Functional Kotlin

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

What is Functional Programming?

Functional programming programming paradigm where programs are constructed by applying and composing functions.

Compare the imperative example (left) to the functional example (right). While the imperative program focuses on *how to do something*, the functional program focuses instead on the *operations to perform*.

```
var sum = 0
for (item in list) {
    if (item > 0) {
        sum += item * item
    }
}
```

```
list.filter { it > 0 }
    .map { it * it }
    .sum()
```

Functional code is often easier to read, and reason about [1].

Characteristics of Functional Programs

First-class functions means that functions are treated “first-class citizens” in the language. We can assign functions to variables, pass them as parameters, return functions from other functions.

Pure functions are functions that have no side effects. The output of a pure function is derived completely from its arguments.

Immutability suggests that an object cannot be changed after its creation; we prefer to instantiate new objects over mutating existing ones. This is important to avoid side-effects.

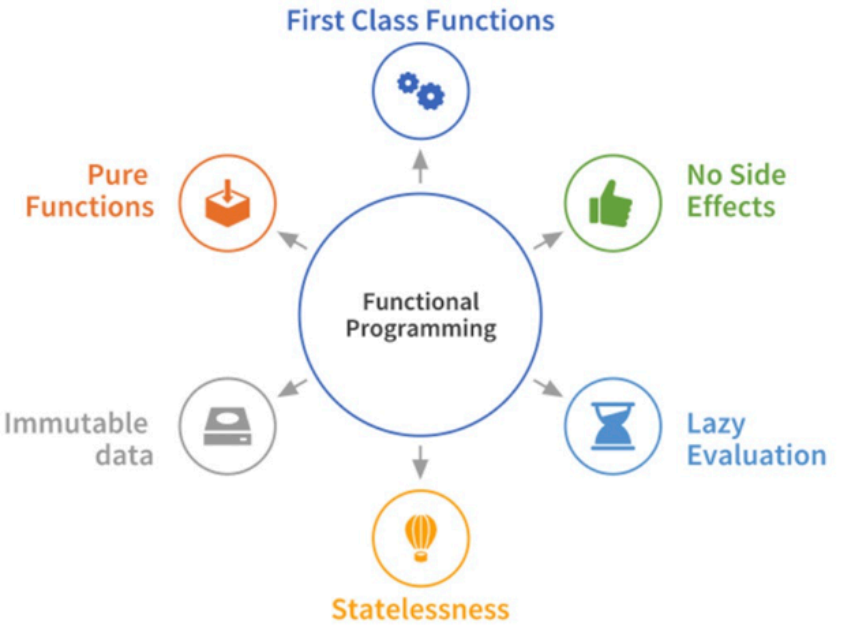

Figure 1: The Functional programming paradigm. From https://towardsdatascience.com/

Functional Kotlin

How is Kotlin functional?

Kotlin supports both OO and FP programming features.

- We mostly use it as an OO language, and apply FP concepts.
- Arrow is the standard library for Typed Functional Programming (FP) in Kotlin.

Functions in Kotlin are first-class citizens. The language uses a family of function types to represent functions and provides specialized language constructs for working with them e.g., lambdas, closures.

Higher-Order Functions

Functions that take other functions as arguments are called **higher order functions**. We can write our own higher-order functions if we wish.

The Kotlin `stdlib` includes some canonical operations:

- map allows us to perform a function over each element in a collection:
- filter lets us reduce a set based on some predicate function.
- fold creates a mutable accumulator, which is updated on each round of the `fold` and returns one value.
- reduce is an aggregate operation used to combine all elements of a collection into a single result by repeatedly applying a binary operation.

In Kotlin, these functions are applied to collection classes e.g., Lists, Sets.

Higher-Order Functions

map

`map` allows us to perform a function over each element in a collection.

The result of a `map` is a new collection containing the result of applying a function to each element.

```
val nums = (1..100).toList()

// numerical operations
val doubled = nums.map { it * 2 } // [2, 4, 6, 8, ...]
val squared = nums.map { it * it } // [1, 4, 9, 16, ...]

// string operations
val names = ["jerry", "alice", "marge", "susan"]
val caps = names.map { it.capitalize() } // ["Jerry", "Alice", ...]
```

Higher-Order Functions

filter

`filter` lets us apply a predicate function to each element in the collection, and only keeping those elements where the function returns true.

```
// numerical operations
val nums = listOf(1, 2, 3, 4, 5, 6)
val even = list.nums { it % 2 == 0 } // keep even numbers [2, 4, 6]

// string operations
val names = ["jerry", "alice", "marge", "susan"]
val select = names.filter { it ≠ "jerry" } // exclude jerry
```

Higher-Order Functions

fold

`fold` creates a mutable accumulator, which is updated on each round of the `fold` and returns one value:

```
// summing an array of integers
val list = listOf(1, 2, 3)
val folded = list.fold(0) { acc, x → acc + x } // 6
```

Note that fold lets you set an initial value for the accumulator. Also, your input type and accumulator types can be different.

```
// combining elements into a string with a prefix
val list = listOf(1, 2, 3)
val folded = list.fold("A") { acc, x → acc + x } // A123
```

Higher-Order Functions

reduce

`reduce` accumulates values starting with the first element and applying an operation to each element from left to right.

Note that unlike `fold`, `reduce` requires the input and output types to be the same. You also cannot initialize the accumulator.

```
val strings = listOf("a", "b", "c", "d")
val str = strings.reduce { acc, string → acc + string }) // abcd
```

Higher-Order Functions

others

`forEach` calls a function for every element in the collection.

```
val fruits = listOf("advocado", "banana", "cantaloupe" )
fruits.forEach { print("$it ") }  // advocado banana cantaloupe
```

`take` returns a new collection containing just the first n elements.

`drop` returns a new collection without the first n elements.

`first` and `last` return those respective elements.

`slice` allows us to extract a range of elements into a new collection.

```
val list = (1..50)
val e1 = list.take(10) // 1 2 3  ...  10
val e2 = list.drop(10)  // 11 12 13  ...  50

val r1 = list.first() // 2
val r2 = list.last() // 50
val r3 = list.slice(1..3) // 4 6 8
```

Higher-Order Functions

Composing Functions

You can also **compose** functions to perform multiple operations:

```
val list = listOf(1, 2, 3)
list.map { it * it }.map { it + 1 } // [2, 5, 10]
list.map { it * it + 1 } // [2, 5, 10]
```

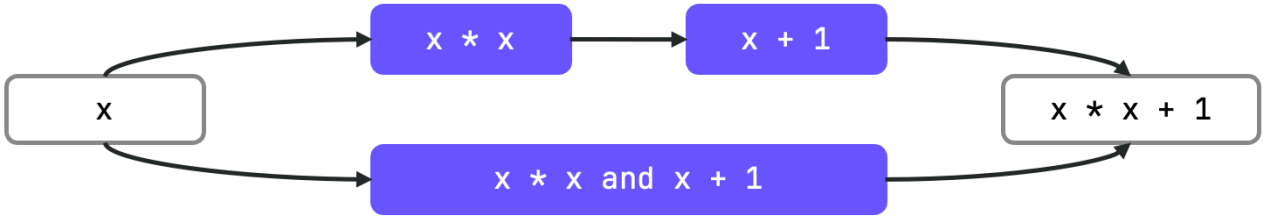

Figure 2: Both expressions return the same result.

Function Types

Kotlin uses function types for declarations that deal with functions. A function type has this form:

```
(Type, ...) → ReturnType
```

Examples

- `(Int) → String` is a function that accepts an Int and returns a String.
- `() → Unit` is a function that takes no arguments and returns Unit (nothing).

Guidelines

- All function types have a parenthesized list of parameter types and a return type: `(A, B) → C` denotes a type that represents functions that take two arguments of types `A` and `B` and return a value of type `C`.
- The list of parameter types may be empty, as in `() → A`.
- The `Unit` return type cannot be omitted.

Function Types

Instantiating a Function Type

We most often instantiate a function literal by providing a lambda representing the function body.

```
// repeatingString is a function reference
val repeat: (String, Int) → String = { str, i → str.repeat(i) }

// invoke the function normally
val s = repeat("abc", 3) // s == "abcabcabc"
```

> **Tip**
>
> Why would you do this instead of just defining the function normally? This notation can be helpful when we want to pass a reference to a function...

Lambda Expressions

Lambda expressions and anonymous functions are `function literals`. Function literals are functions that are not declared but are passed immediately as an expression.

```
max(strings, { a, b → a.length < b.length })
```

The function max is a higher-order function, as it takes a function value as its second argument. This second argument is an expression that is itself a function, called a function literal, which is equivalent to the following named function:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda Expressions

Syntax

The full syntactic form of lambda expressions is as follows:

```
val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }
```

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the ->.
- If the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

```
val sum = { x: Int, y: Int → x + y }
```

Lambda Expressions

if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses. This is called a `trailing lambda`.

```
val product = items.fold(1) { acc, e → acc * e }
```

If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

Lambda Expressions

Closures

A lambda expression or anonymous function (as well as a local function and an object expression) can access its closure, which includes the variables declared in the outer scope.

The variables captured in the closure can be modified in the lambda:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Bibliography

[1] J. Hughes, “Why Functional Programming Matters,” *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.