiles

Solution: Use simpler analysis if pointer analysis unavailable
(call graph queries only)

Con: Can be slow

Solution: Run pointer analysis asynchronously, with option to refresh
Optimize `set<int>` representation further

# Thanks to:

Guru tool:

Robert Griesemer
David Crawshaw
Michael Matloob

Dominik Honnef
Daniel Morsing

Editor integration:

| | |
|---|---|
| Acme | David R Jenni, Matt Jibson |
| Atom.io | Scott Baron |
| Eclipse | Bruno Medeiros |
| Emacs | Dominik Honnef |
| Vim | Fatih Arslan |
| Sublime | Alvaro M, Hewei Liu |
| | Dan Mace, Jesse Meek |
| VSCode | Luke Hoban |

`golang.org/x/tools/cmd/guru`