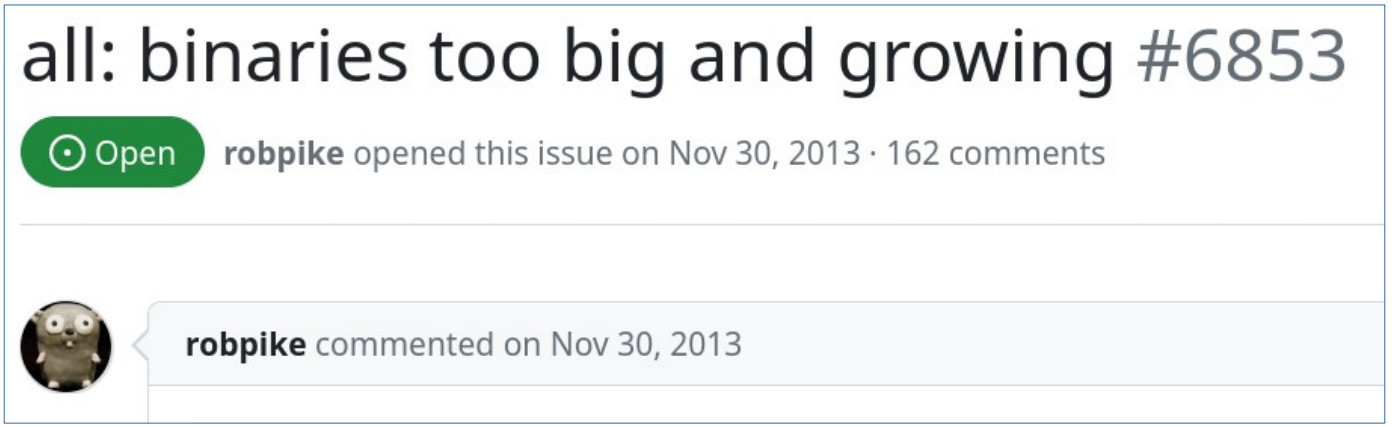
# Getting the most out of Dead Code elimination

# Introduction

- Alessandro Arzilli
  - alessandro.arzilli@gmail.com
  - https://github.com/aarzilli
- Delve contributor since 2015
- Also contributed to Go toolchain

# Short Version

# Motivation




- Dead code elimination helps with this
- However using *some* reflection features will *partially* disable this

# Problematic Reflection

- If your program use one of these
  - `reflect.Value.Method / reflect.Type.Method`
  - `reflect.Value.MethodByName / reflect.Type.MethodByName`
- All public methods of all reachable types will be considered reachable
  - partially disabling deadcode elimination
- Figuring out what makes these reachable is hard
- Use: https://github.com/aarzilli/whydeadcode

# Long Version

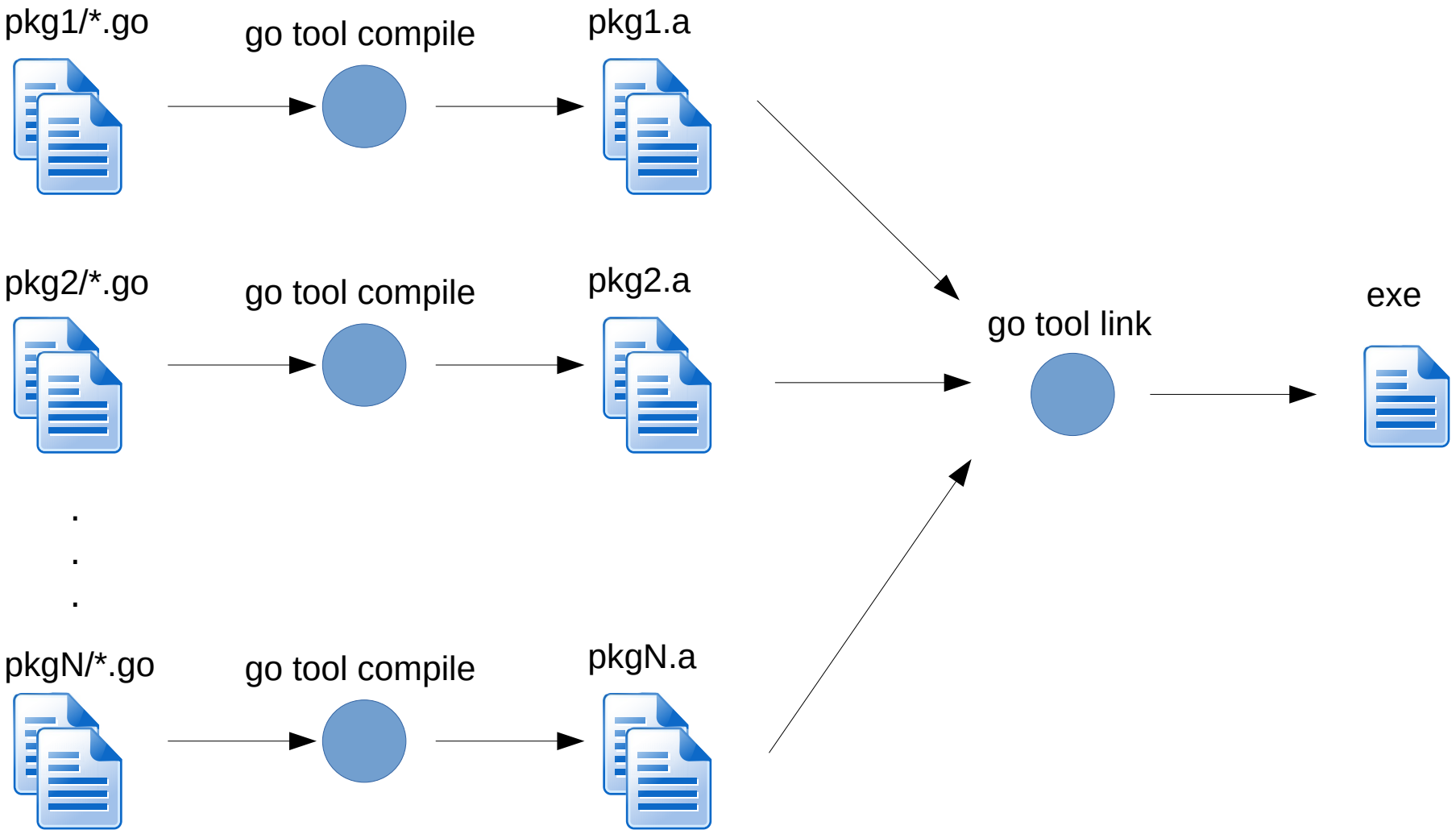
# What is `go build`

- `go build` is not a compiler
- `go build` is to Go as...
  - `make` is to C
  - `cargo` is to Rust
  - `maven` (or ant) is to Java...

# What is `go build` (2)

- Reads go.mod and go.work
- Calculates package dependency graph
- Decides which packages to build, based on build cache
- Calls the compiler (`go tool compile`) on every package
- Calls the linker (`go tool link`) passing all packages to it

# What is go build (3)



# What is `go build` (4)

- Much simplified
- Doesn’t cover cgo
- Compiler and linker are separate binaries
- For more use `go build -x` …

# Object files

- Output of compiler
- Input of linker
- Contains a list of “symbols”

# Symbol

- A symbol:
  - executable code for function/method
  - global variable
  - metadata for reflection
  - GC info
  - Debug info (for functions/methods/types/vars/etc)
- How to see Go symbols:
  - Get name of an object file from `go build -x`
  - Use `go tool nm <object file>`

# Whence deadcode

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Split("hello world", " "))
}
```

- Linker receives object files for fmt, strings, main (and more)
- strings object file contains symbols for every function of strings
  - Clone, Compare, Contains, Count, Cut, ...
- The final executable only needs to contain Split and its dependencies
- Same goes for fmt.Println

# Deadcode detection

- $GOROOT/src/cmd/link/internal/ld/deadcode.go
- Graph visit
  - Starts with the system entry point (eg. `_rt0_amd64_linux`)
  - Marks all symbols reachable using relocations
    - relocation ~ dependency between two symbols
      - in the last example main.main will have relocations for fmt.Println and strings.Split
- This algorithm must make an exception for Method/MethodByName

# Deadcode and reflection

```
package main

import (
    "fmt"
    "reflect"
)

type X struct { }
type Y struct { }

func (*X) One()   { fmt.Println("hello 1") }
func (*X) Two()   { fmt.Println("hello 2") }
func (*X) Three() { fmt.Println("hello 3") }

func (*Y) Four() { fmt.Println(“hello 4”) }
func (*Y) Five()      { fmt.Println(“hello 5”) }

func main() {
    var name string
    fmt.Scanf("%s", &name)
    reflect.ValueOf(&X{}).MethodByName(name).Call(nil)
    var y Y
    y.Five()
}
```

# Deadcode and reflection (2)

- Symbols One, Two and Three are not reachable from anywhere
- But the final executable can still call them depending on user input
- Therefore when the linker determines that MethodByName is reachable it must keep all three of them
- Once they are reachable fmt.Println also becomes reachable

# Detecting problematic reflection

- `grep -Rn Method`?
- Not very good
  - Method is a pretty common identifier (`net/http.Request.Method`)
  - could be from a dependency
  - could be from a standard library package
  - not all calls to Method/MethodByName count
    - they have to be reachable from the entry point

# Detecting problematic reflection (2)

- Use the dumpdep linker flag
  - `go build -ldflags=-dumpdep …`
- Prints the reachability graph as the deadcode algorithm is executed
  - `runtime.throw -> runtime.systemstack`
  - means that `runtime.systemstack` is reachable from `runtime.throw`
- This works but hard to interpret
  - Output for simple program above is >10k lines

# Detecting problematic reflection (3)

- Postprocess -dumpdep output using whydeadcode
  - https://github.com/aarzilli/whydeadcode
- go build -ldflags=-dumpdep … |& whydeadcode
- Prints reachability paths similar to “stacktraces”:

```
reflect.Value.MethodByName reachable from:
        text/template.(*state).evalField
        text/template.(*state).evalFieldChain
        text/template.(*state).evalCommand
        text/template.(*state).evalPipeline
        text/template.(*state).walk
        text/template.(*Template).execute
        github.com/spf13/cobra.tmpl
        github.com/spf13/cobra.(*Command).execute
        github.com/spf13/cobra.(*Command).ExecuteC
        github.com/spf13/cobra.(*Command).Execute
        main.main
        ...
```

# Whydeadcode caveats

- First path it prints is always correct
- Everything after the first one could be a “false positive”
  - After seeing Method/MethodByName the linker switches to marking all public methods as reachable
- There could be ways to reach Method/MethodByName that whydeadcode doesn’t print
  - dumpdep does not print all the paths that lead to a symbol

# Whydeadcode caveats (2)

- Run whydeadcode
- Read the first path it prints, comment out code that leads to it
- Repeat until whydeadcode prints nothing

# Examples

# Delve

- Go Debugger
  - written in Go
  - powers debugging in GoLand, VSCode-go
- Has the deadcode problem described here
  - Somewhere in the code there are reachable calls to MethodByName and Method
  - Can we remove them without breaking backwards compatibility?
  - How much disk space do we save if we do it?

# Starlark

```
reflect.Value.MethodByName reachable from:
        go.starlark.net/starlark.unpackOneArg.func1
        go.starlark.net/starlark.unpackOneArg
        go.starlark.net/starlark.UnpackPositionalArgs
        go.starlark.net/starlark.abs
        go.starlark.net/starlark.abs·f
        go.starlark.net/starlark.init.0
        go.starlark.net/starlark..inittask
        go:main.inittasks
        _
```

- Through a dependency
- Used to provide scripting to the debugger

# Starlark (2)

```
// Report Starlark dynamic type error.
//
// We prefer the Starlark Value.Type name over
// its Go reflect.Type name, but calling the
// Value.Type method on the variable is not safe
// in general. If the variable is an interface,
// the call will fail. Even if the variable has
// a concrete type, it might not be safe to call
// Type() on a zero instance. Thus we must use
// recover.

// Default to Go reflect.Type name
paramType := paramVar.Type().String()

// Attempt to call Value.Type method.
func() {
        defer func() { recover() }()
        paramType = paramVar.MethodByName("Type").Call(nil)[0].String()
}()
```

# Starlark (3)

- `paramVar` is of type reflect.Value
- The type `starlark.Value` that the comment talks about is an interface, like this:

```
type Value struct {
    ...
    Type() string
    ...
}
```

# Starlark (4)

- We can replace the MethodByName call like this:

```
- paramType = paramVar.MethodByName("Type").Call(nil)[0].String()
+ typer, _ := paramVar.Interface().(interface{ Type() string })
+ if typer != nil {
+         paramType = typer.Type()
+ }
```

- Every time you need to call a method with a know signature you can use a type assertion
- Change already submitted to starlark
  - https://github.com/google/starlark-go/pull/444

# JSON-RPC

```
github.com/go-delve/delve/service/rpccommon.suitableMethods reachable from:
        github.com/go-delve/delve/service/rpccommon.(*ServerImpl).Run
        github.com/go-delve/delve/cmd/dlv/cmds.execute
        github.com/go-delve/delve/cmd/dlv/cmds.New.func4
        github.com/go-delve/delve/cmd/dlv/cmds.New.func4·f
        github.com/go-delve/delve/cmd/dlv/cmds.New
        main.main
        runtime.main_main·f
        runtime.main
        runtime.mainPC
        runtime.rt0_go
        _rt0_amd64
        _rt0_amd64_linux
        _
```

# JSON-RPC (2)

- Why is it complaining about rpccommon.suitableMethods?
  - isn’t this about reflect.Value.MethodByName and reflect.Value.Method?
  - reflect.Value.Method was inlined into suitableMethods
  - there is no symbol for reflect.Value.Method but suitableMethods is flagged as being the same thing due to the inlining

# JSON-RPC (3)

- Delve has a JSON-RPC API that clients can use it
  - GoLand starts a headless instance of Delve then uses a TCP/IP connection to debug a program
  - VSCode-go also used to work like this (now uses DAP instead of the JSON-RPC API)

# JSON-RPC (4)

```
var methodMap = map[string]reflect.Value{}
suitableMethods(&rpcServer, methodMap)
...

for {
    header := readRequestHeader()
    method := methodMap[header.ServiceMethod]
    argv := reflect.New(method.Type().In(1).Elem())
    readRequestBody(&argv)
    replyv := reflect.New(method.Type().In(2).Elem())
    errValue := method.Call([]reflect{argv, replyv})
}
```

- Despite using a lot of reflection none of this is a problem
- Except suitable methods

# JSON-RPC (5) - suitableMethods

- Scans its argument looking for methods with this signature:

```
func (s *RPCServer) RPCCallableMethod(input *InputType, output *OutputType) error
```

- All methods like this become API calls
- Pseudocode:

```
func suitableMethods(s *RPCServer, m map[string]reflect.Value) {
    val := reflect.ValueOf(s)
    for i := 0; i < val.NumMethod(); i++ {
        method := val.Method(i)
        if isRPCMethodSignature(method) {
            m[method.Name] = method
        }
    }
}
```

# JSON-RPC (6) - codegen

- Instead of determining the list of methods at runtime do it at compile time
  - `golang.org/x/tools/go/packages`

```
func suitableMethods(s *RPCServer, m map[string]reflect.Value) {
    m["CreateBreakpoint"] = reflect.ValueOf(s.CreateBreakpoint)
    m["AmendBreakpoint"] = reflect.ValueOf(s.AmendBreakpoint)
    m["EvalSymbol"] = reflect.ValueOf(s.EvalSymbol)
    ...
}
```

- Whenever you do codegen add also a test that checks that it is up-to-date

# text/template

```
reflect.Value.MethodByName reachable from:
        text/template.(*state).evalField
        text/template.(*state).evalFieldChain
        text/template.(*state).evalCommand
        text/template.(*state).evalPipeline
        text/template.(*state).walk
        text/template.(*Template).execute
        github.com/go-delve/delve/pkg/version.moduleBuildInfo
        github.com/go-delve/delve/pkg/version.moduleBuildInfo·f
        github.com/go-delve/delve/pkg/version.init.0
        github.com/go-delve/delve/pkg/version..inittask
        go:main.inittasks
        _
```

# text/template

- Used for `dlv version`

```
var buildInfoTmpl = ` mod    {{.Main.Path}} {{.Main.Version}}   {{.Main.Sum}}
{{range .Deps}} dep {{.Path}} {{.Version}}   {{.Sum}}{{if .Replace}}
     => {{.Replace.Path}}    {{.Replace.Version}}    {{.Replace.Sum}}{{end}}
{{end}}
...
buf := new(bytes.Buffer)
err := template.Must(template.New("buildinfo").Parse(buildInfoTmpl)).Execute(buf, info)
if err != nil {
     panic(err)
}
return buf.String()
```

# text/template (2)

- Using `text/template` will always make MethodByName reachable

```
type X struct {}
func (*X) One() string   { return "hello 1" }
func (*X) Two() string   { return "hello 2" }
func (*X) Three() string { return "hello 3" }

func main() {
    var name string
    fmt.Scanf("%s", &name)
    template.Must(template.New("temp").Parse("{{." + name + "}}\n"))
        .Execute(os.Stdout, &X{})
}
```

# text/template (3)

- It’s a small, fixed template
- Just replace it with Go code calling fmt.Fprintf
  - tradeoff between clean code and executable size

```
buf := new(bytes.Buffer)
fmt.Fprintf(buf, " mod\t%s\t%s\t%s\n",
     info.Main.Path, info.Main.Version, info.Main.Sum)
for _, dep := range info.Deps {
     fmt.Fprintf(buf, " dep\t%s\t%s\t%s", dep.Path, dep.Version, dep.Sum)
     if dep.Replace != nil {
          fmt.Fprintf(buf, "\t=> %s\t%s\t%s",
               dep.Replace.Path, dep.Replace.Version, dep.Replace.Sum)
     }
     fmt.Fprintf(buf, "\n")
}
return buf.String()
```

# Cobra

```
reflect.Value.MethodByName reachable from:
        text/template.(*state).evalField
        text/template.(*state).evalFieldChain
        text/template.(*state).evalCommand
        text/template.(*state).evalPipeline
        text/template.(*state).walk
        text/template.(*Template).execute
        github.com/spf13/cobra.tmpl
        github.com/spf13/cobra.(*Command).execute
        github.com/spf13/cobra.(*Command).ExecuteC
        github.com/spf13/cobra.(*Command).Execute
        main.main
        runtime.main_main·f
        runtime.main
        runtime.mainPC
        runtime.rt0_go
        _rt0_amd64
        _rt0_amd64_linux
        _
```

# Cobra (2)

- `text/template` again…
- Cobra is a famous CLI library
- Cobra uses `text/template` to print the command line help (and usage)
- There’s even a method to change it
  - `func (c *Command) SetHelpTemplate(s string)`

# Cobra (3)

- The situation we are in is
  - Cobra uses text/template
    - text/template uses MethodByName
  - can’t replace Cobra in Delve because of backwards compatibility
  - can’t remove text/template in Cobra because it’s part of its public API
- Game over?
  - No. We can still make text/template unreachable

# Cobra (4)

- Introduce the `tmplFunc` type

```
type tmplFunc struct {
    tmpl string
    fn func(io.Writer, interface{}) error
}
```

- Use it to store template strings:

```
type Command struct {
...
-   helpTemplate string
+   helpTemplate *tmplFunc
...
}
```

# Cobra (5)

- Change SetHelpTemplate:

```
func (c *Command) SetHelpTemplate(tmpl string) {
    c.helpTemplate = &tmplFunc{
        tmpl,
        func(w io.Writer, data interface{}) error {
            return template.Must(template.New("top").Parse(tmpl))
                .Execute(w, data)
        }
    }
}
```

- Where Cobra needs to generate the help text:

```
- template.Must(template.New("top").Parse(c.helpTemplate))
-       .Execute(c.OutOrStdout(), c)
+ c.helpTemplate.fn(c.OutOrStdout(), c)
```

# Cobra (6)

- Rewrite the default help to use a function instead of a template:

```
c.helpTemlpate = &tmplFunc{
    tmpl: `template string...`,
    fn: func(out io.Writer, arg interface{}) error {
        fmt.Fprintf(out, "…")
        ...
    }
}
```

- Public API is unchanged
- As long as clients do not use SetHelpTemplate text/template is unreachable

# Cobra (7)

- This is also a tradeoff
  - The default help template is complicated and the replacement Go code more so
  - But Cobra is a popular library and this is making many executables bigger
- Code shown here is a simplified version
- Real version was submitted as PR
  - https://github.com/spf13/cobra/pull/1956
  - No response yet

# Is it worth it?

Before: 17’827’776

After: 15’620’567

- 2MB of deadcode
- 12% of the executable is deadcode
- Delve is downloade 2500 times per day
  - (not counting GoLand, Goproxy and distro installs)
  - 5GB of extra hard drive wear due to deadcode