Compare Functional Programming in Go with Haskell

Table of Contents

Introduction

This article compares the programming languages Haskell and Go. For this purpose, concepts of functional programming are compared with each other using examples from both programming languages, but the focus is going be on Haskell, because our lecture was based in Go(lang). **Go(lang)** is not a functional language, it's a multi paradigm programming language, but has a lot of features applies functional principles like functions, lambdas, closures and higher order functions. It doesn't have Sum/product types, Immutable types (beyond the built-in string type), a type system that aids functional abstractions and pattern matching. [1: https://www.quora.com/Is-golang-suitable-for-functional-programming]

The main points are

- General overview of Haskell
- Feature comparison
- Code comparison for selected features like higher order functions, lambdas, functional composition, currying and closure

Haskell

Haskell is a purely functional programming language named after the US mathematician Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. The first Version of Haskell appeared 1990.

Haskell is based on the lambda calculus, which is a formal mathematical system for expressing the notion of computation. Most functional languages are based on the lambda calculus.

Haskell's core information

- everything is immutable. If value is set, it's set forever
- functions can be passed to other function as parameters
- recursion is mandatory
- for, while or technically variables do not exist
- Haskell is lazy, only what is needed is executed
- the compiler is very strict and checks each compilation for errors
- different implementations like Haskell Platform [2: One of the most important implementation is Haskell Platform (https://www.haskell.org/platform/)]
- Standards: Haskell 98 and Haskell 2010 [3: The current Haskell standard is Haskell 2010 (https://wiki.haskell.org/Haskell_2010)]

Haskell Syntax

A quick tour of Haskell syntax where the base types, prefix operators, lists, tuples, functions, recursion and guards are displayed briefly.

Basic Types

Haskell has a static type system. The types are known at compile time for each expression this leads to type safety. So, it won't compile if there are any type errors. In Haskell everything has a type, so the compiler can tell a lot about the program before compiling.

```
-- Char
a = 'a' :: Char

-- String - strings are a list of characters
hello = "Hello World" :: [Char]
-- Indexing Strings - hello !! 2 -- 'l'

-- Bool True and False
t = True :: Bool -- True
f = False :: Bool -- False

-- Int -2^63 2^63
maxInt = maxBound :: Int --  9223372036854775807
minInt = minBound :: Int -- -9223372036854775808

-- Integer
{-
A signed integer of unbounded size. It's not often used because
it's more expensive, but are more reliable because they do not
overflow.
-}
bigInteger = 123456789101112131415161718192O :: Integer --
123456789101112131415161718192O

-- Double
{-
Used for floating-point numbers. A Double value is typically
64 bits wide and use the system's native floating-point
representation.
-}
bigFloat = 3.999999999999 + 0.0000000005 :: Double -- 4.000000000499
```

Infix operator

Functions are usually called using prefix notation, or simpler function name followed by arguments.

```
-- prefix notation
mod 5 4 -- 1
-- or
(+) 5 4 -- 9
```

Some functions are called with infix notation. In this case the function is put between two arguments. The function must be between back ticks (``).

```
-- infix operator
5 `mod` 4 --1
```

List and Tuples

List are elements together with `:`. They can be declared like `'a' : 'b' : 'c' : []` which is the same like `['a', 'b', 'c']`. Every element in a list must have the same type. For different types use tuples `(elem1, elem2, ⋯)`. Tuple can contain different types.

```
-- lists
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
-- The above and below lists are equal
[1..10]
-- this works with characters too
['A', 'B', 'C', 'D']
-- equals to
['A'..'D']

-- ranges with steps
[0,5..20] -- [0, 5, 10, 15, 20]
[10,8..0] -- [10, 8, 6, 4, 2, 0]

-- indexing a list
[1..5] !! 3 -- 2

-- list operations
head [1..5] -- 1
last [1..5] -- 5

-- concat list
[3,5,7,11] ++ [13,17,19,23,29]

-- tuples
("haskell", 1337)

-- accessing elements, this functions only works on tuple length 2
fst ("haskell", 1337) -- "haskell"
snd ("haskell", 1337) -- 1137
-- to access a triple tuple you can usw Data.Tuple.Extra
thd3 ("haskell", 1337, 42) -- 42

-- zip -- combines value in two different lists into two pairs
names = ["Mike", "Tim", "Amy"]
address = ["A 123", "B 456", "C 789"]

namesAddress = zip names address -- [("Mike","A 123"),("Tim","B 456"),("Amy","C 789")]
```

Functions

Mathematically speaking, a function relates all values in a set `A` to values in a set `B`. The function `square x = x^2`, given that `x` is an integer, will map all elements of a set of integers into another set of integers. [4: https://wiki.haskell.org/Function]

A Function has an input and an output. It describes how the output is produced using the input. In Haskell first you declare the function. It begins with the `function name` which is separated by `::` from the type definition. The first arguments will be the receiving types, the last one the returning type.

```
-- Simple add function
-- Type declaration | receiving a Integer, receiving a Integer, returning a Integer
add :: Integer -> Integer -> Integer
-- actual creation of the function
add param1 param2 = param1 + param2

-- function call
add 30 12 -- 42
```

Haskell functions are first class entities, which means that they [4: https://wiki.haskell.org/Function]

- can be given names
- can be the value of some expression
- can be members of a list
- can be elements of a tuple
- can be passed as parameters to a function
- can be returned from a function as a result

Recursion

Recursive functions play a central role in Haskell. Recursion is basically a form of repetition.

```
-- Recursion on factorial example
factorial :: Int -> Int

factorial 0 = 1
factorial n = n * factorial (n-1)

factorial 4 -- 120
```

Pattern Guards

Guards are a way of testing whether a property is true or false.

```
-- Pattern Guards
isOdd :: Int -> Bool

isOdd n
    | n `mod` 2 == 0 = False
    | otherwise = True

isOdd 5 -- True
isOdd 4 -- False
```

Haskell Features

Statically typed

Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler. Types become not only a form of guarantee, but a language for expressing the construction of programs. [5: Features (https://www.haskell.org/)]

```
-- Character
c = 'A' :: Char
-- Integer
i = 42 :: Int
-- Function
f = Int -> [Char]
```

Purely functional

Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers. [5: Features (https://www.haskell.org/)]

```
add :: Integer -> Integer -> Integer
add param1 param2 = param1 + param2
```

Type inference

You don't have to explicitly write out every type in a Haskell program. Types will be inferred by unifying every type bidirectionally. However, you can write out types if you choose, or ask the compiler to write them for you for handy documentation. [5: Features (https://www.haskell.org/)]

Concurrent

Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions. [5: Features (https://www.haskell.org/)]

Lazy

Functions don't evaluate their arguments. This means that programs can compose together very well, with the ability to write control constructs (such as if/else) just by writing normal functions. The purity of Haskell code makes it easy to fuse chains of functions together, allowing for

performance benefits. [5: Features (https://www.haskell.org/)]

Packages

Open source contribution to Haskell is very active with a wide range of packages available on the public package servers. [5: Features (https://www.haskell.org/)]

Haskell Vs. Go

Comparison

The following table displays some of the differences between Haskell and Go(lang) and compares some of their features.

Table 1. Haskell Vs. Go

Feature	Haskell	Go
First appeared	1990	2009
Paradigm	functional, lazy/non-strict, modular	Multi-paradigm: concurrent, functional, imperative, object-oriented
Typing discipline	static, strong, inferred	static, strong, inferred, structural
Operation System (OS)	Cross-platform	Linux, macOS, Windows, DragonFly BSD, FreeBSD, NetBSD, OpenBSD, Plan 9, Solaris
Influenced by	Clean, FP, Gofer, Hope and Hope+, Id, ISWIM, KRC, Lisp, Miranda, ML and Standard ML, Orwell, SASL, Scheme, SISAL	Alef, APL, BCPL, C, CSP, Limbo, Modula, Newsqueak, Oberon, occam, Pascal, Smalltalk
Influenced	Agda, Bluespec, C++11/Concepts, C#/LINQ, Cayenne, Clean, Clojure, CoffeeScript, Curry, Elm, Escher, F#, Frege, Hack, Idris, Isabelle, Java/Generics, LiveScript, Mercury, Perl 6, PureScript, Python, Rust, Scala, Swift, Timber, Visual Basic 9.0	Crystal
Multiple return values	Yes	Yes
Anonymous functions	Yes	Yes
Compiled	Yes	Yes

Stats

Some stats from Hacker News, Reddit and Stack Overflow to compare the activities of the communities.

Table 2. Stats (18.12.2018)

Site	Haskell	Go
Hacker News	10.3K	763
Reddit	8.28K	10.7K
Stack Overflow Stats	34.6K	39.4K

Higher order functions

One of the important benefits of functional programming is that functions are just like any other value and can be handled like regular values.

A function that takes another function (or several functions) as an argument is called a higher-order function.

The example below will illustrate, how functions can be written as parameters in Haskell. In this example the parameter is a function and the result is an int. The result will always be a 3.

```
f :: (Int -> Int) -> Int
f x = 3
```

Every Haskell function takes one argument and return one value. In the following example we are creating a series of functions and are applying each one to the next value. The first int takes a value and returns a function, that function takes an int and returns another function and the last function takes a function and returns an int.

```
sum3 :: Int -> (Int -> (Int -> Int))
sum3 a b c = a + b + c

sum3 1 2 3 -- 6
-- Or
((sum3 1) 2) 3 -- 6
```

Now a simple example with the higher order function map. Map takes a function and a list and applies that function to every element in the list, producing a new list.

```
-- map takes a function (a -> b) and a list [a] as parameter and returns a new list
[b] as result
map :: (a -> b) -> [a] -> [b]

-- if the list is empty it will return a empty list
map _ [] = []
map f (x:xs) = f x : map f xs

-- a simple add function to display the usage of map above
add3 :: Int -> Int
add3 x = x + 3

-- Now we use the map and the add3 function
map add3 [1,2,3,4,5] -- [4,5,6,7,8]
```

Now the same example in Go. Here a new function `add3` is created which will be used in a `Map` function to add `3` to each element of an int list.

```
-- a simple add function
func add3(x int) int {
    return x + 3
}

y := [5]int{1, 2, 3, 4, 5}
Map(add3, y) -- [4,5,6,7,8]
```

Lambdas

Lambdas are anonymous functions. These functions are unnamed and are passed as parameters to other functions. They work like compositions <color rgb="#999999">[6: http://www.mathwords.com/c/composition.htm]</color> in math. Lambdas are named after the lambda calculus <color rgb="#999999">[7: https://www.wolframscience.com/nks/notes-11-12—lambda-calculus/]</color>. When using maps it’s sometimes more convenient to use a lambda instead of a function. They are mainly used if we need a function only once.

In Haskell an anonymous function is a function without a name and can be written like `(\x → x * 5)`. That backslash is expressing a λ ind Haskell and is supposed to look like a lambda.

```
-- anonymous function called with parameter 4
(\x -> x * 2) 4 -- 8

-- or with two parameters
(\x y -> x + y) 2 4 -- 6
```

Another way to use anonymous functions is to name them, if you want to reuse them.

```
-- Define a function and assign to variable
multi2 = (\x -> x * 2)
-- call the variable with parameter
multi2 4 -- 8
```

Example for using a lambda in a map.

```
map (\x -> x * 2) [1,2,3] -- [2,4,6]
```

For the go example of lambda. Here we use a anonymous function in a `sort` with following function `func Slice(slice interface{}, less func(i, j int) bool)`. `Slice` is the interface for list structure and `less` is the anonymous function with parameters `i, j int` and return value `bool`.

```
developer := []string{"Rob Pike", "Robert Griesemer", "Ken Thompson"}

sort.Slice(developer, func(i, j int) bool {
        return len(developer[i]) > len(developer[j])
})

fmt.Println(developer)// --> [Robert Griesemer Ken Thompson Rob Pike]
```

Functional Composition

Function Composition is the process of using the output of one function as an input of another function. In mathematics it would be like `f{g{x}}` where `g()` and `f()` are functions. The output of `g()` is used as Input in `f()`.

The dot operator (.) is used in Haskell to implement function composition.

As example we will use the prelude functions `even` and `not` to define a `odd` function with functional composition.

```
even :: Int -> Bool
not :: Bool -> Bool
```

Without functional composition the definition could look like the example below.

```
odd :: Int -> Bool
odd x = not (even x)
```

But this can be written with functional composition.

```
odd :: Int -> Bool
odd x = not . even
```

A map is a good use case to show its usefulness.

```
map (not . even) [1,2,3] -- [True,False,True]
```

Both programming languages support compositions as arguments in a function. But in Haskell you can use the dot operator to composite the functions. Below the `odd` example in go.

```
// Even function
even := func(x int) bool{
        return x%2 == 0
}
// Not function
not := func(x bool) bool {
    return !x
}

// Function Composition: odd
odd := func(x int) bool{
    return not(even(x))
}

fmt.Printf("%v\n", even(2)) // --> true
fmt.Printf("%v\n", odd(2)) // --> false
```

Currying

Currying is the conversion of a function with several arguments into a function with one argument. All the functions that accept several parameters are curried functions. A good example to understand curried functions is the `max` function. This function looks like it takes two arguments `max 3 9`. Its definition is: `max :: (Ord a) ⇒ a → a → a`.

Easier said, high order functions enable currying. This is the ability to take a function with n parameters and turn it into a composition of n functions with each taking 1 parameter.

Another simple example in Haskell for curried functions.

```
multiThree :: (Num a) => a -> a -> a -> a
multiThree x y z = x * y * z

((multiThree 3) 5) 9 -- 135
-- Or
multiThree 3 5 9 -- 135
```

Now a currying example from go for comparison . Here is an operator function defined and will be applied to an operation on a slice of data. The mapper only get one argument that will be mapping function. That function gets a slice of data and operates on them in a separate step.

```
func mapper(operator func(interface{}) interface{}, m interface{}) (result
interface{}) {
    switch m.(type) {
    case []int:
        result := m.([]int)
        for i, n := range result {
            result[i] = operator(n).(int)
        }
        return result
    }
    return result
}

func main() {
    fmt.Println("Hello, playground")
    add3:= func(value interface{}) interface{} {
        switch value.(type) {
        case int:
            return value.(int) + 3
        }
        return nil
    }
    fmt.Println(mapper(add3, []int{1, 2, 3}))

}
```

Closure

A closure is a function, that makes use of free variables in its definition. A free variable is not bound. `(\x → x y)` In this example `y` is a free variable. The context defines if the variable is free.

```
f x = (\y -> x + y)
-- f returns a closure, because the variable x which is bound outside

f 2 5 -- 7
```

In the Go(lang) example of closure, a function intSeq is defined which returns a function, in which a anonymously body is defined. The returned function contains the variable `i` to form a closure.

```
func add_x(x int) func() int {

    return func(y int) int { // anonymous function

        return x + y

    }

}

add_2 := add_x(2)

add_5 := add_x(5)

add_2(5) // --> result 7

add_5(2) // --> result 7
```

Conclusion

As always, everything depends on your preferences and for which application you want to use the language. I personally think Haskell is more elegant and easier to read than Go(lang) and would rather use it. Haskell makes lazy immutable functional programming really elegant.

Bibliography

Web

- https://www.haskell.org/
- https://wiki.haskell.org/Haskell
- https://golang.org/
- http://www2.informatik.uni-freiburg.de/~thiemann/haskell/haskell98-report-html/basic.html
- https://learnxinyminutes.com/docs/haskell/
- http://learnyouahaskell.com/chapters
- https://gobyexample.com

Books

- Real World Haskell, O'Reilly (2008)

Lecture

- Johannes Weigend at Technical University of Applied Sciences Rosenheim

by Carlos Haselmaier