Object Meets Function

Do It with Functions

Steven Lolong

steven.lolong(at)uni-tuebingen.de

Programming Language Research Group
Tuebingen University

Nov. 20, 2023

Eberhard Karls
Universität
Tübingen

Summary

1 Function
2 First-class Function (FcF)
3 Higher-order Function (HoF)
4 Currying and Uncurrying
5 Function Application
6 Evaluation and Termination

Function

Function is an organized block that contains expression(s) to perform one or more action.

or **in Math**: to map from ***input*** to ***output***.

Function's Definition

- Function declaration with a name. In Scala, called **method**.
- Function expression without a name (anonymous / lambda). In Scala, called **function**.

Functions terms in this Praktikum

But, in this Praktikum use these terms to distinguish function:

- Function declaration with a name but not a member of an object or class, called **function**.
- Function declaration with a name and a member of an object or class, called **method**.
- Function expression without a name (anonymous),called **lambda**.

Method / Function

```
def funName [typParam] (param1: typ1, ..., paramN: typN) : retType =
  exp1
  ...
  expN
  lastExp
```

Listing: Method declaration format

def	method annotation (keyword)
funName	method's name
typParam	type paramenter
param1 ... paramN	parameter
typ1 ... typN	type of parameter
retType	type of return value
exp1 ... expN lastExp	list of expressions in the body of the method
lastExp	last expression is the return value for method. Type of lastExp must be the same with retType.

Method / Function - example

```
def multiply (fstNum: Int, sndNum: Int) : Int =
  val returnValue = fstNum * sndNum
  returnValue

val nine = multiply(3,3)
```

Listing: Method/function - example1

Method / Function - example

```scala
def idFun[A](a: A) : A = a

val nineIsNime = idFun(9)

val mulFun = idFun(multiply)
val anotherNine = mulFun(3,3)
```

Listing: Method/function - example2

The **identity function** is a function that always returns the value that was used as its argument, unchanged. That is, when f is the identity function, the equality $f(X) = X$ is true for all values of X to which f can be applied.

Method / Function - Nested

```
def multThenSquare(a: Int, b: Int) : Int =
  def squareIt(x: Int) : Int =
    x * x
  squareIt(a * b)

val _36 = multThenSquare(2,3)
```

Listing: Method/function - nested

Eberhard Karls
Universität
Tübingen

Function (a.k.a Lambda)

```
(param1: typ1, ... , paramN: typN) =>
  exp1
  ...
  expN
  lastExp
```

Listing: Lambda declaration format

param1 ... paramN parameter

=> separator between param and body

typ1 ... typN type of parameter

exp1 ... expN lastExp list of expressions in the body of the method

lastExp last expression is the return value for lambda.

Lambda - example

```scala
val add = (a: Int, b: Int) => a + b

val seven = add(3,4)
```

Listing: Lambda - example1

First-class Function (FcF)

FcF is a function that **only accepts values** as input (parameter) and returns **only values** as output.

FcF - example

```
def multiply (fstNum: Int, sndNum: Int) : Int =
  val returnValue = fstNum * sndNum
  returnValue

val nine = multiply(3,3)
```

Listing: FcF - example1

Is this an FcF?

```scala
def pow(n: Int, pw: Int): Int = pw match
  case 0 => 1
  case x if x < 0 => sys.error("It can only be raised to the power of positive numbers.")
  case 1 => n
  case x if x > 1 => n * pow(n, (pw-1))
```

Listing: Is this an FcF?

Higher-order Function (HoF)

HoF is a function that can takes value(s) or function(s) as parameter(s) and return a value or function.

HoF - Method / Function

```
def funName [typParam] (param1: typ1, ..., paramN: typN) : retType =
  exp1
  ...
  expN
  lastExp
```

Listing: Method declaration format

def	method annotation (keyword)
funName	method’s name
typParam	type paramenter
param1 ... paramN	parameter, **value(s) or function(s)**
typ1 ... typN	type of parameter
retType	type of return value
exp1 ... expN lastExp	list of expressions in the body of the method
lastExp	last expression is the return value for method. Type of lastExp must be the same with retType, can be **value(s) or function(s)**

HoF - Lambda

```
(param1: typ1, ... , paramN: typN) =>
  exp1
  ...
  expN
  lastExp
```

Listing: Lambda declaration format

param1 ... paramN parameter, **value(s) or function(s)**

=> separator between param and body

typ1 ... typN type of parameter

exp1 ... expN lastExp list of expressions in the body of the method

lastExp last expression is the return value for lambda.

Higher-order Function (HoF)

```
def hofArithOperator(a: Int, b: Int, f: (Int, Int) => Int) : Int =
  f(a,b)

def addOptr(a: Int, b: Int) : Int = a + b
def multOptr(a: Int, b: Int) : Int = a * b
def subtrOptr(a: Int, b: Int) : Int = a - b
def divOptr(a: Int, b: Int) : Int = b match
   case 0 => sys.error("arithmatic error, devide by zero")
   case _ => a / b

val twoPlusThree = hofArithOperator(2,3,addOptr)
```

Listing: HoF - Ex-1

Higher-order Function (HoF)

```
def urlBuilder(ssl: Boolean, domainName: String) : (String, String) => String =
  val schema =  if ssl then "https://" else "http://"
  (endPoint: String, query: String) => s"$schema$domainName/$endPoint?$query"

val domName = "www.example.com"
def getUrl = urlBuilder(true, domName)
val endPoint = "users"
val query = "id=1"
val url = getUrl(endPoint,query)
```

Listing: HoF - Ex-2

Higher-order Function (HoF)

```
def urlBuilder(ssl: Boolean, domainName: String) : (String, String) => String =
  val schema =  if ssl then "https://" else "http://"
  (endPoint: String, query: String) => s"$schema$domainName/$endPoint?$query"

val domName = "www.example.com"
def getUrl = urlBuilder(true, domName)
val endPoint = "users"
val query = "id=1"
val url = getUrl(endPoint,query)
```

Listing: HoF - Ex-2

Currying

Currying is a function that takes an input (one parameter) at a time.

```
def fHoFArithOptCurrying (a: Int) (b: Int) (f: (Int, Int) =>
     Int) : Int = f(a,b)
```

Listing: Currying

Take a parameter at a time

```
val needB = fHoFArithOptCurrying(4)
val needF = needB(3)
val ex2_seven = needF(addOptr)
```

Listing: Currying - function application1

Take all parameters at a time

```
val ex1_seven = fHoFArithOptCurrying(4)(3)(addOptr)
```

Listing: Currying - function application2

Uncurrying

Uncurrying is a function that takes all inputs (parameters) at a time.

```scala
def fHoFArithOptUncurrying (a: Int, b: Int, f: (Int,Int)=>
    Int) : Int = f(a,b)
```

Listing: Uncurrying

Take all parameters at a time

```scala
// val anError = fHoFArithOptUncurrying(4)
val ex3_seven = fHoFArithOptUncurrying(4, 3, addOptr)
```

Listing: Uncurrying - function application

Eberhard Karls
Universität
Tübingen

Uncurrying to Currying

In Scala, the wildcard '_' will make the **Uncurrying function** act like a **Currying function**.

```
def fHoFArithOptUncurrying (a: Int, b: Int, f: (Int,Int)=>
    Int) : Int = f(a,b)
```

Listing: Uncurrying

Take all parameters at a time

```
val needBUc = fHoFArithOptUncurrying(4, _ , _)
val needFUc = needBUc(3, _)
val exUc_seven = needFUc(addOptr)
```

Listing: Uncurrying - function application

Function application (Calling a function)

There are two type of function application:

- **Total application**, call a function with **complete** input (parameter).
- **Partial application**, call a function with **incomplete** input (parameter).

```
def fHoFArithOptUncurrying (a: Int, b: Int, f: (Int,Int)=>
    Int) : Int = f(a,b)
```

Listing: Uncurrying

Total application

```
val ex3_seven = fHoFArithOptUncurrying(4, 3, addOptr)
```

Listing: Function application - Total

Partial application

```
val needBUc = fHoFArithOptUncurrying(4, _ , _)
val needFUc = needBUc(3, _)
```

Listing: Function application - Partial

Evaluation and Termination

This topic aims to explain how an expression is evaluated and how it has ended. There are two types of evaluation:

- **Call by Value** (CbV), *immediately evaluate an expression to a value*. This is the default expression evaluation method in Scala.
- **Call by name** (CbN), substitute the variable with the expression and *evaluate it when needed*. The annotation to distinct is '**=>**'.

Evaluation and Termination

```
def addOptr(a: Int, b: Int) : Int = a + b
```

Listing: a square function

```
def sumOfSquareCbV(a: Int, b: Int) : Int =
  multOptr(a) + multOptr(b)

val cbv_exm = sumOfSquareCbV(2, 3+3)
```

Listing: CbV

```
def sumOfSquareCbN(a: Int, b: => Int) =
  multOptr(a) + multOptr(b)

val cbv_exm = sumOfSquareCbN(2, 3+3)
```

Listing: CbN

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)

Eberhard Karls
Universität
Tübingen

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)
4	(2 * 2) + multOptr(6)	(4) + multOptr(3+3)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)
4	(2 * 2) + multOptr(6)	(4) + multOptr(3+3)
5	(4) + multOptr(6)	(4) + multOptr(6)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)
4	(2 * 2) + multOptr(6)	(4) + multOptr(3+3)
5	(4) + multOptr(6)	(4) + multOptr(6)
6	(4) + (6 * 6)	(4) + (6 * 6)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)
4	(2 * 2) + multOptr(6)	(4) + multOptr(3+3)
5	(4) + multOptr(6)	(4) + multOptr(6)
6	(4) + (6 * 6)	(4) + (6 * 6)
7	(4) + (36)	(4) + (36)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	sumOfSquareCbV(2, 3+3)	sumOfSquareCbN(2, 3+3)
2	sumOfSquareCbV(2, 6)	multOptr(2) + multOptr(3+3)
3	multOptr(2) + multOptr(6)	(2 * 2) + multOptr(3+3)
4	(2 * 2) + multOptr(6)	(4) + multOptr(3+3)
5	(4) + multOptr(6)	(4) + multOptr(6)
6	(4) + (6 * 6)	(4) + (6 * 6)
7	(4) + (36)	(4) + (36)
8	40	40

Evaluation and Termination

Will all expressions terminate?

```scala
def loop : Int = loop

def exCbV (a: Int, b: Int) : Int = a
def exCbN (a: Int, b: => Int) : Int = a

val cbv = exCbV(4, loop)
val cbn = exCbN(4, loop)
```

Listing: cbv vs cbn

Eberhard Karls Universität Tübingen

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)
2	evaluate 4 then evaluate *loop*	evaluate 4 and binding *b* with *loop*

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)
2	evaluate 4 then evaluate *loop*	evaluate 4 and binding *b* with *loop*
3	evaluate *loop* because it is waiting for the value of loop	return 4 because it doesn't need b, so b has never evaluated

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)
2	evaluate 4 then evaluate *loop*	evaluate 4 and binding *b* with *loop*
3	evaluate *loop* because it is waiting for the value of loop	return 4 because it doesn't need b, so b has never evaluated
4	evaluate *loop*	

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)
2	evaluate 4 then evaluate *loop*	evaluate 4 and binding *b* with *loop*
3	evaluate *loop* because it is waiting for the value of loop	return 4 because it doesn’t need b, so b has never evaluated
4	evaluate *loop*	
5	...	

Evaluation and Termination

Table: CbV vs CbN

Step	CbV	CbN
1	exCbV(4, loop)	exCbN(4, loop)
2	evaluate 4 then evaluate *loop*	evaluate 4 and binding *b* with *loop*
3	evaluate *loop* because it is waiting for the value of loop	return 4 because it doesn't need b, so b has never evaluated
4	evaluate *loop*	
5	...	
6	infinite evaluate *loop*	