Functional Programming with Go

Concepts of Programming Languages
25 October 2018

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim

What is Functional Programming?

- the combination of pure functions;
- avoiding shared state, mutable data, and side-effects;
- the prevalence of declarative approach rather than imperative approach.

2

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports higher-order functions and lazy evaluation features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism

3

Functional programming languages are categorized into two groups

- **Pure** Functional Languages

These types of functional languages support only the functional paradigms and have no state. For example − Haskell.

- **Impure** Functional Languages

These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

4

Functional programming offers the following advantages

- Bugs-Free Code

Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

- Efficiency

Functional programs consist of independent units that can **run concurrently.** As a result, such programs are more efficient.

- Lazy Evaluation

Functional programming supports **lazy evaluation** like Lazy Lists, Lazy Maps, etc.

- Distribution

Functional programming supports distributed computing

Functional Composition

Functions can be composed to new functions

g(f(x)) -> (g ∘ f)(x)

```
// Function f()
f := func(x int) int {
    return x * x
}

// Function g()
g := func(x int) int {
    return x + 1
}

// Functional Composition: (g∘f)(x)
gf := func(x int) int {
    return g(f(x))
}

fmt.Printf("%v\n", gf(2)) // --> 5
```

Functional Composition (2)

Functions can be composed with functions as parameters

g(f(x)) -> (g ∘ f)(x)

```
// Type any makes the code readable
type any interface{}
type function func(any) any
```

```
compose := func(g, f function) function {
    return func(x any) any {
        return g(f(x))
    }
}
```

```
square := func(x any) any { return x.(int) * x.(int) }

fmt.Printf("%v\n", compose(square, square)(2)) // --> 4*4 = 16

fmt.Printf("%v\n", compose(compose(square, square), square)(2)) // --> 256
```

Clojures (Only impure if you modify the closed-over variable)

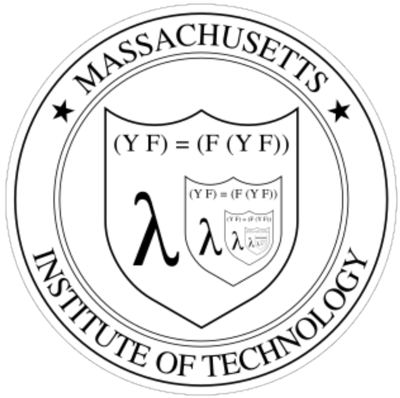
```go
// intSeq returns another function, which we define anonymously in the body of intSeq.
// The returned function closes over the variable i to form a closure.
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
func main() {
    // We call intSeq, assigning the result (a function) to nextInt.
    // This function value captures its own i value, which will be updated each time we call nextInt.

    nextInt := intSeq()
    // See the effect of the closure by calling nextInt a few times.
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // To confirm that the state is unique to that particular function, create and test a new one.
    newInts := intSeq()
    fmt.Println(newInts())
}
```

Run

History: The Lambda Calculus



- What is it?
- Why is it useful?
- Where did it came from?

[Professor Graham Hutton explains the Lambda Calculus (Cool Stuff :-)](https://www.youtube.com/watch?v=eis11j_iGMs) (https://www.youtube.com/watch?v=eis11j_iGMs)

Hint: To understand this video you will watch it at least three times :-)

Summary of the Introduction to Lambda Calculus

- Pure Functions have no internal state
- The Lambda Calculus is very different to the Turing Machine in this way
- The lambda calculus knows only three primitives: Variables (x,y,z), building functions λx.x, applying functions "(λx.x) 5" with values
- There are no datatypes (number, logical values) - values can be functions, No build in recursion!
- It can encode any computation (Church-Turing thesis)
- Lambda Calculus is present in most major programming languages

10

Lambda Calculus in Go

play.golang.org/p/IFknxxxd1vx (https://play.golang.org/p/IFknxxxd1vx)

```
// Lambda Calculus in Golang --> See Video Graham Hutton
// https://www.youtube.com/watch?v=eis11j_iGMs

// This is the key: A Recursive function definition for all functions!!!
type fnf func(fnf) fnf

ID := func(x fnf) fnf { return x }

// TRUE as function: λx.λy.x
True := func(x fnf) fnf {
    return func(y fnf) fnf {
        return x
    }
}

// FALSE as function: λx.λy.y
False := func(x fnf) fnf {
    return func(y fnf) fnf {
        return y
    }
}
```

Application

```
fmt.Printf("Id = %p\n", ID)
fmt.Printf("True = %p\n", True)
fmt.Printf("False = %p\n", False)

// debugging functions
f := func(x fnf) fnf { fmt.Printf("f()\n"); return x }
g := func(y fnf) fnf { fmt.Printf("g()\n"); return y }

// select and call first function f(ID)
False(False)(True)(f)(g)(ID)

// select and call second function g(ID)
True(False)(True)(f)(g)(ID)
```

12

Lambda Calculus in Go: NOT

```
// NOT as function: λb.b false true
Not := func(b fnf) fnf {
    return b(False)(True)
}

// should print false
fmt.Printf("Not(True) = %p\n", Not(True))

// should print true
fmt.Printf("Not(False) = %p\n", Not(False))

// select and call first function f(ID)
Not(False)(f)(g)(ID)

// select and call second function g(ID)
Not(True)(f)(g)(ID)
```

13

Lambda Calculus in JavaScript

```
TRUE = a => b => a;
FALSE = a => b => b;
NOT = f => a => b => f(b)(a);

f = x => x + 10
g = x => x + 20

TRUE(f)(g)(3)       // -> 13
FALSE(f)(g)(3)      // -> 23

NOT(TRUE)(f)(g)(3)  // -> 23
NOT(FALSE)(f)(g)(3) // -> 13
```

Fundamentals of Lambda Calculus & Functional Programming in JavaScript

(https://www.youtube.com/watch?v=3VQ382QG-y4)

Famous Functional Languages inspired by the Lamda Calculus

- Haskell

www.youtube.com/watch?v=1jZ7j21g028 (https://www.youtube.com/watch?v=1jZ7j21g028)

- ML
- Clojure
- F#
- Scala
- JavaScript

Palindrome Problem in Functional (pure) Languages

- Haskell

```
is_palindrome x = x == reverse x
```

- Clojure

```
(defn palindrome? [x]
    (= x (clojure.string/reverse x)))
```

Palindrome Problem in Functional (impure) Languages

- F#

```
let isPalindrome (x: string) =
    let arr = x.ToCharArray()
    arr = Array.rev arr
```

- Scala

```
def isPalindrome[A](l: List[A]):Boolean = {
    l == l.reverse
}
```

- Go

```
func IsPalindrome3(x string) bool {
    return x == strings.Reverse(x)
}
```

Functions as First Class Citizens in Go

- Go supports functions as 1st Class Citizens: Clojures und Lambdas
- Functions can be assigned to variables
- Functions can be used as function parameters and return values (High Order Functions)
- Functions can be created inside functions
- The Go standard library uses functional constructs

Sample from the Go Standard Library

- strings.map

```
// Map returns a copy of the string s with all its characters modified
// according to the mapping function. If mapping returns a negative value, the character is
// dropped from the string with no replacement.
func Map(mapping func(rune) rune, s string) string
```

- Usage

```
s := "Hello, world!"
s = strings.Map(func(r rune) rune {
    return r + 1
}, s)
fmt.Println(s) // --> Ifmmp-!xpsme"
```

Go does not have an API similar to Java Streams

- It is possible to build such an API in Go

```
// array of generic interfaces.
stringSlice := []Any{"a", "b", "c", "1", "D"}

// Map/Reduce
result := ToStream(stringSlice).
    Map(toUpperCase).
    Filter(notDigit).
    Reduce(concat).(string)

if result != "A,B,C,D" {
    t.Error(fmt.Sprintf("Result should be 'A,B,C,D' but is: %v", result))
}
// lambda (inline)
```

20

Classic Word Count Sample

```go
// Classic wordcount sample
// ========================
//
func TestWordCount(t *testing.T) {
    strings := []Any{"a", "a", "b", "b", "D", "a"}

    // Map/Reduce
    result := ToStream(strings).
        Map(func(o Any) Any {
            result := []Pair{Pair{o, 1}}
            return result
        }).
        Reduce(sumInts).([]Pair)

    for _, e := range result {
        fmt.Printf("%v:%v, ", e.k, e.v)
    }
}
```

21

Questions

- How can you implement parallel execution for our API?
- How can you implement distributed execution for our API?

Summary

- You can do functional programming with Go
- Generics and type inference for functions are missing (maybe 2.0?)
- Type definitions for functions make code readable
- You can use functional patterns and generic programming with extra casting (type assertions)
- Functional patterns like Map/Filter/Reduce are easy to implement in Go
- Reflection can help to avoid casting, but it is slow!

Thank you

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim
johannes.weigend@qaware.de (mailto:johannes.weigend@qaware.de)
http://www.qaware.de (http://www.qaware.de)
@johannesweigend (http://twitter.com/johannesweigend)