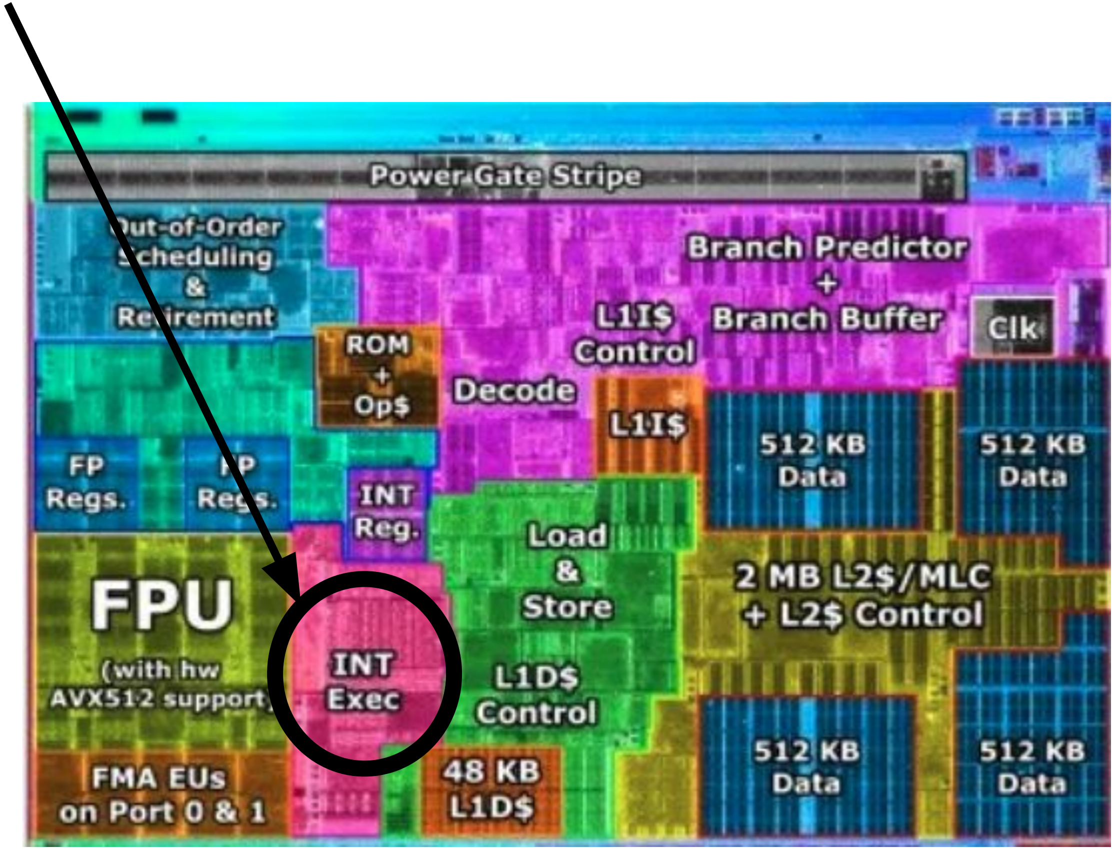
Interface Internals

Keith Randall
@GopherCon, 2024/07/10

```
c := a * b
```

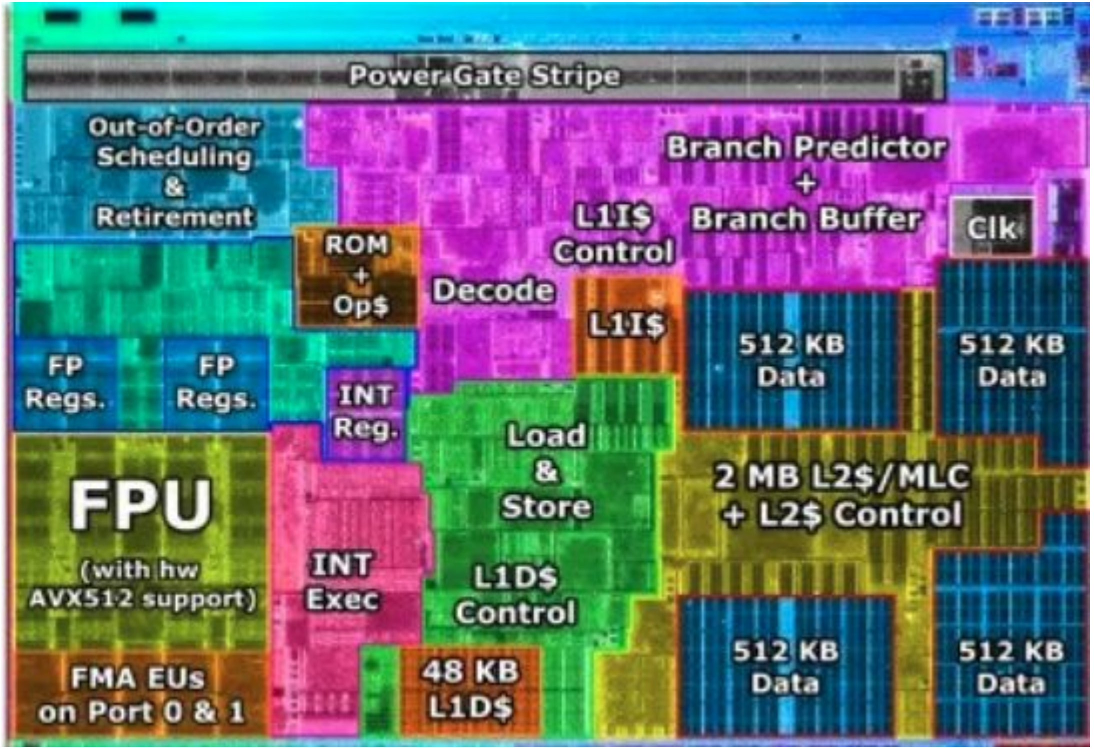
in here somewhere



`c := a * b` → `MUL R1, R0, R0`

```
var c io.Closer = …
err := c.Close()
```

There is no “interface method execution unit” anywhere here.



```
var c io.Closer = …
err := c.Close()
```

```
var c io.Closer = …
err := c.Close()
```

- find the contained type
- get its list of methods
- find the one with the right name
- get its location in the binary
- jump to it

```
var c io.Closer = …
err := c.Close()
```

- find the contained type
- get its list of methods
- find the one with the right name
- get its location in the binary
- jump to it

How many instructions will it take?

```
var c io.Closer = …
err := c.Close()
```

```
MOVD   24(R0), R2
MOVD   R1, R0
CALL   (R2)
```

```
interface {                           io.Reader
    Read(buf []byte) (int, error)
}
interface {                           io.ReadWriter
    Read(buf []byte) (int, error)
    Write(buf []byte) (int, error)
}
interface {                           a.k.a. error
    Error() string
}
interface { }                         a.k.a. any
```

Interface-typed variables can contain any type that has the methods listed in the interface.

```
var r io.Reader

r = &os.File{...}
```

Interface-typed variables can contain any type that has the methods listed in the interface.

```
var r io.Reader

r = &os.File{...}
```

sure!

Interface-typed variables can contain any type that has the methods listed in the interface.

```
var r io.Reader

r = &os.File{...}
```

sure!

```
r = &bytes.Buffer{...}
```

Interface-typed variables can contain any type that has the methods listed in the interface.

```
var r io.Reader

r = &os.File{...}             sure!

r = &bytes.Buffer{...}        yep!
```

Interface-typed variables can contain any type that has the methods listed in the interface.

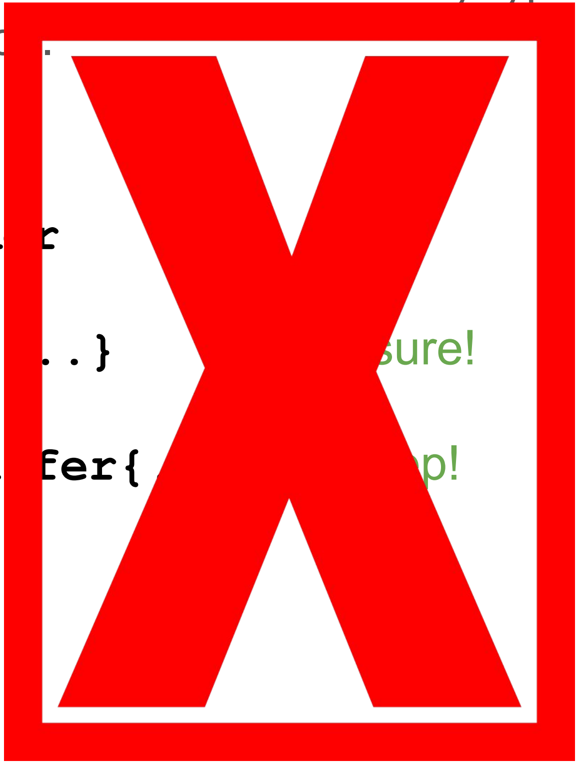
```
var r io.Reader

r = &os.File{...}              sure!

r = &bytes.Buffer{...}         yep!

r = 9
```

Interface-typed variables can contain any type that has the methods listed in the interface.



```
var r io.Reader

r = &os.File{...}                sure!

r = &bytes.Buffer{               p!

r = 9
```

Interface variables bridge the gap between static and dynamic worlds.

Static

- compile time
- fixed type
- interface type

Dynamic

- run time
- type can change
- value can change
- non-interface (“concrete”) type

Interface variables bridge the gap between static and dynamic worlds.

Static

- compile time
- fixed type
- interface type

Dynamic

- run time
- type can change
- value can change
- non-interface (“concrete”) type

This is what distinguishes interfaces from all other types in Go.

```
var r io.Reader
r = &os.File{...}
n, err := r.Read(buf)
r = &bytes.Buffer{...}
n, err = r.Read(buf)
```

calls os.(*File).Read

```
var r io.Reader
r = &os.File{...}
n, err := r.Read(buf)
r = &bytes.Buffer{...}
n, err = r.Read(buf)
```

```
var r io.Reader
r = &os.File{...}
n, err := r.Read(buf)
r = &bytes.Buffer{...}
n, err = r.Read(buf)
```

calls os.(*File).Read

calls bytes.(*Buffer).Read

This all sounds kind of magical!

Let’s investigate how it is done!

Conceptually, interfaces contain a pair of

- a concrete (non-interface) type
- a value of that type

How do we represent that at runtime?

Interfaces are just 2-word structs!

From the runtime:

```
type eface struct {
    _type  *_type
    data   unsafe.Pointer
}
```

Interfaces are just 2-word structs!

From the runtime:

```
type eface struct {
    _type  *_type
    data   unsafe.Pointer
}
```

We’ll tackle the data field first.

```
var i any
i = [3]int{1, 2, 3}
```

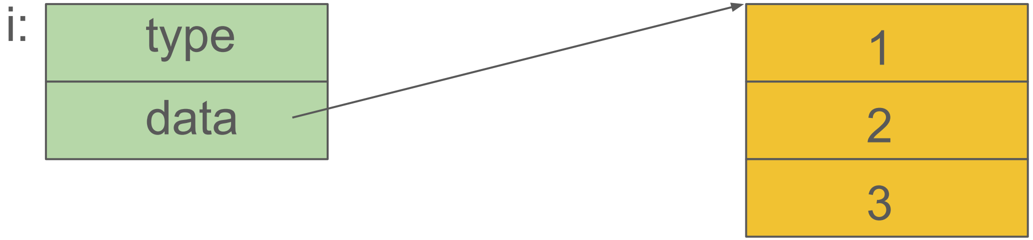
```
var i any
i = [3]int{1, 2, 3}
```

How do we fit a value that big into a 2-word interface?

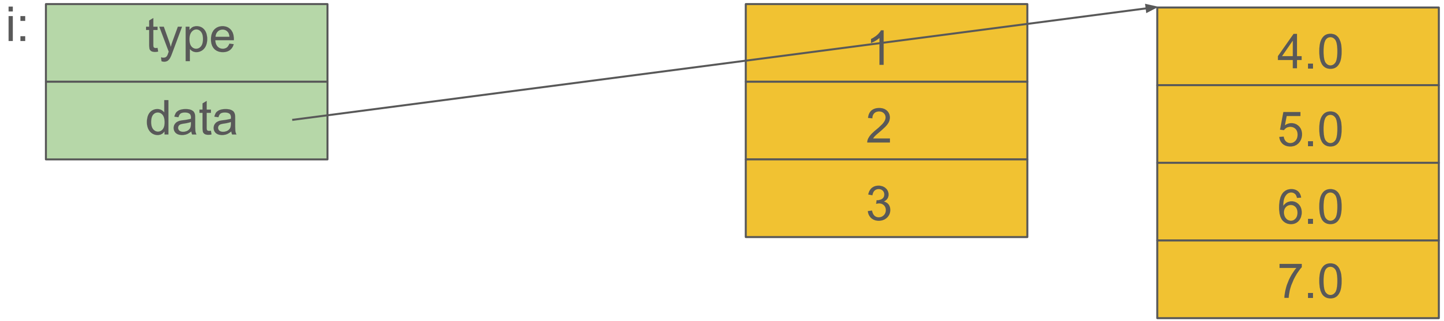
```
var i any
i = [3]int{1, 2, 3}
```

🟩 = registers / stack

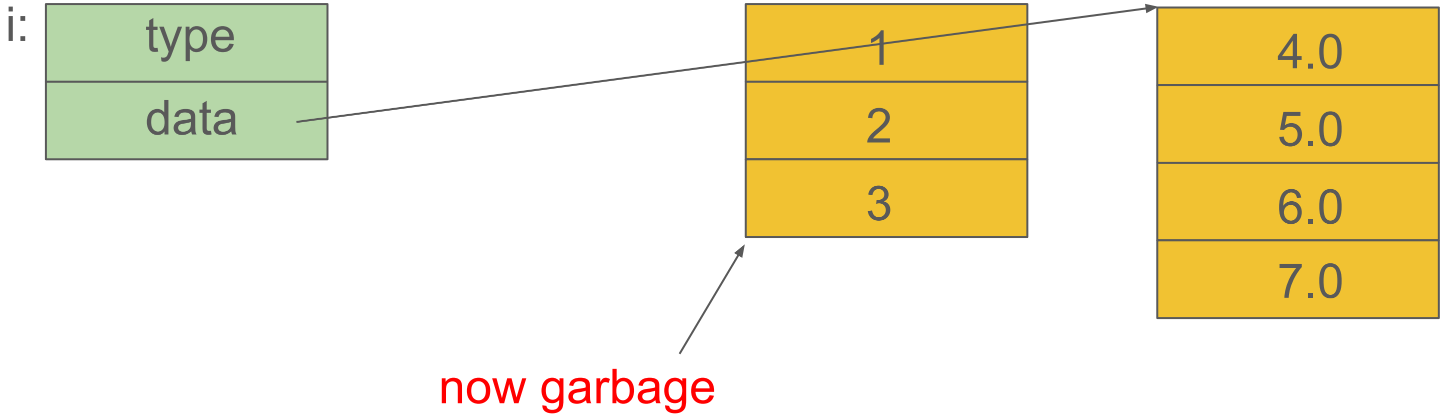
🟨 = heap



```
var i any
i = [3]int{1, 2, 3}
i = [4]float64{4.0, 5.0, 6.0, 7.0}
```



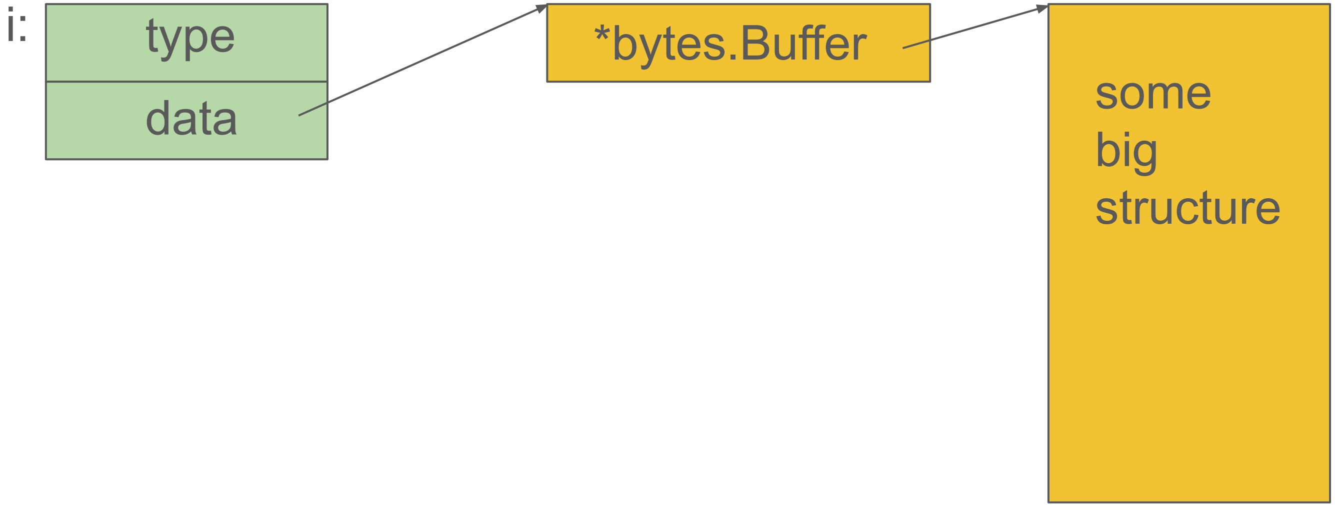
```
var i any
i = [3]int{1, 2, 3}
i = [4]float64{4.0, 5.0, 6.0, 7.0}
```



Optimization: if the data is already a pointer, we can use it directly.

```
var i any
i = &bytes.Buffer{...}
```

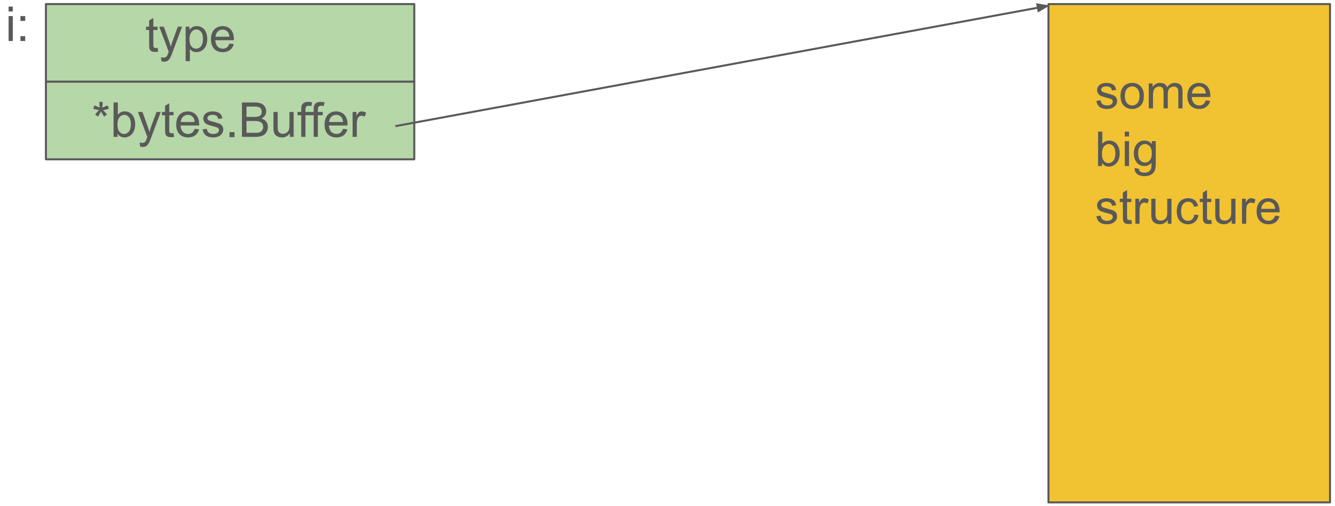
instead of



Optimization: if the data is already a pointer, we can use it directly.

```
var i any
i = &bytes.Buffer{...}
```

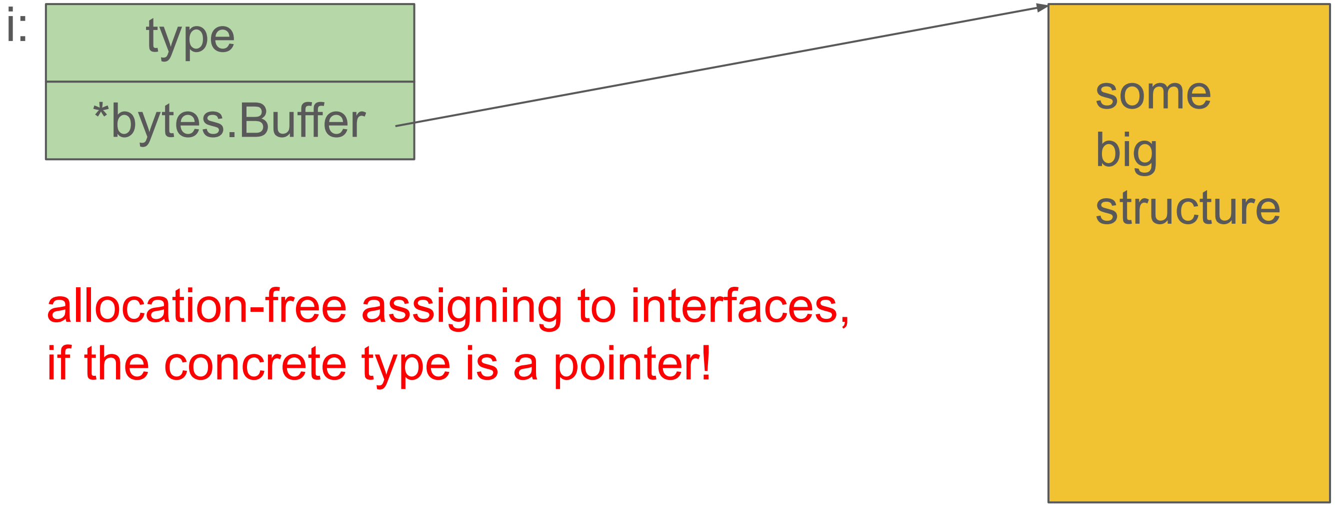
we can do



Optimization: if the data is already a pointer, we can use it directly.

```
var i any
i = &bytes.Buffer{...}
```

we can do



allocation-free assigning to interfaces,
if the concrete type is a pointer!

Interfaces are just 2-word structs!

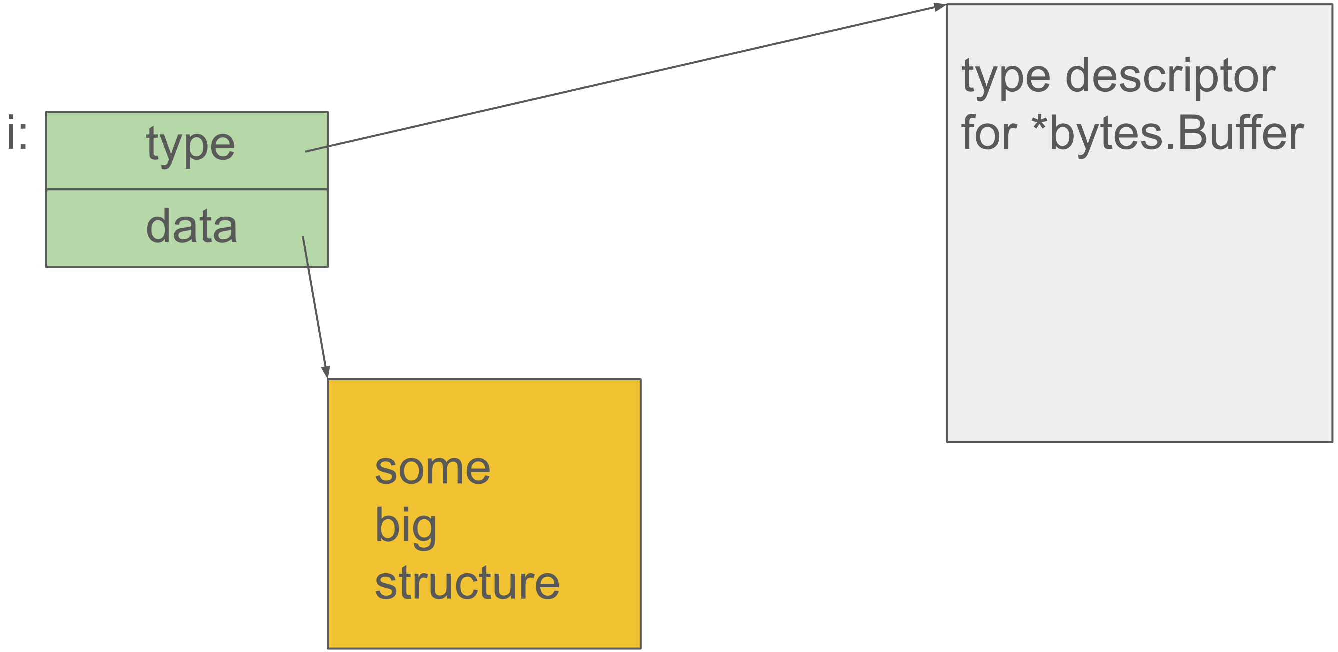
From the runtime:

```
type eface struct {
    _type  *_type
    data   unsafe.Pointer
}
```

Now we’ll look at the type information.

□ = static

```
var i any
i = &bytes.Buffer{...}
```



i:

type

data

type descriptor
for *bytes.Buffer

some
big
structure

Type descriptors

Give the runtime access to information about a type.

- Size
- Pointer fields
- String representation
- How to do == on this type
- List of methods
- Lives in the read-only data section of the binary
  - or maybe allocated by the reflect package

concrete -> interface conversion

```
var b *bytes.Buffer = …
var i any = b
```

… b starts in R0 …

```
MOVD   R0, R1                        // i.data = b
MOVD   $type:*bytes.Buffer(SB), R0   // i._type = constant
```

… i is in R0,R1 …

interface -> concrete conversion

```
var i any = …
if b, ok := i.(*bytes.Buffer); ok { … }
```

… i starts in R0,R1 …

```
MOVD   $type:*bytes.Buffer(SB), R2   // R2 = constant
CMP    R2, R0                        // R2 == i._type?
BEQ    32                            // branch if equal
```

… b is in R1 …

interface -> concrete conversion

- Type descriptors must be unique.
- Deduplicated by the linker.
- The **`reflect`** and **`plugin`** packages have to be careful not to break this property.

Compare against nil

```
err := f()
if err != nil { … }
```

… err is in R0/R1 …

```
CBNZ   R0, 48          // branch if err._type != nil
```

```
var c io.Closer = …
err := c.Close()
```

```
var c io.Closer = …
err := c.Close()
```

- find the contained type
- get its list of methods
- find the one with the right name
- get its location in the binary
- jump to it

```
var c io.Closer = …
err := c.Close()
```

- ~~find the contained type~~ done!
- get its list of methods
- find the one with the right name
- get its location in the binary
- jump to it

```go
var c io.Closer = …
err := c.Close()
```

- ~~find the contained type~~ done!
- get its list of methods
- find the one with the right name
- get its location in the binary
- jump to it

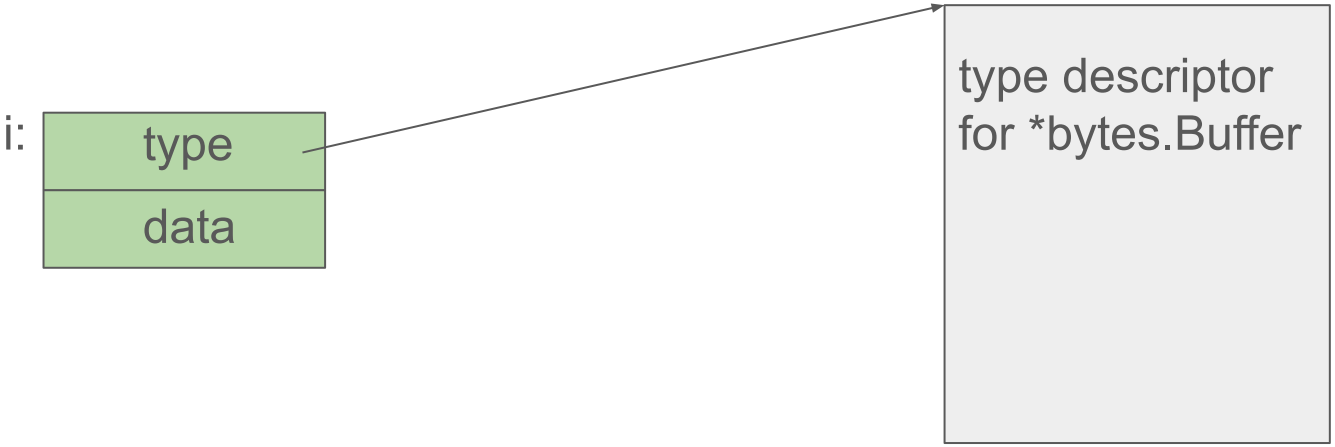
The rest of this work can be expensive!
Can we precompute it somehow?

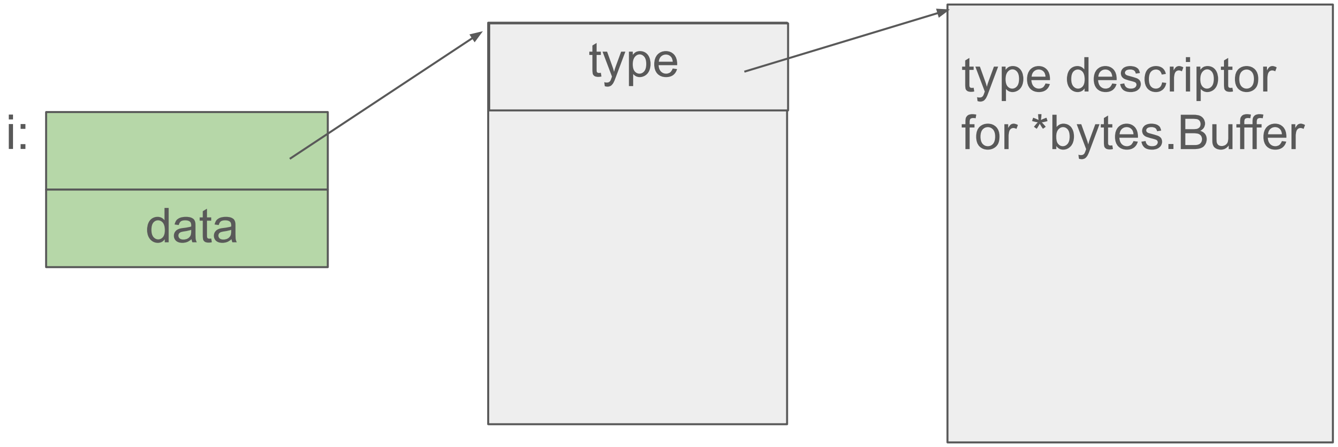
Interfaces are just 2-word structs!

From the runtime:

```
type eface struct {
    _type  *_type
    data   unsafe.Pointer
}
```

Problem: there’s no room!


i:
type
data
type descriptor
for *bytes.Buffer



- The intermediate object is called an “interface table”, or “itab”.
- Provides space to store additional data.

Interfaces are just 2-word structs!

From the runtime:

```
type eface struct
    _type  *_type
    data   unsafe.Pointe
}
```

Interfaces are just 2-word structs!

Empty interfaces, from the runtime:

```
type eface struct {
   _type  *_type
   data   unsafe.Pointer
}
```

Non-empty interfaces, from the runtime:

```
type iface struct {
   tab    *itab
   data   unsafe.Pointer
}
```

Interface Tables

```
type itab struct {
    _type *_type
    inter *interfaceType
    fun   [1]uintptr // variable sized.
}
```

Interface Tables

dynamic, concrete type

```
type itab struct {
    _type *_type
    inter *interfaceType
    fun   [1]uintptr // variable sized.
}
```

Interface Tables

```
type itab struct {
    _type *_type
    inter *interfaceType
    fun   [1]uintptr // variable sized.
}
```

static, interface type

Interface Tables

```
type itab struct {
    _type *_type
    inter *interfaceType
    fun   [1]uintptr // variable sized.
}
```

“method table”, 1 entry for each method of the interface (in sorted order)

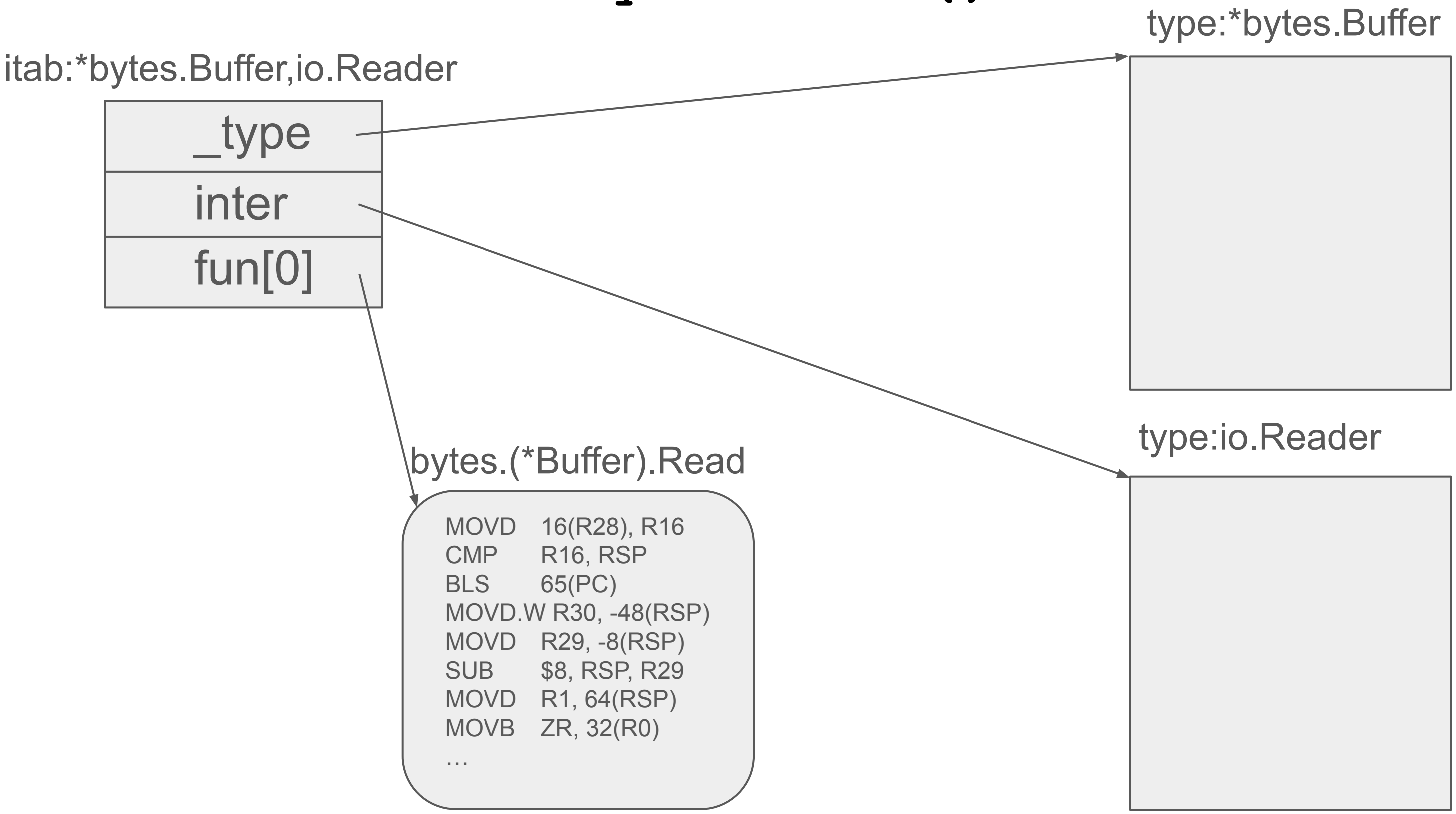
Interface Tables

```
type itab struct {
    _type *_type
    inter *interfaceType
    fun   [1]uintptr // variable sized.
}
```

each entry is a PC for the start of a method

Interface Tables

```
var r io.Reader = &bytes.Buffer{}
```



itab:*bytes.Buffer,io.Reader

_type

inter

fun[0]

type:*bytes.Buffer

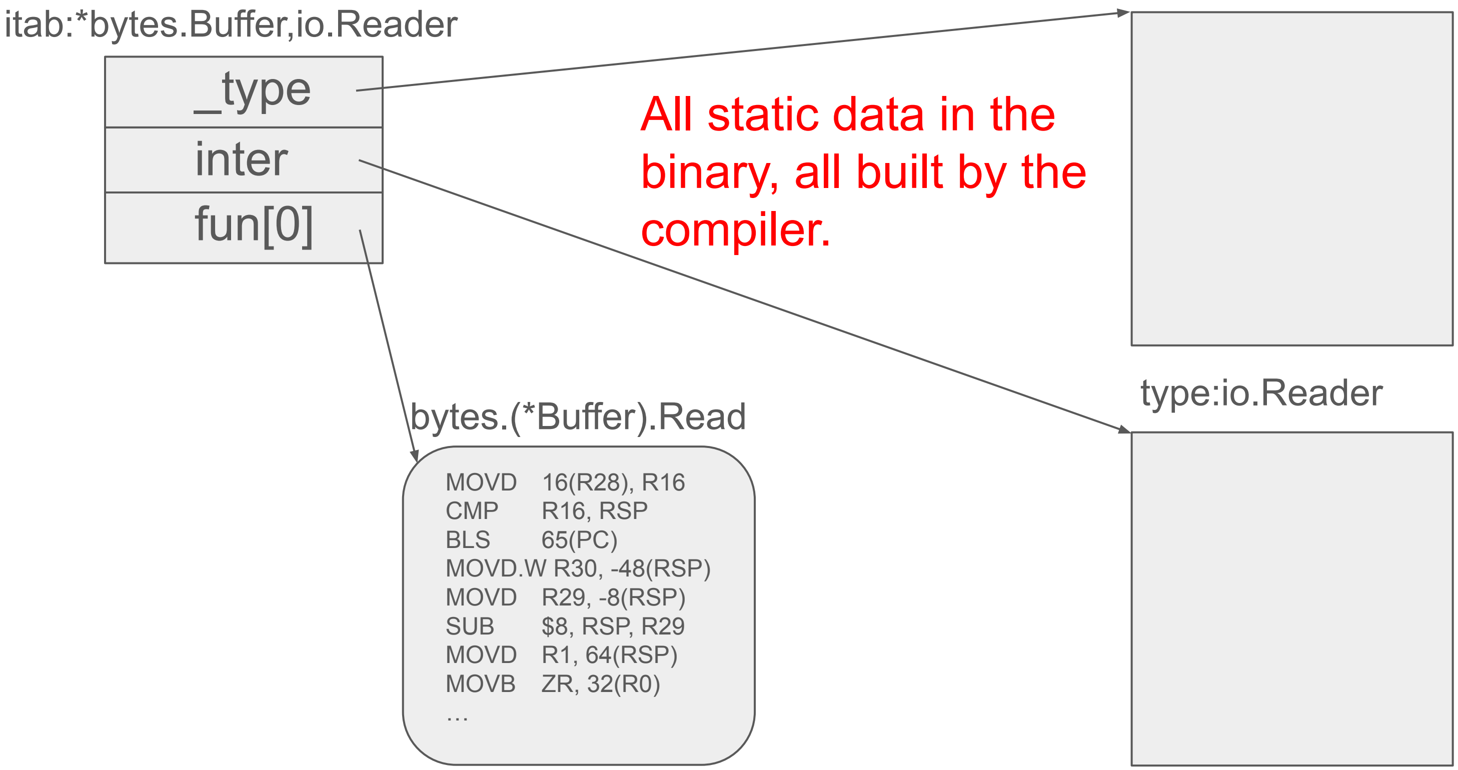
type:io.Reader

bytes.(*Buffer).Read

```
MOVD   16(R28), R16
CMP    R16, RSP
BLS    65(PC)
MOVD.W R30, -48(RSP)
MOVD   R29, -8(RSP)
SUB    $8, RSP, R29
MOVD   R1, 64(RSP)
MOVB   ZR, 32(R0)
…
```

Interface Tables

```
var r io.Reader = &bytes.Buffer{}
```

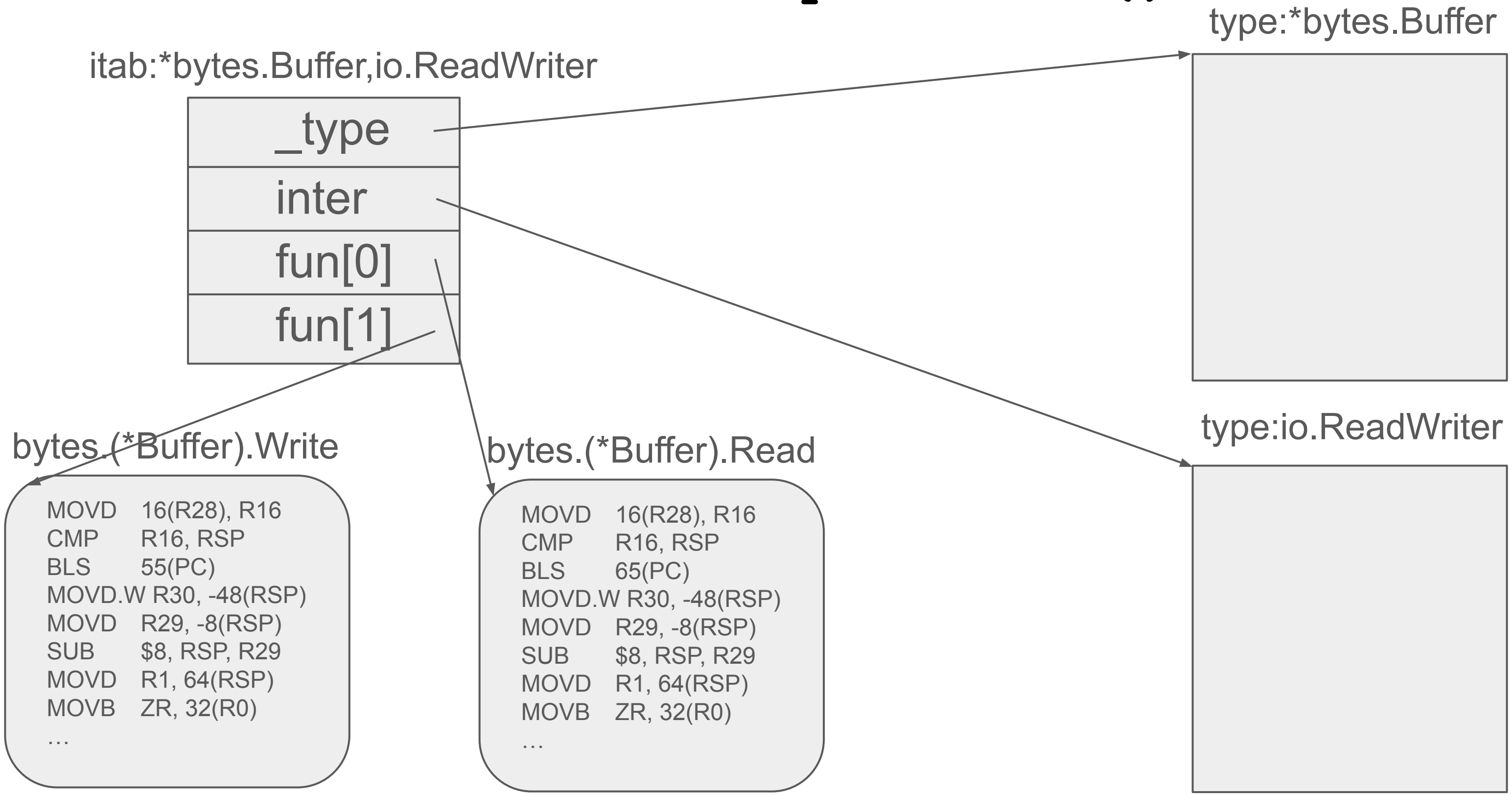


```
MOVD   16(R28), R16
CMP    R16, RSP
BLS    65(PC)
MOVD.W R30, -48(RSP)
MOVD   R29, -8(RSP)
SUB    $8, RSP, R29
MOVD   R1, 64(RSP)
MOVB   ZR, 32(R0)
…
```

All static data in the binary, all built by the compiler.

Interface Tables

```
var r io.ReadWriter = &bytes.Buffer{}
```



bytes.(*Buffer).Write

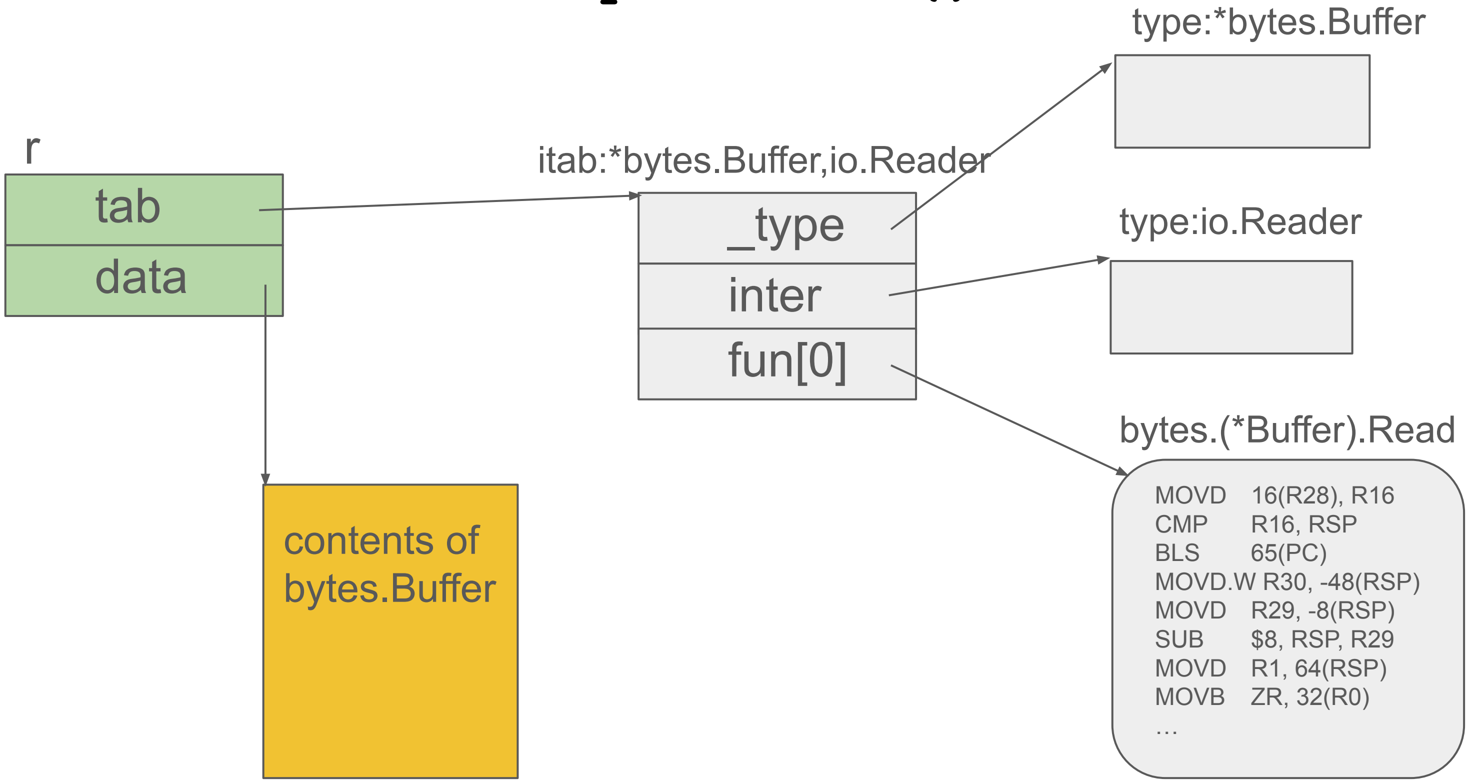
```
MOVD   16(R28), R16
CMP    R16, RSP
BLS    55(PC)
MOVD.W R30, -48(RSP)
MOVD   R29, -8(RSP)
SUB    $8, RSP, R29
MOVD   R1, 64(RSP)
MOVB   ZR, 32(R0)
…
```

bytes.(*Buffer).Read

```
MOVD   16(R28), R16
CMP    R16, RSP
BLS    65(PC)
MOVD.W R30, -48(RSP)
MOVD   R29, -8(RSP)
SUB    $8, RSP, R29
MOVD   R1, 64(RSP)
MOVB   ZR, 32(R0)
…
```

Interface Tables

```
var r io.Reader = &bytes.Buffer{}
```



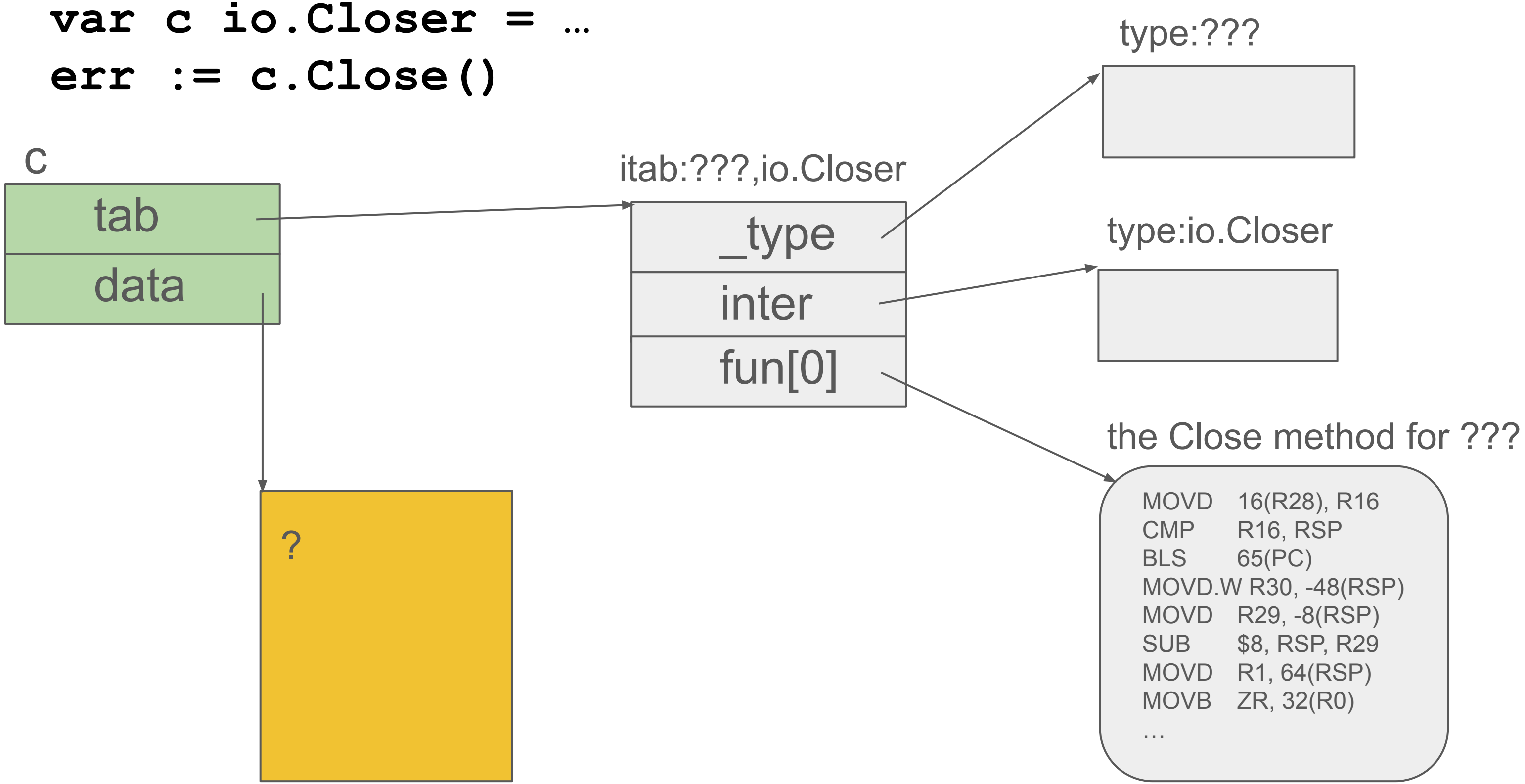
```
MOVD    16(R28), R16
CMP     R16, RSP
BLS     65(PC)
MOVD.W R30, -48(RSP)
MOVD    R29, -8(RSP)
SUB     $8, RSP, R29
MOVD    R1, 64(RSP)
MOVB    ZR, 32(R0)
…
```

interface method invocation

```
var c io.Closer = …
err := c.Close()
```

interface method invocation

```
var c io.Closer = …
err := c.Close()
```



```
MOVD    16(R28), R16
CMP     R16, RSP
BLS     65(PC)
MOVD.W R30, -48(RSP)
MOVD    R29, -8(RSP)
SUB     $8, RSP, R29
MOVD    R1, 64(RSP)
MOVB    ZR, 32(R0)
…
```

interface method invocation

```
var c io.Closer = …
err := c.Close()
```

… c is in R0,R1 …

```
MOVD   24(R0), R2                // pc := c.tab.fun[0]
MOVD   R1, R0                    // receiver := c.data
CALL   (R2)                      // jump to pc
```

… Close method’s receiver is in R0 …

interface method invocation

```
var c io.Closer = …
err := c.Close()
```

24 = offset of fun[0] in the itab struct

… c is in R0,R1 …

```
MOVD   24(R0), R2            // pc := c.tab.fun[0]
MOVD   R1, R0                // receiver := c.data
CALL   (R2)                  // jump to pc
```

… Close method’s receiver is in R0 …

interface -> interface conversion

go1.21 and earlier

```
var r io.Reader = …
if rw, ok := r.(io.ReadWriter); ok { … }
```

… r is in R0,R1 …

```
MOVD   R1, R2
MOVD   R0, R1
MOVD   $type:io.ReadWriter(FP), R0
CALL   runtime.assertI2I2(SB)
CBZ    R0, 64
```

… rw is in R0/R1 …

interface -> interface conversion

go1.21 and earlier

```
var r io.Reader = …
if rw, ok := r.(io.ReadWriter); ok { … }
```

… r is in R0,R1 …

```
MOVD    R1, R2
MOVD    R0, R1
MOVD    $type:io.ReadWriter(FP), R0
CALL    runtime.assertI2I2(SB)
CBZ     R0, 64
```

… rw is in R0/R1 …

The tricky part: we need a new interface table for the result

interface -> interface conversion

go1.21 and earlier

The runtime call:

1. Checks a cache to see if we’ve made this interface table before.
   a. *Required* because interface tables must be unique.
2. Obtains a list of all the methods of the concrete type.
3. Finds each of the interface’s methods in that list.
4. Builds an interface table, caches it, and returns it.

interface -> interface conversion

go1.22 and later

```
var r io.Reader = …
if rw, ok := r.(io.ReadWriter); ok { … }
```

… r is in R0,R1 …

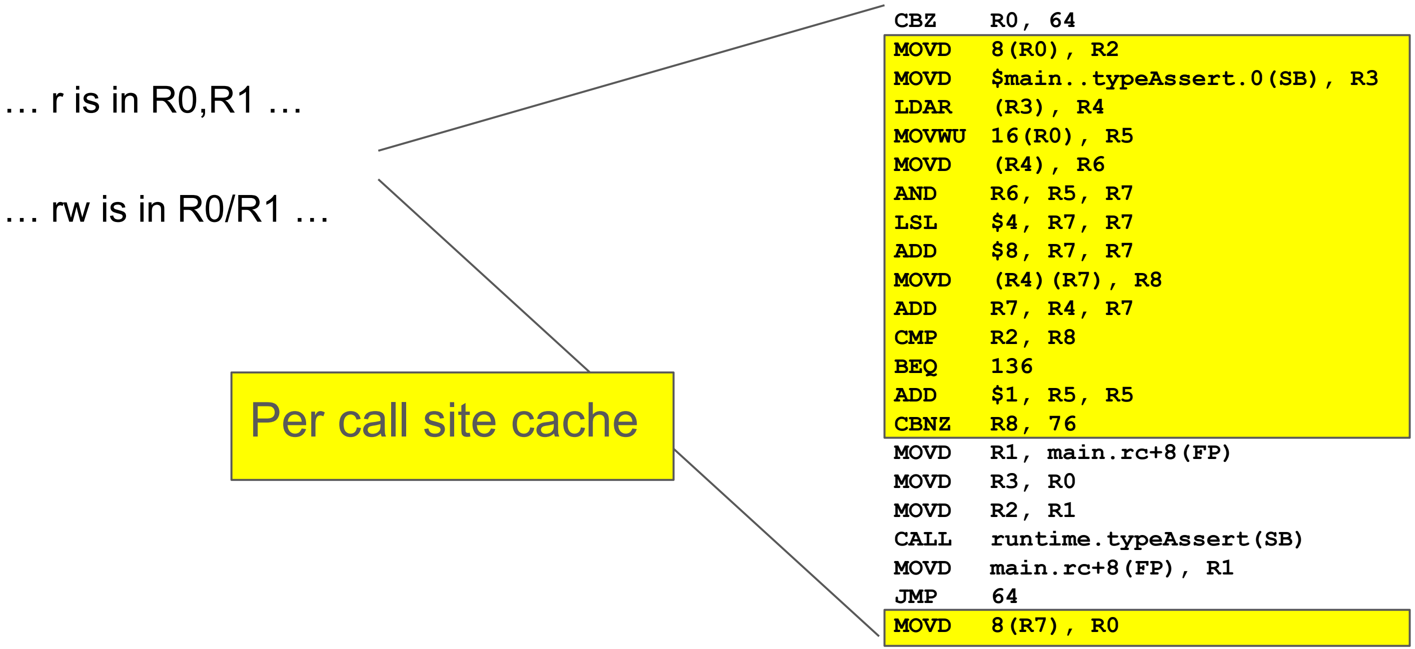
… rw is in R0/R1 …

```
CBZ    R0, 64
MOVD   8(R0), R2
MOVD   $main..typeAssert.0(SB), R3
LDAR   (R3), R4
MOVWU  16(R0), R5
MOVD   (R4), R6
AND    R6, R5, R7
LSL    $4, R7, R7
ADD    $8, R7, R7
MOVD   (R4)(R7), R8
ADD    R7, R4, R7
CMP    R2, R8
BEQ    136
ADD    $1, R5, R5
CBNZ   R8, 76
MOVD   R1, main.rc+8(FP)
MOVD   R3, R0
MOVD   R2, R1
CALL   runtime.typeAssert(SB)
MOVD   main.rc+8(FP), R1
JMP    64
MOVD   8(R7), R0
```

interface -> interface conversion

go1.22 and later

```
var r io.Reader = …
if rw, ok := r.(io.ReadWriter); ok { … }
```



… r is in R0,R1 …

… rw is in R0/R1 …

Per call site cache

```
CBZ     R0, 64
MOVD    8(R0), R2
MOVD    $main..typeAssert.0(SB), R3
LDAR    (R3), R4
MOVWU   16(R0), R5
MOVD    (R4), R6
AND     R6, R5, R7
LSL     $4, R7, R7
ADD     $8, R7, R7
MOVD    (R4)(R7), R8
ADD     R7, R4, R7
CMP     R2, R8
BEQ     136
ADD     $1, R5, R5
CBNZ    R8, 76
MOVD    R1, main.rc+8(FP)
MOVD    R3, R0
MOVD    R2, R1
CALL    runtime.typeAssert(SB)
MOVD    main.rc+8(FP), R1
JMP     64
MOVD    8(R7), R0
```

interface -> interface conversion

go1.22 and later

```
var r io.Reader = …
if rw, ok := r.(io.ReadWriter); ok { … }
```

```
name             old time/op  new time/op  delta
TypeAssert       3.78ns ± 3%  1.00ns ± 1%  -73.53%
TypeSwitch       25.8ns ± 2%   2.5ns ± 3%  -90.43%
```

Conclusion

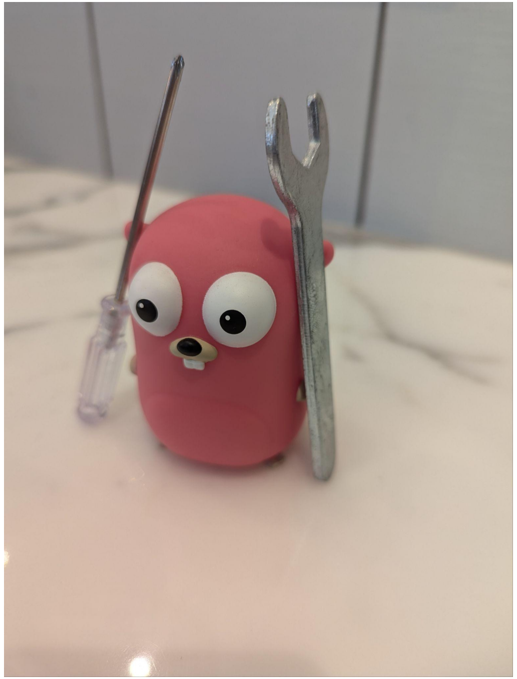
- Interfaces are magic!

Conclusion

- Interfaces are mag

Conclusion

- Interfaces are good engineering.



Things to cover

Interface equality

type asserts, type switches

nil interface vs nil pointer

new faster type switches?

allocation requirements (use pointer types when you can)

interfaces are value types - they can’t be changed except wholesale

method invocation (+ itabs)

type descriptor

Skipping

empty <-> nonempty conversions

comparing 2 interfaces to each other?

wrapper functions for non-pointers

concrete -> interface conversion

```
var b *bytes.Buffer = …
var r io.Reader = b
```

… b starts in R0 …

```
MOVD   R0, R1
MOVD   $itab:*bytes.Buffer,io.Reader(SB), R0
```

… r is in R0,R1 …

interface -> concrete conversion

```
var r io.Reader = …
if b, ok := r.(*bytes.Buffer); ok { … }
```

… r is in R0,R1 …

```
MOVD   $itab:*bytes.Buffer,io.Reader(SB), R2
CMP    R2, R0
BNE    32
```

… b is in R1 …

interface -> any conversion

```
var r io.Reader = ...
var i any = r
```

… r is in R0,R1 …

```
CBZ    R0, 16
MOVD   8(R0), R0
```

… i is in R0/R1 …

interface -> any conversion

```
var r io.Reader = …
var i any = r
```

```
… r is in R0,R1 …
CBZ    R0, 16
MOVD   8(R0), R0
… i is in R0/R1 …
```

copies the Type field out of the interface table

data field unmodified!

interface -> interface down conversion

go1.21 and earlier

```
var rc io.ReadCloser = ...
var r io.Reader = rc
```

… rc is in R0,R1 …

```
MOVD   R1, main.rc+8(FP)
MOVD   R0, R1
MOVD   $type:io.Closer(FP), R0
CALL   runtime.convI2I(SB)
MOVD   main.rc+8(FP), R1
```

… r is in R0/R1 …

interface -> interface down conversion

go1.22 and later

```
var rc io.ReadCloser = ...
var r io.Reader = rc
```

… rc is in R0,R1 …

… r is in R0/R1 …

```
CBZ    R0, 64
MOVD   8(R0), R2
MOVD   $main..typeAssert.0(SB), R3
LDAR   (R3), R4
MOVWU  16(R0), R5
MOVD   (R4), R6
AND    R6, R5, R7
LSL    $4, R7, R7
ADD    $8, R7, R7
MOVD   (R4)(R7), R8
ADD    R7, R4, R7
CMP    R2, R8
BEQ    136
ADD    $1, R5, R5
CBNZ   R8, 76
MOVD   R1, main.rc+8(FP)
MOVD   R3, R0
MOVD   R2, R1
CALL   runtime.typeAssert(SB)
MOVD   main.rc+8(FP), R1
JMP    64
MOVD   8(R7), R0
```