Concurrency in Go

S. Lizeth Tapia Tarifa, Andrea Pferscher, Juliane Päßler, Tobias John

September 27, 2024

University of Oslo

Repetition & Outlook

- Distributed systems and synchronous channels
- Asynchronous channels
- Actors – asynchronous communication without channels
- Futures and Promises
- Active Objects – actors with object model and cooperative scheduling

Concurrent Programming Languages

- Concurrency model part of the language
- Provides abstraction and first-class primitives (in addition to libraries)
- How to fit concurrency nicely into language design?

Go Basics

Background

Growing dissatisfaction with C and C++ as system programming languages in 2000s, when multi-core programs became more important

Common criticisms (back then)

- Concurrency hard to do, even harder to get right – no builtin language support
- Type system overly complex
- Long compilation times, complex build systems
- Memory safety

Emergent Solutions

- Solution 1: Make C++ better (e.g., coroutines in C++20, compositional futures in C++23)
- Solution 2: A new language with simplicity and asynchronous communication first: **Go**
- Solution 3: A new language with type and memory safety first: **Rust**

History of Go

- First plans around 2007 at Google, due to above dissatisfaction
- Public announcement 2009, first release 2012, widely adapted by now
- Inspired by:
  - C (systems programming language)
  - CSP (research model)
  - Newsqueak (research language)
  - Erlang (concurrent systems programming language)
  - Concurrent ML (systems programming language)
  - Python (scripting language)
- Very much a consolidation language along the idea of “less is more”

Go's Non-revolutionary Feature Mix

- Imperative
- Compiled, no VM
- Garbage collected
- Concurrency with light-weight processes (goroutines) and channels
- Strongly typed
- Portable
- Higher-order functions and closures
- No orthodox OO, but common patterns for emulation

Agenda

1. Objects in Go
2. Types in Go
3. Concurrency in Go

Go code on the slides is sometime abbreviated to fit the format

Go Object Model

Go's heterodox take on OO

- No classes
- No class inheritance, also no inheritance on records
- Interfaces as types

Code reuse

Code reuse encouraged by

- Embedding
- Aggregation (internal records)

A First Glimpse at Go

Go

```
type Pair struct { X, Y float64 }

func main() {
  var pair1 Pair
  pair1 = Pair{ 3,4 }
  pair2 := Pair{ 1,2 } //no type needed if initialized
  var res float64 = pair1.Abs() + pair2.Abs()
  fmt.Println(res)
}

func (x *Pair) Abs() float64 { ... }
```

What is a Type?

Views on Types

- Compiler & run-time system
  - A hint for the compiler of memory usage & representation layout
  - Piece of meta-data about a chunk of memory
- Programmer
  - Partial specification for safety
  - Whatever I must do to make the compiler happy
- Orthodox OO
  - A type is essentially a class (at least the interesting ones/custom types)

Milner's dictum on Static Type Systems

"*Well-typed programs cannot go wrong*"

For some notion of going wrong.

How to implement an interface with an object?

- Interfaces contain *methods* (but no fields)
- Records contain *fields* (but no methods)

What is an object?

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

Interfaces

Go

```
type Absl interface { Abs() float64 }
type Triple struct { X, Y, Z float64 }
func (x *Triple) Abs() float64 { ... }
func main() {
  var a Absl //must contain something that implements Absl
  pair := Pair{1,2}
  triple := Triple{3,4,5}

  a = &pair  // a *Pair is ok
  a = &triple // a *Triple is ok
  a = pair // a Pair is not ok, except pair := &Pair{1,2}
}
```

Duck Typing

Duck Typing

"If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

- If interface and record are detached, a method can be called if the record fits its signature
- Dynamic duck typing: check at runtime whether the record fits
- Static duck typing: check at compile time whether the type of the value/variable fits

Beware: Go does *static* duck typing: smaller runtime, no need for time tagging

Code Reuse with Embeddings

Go

```
type Pair struct { X, Y float64 }
type Triple struct {
    Pair // no variable name -> implicit field
    Z float64
}

func main(){
  triple := Triple{ Pair { 1,2 } , 3}
  fmt.Println(triple.X)
}
```

Go Concurrency

Go Concurrency

Go's concurrency mantra

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

- Go does have shared memory via global variables, heap memory etc.
- But you are supposed to only send references – getting a reference transfers ownership, i.e., the permission to write/read it

Go's primitives

- Goroutines – lightweight threads
  - Own call stack, small stack memory (2KB initially), handled by go runtime
  - Very cheap context switch
- Channels
  - Synchronous and typed
  - Communication between (lightweight) threads
  - Main means of synchronization

Goroutines

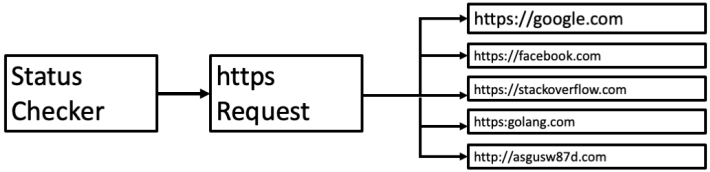
3 ways to call a function

- `f(x)` – ordinary (synchronous) function call, where f is a defined function or a functional definition
- `go f(x)` – called as an asynchronous process, i.e., goroutine
- `defer f(x)` – the call is delayed until the surrounding function returns

Goroutines

3 ways to call a function

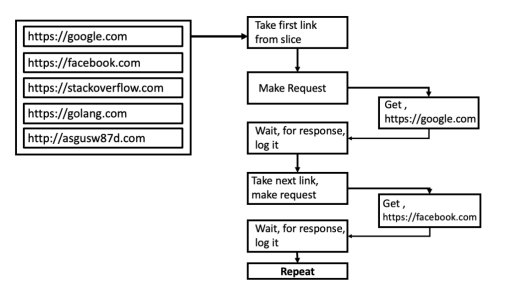
- `f(x)` – ordinary (synchronous) function call, where f is a defined function or a functional definition
- `go f(x)` – called as an asynchronous process, i.e., goroutine
- `defer f(x)` – the call is delayed until the surrounding function returns



Goroutines

3 ways to call a function

- `f(x)` – ordinary (synchronous) function call, where f is a defined function or a functional definition
- `go f(x)` – called as an asynchronous process, i.e., goroutine
- `defer f(x)` – the call is delayed until the surrounding function returns



Channels in Go

- Channels provide a way to send messages form one go routine to another.
- Channels are created with **make**
- The arrow operator (<-) is used both to signify the direction of a channel and to *send* or *receive* data over a channel

Go

```
func main(){
  chl := make(chan float64)
  go sendf(chl);  go receivef(chl)
}
func sendf(ch chan<- float64) {
    ch <- 0.5 }
func receivef(ch <-chan float64){
    v := <-ch }
```

Go Routines - Example 1

back to our server

Go

```go
func main() {
    links := []string{
        "https://google.com",
        "https://facebook.com",
        "https://golang.org",
        "https://stackoverflow.com",
    }
    c := make(chan string)
    for _, link := range links { go checklink(link, c) }
    for i := 0; i < len(links); i++ { fmt.Println(<-c) }
}
```

Goroutines - Example 1

Go

```
func checklink(link string, c chan string) {
    _, err := http.Get(link) //pairs as language builtins
    if err != nil {
        c <- link + "-is-down!"
        return}
    c <- link + "-is-up!"}
```

- The five checklink goroutines starts up concurrently and five calls to `http.Get` are made concurrently as well.
- The main process does not wait until one response comes back before sending out the next request.
- Go scheduler picks up other goroutine in case a goroutine makes a blocking call (e.g. file-based system calls)

Waiting for goroutines to finish in Go

Go offers several synchronization primitives in the `sync` package to avoid using channels in certain situations.

WaitGroup

A `WaitGroup` is a semaphore, used to join over several activities

- The `Add` method is used to add a counter to the `WaitGroup`.
- The `Done` method of `WaitGroup` is scheduled using a defer statement to decrement the `WaitGroup` counter.
- The `Wait` method of the `WaitGroup` type waits for the program to finish all goroutines: The `Wait` method is called inside the main function, which blocks execution until the `WaitGroup` counter reaches the value of zero and ensures that all goroutines are executed.

Waiting for goroutines to finish

Go

```
func main() {
    var wg sync.WaitGroup
    var i int = -1
    var file string
    for i, file = range os.Args[1:] {
        wg.Add(1) // add before asynchronous call!
        go func(file string) { // anonymous function
            compress(file)
            wg.Done()}(file)
    }
    wg.Wait()
    fmt.Printf("compressed %d files\n", i+1)
}
```

Waiting for goroutines to finish

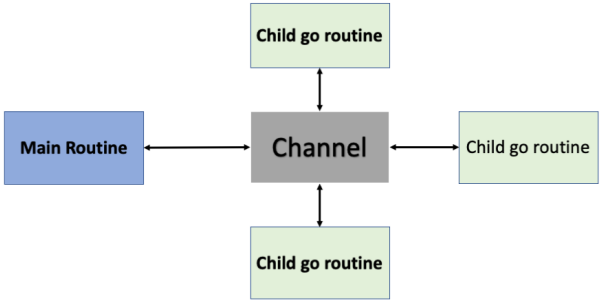
Go

```
func compress(filename string) error { //errors as builtin type
    in, err := os.Open(filename)
    if err != nil {
        return err}
    defer in.Close()
    out, err := os.Create(filename + ".gz")
    if err != nil {
        return err}
    defer out.Close()
    gzout := gzip.NewWriter(out)
    ... // read file and write compressed content
}
```

Channels

Channels in Go

- Channels are bidirectional, synchronous and typed
- Careful which routine is reading and which is writing
- Type support to enforce that



Channel operations

- Send and receive
- Create channels (with buffer)

Go

```go
func main() {
  ch := make(chan int, 2)
  ch <- 1 //does not block!
  ch <- 2
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Channel operations

- Send and receive
- Create channels (with buffer)

Go

```
func main() {
  ch := make(chan int, 2)
  ch <- 1
  ch <- 2
  ch <- 3 //deadlock
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Channel operations

- Send and receive
- Create channels (with buffer)

Channel operations

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func main() {
  ch := make(chan int)
  go write(ch)
  for {
      i, ok := <-ch
      if !ok {break}
      fmt.Println(i) } }

func write(ch chan<- int) {
  ch <- 1; ch <- 2; close(ch) }
```

Channel operations

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func main() {
  ch := make(chan int)
  go write(ch)
  for i := range ch {
      fmt.Println(i)
  }
}

func write(ch chan<- int) {
  ch <- 1; ch <- 2; close(ch) }
```

Channel operations

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func m() {
  ch := make(chan int)
  go write(ch)
  fmt.Println(<-ch)
  fmt.Println(<-ch) } // ?

func write(ch chan<- int) {
  ch <- 1; close(ch) }
```

Channel operations

Selection

- Waiting on several channels in parallel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Channel operations

Selection

- Waiting on several channels in parallel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Go

```
ch1 := make(chan int); ch2 := make(chan int)
go write(ch1); go write(ch2)
select {
  case i1 = <-ch1: fmt.Printf("first call %d \n", i1)
  case i2 = <-ch2: fmt.Printf("second call %d \n", i2) }
```

Channel operations

Selection

- Waiting on several channels in parallel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Go

```
ch1 := make(chan int); ch2 := make(chan int)
go write(ch1); go write(ch2)
select {
  case i1 = <-ch1: fmt.Printf("first call %d \n", i1)
  case i2 = <-ch2: fmt.Printf("second call %d \n", i2)
  default: fmt.Println("I don't block") }
```

Select Operation on Channels in Go

Go

```
func main() {
  done := time.After(30 * time.Second)
  echo := make(chan []byte)
  go readStdin(echo)
  for {
    select {
      case buf := <-echo:
           os.Stdout.Write(buf)
      case <-done:
           fmt.Println("Timed out")
           os.Exit(0) } } }

func readStdin(out chan<- []byte) {
  for {
    data := make([]byte, 1024)
    l, _ := os.Stdin.Read(data)
    if (l > 0) {out <- data} } }
```

Lock Implementation with Channels in Go

Go

```
func main() {
        lock := make(chan bool, 1)
        for i := 0; i < 7; i++ {
                go worker(i, lock)
        }
        time.Sleep(10 * time.Second)
}

func worker(id int, lock chan bool) {
        fmt.Printf("%d wants the lock\n", id)
        lock <- true
        fmt.Printf("%d has the lock\n", id)
        time.Sleep(500 * time.Millisecond)
        fmt.Printf("%d is releasing the lock\n", id)
        <-lock
}
```

Producer-Consumer Implementation in Go

Go

```go
const producerCount int = 4
const consumerCount int = 3

func produce(link chan<- string, id int, wg *sync.WaitGroup) {
        defer wg.Done()
        for _, msg := range messages[id] {
                link <- msg
        }
}

func consume(link <-chan string, id int, wg *sync.WaitGroup) {
        defer wg.Done()
        for msg := range link {
                fmt.Printf("Message %v is consumed by consumer %v\n", msg, id)
        }
}
```

Producer-Consumer Implementation in Go

Go

```
func main() {
        link := make(chan string)
        wp := &sync.WaitGroup{}
        wc := &sync.WaitGroup{}

        wp.Add(producerCount)
        wc.Add(consumerCount)

        for i := 0; i < producerCount; i++ {
                go produce(link, i, wp)
        }

        for i := 0; i < consumerCount; i++ {
                go consume(link, i, wc)
        }

        wp.Wait()
        close(link)
        wc.Wait()
```

Dining Philosophers

Go

```
type Fork struct{ sync.Mutex }

type Philosopher struct {
        id                  int
        leftFork, rightFork *Fork
}
// Goes from thinking to hungry to eating, done eating then starts over.
func (p Philosopher) eat() {
        defer eatWgroup.Done()
        for j := 0; j < 3; j++ {
                p.leftFork.Lock()
                p.rightFork.Lock()
                p.say("eating")
                time.Sleep(time.Second)
                p.rightFork.Unlock()
                p.leftFork.Unlock()
                p.say("finished eating")
                time.Sleep(time.Second)}}
func (p Philosopher) say(action string) {
  fmt.Printf("Philosopher #%d is %v\n", p.id, action) }
```

Dining Philosophers

Go

```
func main() {
  count := 5

  // Create forks
  forks := make([]*Fork, count)
  for i := 0; i < count; i++ {
    forks[i] = new(Fork) }

  // Create philosopher, assign them 2 forks and send them to the dining table
  philosophers := make([]*Philosopher, count)
  for i := 0; i < count; i++ {
    philosophers[i] = &Philosopher{
      id: i, leftFork: forks[i], rightFork: forks[(i+1)%count]}
    eatWgroup.Add(1)
    go philosophers[i].eat()}
  eatWgroup.Wait()}
```

Wrap-Up

Today's Lecture

- Object-orientation in Go: interface types and embeddings
- Goroutines: lightweight threads with builtin support
- Channels in mainstream programming: selection, creation, typing

Next block: actors, active objects and asynchronous communication