Typology of programming languages

An overview of Go

Table of Contents

Go (also referred as Golang)

- First appeared in November 2009
- Some Unix/C-stars:
  - Ken Thompson (Multics, Unix, B, Plan 9, ed, UTF-8, etc. – Turing Award)
  - Rob Pike (Plan 9, Inferno, Limbo, UTF-8, Squeak, etc.)
  - Russ Cox (Plan 9, R2E etc.)
- Derived from C and Pascal
- Open-source
- Garbage Collected, compiled, CSP-style concurrent programming

Go (also referred as Golang)

> *Go is an attempt to combine the safety and performance of statically typed languages with the convenience and fun of dynamically typed interpretative languages.*
>
> *– Rob Pike*

Some compagnies using Go

- Google
- CoreOS
- Dropbox
- Netflix
- MongoDB
- SoundMusic
- Uber
- Twitter
- Dell
- Docker
- Github
- Intel
- Lyft
- ...

Table of Contents

Hello World (1/2)

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Hello ",
                os.Args[1])
}
```

Hello World (2/2)

- Compile and run with:

```
go run hello.go exec
```

- Documentation:

```
godoc -http=":6060"
http://localhost:6060/pkg/
```

Packages

- Every Go program is made up of packages.
- Programs start running in package main

```
package main

import  "fmt"
import  "math/rand"

func main() {
  fmt.Println("My favorite",
              " number is",
              rand.Intn(10))
}
```

Exported Names

- Every name that begins with a capital letter is exported
- ”unexported” names are not accessible from outside the package

```go
package main

import  "fmt"
import  "math"

func main() {
        fmt.Println(math.Pi)
}
```

Declaring variables

- Types come **after** the name
- Variables are introduced via **var**
- A var declaration can include initializers =
- Implicit type declaration can be done using :=

```
func main() {
  var i = 51
  j := 42
  var k int = 51
  l, m := 12, 18
  var n, o int = 12, 18
}
```

Functions

- The return type comes after the declaration, and before the body
- Shared types can be omitted from all but the last parameter
- Return any number of results

```
func add1(x int, y int) int {
  return x + y
}
func add2(x, y int) int {
  return x + y
}
func swap(x, y string)
          (string, string) {
  return y, x
}
```

Named return values & Naked return

- return values may be named
- A return statement without arguments returns the named return values. This is called **naked** returns.

```
func split(input int)
          (x, y int) {
  x = input * 4 / 9
  y = input - x
  return
}
func main() {
  fmt.Println(split(42))
}
```

Types

bool	string	int	int8	int16	int32	int64
uint	uint8	uint16	uint32	uint64	uintptr	byte
rune	float32	float64	complex64	complex128		

- Variables declared without an explicit initial value are given their **zero value** (for string "", for bool **false**, ...)
- The expression T(v) converts the value v to the type T

```
func main() {
  var i int = 42
  var f float64 = float64(i)
  var b bool
  var s string
  fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
```

Constants

- Numeric constants are high-precision values.
- An untyped constant takes the type needed by its context.
- Constants, like imports, can be grouped.

```
const (
  Big = 1 << 100
  Small = Big >> 99
)

func main() {
  fmt.Println(Small)
  fmt.Println(Small*2.01)
}
```

For init;condition; loop { body }

- No parentheses surrounding the three components of the for statement
- The braces are always required.
- The loop will stop iterating once the boolean condition evaluates to false.
- The init and post statement are optional (while loop)
- Omit the loop condition to get a forever loop

```
for i := 0; i < 9; i++ {
    ...
}
```

Conditional testing

- Variables declared by the statement are only in scope until the end of the if
- No parentheses surrounding the declaration plus the condition

```
func main() {
  if v := 42; v < 51 {
    fmt.Println(v)
  }
  else {
    fmt.Println("Ohoh")
  }
}
```

Switch

- A case body breaks automatically, unless it ends with a fallthrough statement
- Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

```
switch os := runtime.GOOS; os {
  case "darwin": //...
  case test():   //...
  case "linux":  //...
  default:       //...
}
```

Pointers

- * allows dereferences
- & generates a pointer to its operand
- No pointer arithmetic

```
func main() {
  var i int = 21
  var p* int = &i
  fmt.Println(*p)
  *p = *p + 2
  fmt.Println(i)
}
```

Structures

- Struct fields can be accessed through a struct pointer (*p).X or p.X

```go
type FooBar struct {
  X int
  Y int
}
func main() {
  v := FooBar{1, 2}
  v.X = 4
  fmt.Println(v.X)
  p := &v
  p.X = 18
  fmt.Println(v.X)
}
```

Anonymous Structures

- Structs can be anonymous
- Structs can be 'raw' compared

```
package main
import "fmt"
func main() {
  a := struct {
    i int
    b bool
  }{51, false}
  b := struct {
    i int
    b bool
  }{51, false}
  fmt.Println(a == b)
}
```

Arrays

- An array has a fixed size
- A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array
- Slices are like references to arrays
- Lower and/or upper bounds can be omitted for slices
- Slices can be increased/decrease. Use len or cap to know length or capacity of a slice.

```go
package main
import "fmt"
func main() {
  primes := [/*size*/]int{2, 3, 5, 7, 11, 13}
  var s []int = primes[1:4]
  fmt.Println(s)
  var s2 []int = primes[:4]
  fmt.Println(s2)
}
```

Dynamic Arrays

- Dynamic arrays are built over slices
- May ise the built-in make function to specify length and capacity
- Use append to add new elements

```
package main
import "fmt"
func main() {
  d := make([]int, 0/*length*/, 0 /*capacity*/)
  // Previous equivalent to   d := []int {}
  d = append(d, 42, 51)
  fmt.Printf("%s len=%d cap=%d %v\n",
             "d", len(d), cap(d), d)
}
```

Range

- Ranges allow Iteration
- Two values per iteration:
  - the index
  - the referenced element
- Skip the index or value by assigning it to _

```
var array = []int{1, 2, 4, 8,
                  16, 32, 64,
                  128}
func main() {
  for i, v := range array {
    fmt.Println("%d,%d", i, v)
  }
}
```

Map

- make function returns a map of the given type, initialized and ready for use.
- The zero value of a map is nil
- A nil map has no keys, nor can keys be added
- Test that a key is present with a two-value assignment

```
package main; import "fmt"
var m map[string] int
func main() {
   m = make(map[string]int)
   m["EPITA"] = 42
   fmt.Print(m["EPITA"])
   delete(m, "EPITA")
   elem, ok := m["EPITA"]
   fmt.Print(elem, ok)
}
// 42 0 false
```

Package Debug

Package debug contains facilities for programs to debug themselves while they are running.

- **FreeOSMemory**: force Garbage Collection
- **PrintStack**: print stack
- **ReadGCStats**: grab stats on Garbage collection
- **SetMaxStack**: set maximum stack size
- **SetMaxThreads**: fix maximum number of threads
- ...

Table of Contents

A word on fonctionnal programming

Functional programming characteristics:

- First-class functions.
  Functions/methods are first-class citizens, i.e. they can be:
  1. named by a variable
  2. passed to a function as an argument
  3. returned from a function as a result
  4. stored in any kind of data structure.
- Closure. Function/method definitions are associated to some/all of the environment when they are defined.

Go Functions are 1st Class

- Functions can be declared at any levels
- Functions can be passed as arguments/return of functions

```
func compute(fn func(int) int,
             value int) int {
  return 42*fn(value)
}
func main() {
  myfun := func(x int) int{
    myfun2 :=
      func(y int) int{ return y*y }
    return myfun2(x)
  }
  fmt.Print(myfun(5), " ",
            compute(myfun, 5))
} // 25 1050
```

Functions closure

- A closure is a function value that references variables from outside its body.
- The function is ”bound” to the variables.

```
func adder() func(int) int {
  sum := 0
  return func(x int) int {
        sum += x
        return sum
  }
}
func main() {
  cumul := adder()
  for i := 0; i < 10; i++ {
        fmt.Println(cumul(i))
  }
}
```

Closures are Weak in Go

Go closures are not as strong as required by pure Fonctionnal Programming

```
func main () {
  counter := 0;
  f1 := func (x int) int {
    counter += x; return counter
  }
  f2 := func (y int) int{
    counter += y; return counter
  }
  fmt.Printf(" %d \n", f1(1))
  fmt.Printf(" %d \n", f2(1))
  fmt.Printf(" %d \n", f1(1))
}
```

Table of Contents

Functions associated to a type 1/3

- No classes, but you can define functions on types
- A function with a special receiver argument

```go
type MyType struct {
  X, Y float64
}
func (v MyType) Abs() float64 {
  return math.Sqrt(v.X*v.X +
                   v.Y*v.Y)
}
func main() {
  v := MyType{3, 4}
  fmt.Println(v.Abs())
}
```

Functions associated to a type 2/3

The receiver is passed by copy unless a pointer is passed as receiver

You do not need to dereference the receiver in this case

```
type My struct {
  X, Y float64
}
func (v* My) SetX(x float64) {
  v.X = x
}
func main() {
  v := My{3, 4}
  v.SetX(18)
}
```

Functions associated to a type 3/3

- We can declare a function on non-struct types
- Possible, only for function with a receiver whose type is defined in the same package as the function

```
type My float64
func (f My) Abs() float64 {
  if f < 0 {
    return float64(-f)
  }
  return float64(f)
}
func main() {
  f := My(-math.Sqrt2)
  fmt.Println(f.Abs())
}
```

Interface

- An interface type is defined as a set of method signature
- A value of interface type can hold any value that implements it

```
type Runner interface {
  Run() int
}
type MyType struct {
  X int
}
func (v MyType) Run() int {
  return 42
}
func main() {
  var a Runner;  v := MyType{3}
  a = v;  fmt.Println(a.Run())
}
```

Stringer Interface

- Useful to print types

```
type Person struct {
  Name string
  Age  int
}
func (p Person) String()
                string {
  return
    fmt.Sprintf("%v (%v years)",
                p.Name, p.Age)
}

//...
fmt.Println(Person{"John Doe", 42})
```

Runtime Polymorphism

```go
package main; import "fmt"

type Runner interface { Run() int }
type MyType1 struct { X int }
type MyType2 struct { X,Y int }

func (v MyType1) Run() int {return 42 }
func (v MyType2) Run() int {return v.X + v.Y }
func run(v Runner) int { return v.Run()}

func main() {
  v1 := MyType1{3}
  v2 := MyType2{3, 4}
  fmt.Println(v1.Run(), v2.Run())
  fmt.Println(run(v1),run(v2))
}
```

Maximum Polymorphism and Reflection

- maximum polymorphism through the empty interface: ”interface {}”
- For example, the printing functions in fmt use it
- Need for some reflection mechanisms, i.e. ways to check at runtime that instances satisfy types, or are associated to functions.
- For instance, to check that x0 satisfies the interface I

```
x1, ok := x0.(I);
```

(ok is a boolean, and if true, x1 is x0 with type I)

Type Dispatch

- Dynamic Dispatch can easily be done

```go
func dispatch(i interface{}) {
  switch v := i.(type) {
        case int:
        //...
        case string:
        //...
        default:
        //...
  }
}
```

Duck Typing (1/2)

- Go functional polymorphism is a type-safe realization of “duck typing”.
- Implicit Rule: If something can do this, then it can be used here.
  - Opportunistic behavior of the type instances.
  - Dynamic OO languages like CLOS or Groovy include duck typing in a natural way

Duck Typing (2/2)

In static languages: duck typing is realized as a structural typing mechanism (instead of nominal in which all type compatibilities should be made explicit – see e.g., implements, extends in Java).

Duck typing uses mechanisms similar to the one we have with C++ Generic Programming.

Go Interfaces and Structuration Levels

- Go interfaces: A type-safe overloading mechanism where sets of overloaded functions make type instances compatible or not to the available types (interfaces).
- The effect of an expression like: x.F(..) depends on all the available definitions of F, on the type of x, and on the set of available interfaces where F occurs
- Dilemma between the functional and modular levels: Go votes for the functional level, but less than CLOS, a little more than Haskell, and definitely more than Java/C# (where almost every type is implemented as an encapsulating class)...

Summary about polymorphism & interface

- Go interface-based mechanism is not new, neither very powerful..
- Haskell offers type inference with constrained genericity, and inheritance

- Go structural-oriented type system is not new, neither very powerful...
- OCaml offers type and interface inference with constrained genericity, and inheritance

Summary about polymorphism & interface

- In Go, no explicit inheritance mechanism. The closest mechanism: some implicit behavior inheritance through interface unions (called “embedding”):

```
type Foo interface {
  F1() int;
type Bar interface {
  F2() int;
}
type FooBar interface {
  Foo // inclusion
  Bar // inclusion
}
```

Rule

If type T is compatible with FooBar, it is compatible with Foo and Bar too

Table of Contents

Concurrency

The idea

Impose a sharing model where processes do not share anything implictly (see Hoare's Communicating Sequential Processes 1978)

Motto

Do not communicate by sharing memory; instead, share memory by communicating.

Objectives

Reduce the synchronization problems (sometimes at the expense of performance)

Three basic constructs

- Goroutines are similar to threads, coroutines, processes, (Googlers claimed they are sufficiently different to give them a new name)
  - Goroutines are then automatically mapped to the OS host concurrency primitives (e.g. POSIX threads)
  - A goroutine does not return anything (side-effects are needed)
- Channels: a typed FIFO-based mechanism to make goroutines communicate and synchronize
- Segmented stacks make co-routines usables

Go Routine

- A goroutine is a lightweight thread managed by the Go runtime
- starts a new goroutine running **go**
- Goroutines run in the same address space
- access to shared memory must be synchronized (see sync package)

```
func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}
func main() {
  go say("world")
  say("hello")
}
```

Channels 1/3

- Channels are a typed conduits
- Send to channel using $ch < -42$
- Receive from channel using $v := < -ch$
- Channels can be buffered: blocking when the buffer is full or empty

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

Channels 2/3

- A sender can close a channel to indicate that no more values will be sent.
- Receivers can test whether a channel has been closed $v, ok := < -ch$
- Sending on a closed channel will cause a panic.
- Channels aren't like files; you don't usually need to close them

```
package main; import "fmt"
func compute(n int, c chan int) {
  for i := 0; i < n; i++ { c <- i }
  close(c)
}
func main() {
  c := make(chan int, 10)
  go compute(cap(c), c)
  for i := range c { fmt.Println(i)  }
}
```

Channels 3/3

- Select lets a goroutine wait on multiple communication operations

```
func main() {
  c1 := make(chan string); c2 := make(chan string)
  go func() { time.Sleep(time.Second * 5)
              c1 <- "one"
            }()
  go func() { time.Sleep(time.Second * 5);
              c2 <- "two"
            }()
  for i := 0; i < 2; i++ {
    select {
      case msg1 := <-c1:
        fmt.Println("received", msg1)
      case msg2 := <-c2:
        fmt.Println("received", msg2)
    }
  }
```

PingPong Time

Demo.

Table of Contents

Reflection & Tags 1/2

Reflection is the ability of program to introspect, and modify its own structure and behavior at runtime

```go
package main

import (
"fmt"
"reflect"
)

type Foo struct {
  FirstName string `tag_name:"tag 1"`
  LastName  string `tag_name:"tag 2"`
  Age       int    `tag_name:"tag 3"`
}
```

Reflection & Tags 2/2

```
func (f *Foo) reflect() {
  val := reflect.ValueOf(f).Elem()
  for i := 0; i < val.NumField(); i++ {
    valueField := val.Field(i)
    typeField := val.Type().Field(i)
    tag := typeField.Tag
    fmt.Printf("Field Name: %s,\t Field Value: %v,\t Tag Valu
           typeField.Name,
           valueField.Interface(),
           tag.Get("tag_name"))
  }
}
func main() {
  f := &Foo{FirstName: "John", LastName:  "Doe",
        Age: 30}
  f.reflect()
}
```

Defer

- Defers the execution of a function until the surrounding function returns
- The deferred call's arguments are evaluated immediately but the function call is not executed until the surrounding function returns.
- Defer is commonly used to simplify functions that perform various clean-up actions (closing file for instance)

```go
func main() {
  defer fmt.Println("world")
  fmt.Println("hello")
}
```

Stacking Defer

- Deferred function calls are pushed onto a stack
- When a function returns, its deferred calls are executed in last-in-first-out order

```go
package main

import "fmt"
func main() {
  fmt.Println("counting")
  for i := 0; i < 10; i++ {
    defer fmt.Println(i)
  }
  fmt.Println("done")
} // counting done 9 8 7 6 5 4 3 2 1 0
```

Panic and Recover 1/2

- Panic is a built-in function that stops the ordinary flow of control and begins panicking.
- Recover is a built-in function that regains control of a panicking goroutine. **Recover is only useful inside deferred functions.**

```go
package main
import "fmt"

func g(i int) {
  fmt.Println("Enter g.")
  panic(i)
  fmt.Println("Exit g.")
}
```

Panic and Recover 2/2

```go
func f() {
  defer func() {
    if r := recover(); r != nil {
      fmt.Println("Recovered in f", r)
    }
  }()
  fmt.Println("Calling g.")
  g(42)
  fmt.Println("Returned normally from g.")
}

func main() {
  f()
  fmt.Println("Returned normally from f.")
}
```

Summary

- Simple and scalable multithreaded and concurrent programming
- All is type
- Tooling and API
- Performance is on the order of C
- Includes a lot of paradigms

- Weak type system
- GC (tricolor concurrent mark-and-sweep algorithm) causes runtime overhead
- Not thread-safe
- No generics