Go language highlights

Martin Steffen

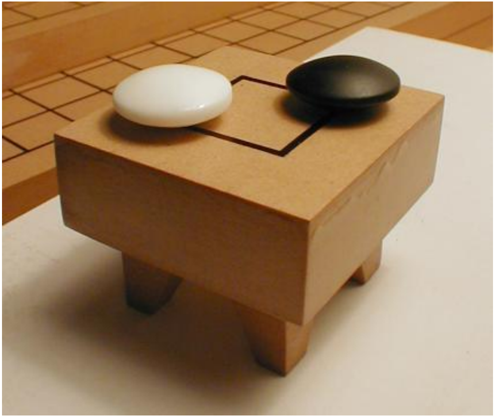
Nov 25, 2016

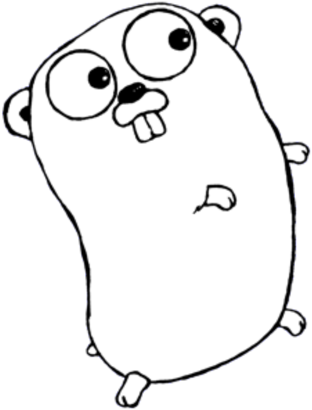
UNIVERSITAS OSLOENSIS • MDCCCXI •

Outline

1. Introduction
2. OO structuring & type system
3. Control
4. Concurrency
5. Memory model
6. Conclusion

Introduction





Go sales pitch

- “language for the 21st century”
- relatively new language (with some not so new features?)
- a lot of *fanfare* & backed by Google no less
- existing show-case applications
    - docker
    - dropbox . . .

Go's stated design principles

- appealing to C programmers
- KISS: "keep it simple, stupid"
- built-in concurrency
- "strongly typed"
- efficient
- fast *compilation*, appealing for scripting

History of Go

- first plans around 2007
- “IPO”: end of 2009
- Precursor languages, resp. inspired by:
    - C
    - CSP / Occam
    - At Bell Labs
        - Squeak, Newsqueak
        - Limbo
        - Alef
    - Erlang, Concurrent ML

Go's non-revolutionary feature mix

- imperative
- object-oriented (?)
- compiled
- concurrent (goroutines)
- "strongishly" typed
- garbage collected
- portable
- higher-order functions and closures

OO structuring & type system

(Sub)-typing, OO, polymorphism, and all that

"In object-oriented programming, the is-a relationship is totally based on inheritance"
– from some random Java tutorial

"overriding is dynamic polymorphism"
– from the blogosphere (stack exchange)

"Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.
– Oracle's Java tutorial, section on polymorphism

“Orthodox” view

- class = type (among other things)
- inheritance = subtyping
- polymorphism = subtype polymorphism (= subtyping = inheritance)

“Orthodox”

- *accepted as true or correct by most people: supporting or believing what most people think is true*
- *accepting and closely following the traditional beliefs and customs of a religion*

```
class Point {
    public int x;
    public Point (int x) {this.x = x;}
}


public class Classroles {
    public static void main(String[] arg) {
        Point x;                        // declaration of x
        x = new Point(5);               // setting x
    }
}
```

Perhaps OO it is, but not as you know it

Go's *heterodox* take on OO

- *no* classes
- not even objects, officially
- no (class) inheritance
- interfaces as types[a]
- code reuse encouraged by
  - embedding
  - aggregation (ok, that one is old hat)
- name of an interface type $\neq$ interface type itself

---

[a] We concentrate here on the "OO" part of Go's type system, i.e., the interfaces. There are other types too, obviously.

Ducks

"We have testimony that you walk like a duck and you quack like a duck. Tell the court—are you a duck?"

No ducks in Java (as in most mainstream OO)

```
interface I1 { int m (int x) ; }
interface I2 { int m (int x); }
class C1 implements I1 {
    public int m(int y) {return y++;  }
}
class C2 implements I2 {
    public int m(int y) {return y++;  }
}

public class Noduck1 {
    public static void main(String[] arg) {
        I1 x1 = new C1();        // I2 not possible
        I2 x2 = new C2();
        x1 = x2;
    }
}
```

I kind of knew that, but what about this?

```
interface I1 { int m (int x) ; }
interface I2 { int m (int x); }
class C1 implements I1 {
    public int m(int y) {return y++;  }
}
class C2 implements I2 {
    public int m(int y) {return y++;  }
}

public class Noduck2 {
    public static void f(I2 x) { return ;}

    public static void main(String[] arg) {
        I1 x1 = new C1(); // I2 not possible
        I2 x2 = new C2();
        x1 = (I1)x2;        // <--- I'll teach you!!!
        x1.m(1);            // both vars support m, right?
    }
}
```

Duck typing

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*

- be careful with Wikipedia's wisdom (or the internet in general)
- Old controversy:

  nominal (or nominative) vs. structual (sub-)typing
- Go: "*static* duck typing"

What's a type?

Well, depends on whom you ask:

- compiler & run-time system?
    - a hint for the compiler of memory usage & representation layout?
    - piece of meta-data about a chunk of memory
- semanticist?
    - what's the *meaning* of a type?
- programmer?
    - types make my programs more safe, it's a partial specification
    - type systems stand in the way of my expert mastering of code
- orthodoxion oo'er?
    - a type is more or less a class (at least the more interesting ones/custom types)

Union types in C

```
union { int a; float b; }
```

> *"Unions provides a way to manipulate different kinds of data in a single area of memory..."*

More grown-up view on types and type systems

- types are abstractions of *"values"* (data items)
- types are "sets"?
- of course: " *memory layout* " view
  - still relevant (for the compiler)
  - only: hidden from the programmer (*abstraction*!)
- cf. abstract data types

What is a datum?

- $\mathbb{N} = \{0, 1, 2 \ldots\}$
- $\mathbb{N} = \{I, II, III, IV, V \ldots\}$
- `Int = 000000000, 00000001, ....`

How can I use a datum?

- How do I get me (new) values?
- How do I (safely) *compute* with them?
- E.g. $+, -, \ldots$ on $\mathbb{N}$

Type systems

- important part of the "static analysis phase" of a compiler
- static vs. dynamic
- decidable typing (when statically typed)
- "Strong" typing

Milner's dictum

" well-typed programs cannot go wrong ".[a]

---

[a] That phrase corresponds to "safe" typing or type safety, not "strong" typing.

- balancing flexibility, safety, notational overhead, etc
- polymorphism

How to *implement* an interface with an *object*?

- interfaces contain *methods* (but not fields)

At the end of the day: What's an "object" anyhow?

data + control + identity

And how to get one, implementing an interface?

Java ...

1. Interface: given
2. name a class which `implements I`
3. "fill" in data (fields)
4. fill in code (methods)
5. instantiate the class

Go

1. Interface: given
2. —
3. choose data (state)
4. bind methods
5. get yourself a data value

What are methods?

- procedures – functions – methods
- the most fundamental (control) *abstraction* in virtually all prog. languages
- Go: methods are “specific” functions

method ∼ function with special first argument

$$f(o, v) \text{ vs. } o.f(v)$$

- elsewhere often: special keyword for first argument: `this` (or `self`)

Methods & functions

```
type Number struct { n int }

func add1 (x Number, y Number) int {return x.n + y.n}
```

Methods & functions

```
func (x      Number) add2 (y int) int {return x.n      + y}
```

Methods & functions

```
func (self Number) add3 (y int) int {return self.n + y}
```

Methods & functions

```
func add4 (x int) (func (int) int)  {
        return func  (y int) (int) { return y+x }
}
```

Methods & functions

```
func main() {
        x1 := 42
        x2 := 1729
        n1 := Number{42}
        n2 := Number{n:1729}

        fmt.Printf("function:␣%v\n", add1(n1,n2))
        fmt.Printf("method1:␣␣%v\n", n1.add2(x2))
        fmt.Printf("method2:␣␣%v\n", n1.add3(x2))
        fmt.Printf("method2:␣␣%v\n", add4(x1)(x2))
        fmt.Printf("???␣␣␣␣:␣␣%v\n", add4(x1))

}
```

Binding methods to a type (from `bufio`)

```
type Writer struct {
        // contains filtered or unexported fields
}
type Reader struct {
        // contains filtered or unexported fields
}

func (b *Writer) Write(p []byte) (int, error) { return 1,1 }
```

Code reuse and inheritance

- different flavors
  - prototype-based inheritance
  - class inheritance
    - single
    - multiple
- inheritance $\neq$ subtyping (even if classes were types)
- other forms of “reuse” or structuring code (in the OO world)
  - traits
  - mixins
- often: *inheritance* vs. *composition* (aggregation)
  - class inheritance persistently criticised but persistent orthodox gold-standard of code reuse
  - inheritance anomaly

Design patterns

- “elements of *reusable* oo software”, or
- 99 sly ways to exploit inheritance and interfaces to arrange code in fancy ways not really supported by plain inheritance

Embedding and aggregation (in a struct)

```
type ColoredPoint struct {
        color.Color // anonymous field (embedding)
                    // Color: interface
        x, y  int   // named field (aggregation)
```

- AKA *delegation* elsewhere (but be careful of terminology)
- anonymous field

Embedding (in an interface)

```
type I1 interface  {
        ying ()
}
type I2 interface  {
        yang ()
}
type I12 interface {
        I1
        I2
}
type I interface {
        ying ()
        I2            // embedd I2
}
func f (o I)    {     // same for I12
        o.ying()
        o.yang()
```

Embedding (in an interface) & duck typing

```
func f12 (o I12)    {   // same for I
        o.ying()
        o.yang()
}

func f1 (o I1 ) { o.ying()}
type O struct {}  // ``so far'' empty
func (o O) ying ()  {}
func (o O) yang ()  {}

func main() {
        o := O {}  // literal struct
        o.ying()
        f(o)       // o of type I
        f1(o)      // I < I1
        f12(o)     // o  of type I12
}
```

Overloading vs. overriding, late-binding vs. dynamic dispatch

- explanation often “Java-centric”
- static vs. dynamic resolution?
- late-binding and dynamic dispatch: In Java etc, basically synonymous
- most OO languages (Java . . . , Go): *single-dispatch*
- multiple-dispatch “OO” language: CLOS
- dynamic dispatch vs. overloading:
  - partly a matter of perspective (esp. for methods):

Late binding . . .

- objects “host” or “contain” methods,
- method is invoked “on an object”
- o’s run-time type (class)
- “ $o.m(v)$ ”

Overloading

- method special kind of function
- method = function with special first argument
- “$m(o, v)$ ”

No method overloading?

```
type I interface {
        ying (bool)
        ying (int)     // nope
}
```

Two “functions” with the same name `X` (overloading)?

```
type cartesianPoint struct{
        x, y float64
}
type polarPoint struct {
        r, theta float64
}

func (p cartesianPoint) X() float64 {return p.x }
func (p cartesianPoint) Y() float64 {return p.y }
func (p polarPoint) X() float64 {
        return p.r*math.Cos(p.theta)
}
func (p polarPoint) Y() float64 {
        return p.r*math.Sin(p.theta)
}
```

```
type Point interface{
        Printer
        X() float64
        Y() float64
}
```

Function with hand-made dynamic dispatch

```
type IPoint interface{}
type polarPoint     struct { r, theta float64 }
type cartesianPoint struct { x, y    float64  }

func X (p IPoint)  float64{
        switch t := p.(type) {// special type assertion + switch
        case cartesianPoint: return t.x
        case polarPoint:  return t.r*math.Cos(t.theta)
        default: return 1
        }
```

Embedding and duck typing, what's the big deal?

So far

- *embedding* in interfaces: "short hand notation"
- *embedding* in structs: "anonymous fields"

Go's take on code reuse

Composition/aggregation + combination of the mentioned concepts:

- interface type embedding
- struct type embedding (anon. fields)
- dynamically dispatched methods

Interfaces from package io

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

cf. earlier: structs `Reader` and `Writer` from `bufio`

ReadWriter struct with explicit fields

```
type Writer struct {
        // contains filtered or unexported fields
}
type Reader struct {
        // contains filtered or unexported fields
}

func (b *Writer) Write(p []byte) (int, error) { return 1,1 }
func (b *Reader) Read(p []byte) (int, error) { return 1,1 }


type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

How to call `Read` on a `ReadWriter`

- subtype polymorphism and subsumption
- In principle:

static duck typing

- a record of type `Writer` implements *interface* `io.Writer`
- = "supports" method `Write`
- analogous for `Reader`, interface `io.Reader` and method `Read`
- record of type `ReadWriter` *supports* both methods indirectly

“Solution” (?): boilerplate wrapper code

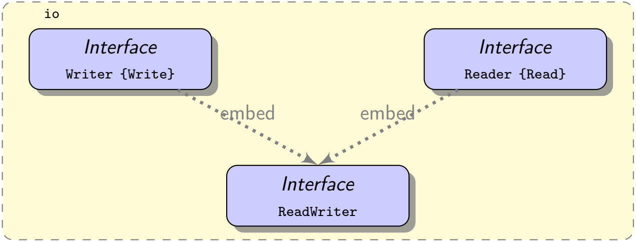
```
}

func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}

func (rw *ReadWriter) Write(p []byte) (n int, err error) {
    return rw.writer.Write(p)
}
```

- Hurrah, `ReadWriter`-structs implement `io.ReadWriter`

ReadWriter


io

Interface

`Writer {Write}`

Interface

`Reader {Read}`

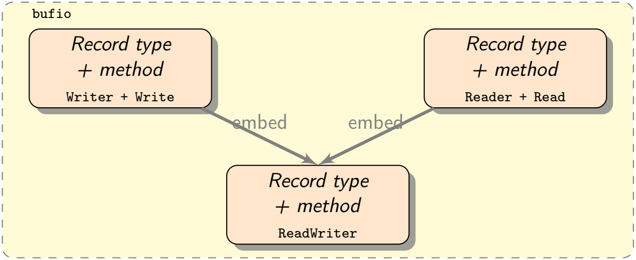
embed

embed

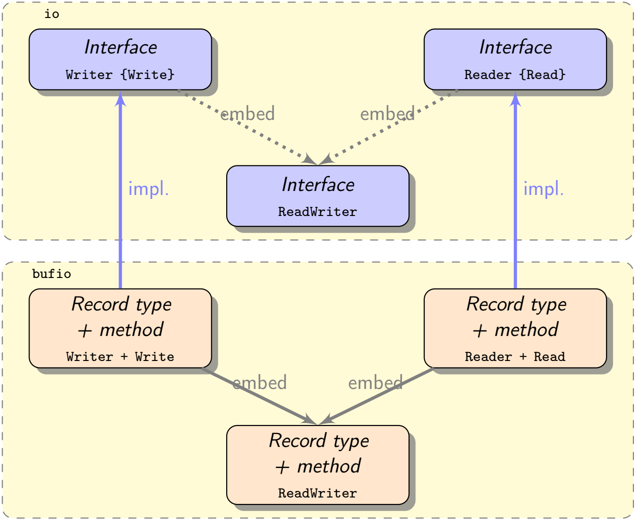
Interface

`ReadWriter`

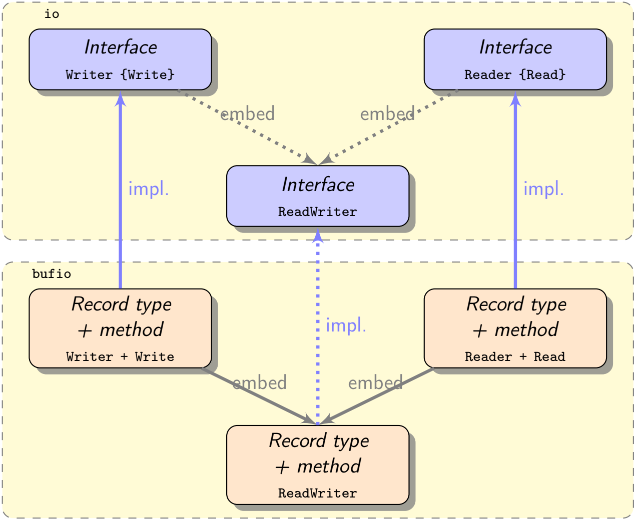
ReadWriter


bufio
Record type
+ method
Writer + Write
Record type
+ method
Reader + Read
embed
embed
Record type
+ method
ReadWriter

ReadWriter


io
Interface
Writer {Write}
Interface
Reader {Read}
embed
embed
impl.
Interface
ReadWriter
impl.
bufio
Record type
+ method
Writer + Write
Record type
+ method
Reader + Read
embed
embed
Record type
+ method
ReadWriter

ReadWriter


io
Interface
Writer {Write}
Interface
Reader {Read}
embed
embed
impl.
Interface
ReadWriter
impl.
bufio
Record type
+ method
Writer + Write
impl.
Record type
+ method
Reader + Read
embed
embed
Record type
+ method
ReadWriter

Embedding

```
type Writer struct {
        // contains filtered or unexported fields
}
type Reader struct {
        // contains filtered or unexported fields
}

// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}

func (b *Writer) Write(p []byte) (int, error) { return 1,1 }
func (b *Reader) Read(p []byte) (int, error) { return 1,1 }
```

Control

Stack-allocated local variables

Memory layout for a program

- code segment
- data
  - static
  - dynamic
    - stack
    - heap
- recursive procedures/functions => stack allocated, or?

Higher-order functions

- known from functional languages
- non-higher-order functions:
  - function takes data and returns data
  - what's data? everything but not functions
- languages with higher-order functions
  - functions as "first-class" data ⇒
  - functions as
    - arguments *and*
    - return values *and*
    - locally definable

```
func add4 (x int) (func (int) int)  {
        return func  (y int) (int) { return y+x }
}
```

$$add_4 : \text{int} \to (\text{int} \to \text{int}) = \lambda x{:}\text{int}.\lambda y{:}\text{int}.\ x + y$$

Lifetime of variables

- function-local variables: “live” (traditionally) in a *stack-frame*
  - call = allocate / “push” a stack frame
  - return = deallocate / “pop” a stack frame
- ⇒ lifetime of local vars = lifetime of “function body incarnation” (= stack frame)

```
var f = func () (func (int) int)  {// int −> int
        var x = 42                  // local variable
        var g = func (y int) int {  // nested function
                return x + 1        // ‘‘non−local’’
        }
        return g                    // function as return value
}
```

Closure

- “construct” of the run-time environment (just like stack-frames)
- *heap*-allocated!
- needed for languages with both
    - full higher-order functions
    - static binding (lexical binding)

- “classic” Lisp (and Emacs Lisp): dynamic binding, Scheme: static (= correct) binding
- *all* modern ho languages have closures

Closure

function + bindings for “non-local” variables

Imperative closures

```
var f = func () (func (int) int)  { // unit -> (int -> int)
        var x = 40                  // local variable
        var g = func (y int) int { // nested function
                return x + 1
        }
        x = x+1                     // update x
        return g                    // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf("␣r␣=␣%v", r)
}
```

Why not simply pass the “hidden” argument *officially*?

- λ -lifting a closure

```
var f = func () (func (int) int)  {
        var x = 40
        var g = (func (x int) (func (int) int) {
                var fr = func (y int) int {
                        return x + 1
                }
                return fr
        }) (x)                  // offically feeding in x
        x = x+1
        return g
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf("␣r␣=␣%v", r)
}
```

But how actually to pass it?

```
var f = func () (func (int) int)  {
        var x = 40                 //
        var g = (func (x *int) (func (int) int) { // call by refer
                return (func (y int) int {
                        return *x + 1
                })
        }) (&x)                          // feeding in address of
        x = x+1
        return g
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf("␣r␣=␣%v", r)
}
```

Call-by-reference and call-by-value

- for immutable data: no difference

By-value

```
func (x int) bool { .... }
```

By-reference

```
func (x *int) bool { .... }
```

Closures in Go

non-local variables are passed by reference

Non-local control flow

- different constructs, like
  - `goto`
  - `break` and `continue`
- Go frowns up using “exceptions” as programming pattern

“ exceptional ” control flow

1. `defer`
2. `panic`
3. `recover`

Deferred functions

- each function/method can be called:
  1. conventionally
  2. deferred
  3. asynchronosuly (see later)

  - Also in Apple's *Swift* language

Deferred call

```
func main() {
        defer fmt.Println ("␣␣1")
        fmt.Println ("2")
}
```

Deferred call

- A deferred call is (guaranteed to be) executed when the surrounding function body *returns*
- eval'd for side-effect only, returned value irrelevant
- deferred calls can be nested, too

And if there's more than one?

```
func main() {
        defer fmt.Println ("␣␣1")
        defer fmt.Println ("␣␣2")
        fmt.Println ("3")
        fmt.Println ("4")
        defer fmt.Println ("␣␣5")
        fmt.Println ("6")
}
```

Deferred calls

Deferred calls are stacked

Also here: closures needed

- deferred call: variable can outlive surrounding scope

```
func main() {
    var x = 0
    {   var x = 7            // local, nested scope
        defer func () {
            fmt.Println(x)   // = 8
        } ()
        x = x+1
    }
    x++
}
```

Deferred functions: what's the point?

- Guaranteed[1] to be executed when returning

  even if the function body panics
- good for *clean up jobs* if something unexpected throws the planned control flow off the track = "panics"
  - out-of-memory
  - nil-pointer derefence
  - out-of-bound access to slices/arrays
  - deadlocks
  - ...
- clean-up jobs
  - close open files
  - close channels
  - ...
  - if clean-up means: "fiddling with the return value", use return parameter in the signature
- more flexible than *finally*-clauses

---

[1] no 100% guarantee (divergence) Also: wait for goroutines

Panic

- cf. *exceptions*
- "jumps out" of the normal control flow
- right to the end of procedure
- panics "propagate" from callee too caller
  - but not before *deferred* functions are done as well
- unravel the call-stack
- deferred code: can raise panic as well

Panic & recover

- cf. `thow` (or `raise`) and `catch` for exceptions
- `recover`: useful (and with any effect) in deferred code, only

```
            panic (1337)     // pass a value to panic
....
         var x = recover ... // retrieve value in case of panic
```

Concurrency

Shared var's considered harmful

Go's concurrency mantra

"Don't communicate by sharing memory, share memory by communicating!"

Concurrency in Go

- concurrency vs. parallelism

Go concurrency

goroutines + channels

- claimed to be “easy”
- first-class, typed channels

Coroutines

- control-structure
- “cooperating” procedures, collaborative
- a sub-routine/procedure with “multiple entry points”
- control passed back and forth between procedures
- `yield` vs. `return`
- as such: no real parallellism.
- kind of oldish concept, superseded by threads, actors, continuations . . .
- multiple stacks
- often implemented with continuations

Generator (here Python)

```
>>> def letters_generator():
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current)+1)

>>> for letter in letters_generator():
        print(letter)
```

Goroutines

- Go's name for its unit of concurrency
- executing function calls asynchronously

- goroutine vs. threads
- "green threads"
- "lightweight" threads
- "threads minus monitor communication"
- goroutine dies when parent dies

function call

```
f(v)
```

async. function call

```
go f(v)
```

Channels

- "named pipes"
- FIFO, bounded, non-lossy communication
- crucial data type with synchronization power (see later)
- taking a back-seat:
    - locks
    - mutexes
    - monitors
    - semaphores. . .
- channels: *first-class* data
    - channels can send (reference to) channels
    - can be passed around by functions . . . .
    - inspired by CSP (and CCS, and, actually π)
- *directed* channels

Channel operations

- create channels (with capacity)
- close a channel[2]
- send and receive
- choice over channel communication
- different from `switch`
  - synchonization statement (`select {}`)!
  - no *first match*
- typical use: select over input

```
select {
    case i1 = <-c1:
          ...
    case i2 = <-c2:
  ...
  }
```

- “mixed” choice: possible

---

[2] don’t forget, otherwise deadlocking!

Channel

```
package main
import "fmt"

func main() {
        messages := make(chan string, 0)   // declare + initialize

        go func() { messages <- "ping" }() // send
        msg := <-messages                  // receive
        fmt.Println(msg)
}
```

Channels for synchronizing

Semaphores by channels:

```
type dummy interface {}                 // dummy type,
type Semaphore chan dummy               // type definition

func  (s Semaphore) Vn (n int) {
        for i:=0; i<n; i++ {
                s <- true               // send something
        }
}
func  (s Semaphore) Pn (n int) {
        for i:=0; i<n; i++ {
                <- s                    // receive
        }
}

func  (s Semaphore) V () {
        s.Vn(1)
}
func  (s Semaphore) P () {
        s.Pn(1)
}
```

“Generator” with channels

```go
package main
import ("fmt")


func letters_generator (c chan rune) {
        for x := 'a'; x < 'e'; x++  {
                c <- x                          // send
        }
        close(c)                            // don't forget
}

func main() {
        c := make (chan rune)           // synchr. channel
        go letters_generator (c)        // goroutine
        for r := range c {              // iterate reception
                fmt.Printf("%c\n", r)
        }
}
```

Select

```
import "fmt"


func main() {
    c1 := make(chan string)
    c2 := make(chan string)
    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

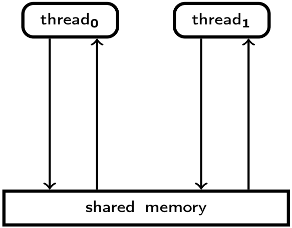
Memory model

Concurrency

> *"Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other"*
> *– (Wikipedia)*

- performance increase, better latency
- many forms of concurrency/parallelism: multi-core, multi-threading, multi-processors, distributed systems

Shared memory: a simplistic picture



- one way of "interacting" (i.e., communicating and synchronizing): via shared memory
- a number of threads/processes/goroutines. . . : access common memory/address space
- interacting by sequence of read/write (or load/stores etc)

however: considerably *harder* to get correct and efficient programs

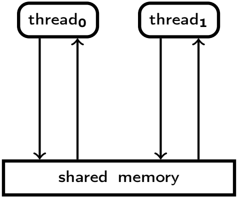
Perhaps disquieting trivial example

```
thread_0          |        thread_1
------------------+------------------------------
    x := 1        |     y := 1
    print y       |     print x
```

Results?

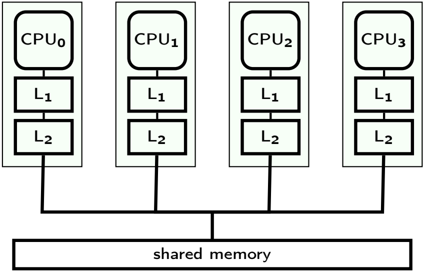
Is the result x,y = 0,0 observable?

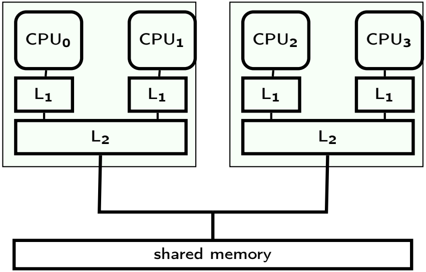
Shared memory concurrency in the real world

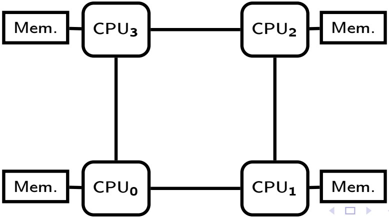


- simplistic memory architecture does not reflect reality
- *out-of-order* executions:
  - modern systems: complex memory hierarchies, caches, buffers . . .
  - compiler optimizations,

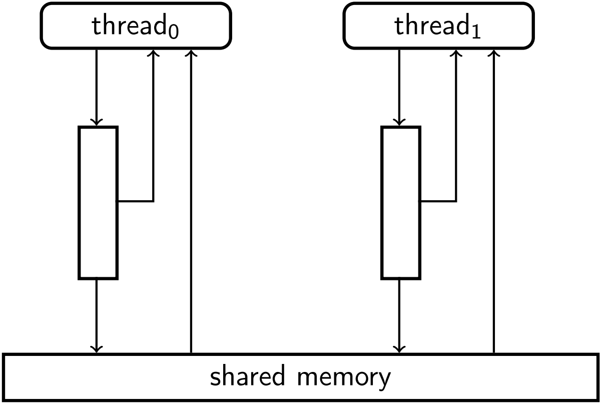
SMP, multi-core architecture, and NUMA







Hardware optimization: Write buffers



Dekker's solution to mutex

- As known, shared memory programming requires synchronization: mutual exclusion

Dekker

- simple and first known mutex algo
- here (rather) simplified

```
  initially flag_0 = flag_1 = 0
-------------------------------------
flag_0 := 1;       |   flag_1 := 1
if (flag_1 = 0)    |   if (flag_0 = 0)
then               |   then
      CRITICAL     |         CRITICAL
```

Compiler optimizations

- *many* optimizations with different forms:
    - *elimination* of reads, writes, sometimes synchronization statements
    - re-ordering of independent non-conflicting memory accesses
    - *introductions* of reads
- examples
    - constant propagation
    - common sub-expression elimination
    - dead-code elimination
    - loop-optimizations
    - call-inlining
    - . . . and many more

Compilers vs. programmers

- What are valid (semantics-preserving) compiler-optimations?
- What is a good memory model as compromise between programmer's needs and chances for optimization

Programmer

- want's to understand the code
- ⇒ profits from strong memory models

Compiler/HW

- want to optimize code/execution (re-ordering memory accesses)
- ⇒ take advantage of weak memory models

Sad facts and consequences

- *error-prone* concurrent code, “unexpected” behavior
    - Dekker (and other well-know mutex algo’s) is incorrect on modern architectures
- unclear/obstruse/informal hardware specifications, compiler optimizations may not be transparent
- understanding of the memory architecture: crucial for *performance*!

Need for unambiguous description of the behavior of a chosen platform/language under shared memory concurrency $\Longrightarrow$ memory models

Memory (consistency) model

What's a memory model?

“A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system.”
– Adve, Gharachorloo

- MM specifies:
  - How threads interact through memory.
  - What value a read can return.
  - When does a value update become visible to other threads.
  - What assumptions are allowed to make about memory when writing a program or applying some program optimization.

The bottom line

- naive programmer: unspoken assumptions/simplistic hardware
  - Program order: statements executed in the order written/issued (Dekker).
  - atomicity: memory update is visible to everyone at the same time

Sequential consistency (Lamport 1979)

"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Go's memory model

- quite conventional weak memory model
- similarly defined for
    - Java (Java 5 JSR-133)
    - C/C++11/
- "data-race free model"
- based on the notion of " Happens-before "

There's hope, though

Data race free model

data race free programs/executions are sequentially consistent!

Data race

- A data race is the "simultaneous" access by two threads to the same shared memory location, with at least one access a write.
- a program is race free, if *no execution reaches* a race.

Especially

Sequential programs behave as one would expect (phew . . .)

There's hope, though

Data race free model

data race free programs/executions are sequentially consistent!

Data race

- A data race is the "simultaneous" access by two threads to the same shared memory location, with at least one access a write.
- a program is race free, if *no sequentially consistent execution reaches* a race.

Especially

Sequential programs behave as one would expect (phew . . .)

Better synchronize properly

- the weak mm is
  - well-defined, but
  - *complex*
- make programs properly synchronized (serialized)

"If you must read the rest of this document [about Go's mm] to understand the behavior of your program, you are being too clever. Don't be clever.
– from Go's memory model description

- in other words: if there's a race, *game over*.
- how to synchronize properly: use " synchronization "

The art of concurrent programming = the art of *synchronization* (and communication)

Shared var's considered harmful

Go's concurrency mantra

"Don't communicate by sharing memory, share memory by communicating!"

Order relations

synchronizing actions: channel communication, lock access (, access to *volatile* variables in Java) . . .

- synchronization order $<_{sync}$: total order on all synchronizing actions (in an execution)
- an s-action synchronizes-with all $<_{sync}$ subsequent s-actions by any thread
- happens-before ($<_{hb}$): transitive closure of program order and synchronizes-with order

Is it clear what it means that something happens-before?

"To specify the requirements of reads and writes, we define happens before, a partial order on the execution of memory operations in a Go program. If event e1 happens before event e2, then we say that e2 happens after e1. Also, if e1 does not happen before e2 and does not happen after e2, then we say that e1 and e2 happen concurrently."

"Within a single goroutine, the happens-before order is the order expressed by the program."

Let's have another look

- *program* order:

  "Within a single goroutine, the happens-before order is the order expressed by the program."

```
x = 5
y = 2
```

- in a run: the x-assignment "is happening" before y-assignment?

Let's have another look

- *program* order:

  "Within a single goroutine, the happens-before order is the order expressed by the program."

```
x = 5
y = 2
```

- in a run: the x-assignment "is happening" before y-assignment?
- NO!!! not guaranteed!
- x-assignment " `happens-before` " y-assignment ($<_{hb}$)
- $<_{hb}$ determines what *may be observed*

The spec again

Observability

A read r of a variable v observes a write w to v if both of the following hold:

- *r does not happen before w.*
- *There is no other write w' to v that happens after w but before r.*

The spec again

Observability: the real deal

A read r of a variable v is allowed to observe a write w to v if both of the following hold:

- *r does not happen before w.*
- *There is no other write w' to v that happens after w but before r.*

Happens before for send and receive

```
x := 1     |   y := 2
c!()       |   c?()
print y    |   print x
```

which read is guaranteed / may happen?

Message passing and happens-before

Send before receive

"A send on a channel happens before the corresponding receive from that channel completes."

Receives before send

"The kth receive on a channel with capacity C happens before the $k + C$th send from that channel completes."

Message passing and happens-before

Send before receive

"A send on a channel happens before the corresponding receive from that channel completes."

Receives before send

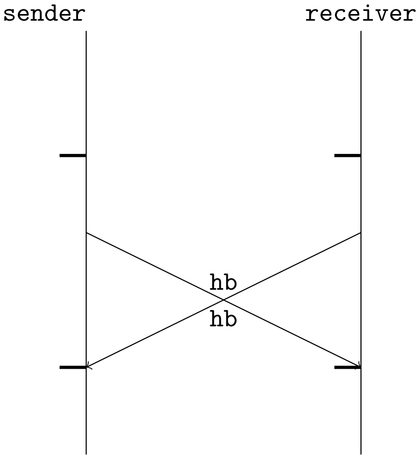
"The kth receive on a channel with capacity C happens before the $k + C$th send from that channel completes."

Receives before send, unbuffered

A receive from an unbuffered channel happens before the send on that

Happens-before for send and receive

```
x := 1    |   y:=2
c!()      |   c?()
print(y)  |   print x
```



Go memory model

- catch-fire / out-of-thin-air (≠ Java)
- standard: DRF programs are SC
- Concrete implementations:
  - more specific
  - platform dependent
  - difficult to “test”

```
[msteffen@rijkaard wmm] go run reorder.go
 1 reorders detected after 329 interations
 2 reorders detected after 694 interations
 3 reorders detected after 911 interations
 4 reorders detected after 9333 interations
 5 reorders detected after 9788 interations
 6 reorders detected after 9951 interations
```

Synchonizing in Go specifically

1. firing off a goroutine
    - `go f a` $<_{hb}$ `f a` starts executing
2. Channel communication
    - channel *send* $<_{hb}$ corresponding channel *receive*
    - *closing* a channel $<_{hb}$ *receiving* the info that it's closed
    - unbuffered/sync. channel: *receive* $<_{hb}$ *send* completes
    - a corresponding generalizatiom for of a k-sized channel
3. further conditions for other constructs with synchronizing power
    - *locks*,
    - *once*

Conclusion

Things I left out

- packaging
- range of mundane data structures
- overview over the library
- go “tools”

Go criticism

- underwhelming type system
- kindergarden type inference
- overloading, inflexibility
- exceptions, nil
- no generics!
- pattern matching
- oo-bias
- trivial (and "implicit") way of regulating *visibility* & *export* on package level

Should I stay or should I Go?