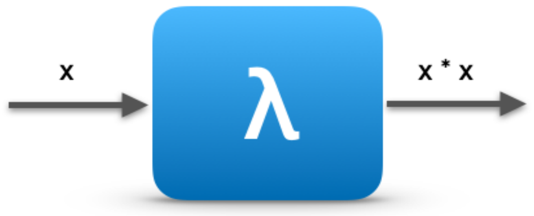
Functional Programming with Go

Concepts of Programming Languages

7 November 2019

Johannes Weigend
Rosenheim Technical University

What is Functional Programming?



-
-
-

Functional Programming – Characteristics

-
-
-
-

Functional programming languages are categorized into two groups

- Pure
- Impure

Functional programming offers the following advantages

-

- run concurrently

- lazy evaluation

-

Functions are Values

```go
func aBlock(i int) {
    fmt.Printf("Entering block: i=%v\n", i)
}

func do(f func (int), loops int) {
    for i := 0; i < loops; i++ {
        f(i)
    }
}

func main() {
    do(aBlock, 5)
}
```

(https://play.golang.org/p/3Lg5BCPiiPA)

Many Functional Languages only support Single Argument Functions

-

```
// ADD with 2 parameters
ADD := func(x, y int) int {
    return x + y
}
```

```
ADD(1,2) -> 3
```

```
// Curried ADD
ADDC := func(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}
```

```
ADDC(1)(2) -> 3
```

Functional Composition

```go
// 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)

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

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

```go
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

●

●

●

(https://www.youtube.com/watch?v=eis11j_iGMs)

Summary of the Introduction to Lambda Calculus

-
-
-
-
-
-

Lambda Calculus in Go

(https://play.golang.org/p/1bLmezdD2zt)

```go
// 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

```go
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)
```

Lambda Calculus in Go: NOT

```go
// 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)
```

Functional Numbers

```go
// Functional Numbers 1
ONCE := func(f fnf) fnf {
    return func(x fnf) fnf {
        return f(x)
    }
}
```

```go
// Functional Numbers 2
TWICE := func(f fnf) fnf {
    return func(x fnf) fnf {
        return f(f(x))
    }
}
```

```go
// Function Numbers 3
THRICE := func(f fnf) fnf {
    return func(x fnf) fnf {
        return f(f(f(x)))
```

```go
    }
```

```go
}
```

Functional Numbers

```go
// Functional Numbers SUCCESSOR(N) = N + 1
SUCCESSOR := func(w fnf) fnf {
    return func(y fnf) fnf {
        return func(x fnf) fnf {
            return y(w)(y)(x)
        }
    }
}
```

```go
Printer := func(x fnf) fnf { fmt.Print("."); return x }
```

```go
SUCCESSOR(TWICE)(Printer)(ID)
fmt.Println("SUCCESSOR(TWICE) = 3")

SUCCESSOR(THRICE)(Printer)(ID)
fmt.Println("SUCCESSOR(THRICE) = 4")
```

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
```

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

Famous Functional Languages inspired by the Lamda Calculus

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

Palindrome Problem in Functional (pure) Languages

-

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

-

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

Palindrome Problem in Functional (impure) Languages

-

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

-

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

-

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

Functions as First Class Citizens in Go

-
-
-
-
-

Sample from the Go Standard Library

-

```go
// 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
```

-

```go
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

-

```
// 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)
```

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)
    }
}
```

Questions

-
-

Summary

-
-
-
-
-
-

Thank you

Johannes Weigend
Rosenheim Technical University
johannes.weigend@qaware.de
http://www.qaware.de