Efficient VM with JIT in Go

quasilyte @ GoWayFest 4.0 (2020)

Part 0/7

Backstory

Once upon a time:

“Can we use Lucene from Go?”

Once upon a time:

"Can we use Lucene from Go?"

Sure...

Once upon a time:

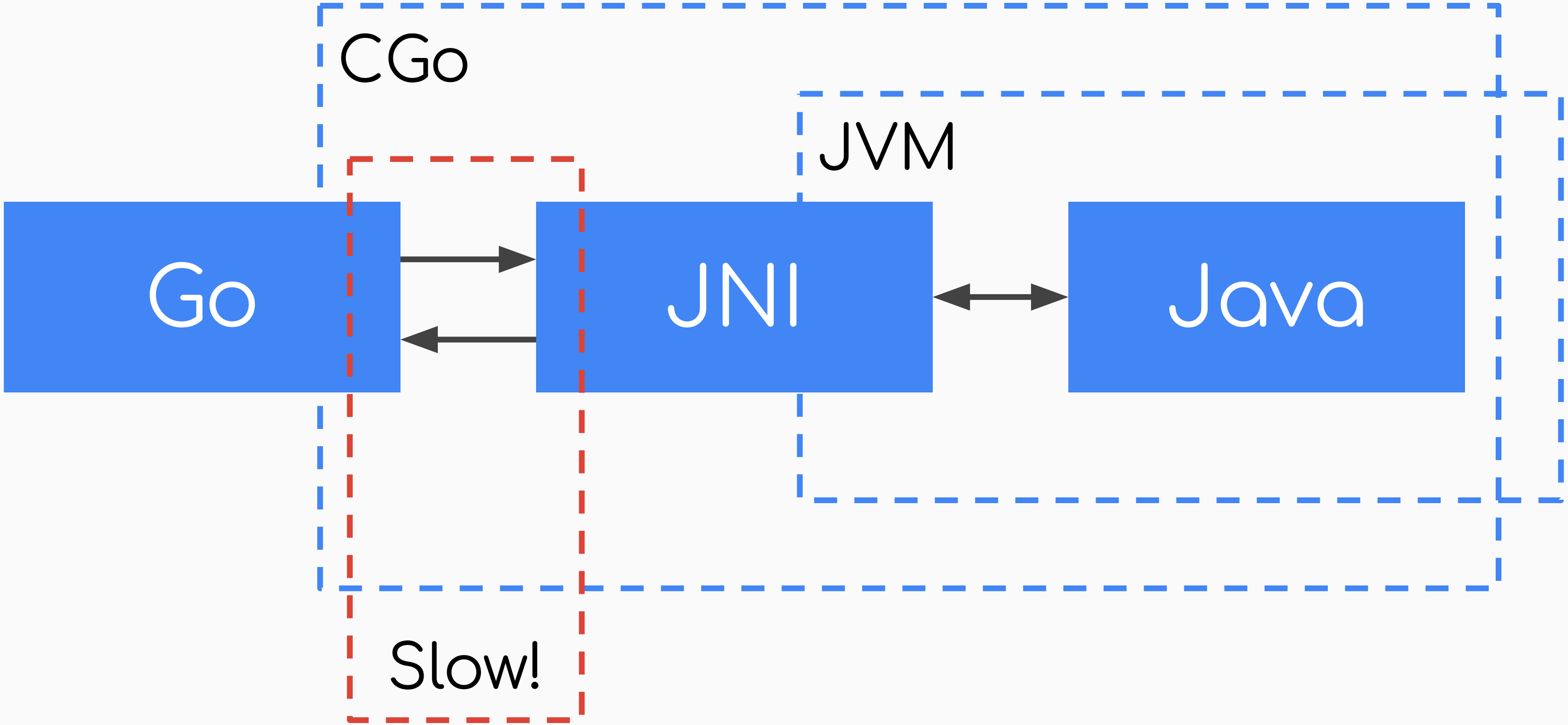
“Can we use Lucene from Go?”

Sure…

How to use Java from Go?

- JNI (with CGo blessing)
- Pass arguments through serialization

https://github.com/timob/jnigi



Go→Java interop with JNI overview

Why JNI is not good for Go?

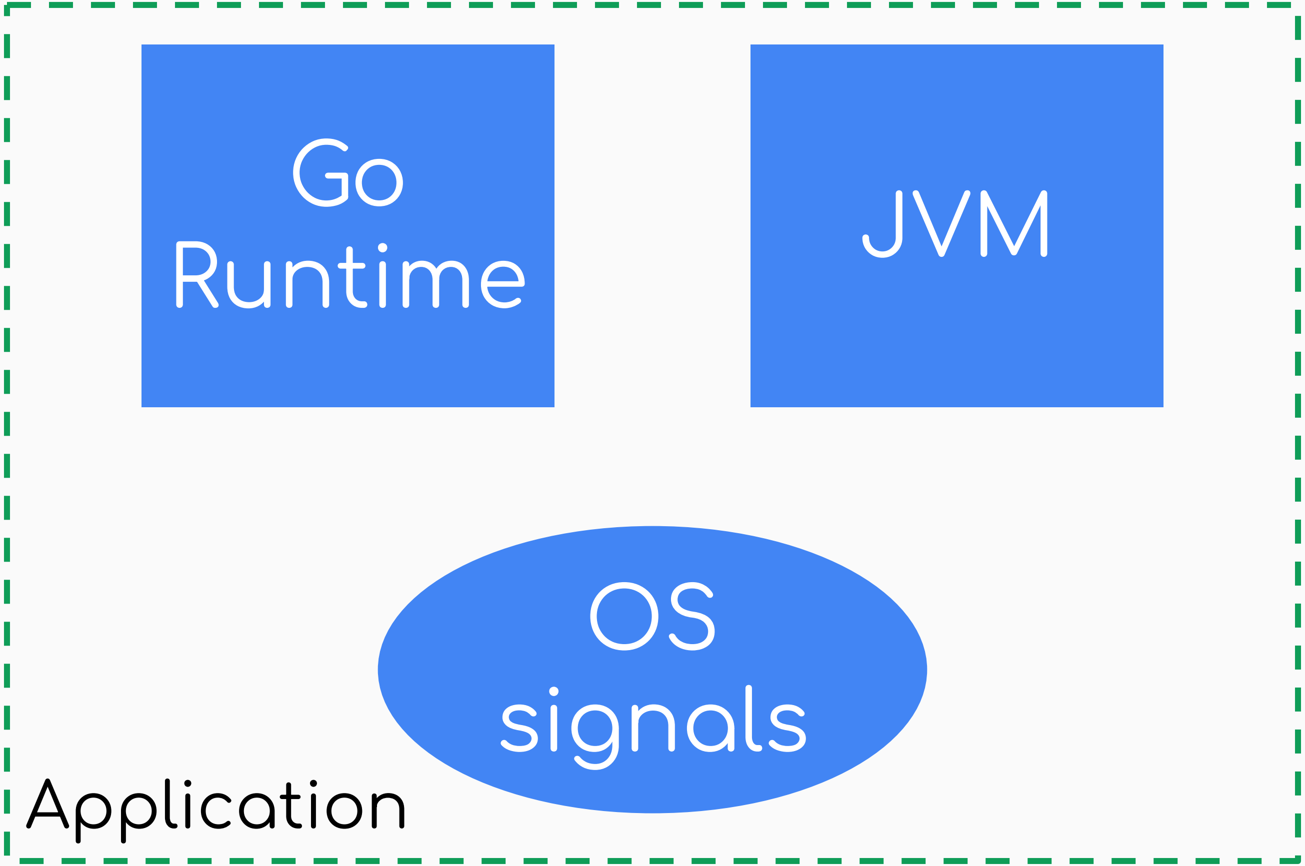
- Locked OS thread for JVM goroutines

Why JNI is not good for Go?

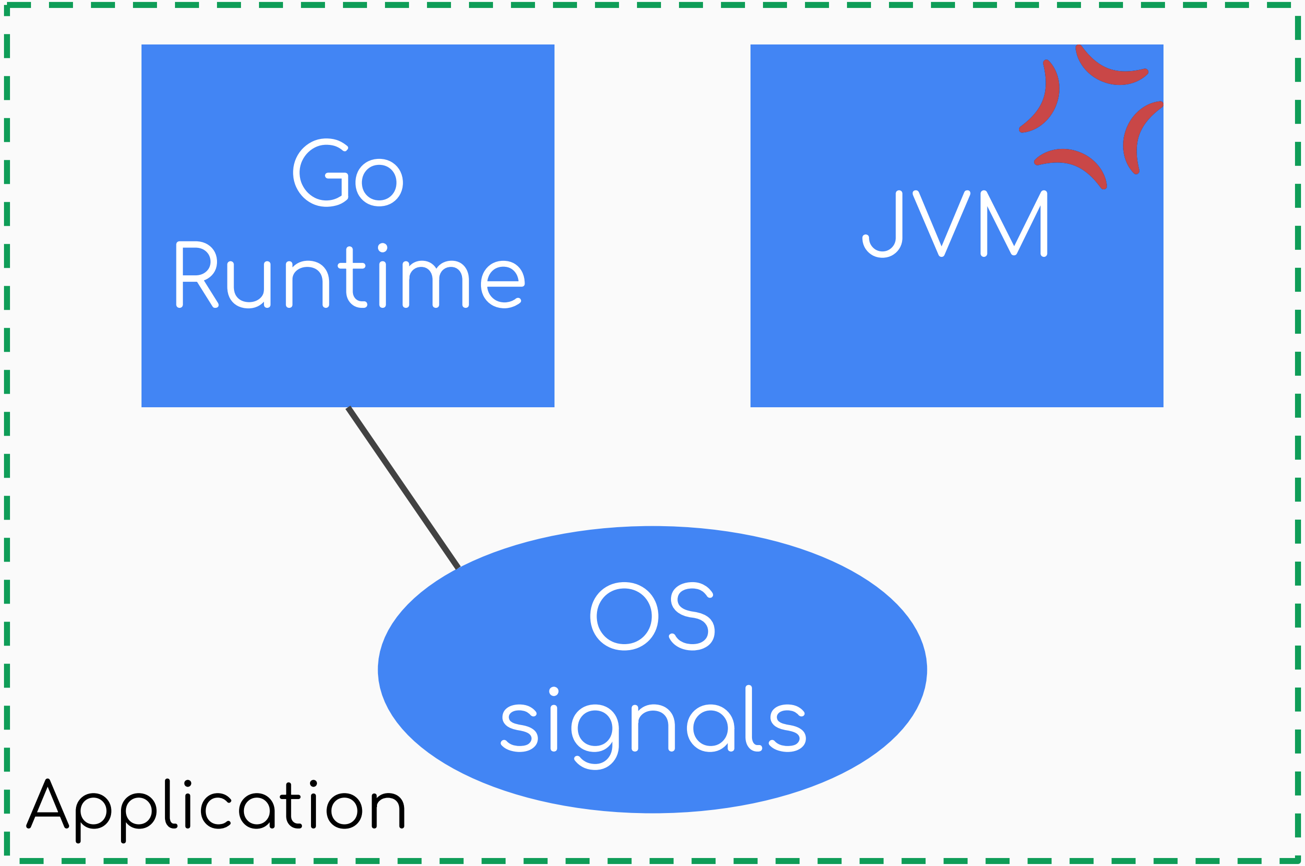
- Locked OS thread for JVM goroutines
- Every JNI call has CGo call overhead

Why JNI is not good for Go?

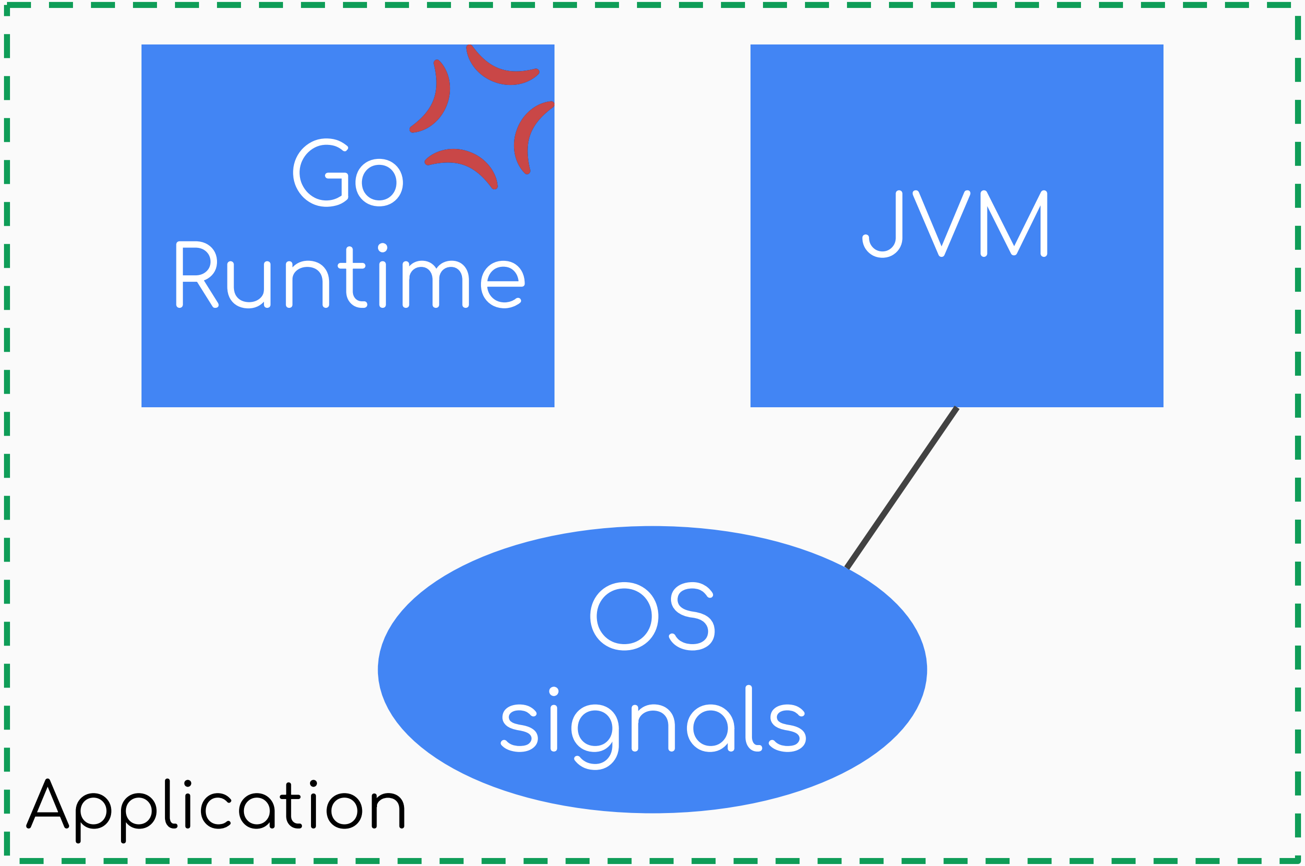
- Locked OS thread for JVM goroutines
- Every JNI call has CGo call overhead
- Expensive Go↔JNI values conversion



Two active runtimes in one application



Two active runtimes in one application



Two active runtimes in one application

Long story short...

We're now using Lucene from our Go application, but

Long story short...

We're now using Lucene from our Go application, but

it bothers me how inefficient it is. Can we do better?

Part 1/7

go-jdk overview

Let’s try build an <u>efficient</u> JVM that can be easily embedded into <u>Go applications</u>.

Me (just now)

Quote

So, what exactly do we want?

- Cheap Go↔Java calls (and no CGo)

So, what exactly do we want?

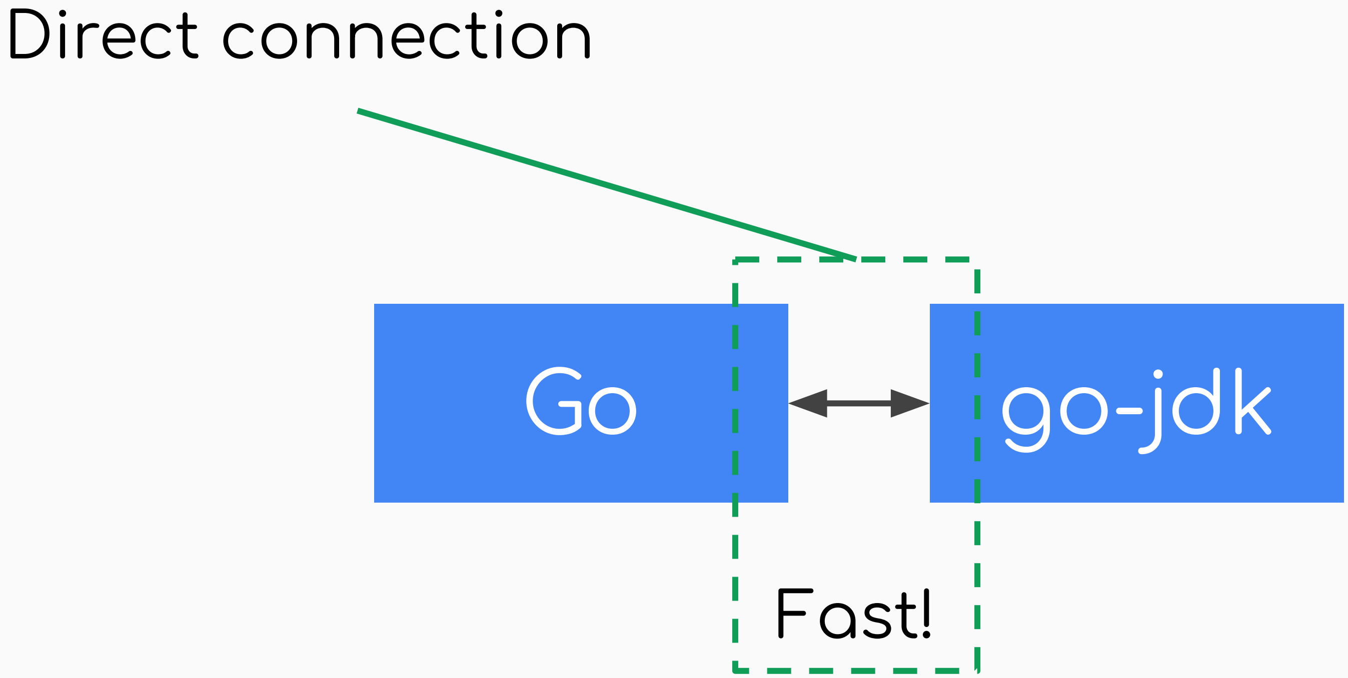
- Cheap Go↔Java calls (and no CGo)
- Optimized machine code (no interpretation)

So, what exactly do we want?

- Cheap Go↔Java calls (and no CGo)
- Optimized machine code (no interpretation)
- Efficient objects layout and allocation

So, what ... want?

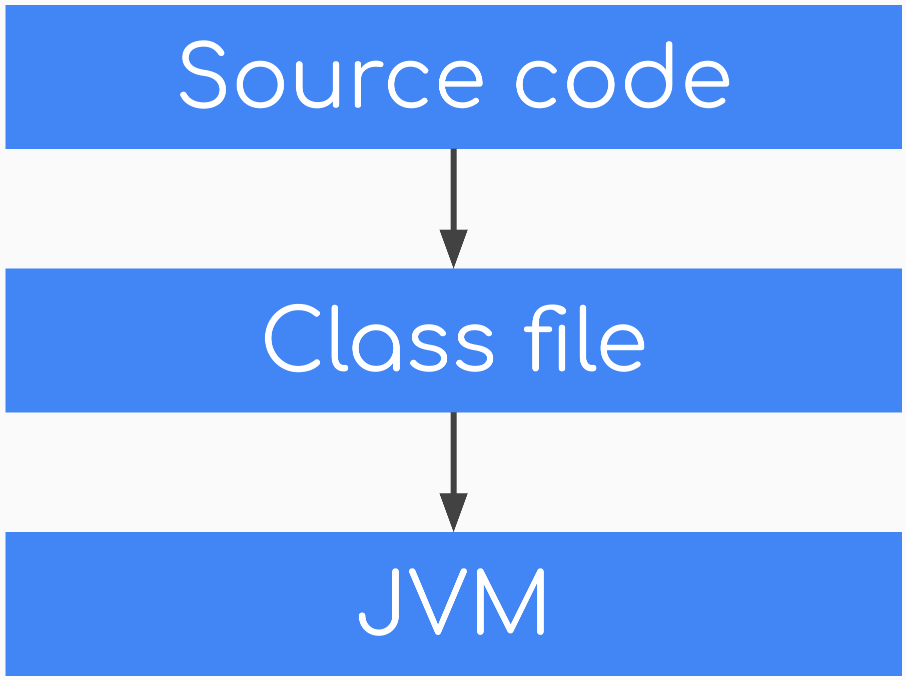
- Cheap ... d no CGo)
- Opti ... terpretation)
- Efficient obj ... allocation


Direct connection
Go
go-jdk
Fast!

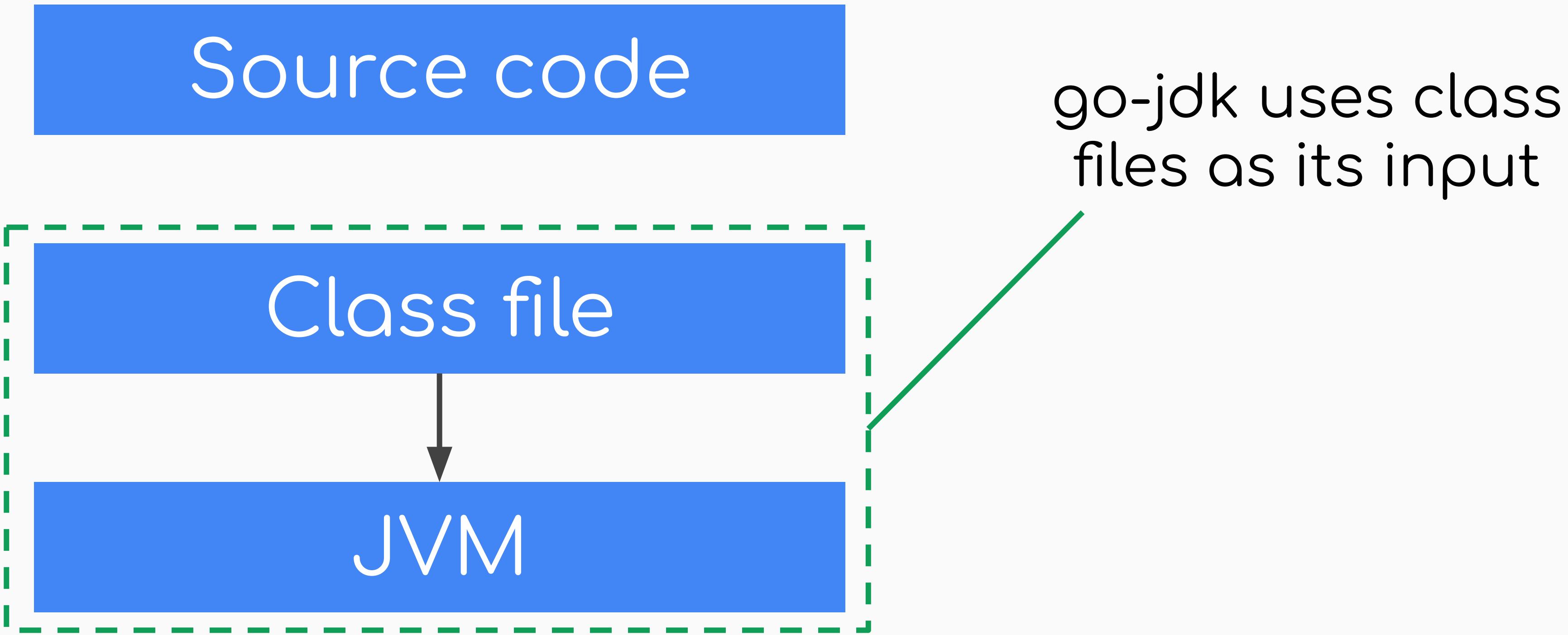
go-jdk project

- Java class file loader
- JIT compiler (non-tracing)
- Runtime and interop primitives
- Utility tools like “javap”

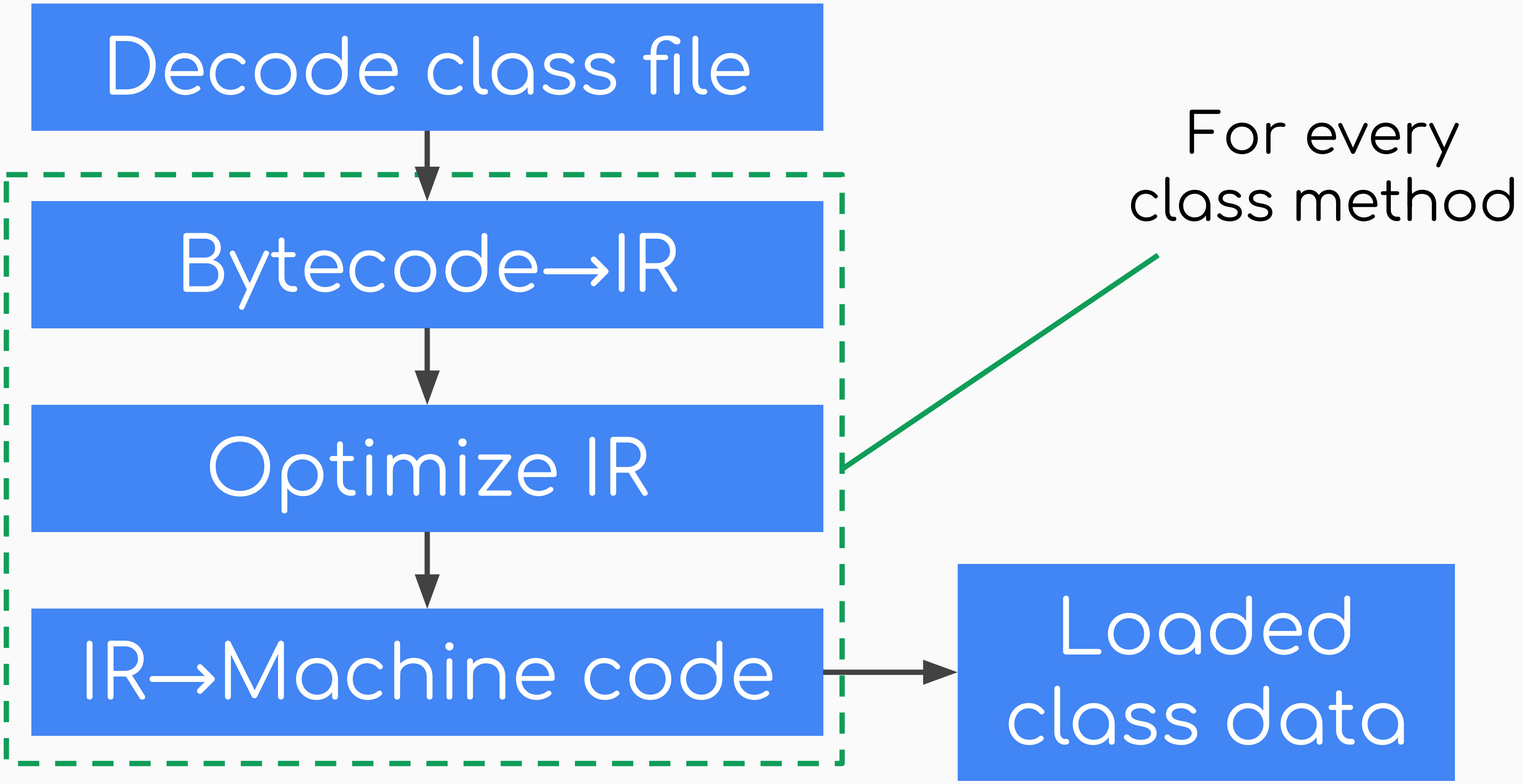
https://github.com/quasilyte/go-jdk



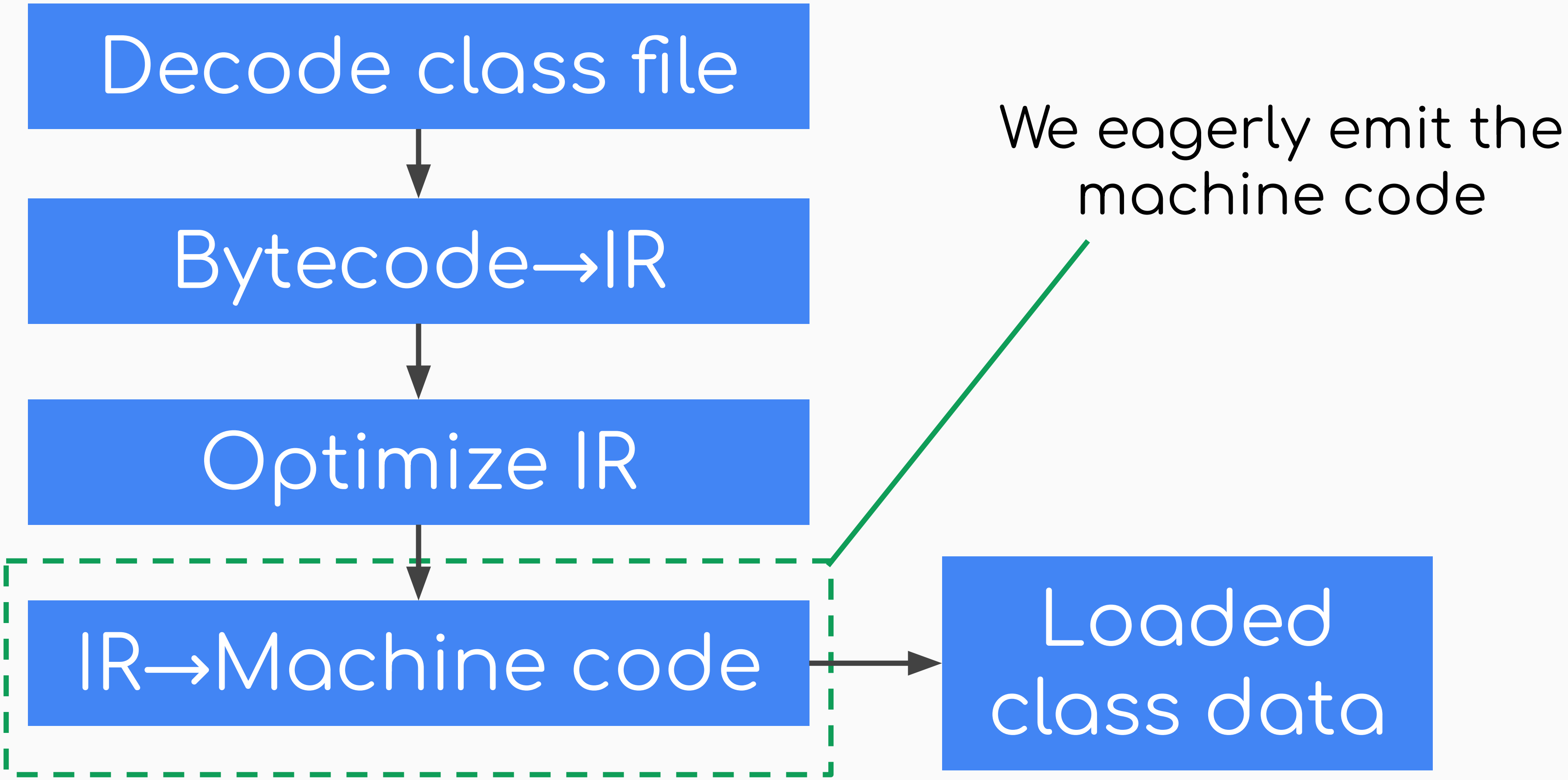
go-jdk inputs



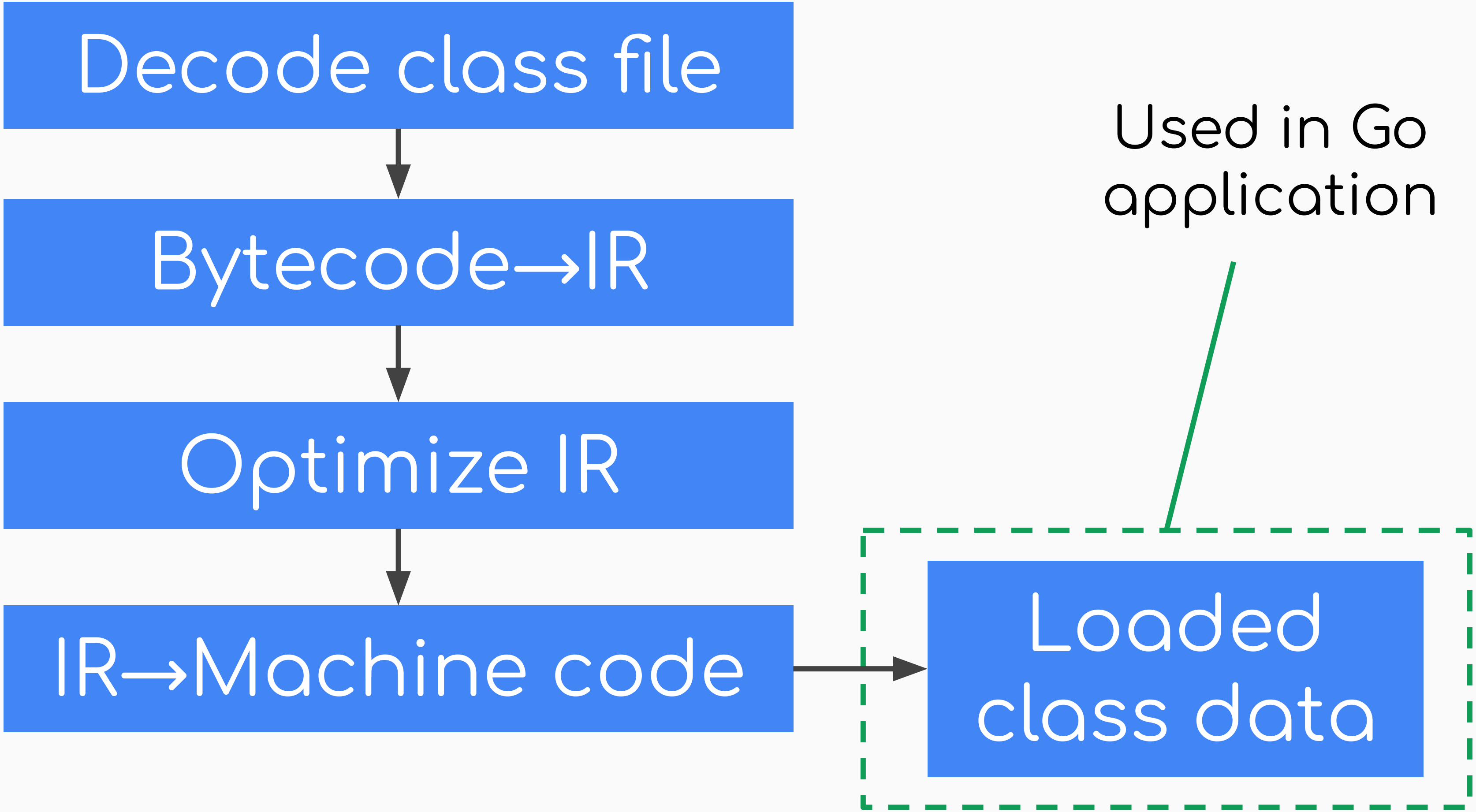
go-jdk inputs



How class is loaded



How class is loaded

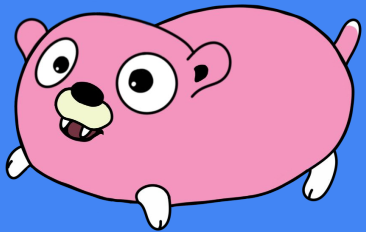


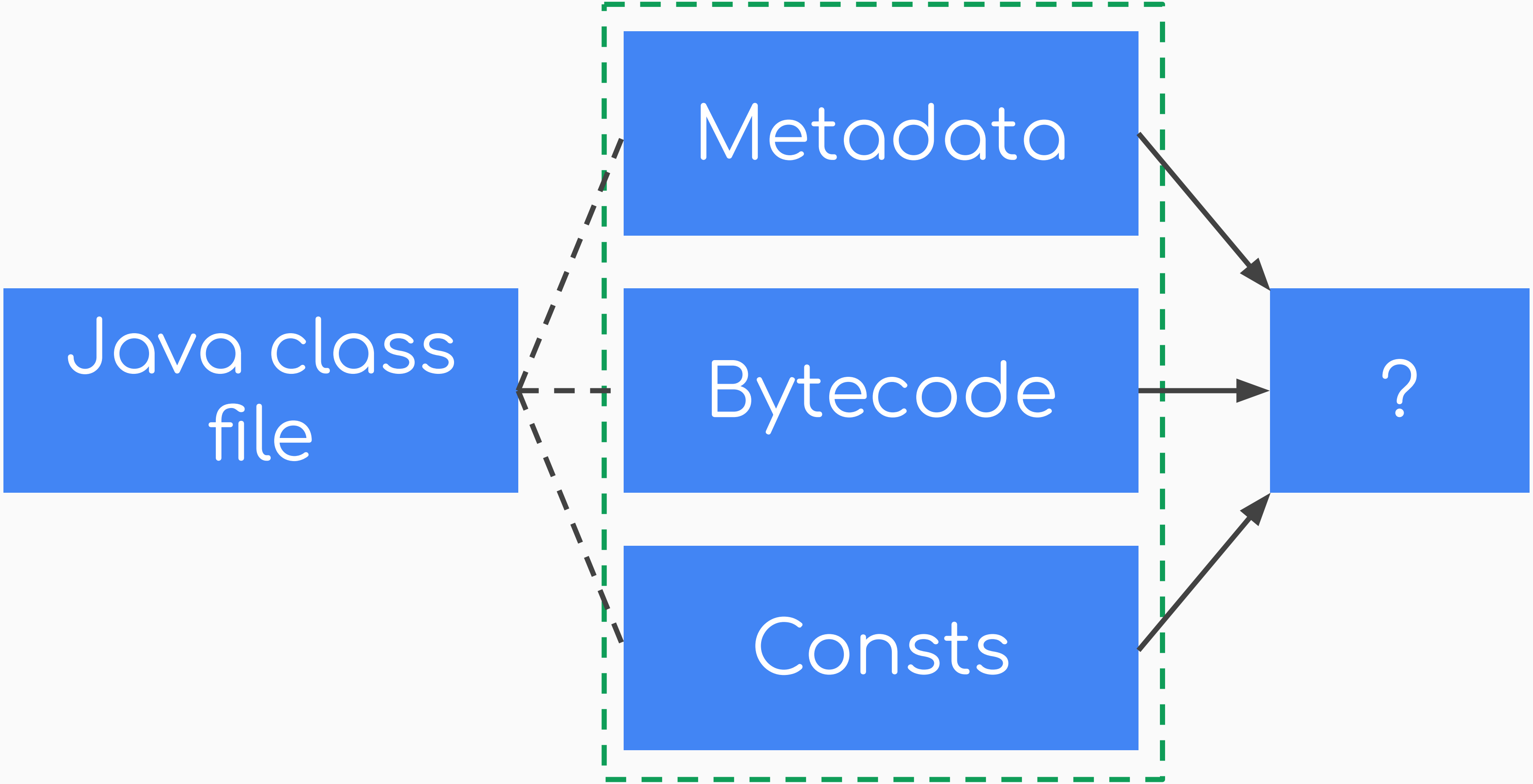
How class is loaded

Part 2/7

Making the code run fast

```
  0 - Backstory
  1 - go-jdk overview
> 2 - Making the code run fast
  3 - GC-friendly slots
  4 - Interop / FFI
  5 - Object layout / mem alloc
  6 - Challenges & limitations
  7 - Closing words
```





Convert Java class file into... what?

Idea 1: direct bytecode to machine code translation

Our example class and example static method

```
class Example {
   public static int add1(int x) {
      return x + 1;
   }
}
```

bytecode→amd64

`iload_0`	`MOVQ local_0(CX), AX`
	`MOVQ AX, (CX)`
	`ADDQ $8, CX`

`iconst_1`	`MOVQ $1, (CX)` `ADDQ $8, CX`

`iadd`	`MOVQ -16(CX), AX`
	`ADDQ -8(CX), AX`
	`MOVQ AX, -16(CX)`
	`SUBQ $8, CX`

`ireturn`	`RET`

bytecode→amd64

iload_0	MOVQ local_0(CX), AX
	MOVQ AX, (CX)
	ADDQ $8, CX

iconst_1	MOVQ $1, (CX) ADDQ $8, CX

iadd	MOVQ -16(CX), AX
	ADDQ -8(CX), AX
	MOVQ AX, -16(CX)
	SUBQ $8, CX

ireturn	RET

Stack bookkeeping

bytecode→amd64

```
iload_0


iconst_1

iadd



ireturn
```

```
MOVQ local_0(CX), AX
MOVQ AX, (CX)
ADDQ $8, CX
MOVQ $1, (CX)
ADDQ $8, CX
MOVQ -16(CX), AX
ADDQ -8(CX), AX
MOVQ AX, -16(CX)
SUBQ $8, CX
RET
```

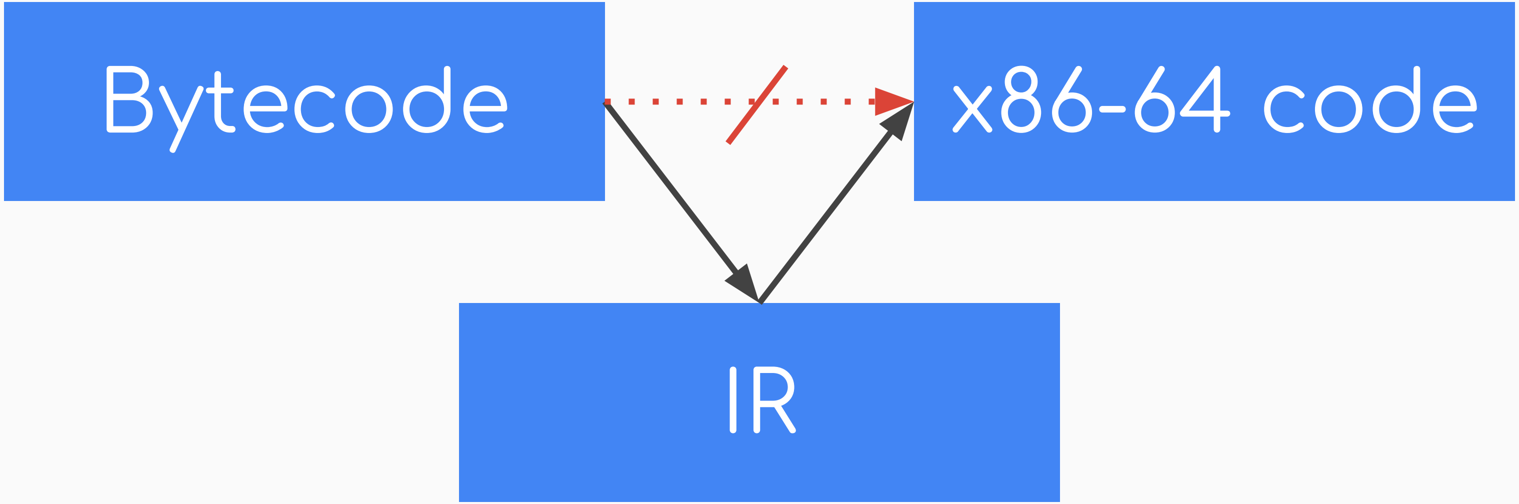
Hard to analyze and optimize

Stack vs Register architecture

Suggested reading:

[VM Showdown: Stack Versus Registers]

We can't change the input bytecode format, but we can add intermediate representation.



Idea 2: add intermediate representation

bytecode→IR

```
iload_0
iconst_1
iadd

ireturn
```

```
r1 = iadd r0 1



iret r1
```

IR→amd64

```
r1 = iadd r0 1



iret r1
```

```
MOVQ local_0(CX), AX
ADDQ $1, AX
MOVQ AX, local_1(CX)

MOVQ local_1, AX
RET
```

IR→amd64

```
r1 = iadd r0 1

iret r1
```

```
MOVQ local_0(CX), AX
ADDQ $1, AX
MOVQ AX, local_1(CX)

MOVQ local_1, AX
RET
```

Can be optimized-out

IR→amd64

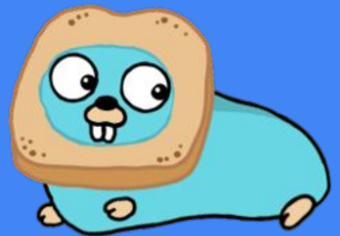
```
ret = iadd r0 1



iret ret
```

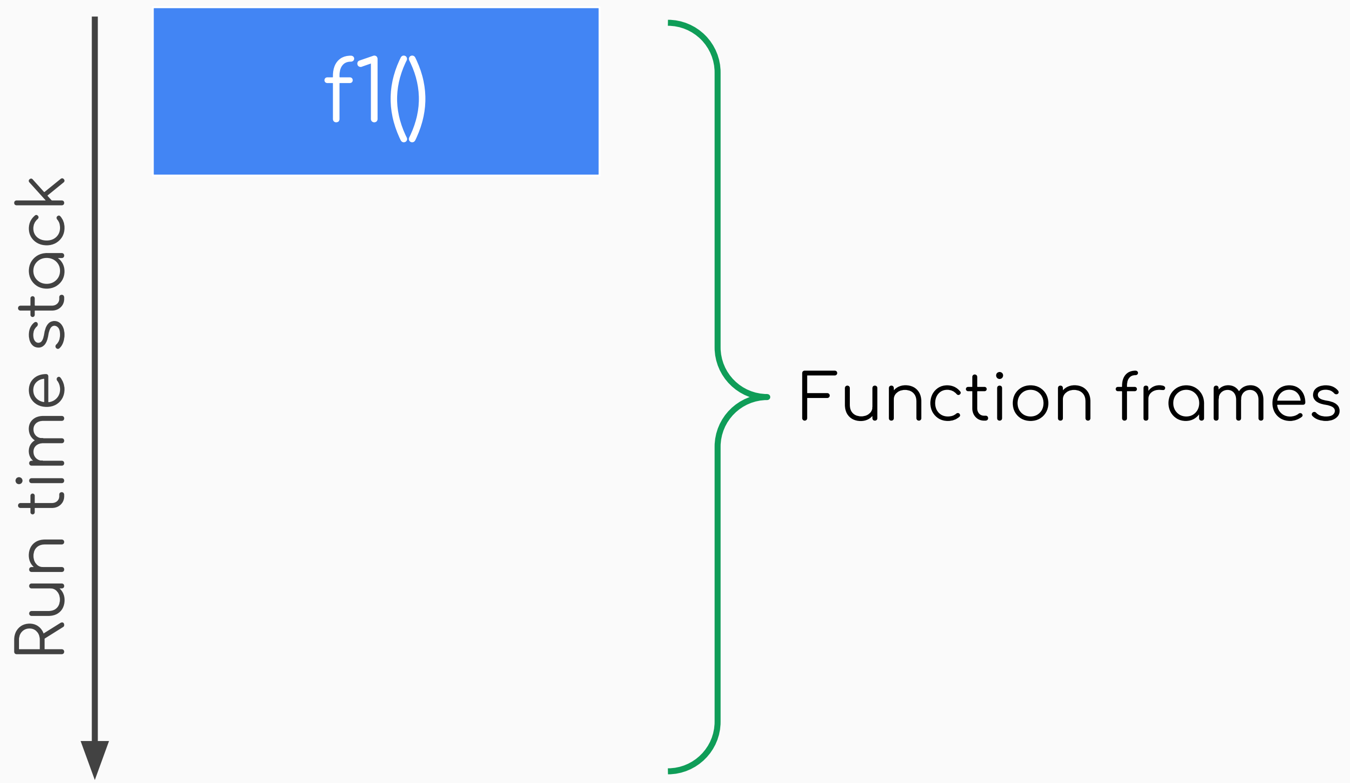
Mapped to
AX (or X0)

```
MOVQ local_0(CX), AX
ADDQ $1, AX



RET
```

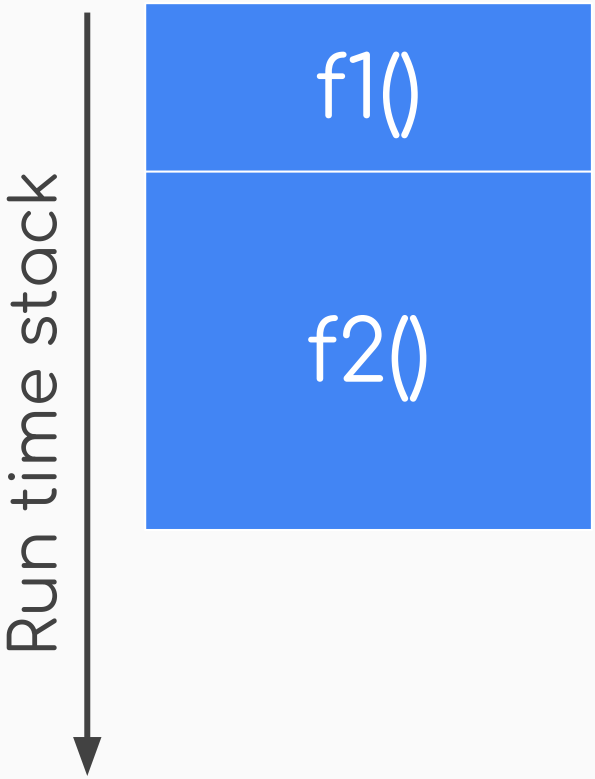
Part 3/7

GC-friendly slots



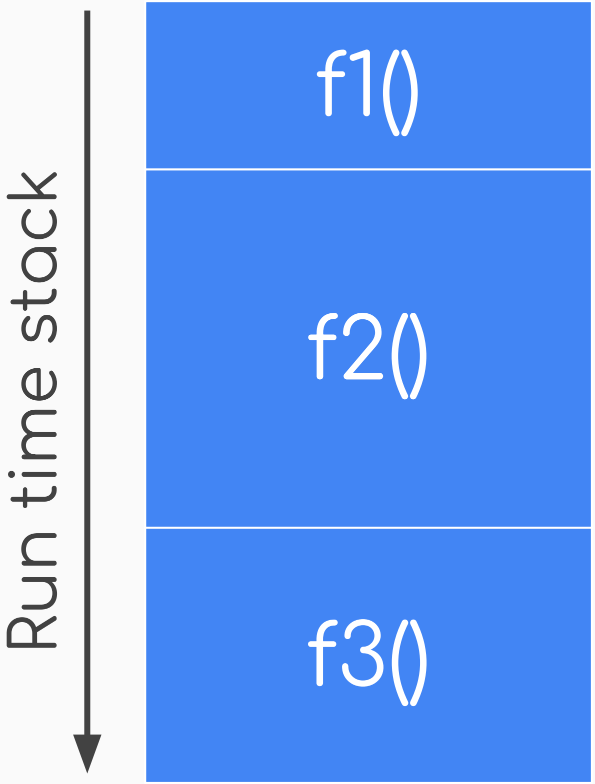


Run time data lives inside the run time stack



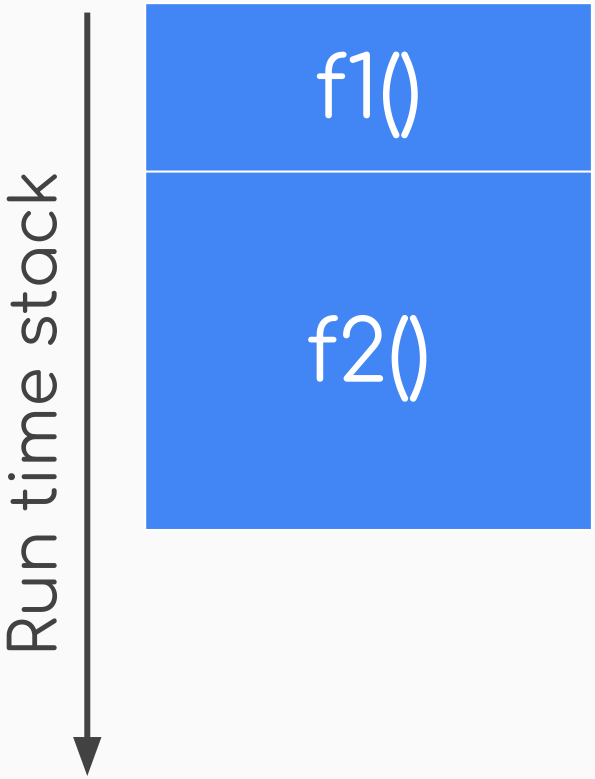
f1() calls f2()

Run time data lives inside the run time stack



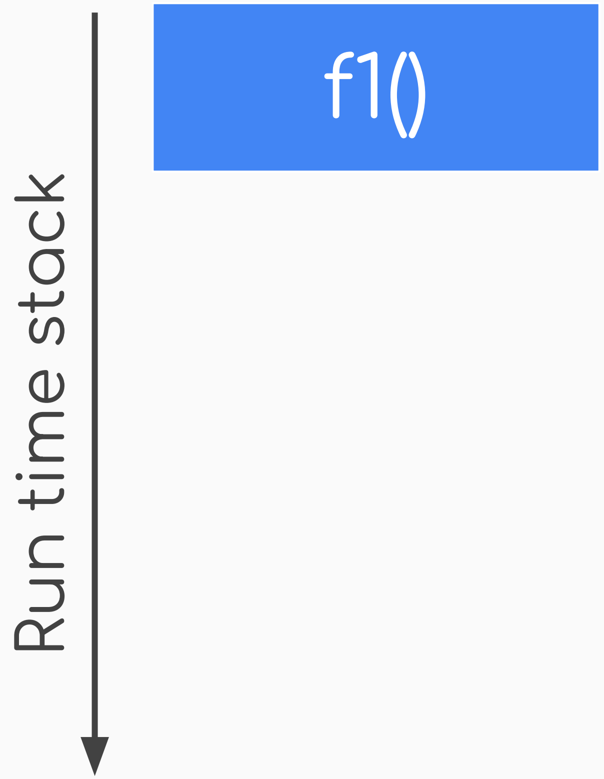

```
f1() calls f2()
f2() calls f3()
```

Run time data lives inside the run time stack



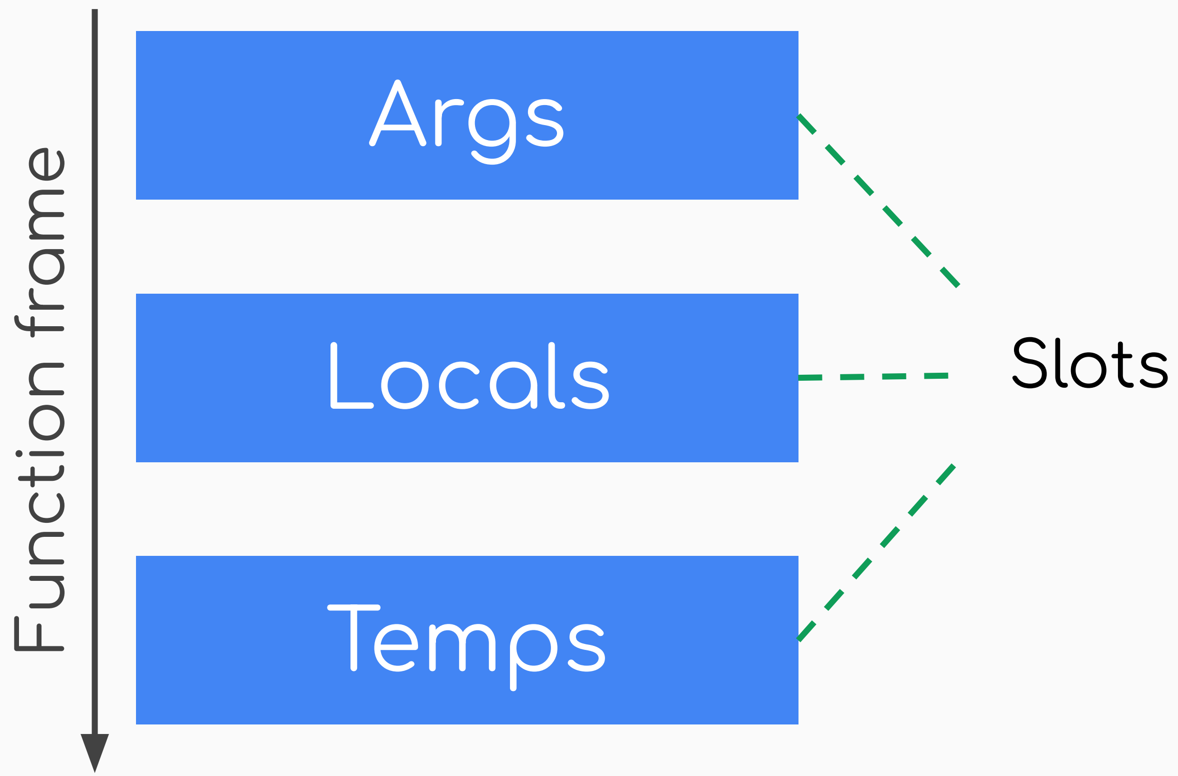

```
f1() calls f2()
f2() calls f3()
f3() returns
```

Run time data lives inside the run time stack

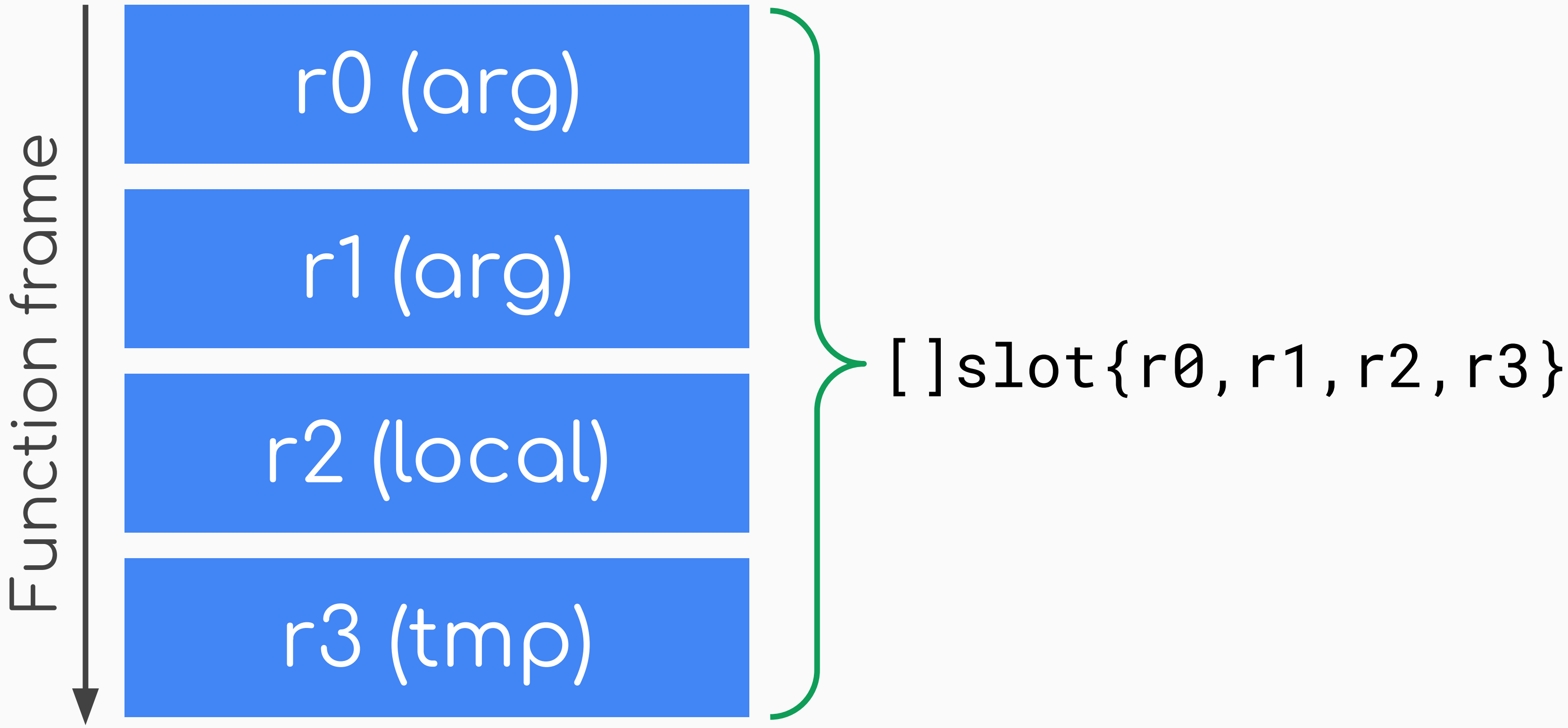



```
f1() calls f2()
f2() calls f3()
f3() returns
f2() returns
```

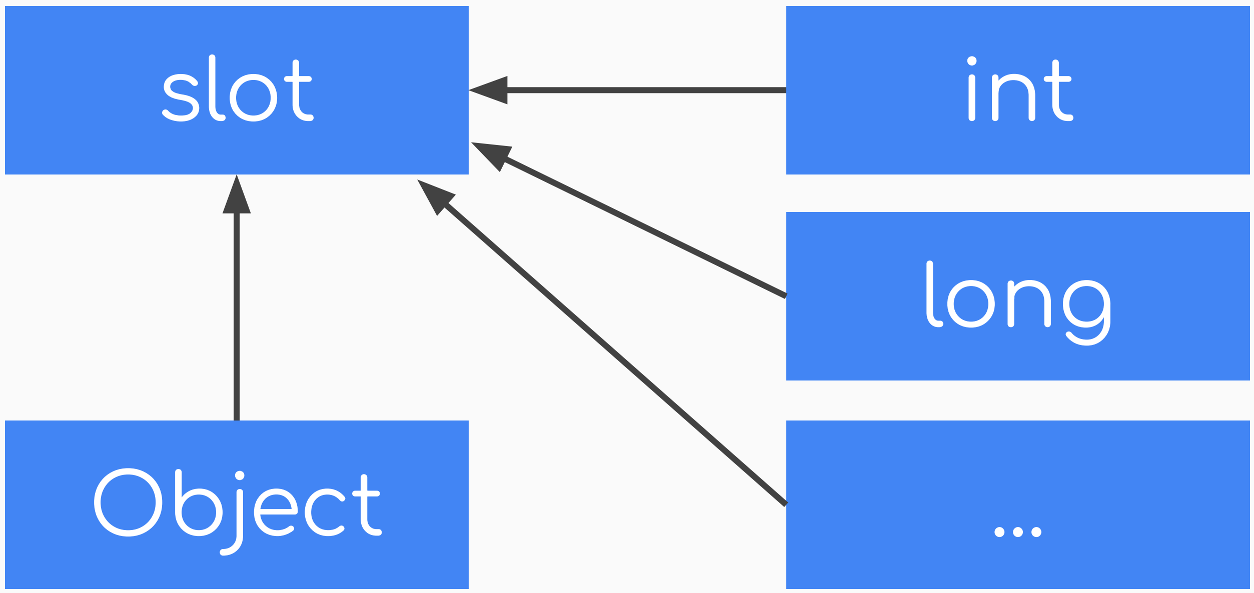
Run time data lives inside the run time stack



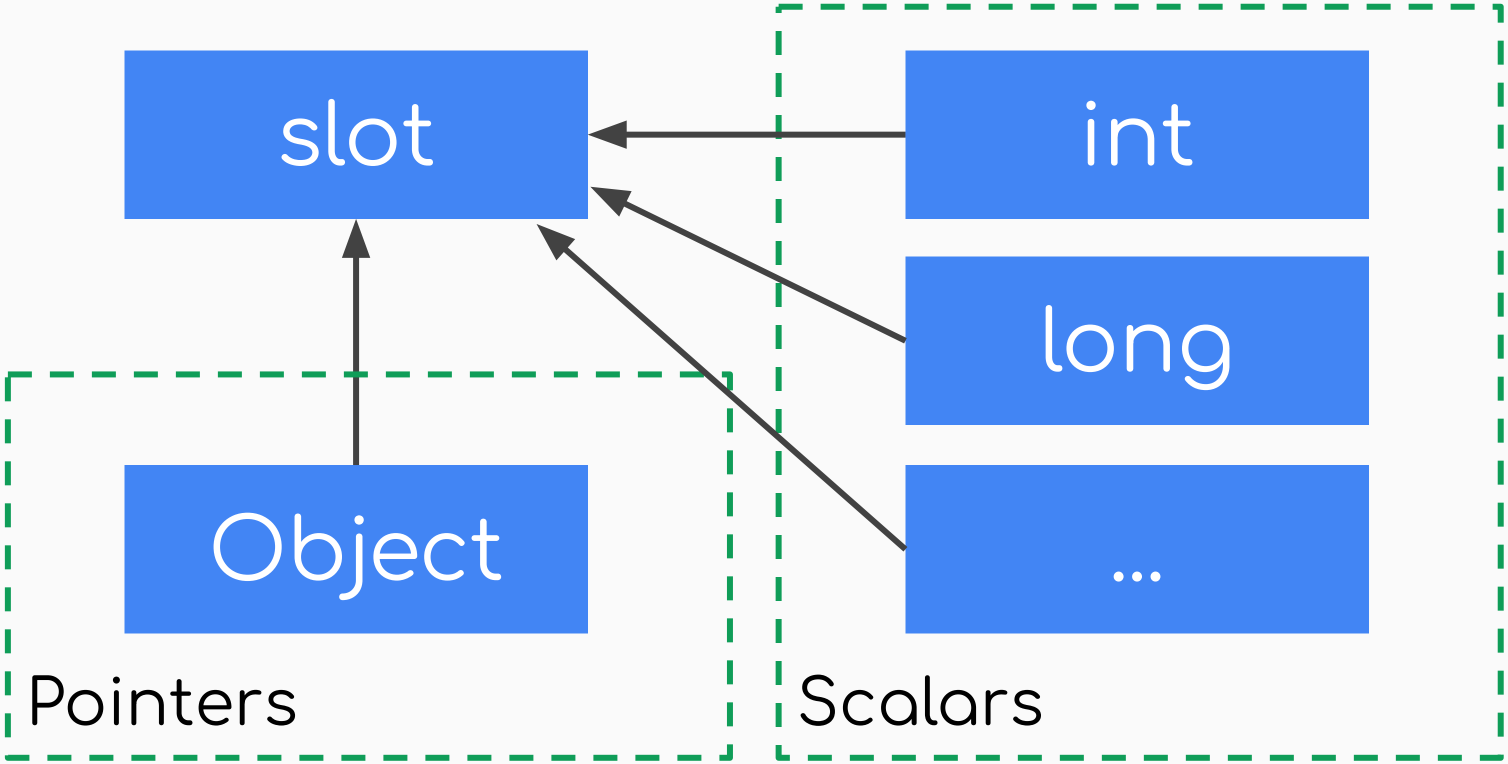
Function frame model (abstract)



Function frame model (concrete)

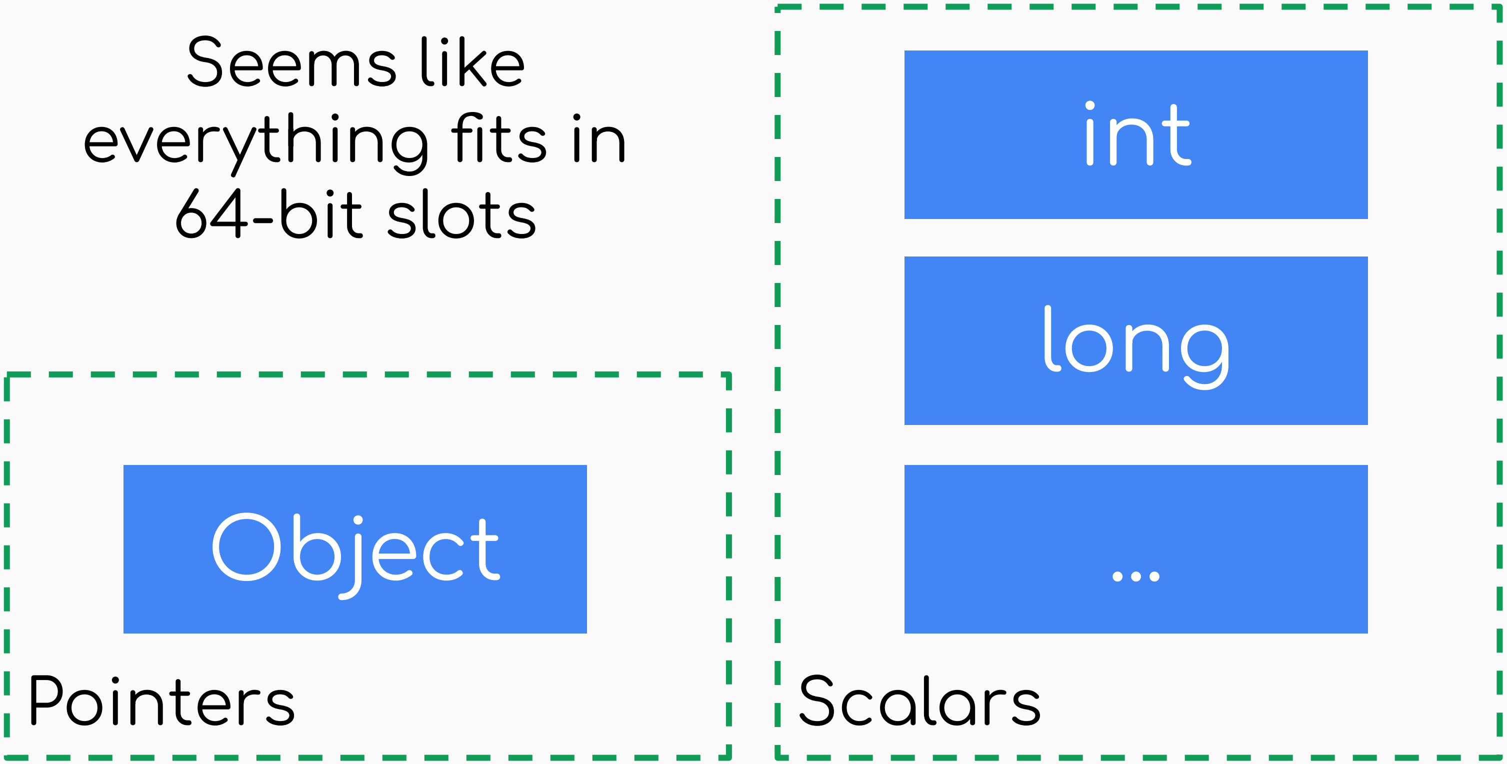


What do we store inside a slot?



What do we store inside a slot?

Seems like everything fits in 64-bit slots



What do we store inside a slot?

Function frame

r0	r1	r2	...

```go
type slot struct {
   value uint64
}
```

Uint64 slots

Function frame

r0	r1	r2	...

```go
type slot struct {
  value uint64
}
```

Not safe to store pointers there!

Uint64 slots

Function frame

r0 r1 r2 ...

```go
type slot struct {
  value uintptr
}
```

uintptr does not retain pointers neither

Uintptr slots

Function frame

r0 r1 r2 ...

```
type slot struct {
   value unsafe.Pointer
}
```

Not safe to store scalars there!

Pointer slots

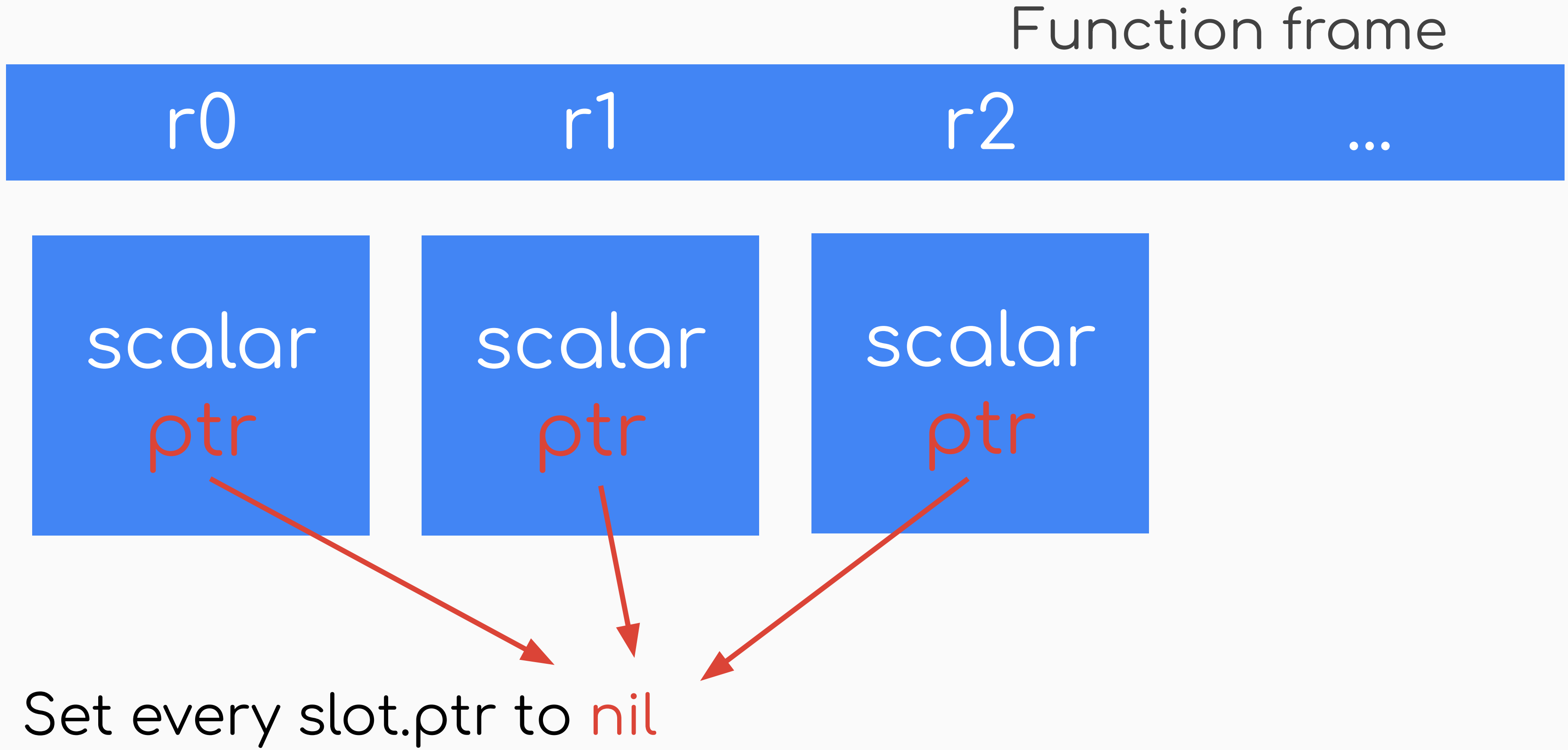
Function frame

r0 r1 r2 ...

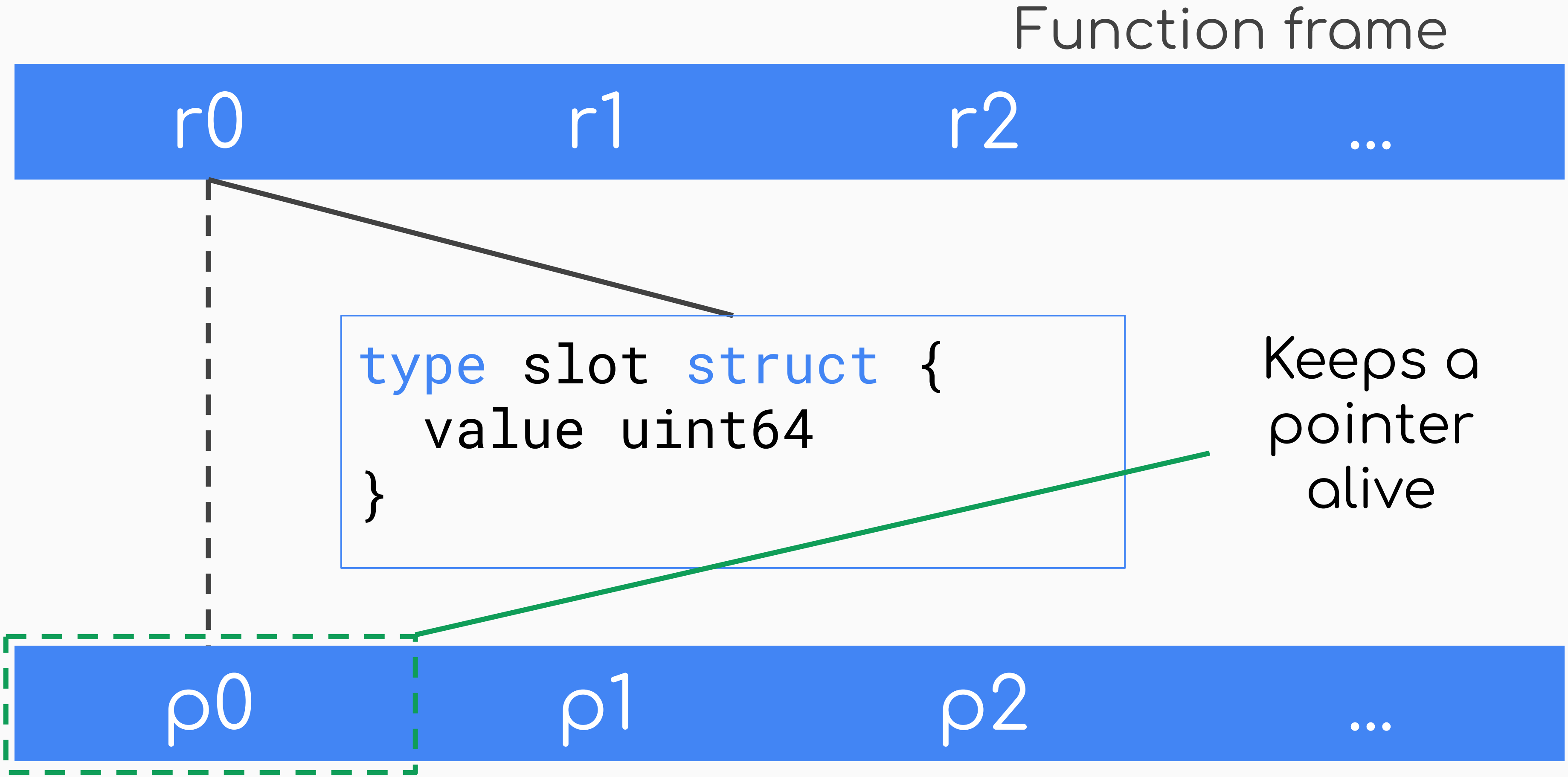
```go
type slot struct {
  scalar uint64
  ptr    *Object
}
```

Paired {scalar, ptr} slots are a safe fix

{uint64,pointer} slots



Memory reclaim



Uint64 + second frame for pointers (alt solution)

Part 4/7

Interop / FFI

Calling Java from Go

- Mark machine code buf as executable
- Call as func or do JMP in asm

Simple, boring.

Calling Go from Java

This is more involved (take a breath):

Calling Go from Java

This is more involved (take a breath):

- Obtain Go function address (simple)

Calling Go from Java

This is more involved (take a breath):

- Obtain Go function address (simple)
- Follow the Go calling convention (normal)

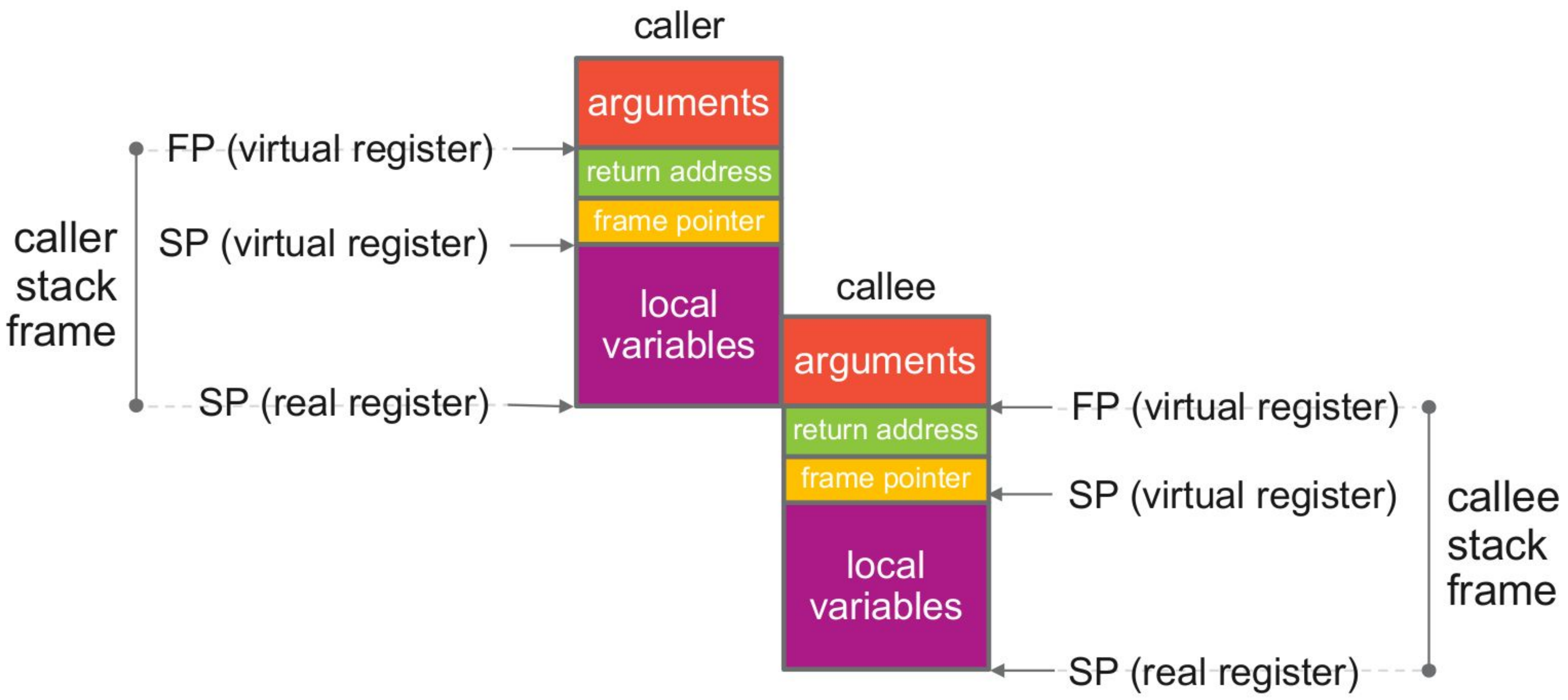
Calling Go from Java

This is more involved (take a breath):

- Obtain Go function address (simple)
- Follow the Go calling convention (normal)
- Deal with fatal error issues (hard)

How to get a Go function code address?

```go
func funcAddr(fn interface{}) uintptr {
    type eface struct {
        typ   uintptr
        value *uintptr
    }
    e := (*eface)(unsafe.Pointer(&fn))
    return *e.value
}
```



Go calling convention (source)

Assembling Java→Go call

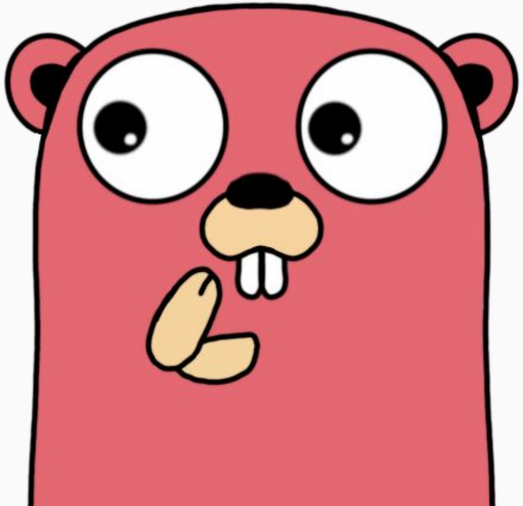
1. Push arguments to the stack
2. CALL $\boxed{\text{\$func\_addr}}$
3. Move results to local slots

Use funcAddr to get that

The exact actions depend on the current Go calling convention.

Let's try it!

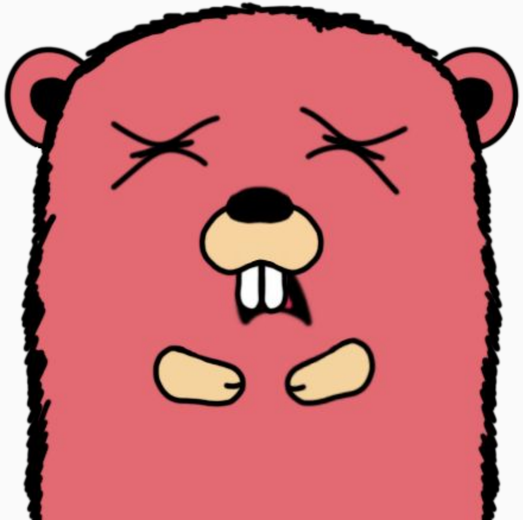
...

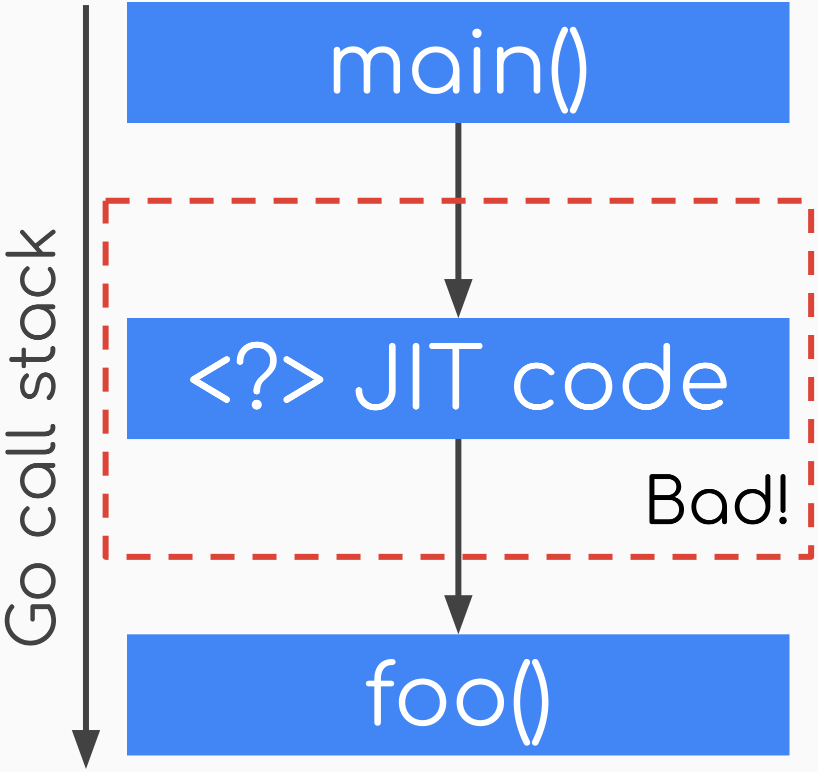


Go runtime is *not* impressed!

- "Unknown caller PC"
- "Unknown return PC"
- "Missing stackmap"

```
=== RUN   TestFoo
runtime: frame <censored> untyped locals 0xc00008ff38+0x8
fatal error: missing stackmap
```





Calling Go directly from the JIT'ed code

You've run into a really hairy area of asm code.

My first suggestion is not try to call from assembler into Go.

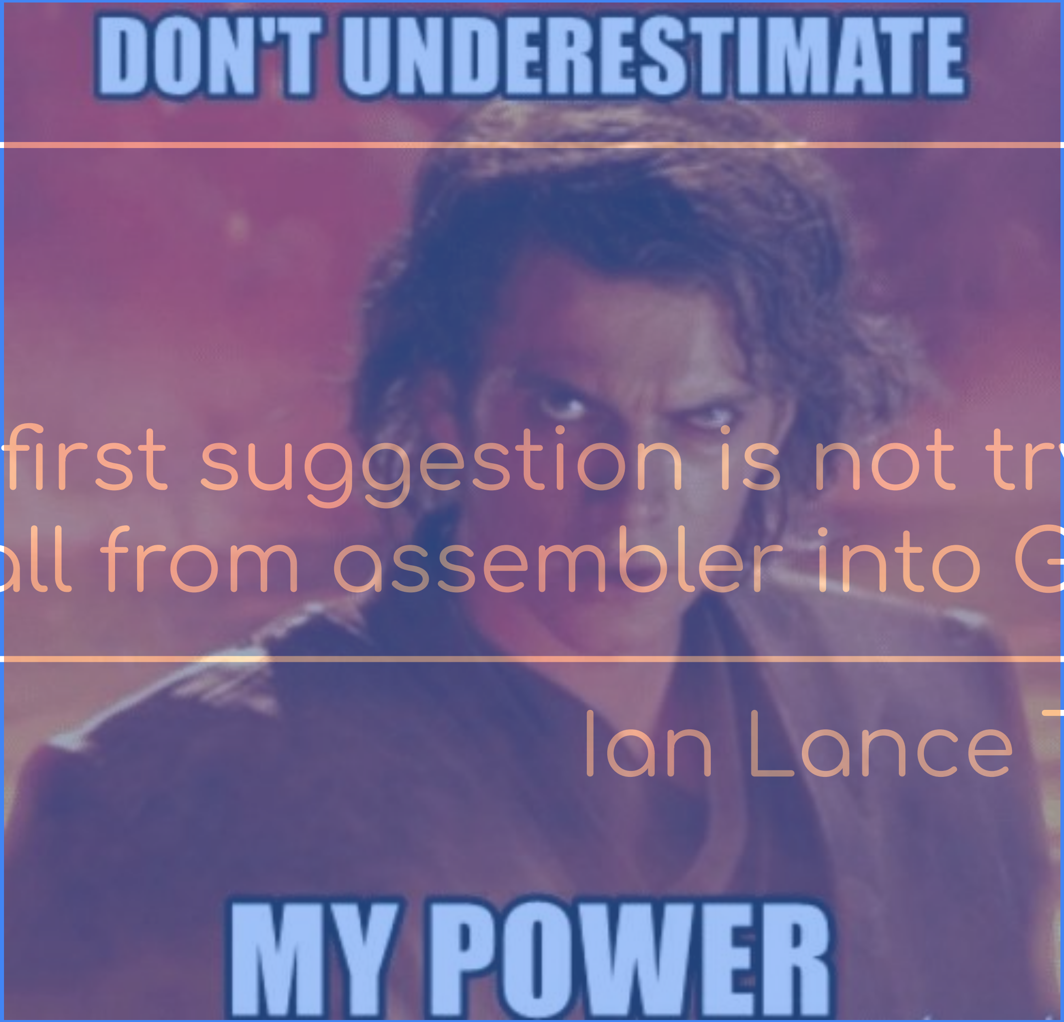
Ian Lance Taylor

Quote

My first suggestion is not try to call from assembler into Go.

Ian Lance Taylor

Quote



My first suggestion is not try to call from assembler into Go.

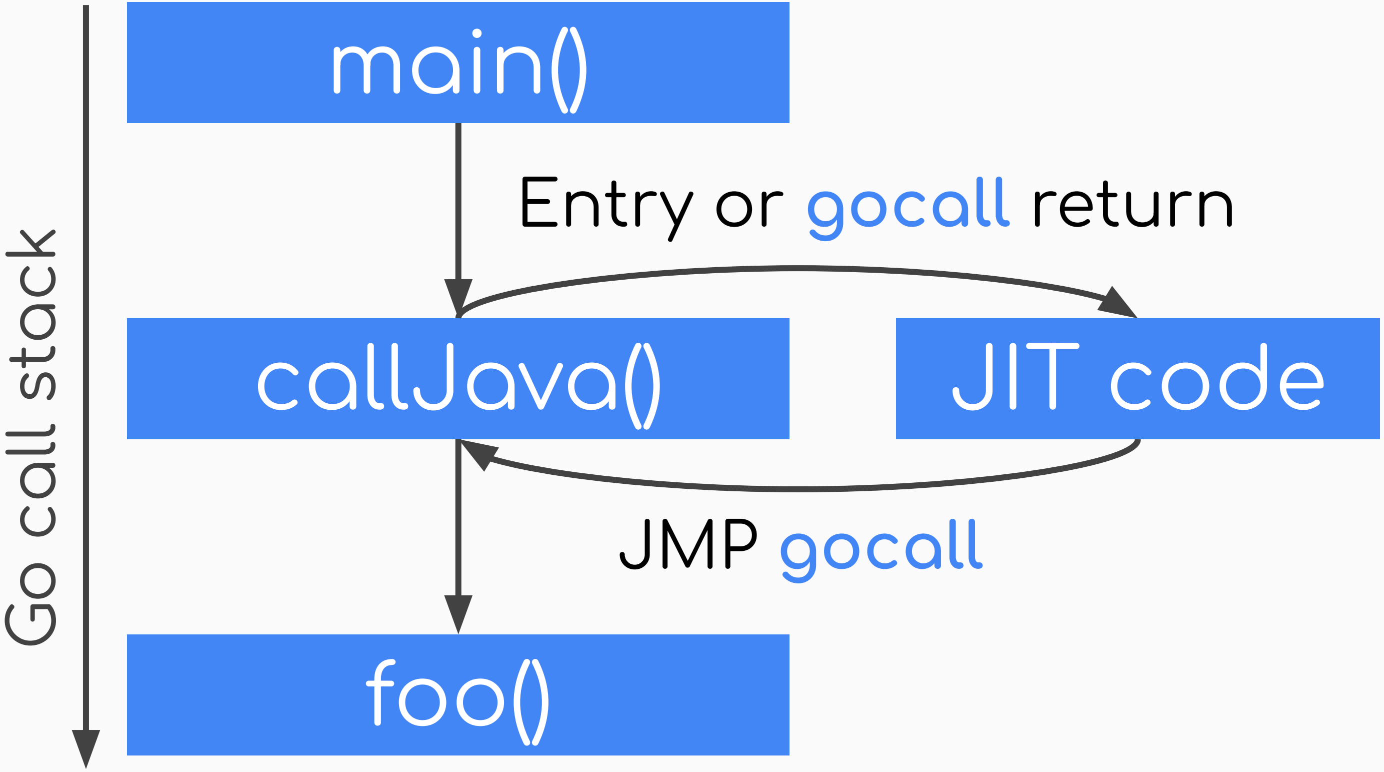
Ian Lance Taylor

Quote

How to fix these fatals?

Add a Go→Java calls proxy.
Java→Go calls via trampoline.

- Provides a stackmap for Java→Go calls
- Provides a known caller/return PC



Calling Go via proxy

Go→Java call proxy (simplified)

```
// callJava(e *Env, code *byte)
TEXT ·callJava(SB), 0, $96-16
	NO_LOCAL_POINTERS
	JMP code+8(FP)
	RET
gocall:
	CALL CX
	JMP -8(BP)
```

Go→Java call proxy (simplified)

```
// callJava(e *Env, code *byte)
TEXT ·callJava(SB), 0, $96-16
    NO_LOCAL_POINTERS
    JMP code+8(FP)
    RET
gocall:
    CALL CX
    JMP -8(BP)
```

Stackmap fix

NO_LOCAL_POINTERS macro

It's safe for us, as long as:

- We never rely on Go stack values address
- Our heap values are reachable elsewhere

Go→Java call proxy (simplified)

```
// callJava(e *Env, code *byte)
TEXT ·callJava(SB), 0, $96-16
     NO_LOCAL_POINTERS
     JMP code+8(FP)
     RET
gocall:
     CALL CX
     JMP -8(BP)
```

Caller PC fix

Go→Java call proxy (fixing return PC)

```
// callJava(e *Env, code *byte)
TEXT ·callJava(SB), 0, $96-16
	NO_LOCAL_POINTERS
	MOVQ code+8(FP), CX
	JCALL(CX)
	RET
gocall:
	CALL CX
	JMP -8(BP)
```

Return PC fix

Go→Java call proxy (fixing return PC)

```
// callJava(e *Env, code *byte)
TEXT ·callJava(SB), 0, $96-16
    NO_LOCAL_POINTERS
    MOVQ code+8(FP), CX
    JCALL(CX)
    RET
gocall:
    CALL CX
    JMP -8(BP)
```

Saves following RET inst addr and Jumps to CX (see next slide)

JCALL macro

```
// Encoding `lea rax, [rip+N]` with BYTE
// since Go has no real RIP-relative
// addressing mode.
#define JCALL(fnreg) \
    BYTE $0x48; … 8d0509000000 \ // Lea
    MOVQ AX, (SI) \ // Store RET addr
    ADDQ $16, SI \  // Move to next slot
    JMP fnreg       // Run JIT code
```

Java native methods

```java
// In Java file:
public class Foo {
  public static native void printInt(int x);
}
```

Java native methods

```
// In Java file:
public class Foo {
  public static native void printInt(int x);
}
// In Go file:
func fooPrintInt(x int32) {
    fmt.Println(x)
}
```

Java native methods

```
// In Java file:
public class Foo {
  public static native void printInt(int x);
}
// In Go file:
func fooPrintInt(x int32) {
    fmt.Println(x)
}
// Before loading Foo class:
vm.Bind("Foo.printInt", fooPrintInt)
```

Why do we need fast Java→Go?

If calls to Go are fast, we can:

- Implement runtime funcs as Go funcs
- Re-use Go code easily in out Java code

Part 5/7

Object layout and memory allocation

Foo class

```
public class Foo {
  public int x;   // scalar 1
  public int y;   // scalar 2
  public Bar bar; // pointer field
}
```

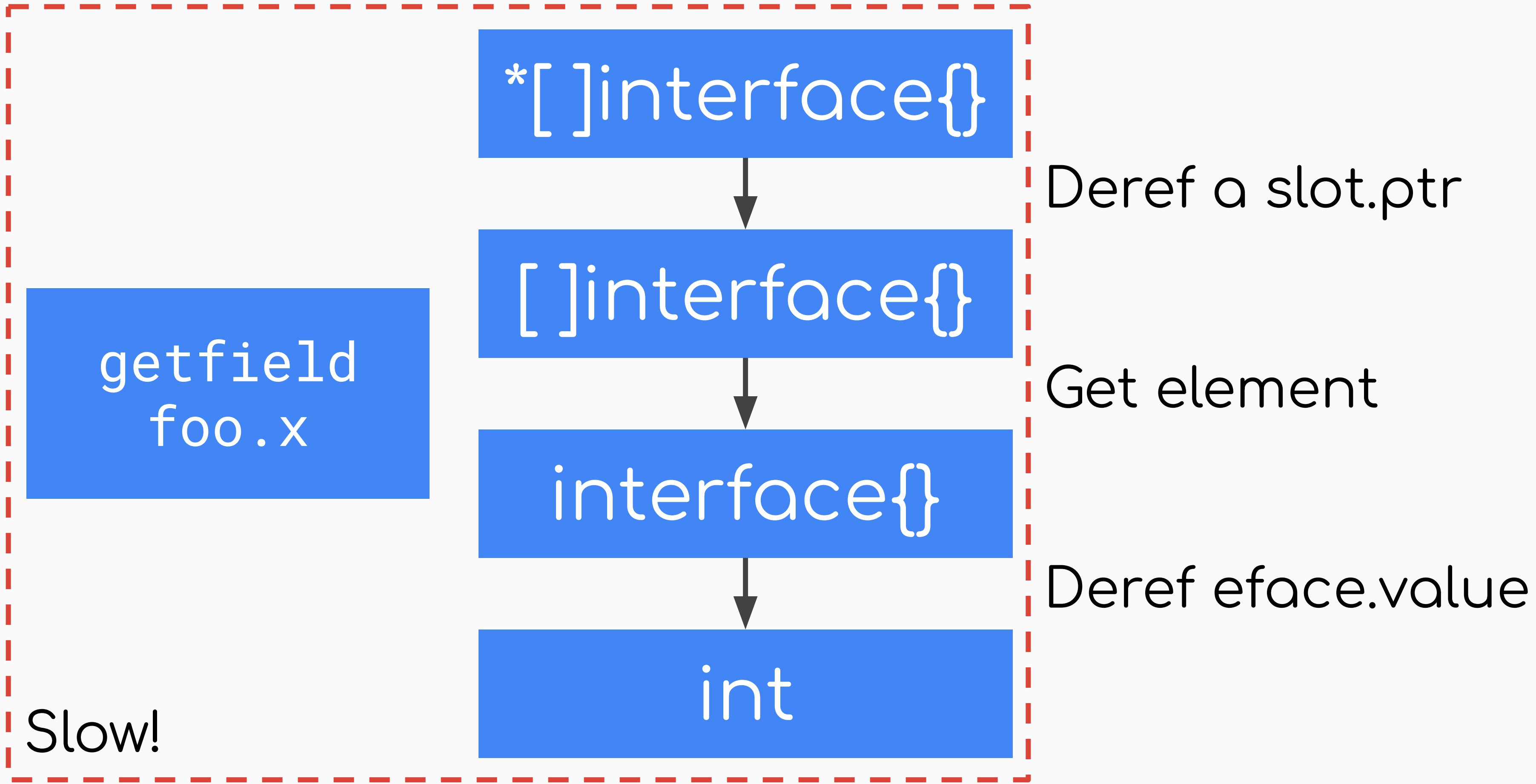
Foo class

```
public class Foo {
  public int x;   // scalar 1
  public int y;   // scalar 2
  public Bar bar; // pointer field
}
type Foo struct {
  X   int32
  Y   int32
  Bar *Bar
}
```

Perfect, but impossible

Foo class values (naive version)

```
// Object is a slice of interface{} fields.
// Pointer slot gets a slice pointer.

foo = []interface{}{x, y, bar}
slot.ptr = &foo
```



Read x:int field from Foo

Proposed object layout

```
type Object struct {
  Class   *ClassInfo
  Ptrdata **Object
}

type Object64 struct {
  Object
  Data [8]byte
}
```

Proposed object layout

```
type Object struct {
  Class   *ClassInfo
  Ptrdata **Object
}

type Object64 struct {
  Object
  Data [8]byte
}
```

Common object header

Proposed object layout

```
type Object struct {
  Class   *ClassInfo
  Ptrdata **Object
}

type Object64 struct {
  Object
  Data [8]byte
}
```

All object pointer fields are stored here

Proposed object layout

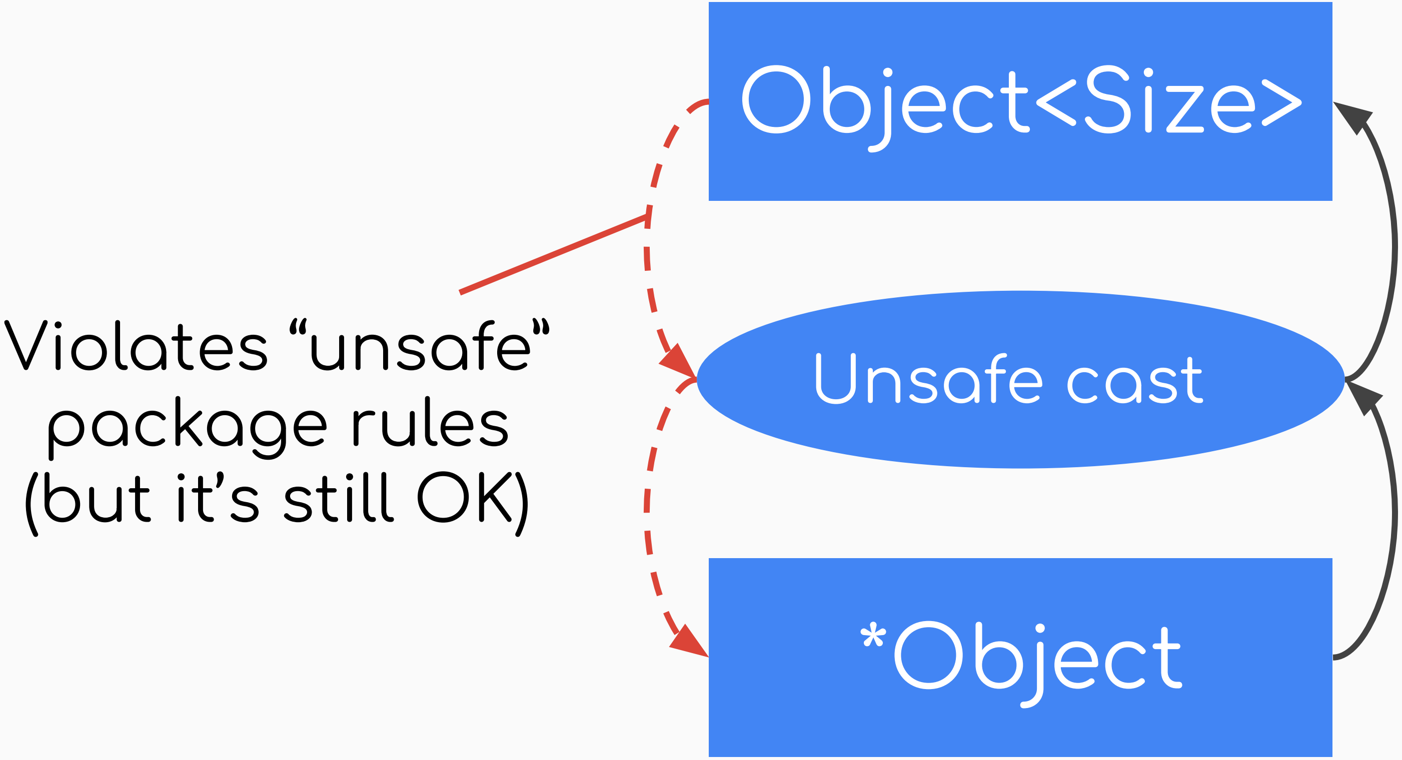
```
type Object struct {
  Class   *ClassInfo
  Ptrdata **Object
}

type Object64 struct {
  Object
  Data [8]byte
}
```

Object with
8-byte storage
for scalar fields,
Object<64>

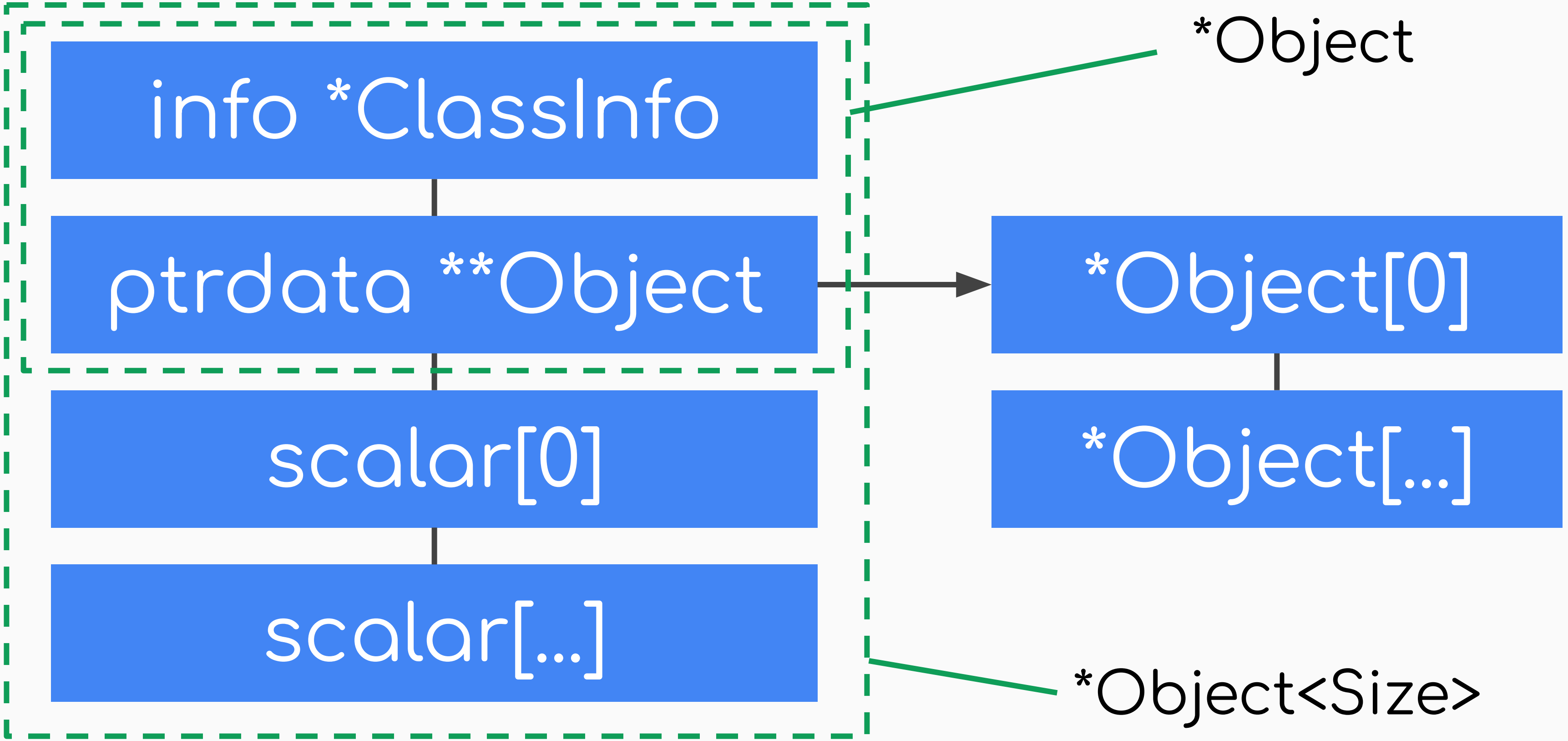
Proposed object layout

```
type Object struct {
  Class   *ClassInfo
  Ptrdata **Object
}

type Object64 struct {
  Object
  Data [8]byte
}
```

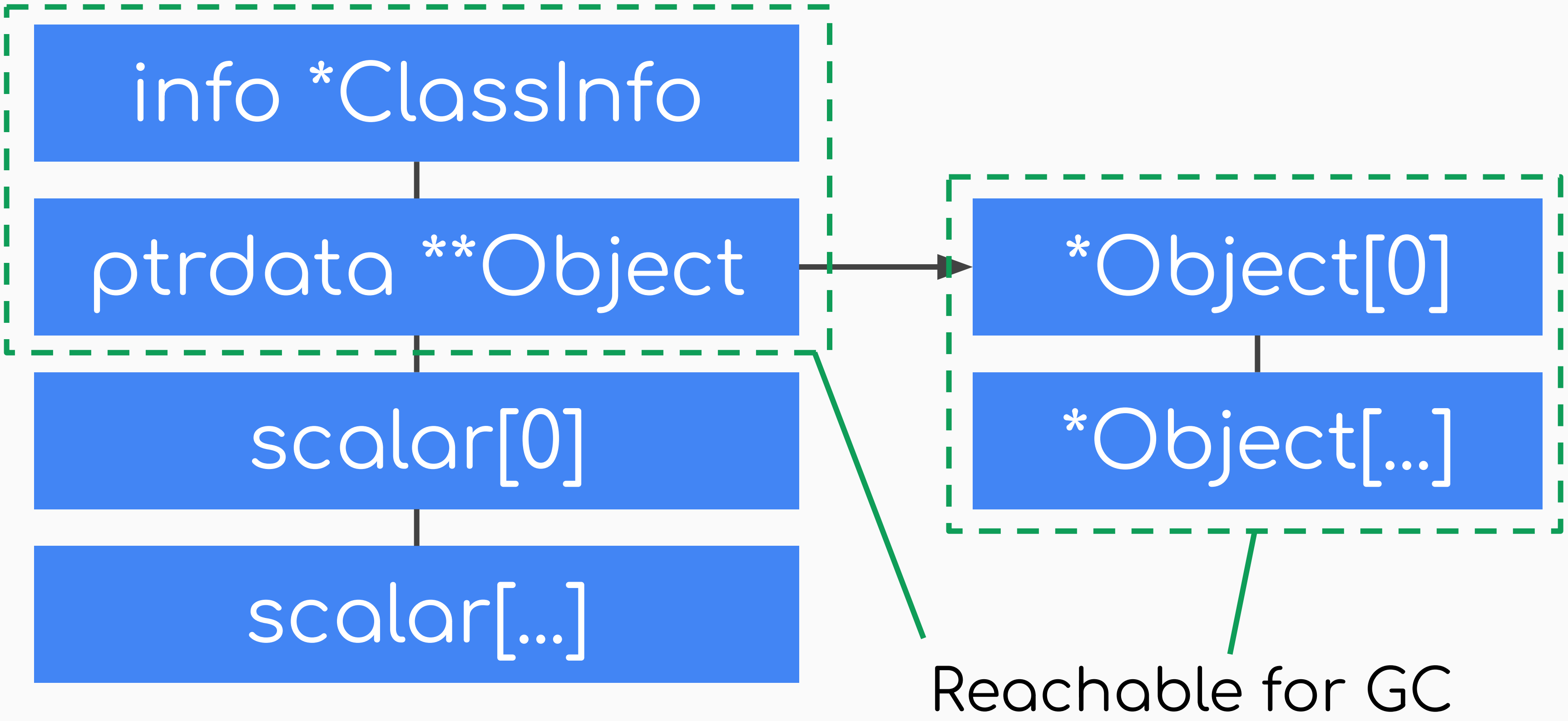
X and Y fields can be stored here



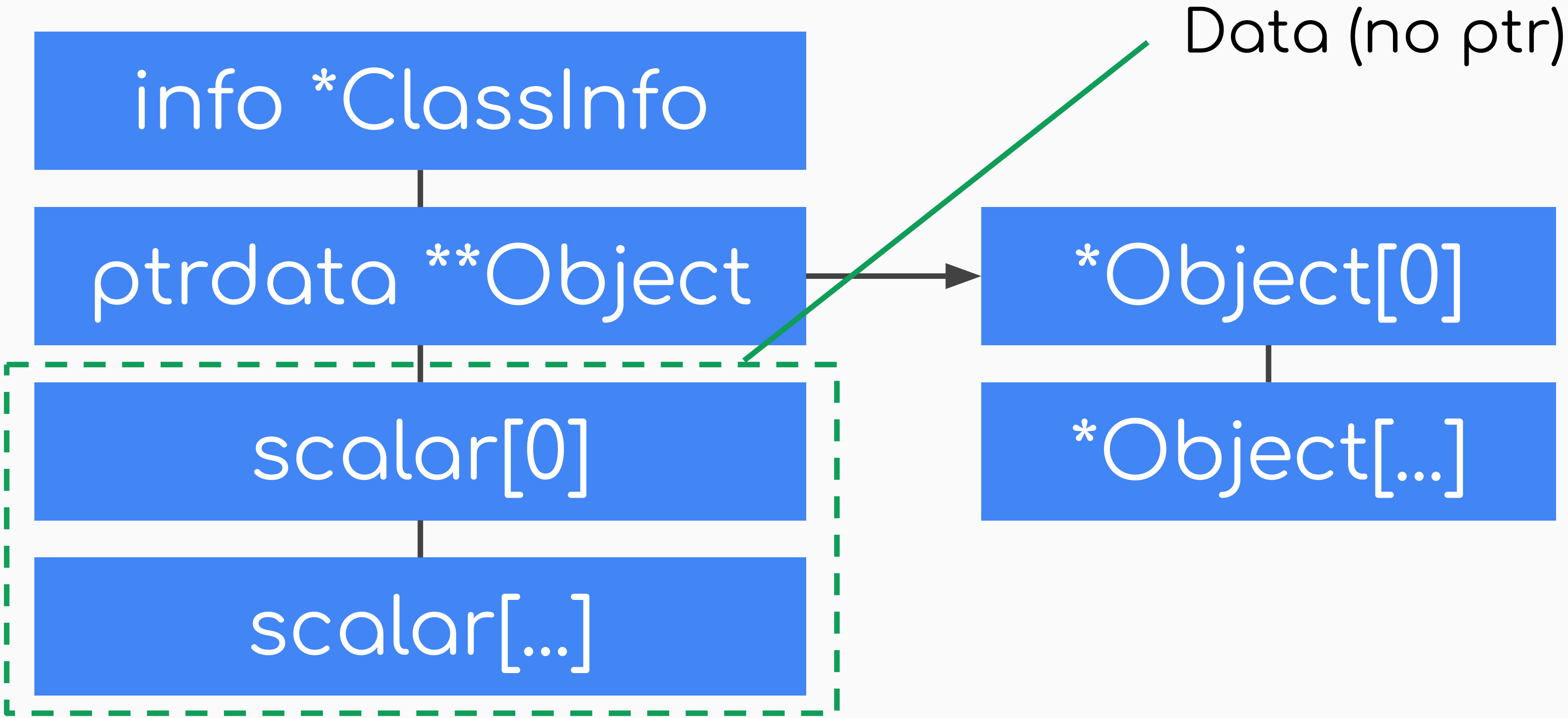
Conversion between Object and Object<Size>



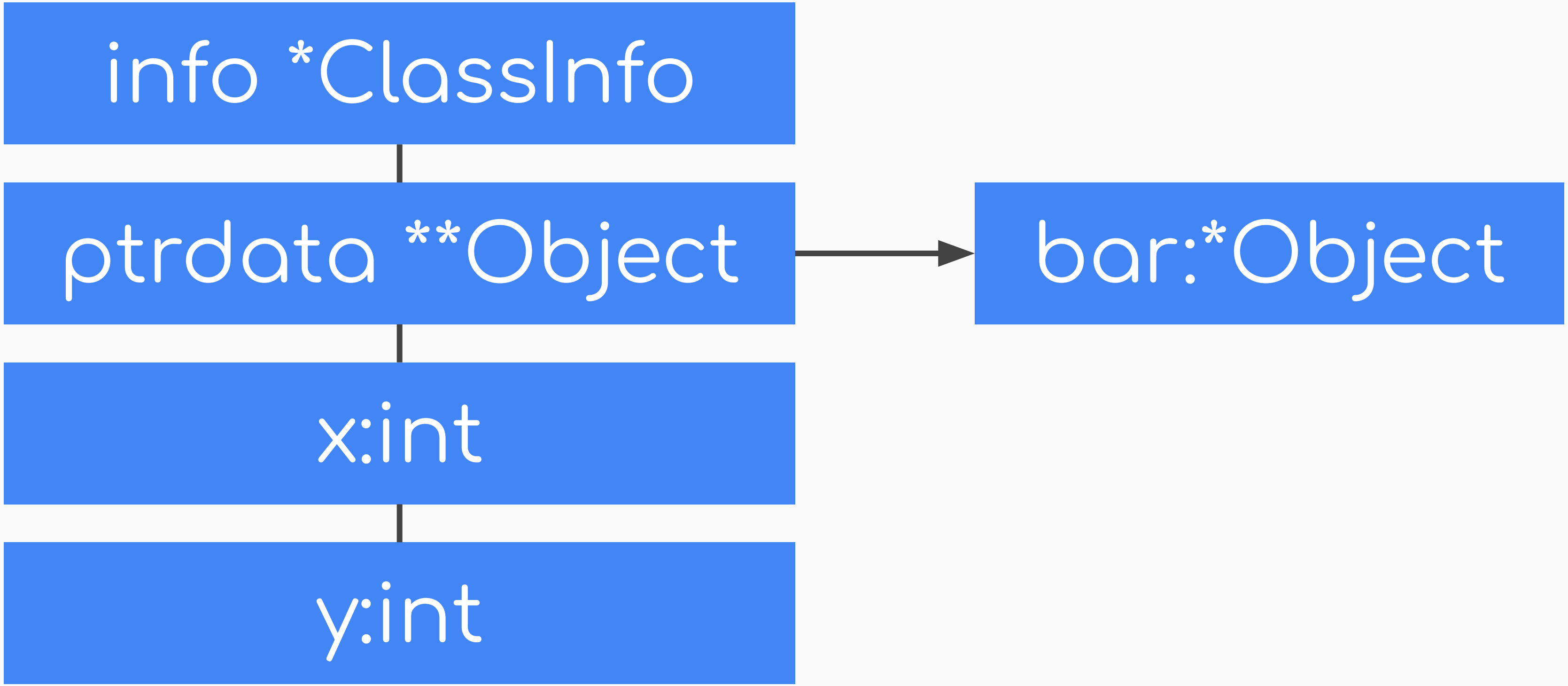
Abstract Object<Size> layout



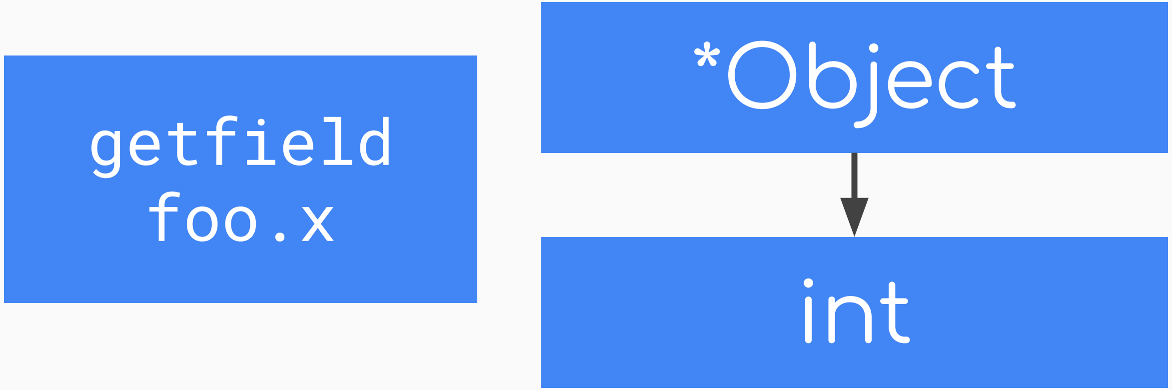
Abstract Object<Size> layout



Abstract Object<Size> layout

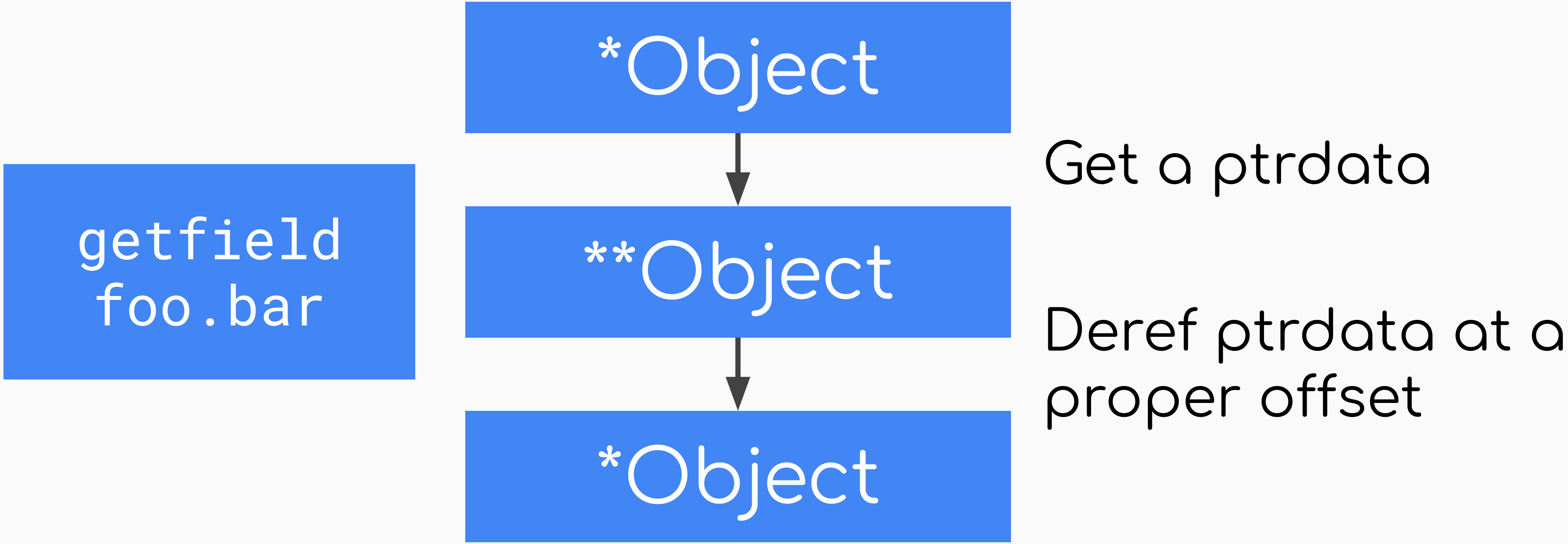


Foo layout in memory



Deref a slot.ptr
At a proper offset

Read x:int field from Foo



Read bar:Bar field from Foo

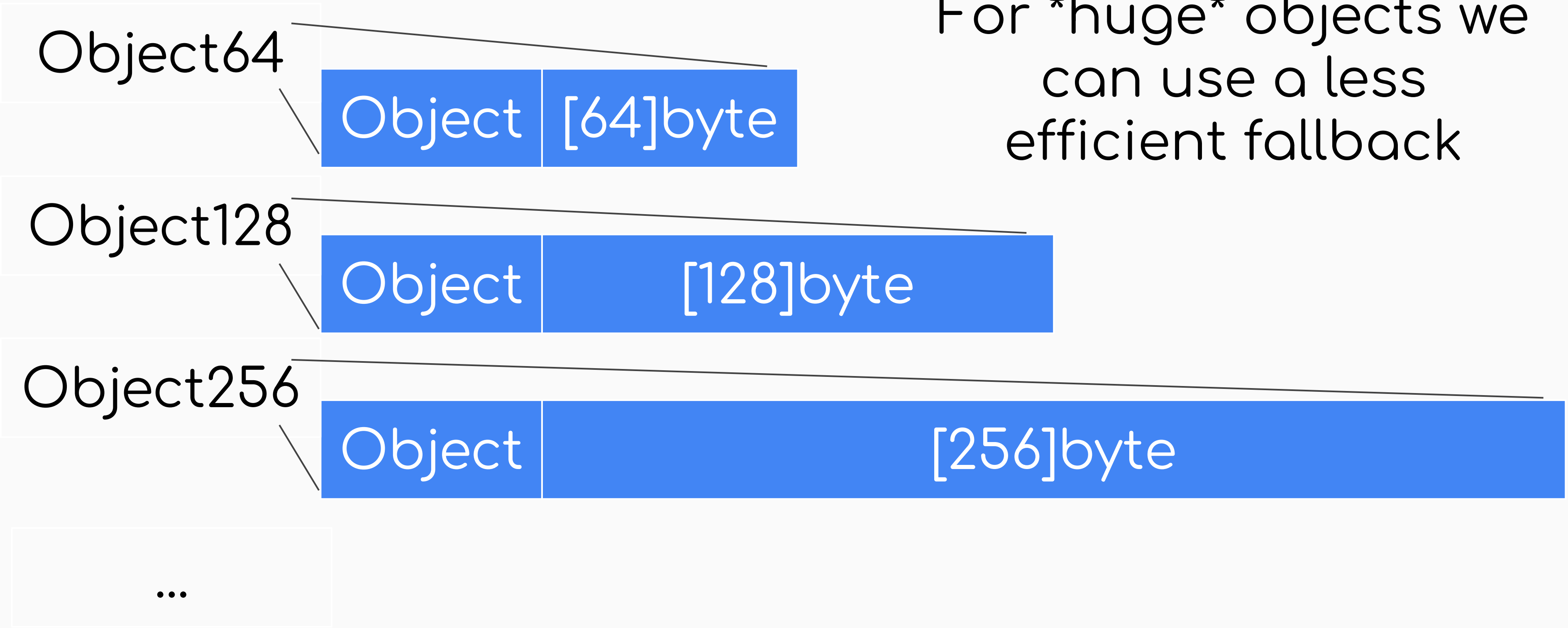
Can we use []byte allocations?

No, Go GC will not track any pointers that are stored inside that memory.

So, how to allocate?

- Choose the closest Object<Size>
- Allocate Object<Size>
- Return as *Object

May want to adjust sizes to the Go memory allocator size classes.



For *huge* objects we can use a less efficient fallback

How many Object<Size> types do we need?

Part 6/7

Challenges and limitations

Null pointer check / explicit

```
var p *int  // p is nil

println(*p)
```

Null pointer check / explicit

```
var p *int   // p is nil
nilcheck(p) // Inserted by a compiler
println(*p)
```

Null pointer check / explicit

```
var p *int  // p is nil
nilcheck(p) // Inserted by a compiler
println(*p)
```

Simple, but not very efficient

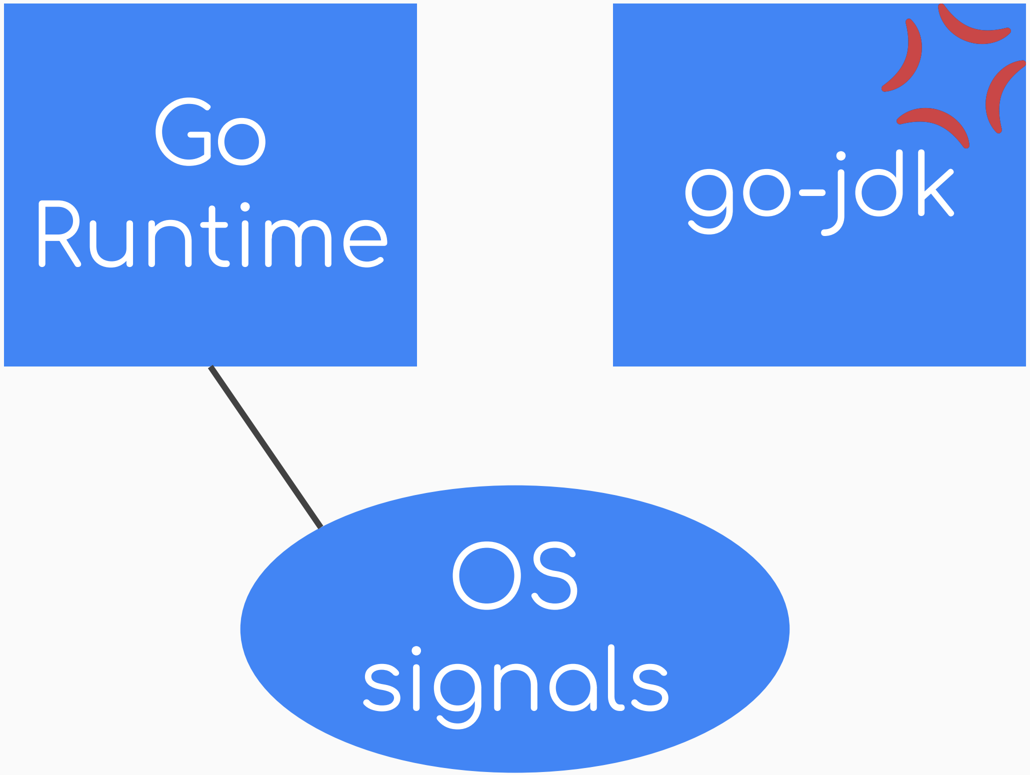
Null pointer check / signals

Hardware exceptions and interrupts

+

OS signals handling

More: https://stackoverflow.com/a/36955888/4017439



Remember this picture?

Limitation: bytecode patching

For some reasons, it's quite common in Java world to modify the bytecode that is being loaded...

Limitation: bytecode patching

For some reasons, it's quite common in Java world to modify the bytecode that is being loaded...

Since we convert bytecode into the machine code, we have a problem...

Challenge: method re-load

If method changes and we can’t fit its code into the old executable buffer, method address will change...

Challenge: method re-load

If method changes and we can’t fit its code into the old executable buffer, method address will change…

This requires re-linking all method callers.
If calls were inlined it’s even harder.

Part 7/7

Closing words

```
  0 - Backstory
  1 - go-jdk overview
  2 - Making the code run fast
  3 - GC-friendly slots
  4 - Interop / FFI
  5 - Object layout / mem alloc
  6 - Challenges & limitations
> 7 - Closing words
```

Testing

```
import testutil.T;

class Test {
  public static void run(int x) {
    T.println(x + 5);
  }
}
```

System.out.println in OpenJDK,
fmt.Println in go-jdk

N-body benchmark results

OpenJDK	3.9s
go-jdk	4.8s
OpenJDK (no JIT)	~11s
go-jdk (no JIT)	~22s

N-body benchmark results

OpenJDK	**3.9s**
go-jdk	**4.8s**
OpenJDK (no JIT)	~11s
go-jdk (no JIT)	~22s

N-body benchmark results

OpenJDK	3.9s
go-jdk	4.8s
OpenJDK (no JIT)	**~11s**
go-jdk (no JIT)	**~22s**

Resources

- [go-jdk repository]
- [VM Showdown: Stack Versus Registers]
- [Calling Go funcs from asm] (ru)
- [Go calling convention]
- [JNI bindings for Go]

Efficient VM with JIT in Go

quasilyte @ GoWayFest 4.0 (2020)