50.054 Introduction to Scala Part 1

ISTD, SUTD

Recap

- What are the three alternatives of a lambda term?

Recap

- What are the free variables in the following lambda term?

$$\lambda x.(x\ (\lambda y.y\ z)\ y)$$

Recap

- What are the two main rules to “execute” a lambda term program?

Recap

- What is the additional rule to avoid capturing the free variables?

Learning Outcomes

- Develop simple implementation in Scala using List, Conditional, and Recursion

What's Scala

A hybrid paradigm language

- Object Oriented
- Functional Programming

Scala is widely used in the indstry and academia

- Scala at Scale at Databricks
- Why Scala is seeing a renewed interest for developing enterprise software
- Who is using Scala, Akka and Play framework
- Type-safe Tensor

Projects that were implementd in Scala

- Akka - a concurrent/distributed frameworks
  - Pekko - a descendant of Akka
  - ZIO - a concurrent programming framework
- Kafka - a distributed event/message platform
- Spark - a in-memory parallel computing framework
- SynapseML - a distributed ML framework by Microsoft
- Optimus-cirrus - a set of software infrastructure that Morgan Stanley use internally to build and run highly-performant and parallelizable applications
- Many commercial projects in the banks, such as JP Morgan, HSBC, Deutsche Bank

How well are Scala Developers paid?

According to Glassdoor, in USA 2025 (compared to 2023), annual salary in thousand USD

Role	Min	Mean	Max	Num Jobs
Kotlin	79(78)	98(96)	120(121)	632
JavaScript	75(81)	99(107)	132(143)	14,000
Python	77(89)	100(113)	130(146)	99,999
Scala	122(113)	147(143)	178(183)	738
Golang	83(152)	107(187)	137(236)	994

Scala History

- Started in 2001 by Martin Odersky at EPFL
- Became popular in the academia in 2005
- Became more popular in the industry thanks to projects like Akka and Spark in 2009
- Scala Version 3 launched in 2021-2022

Scala Compiler and Executor

Given a `HelloWorldApp.scala` source file

```
object HelloWorldApp {
    def main(args:Array[String]) =
        println("hello world")
}
```

- `scalac` - the compiler

```
$ scalac HelloWorldApp.scala
```

generates `HelloWorldApp.class` and `HelloWorldApp.tasty`

- `scala` - the executor

```
$ scala HelloWorldApp
hello world
```

Scala REPL

- scala double up an REPL

```
$ scala
scala> println("hello world")
hello world

scala> val s = "hello"
val s: String = hello

scala> :t s
String
```

Scala Interpretor

Assuming we have `HelloWorldScript.scala`

```
println("hello world")
```

```
$ scala
scala> :load HellWorldScript.scala
hello world

scala> :exit

$ scala HellWorldScript.scala
hello world
```

Scala Imperative Programming

```
def isort(vals:Array[Int]):Array[Int] = {
    for (i <- 1 to (vals.length-1)) {
        var curr = i
        for (j <- i to 1 by -1) {
            if (vals(curr) > vals(j-1)) {
                val t = vals(curr)
                vals(curr) = vals(j-1)
                vals(j-1) = t
                curr = j-1
            }
        }
    }
    vals
}
```

- `def` defines a function. `:Array[Int]` is a type annotation.
- `1 to 3` generates a sequence with 1,2,3
- `var` defines a mutable variable, `val` defines an immutable variable.
- `Array[Int]` is a mutable array
- `return` can be omitted, `;` can be omitted.

Scala Object Oriented Programming

```
trait FlyBehavior {
    def fly()
}

abstract class Bird(species:String, fb:FlyBehavior) {
    def getSpecies():String = this.species
    def fly():Unit = this.fb.fly()
}

class Duck extends Bird("Duck", new FlyBehavior() {
    override def fly() = println("I can't fly")
})

class BlueJay extends Bird("BlueJay", new FlyBehavior() {
    override def fly() = println("Swwooshh!")
})
```

- `trait` defines an interface.
- class constructors are in-line.
- `Unit` is a type, similar to `void`
- Functions and methods' return types can be inferred by the compiler

Scala Functional Programming

- mapping Lambda Calculus to Scala

	Lambda Calculus	Scala
Variable	x	`x`
Constant	c	`1, 2, true, false`
Lambda abstraction	$\lambda x.t$	`(x:T) => e`
Function application	$t_1\ t_2$	`e1(e2)`
Conditional	*if* t_1 *then* t_2 *else* t_3	`if (e1) { e2 } else { e3 }`
Let Binding	*let* $x = t_1$ *in* t_2	`val x = e1 ; e2`
Recursion	*let* $f = (\mu g.\lambda x.g\ x)$ *in* $f\ 1$	`def f(x:Int):Int = f(x); f(1);`

where `T` denotes a type and `:T` denotes a type annotation. `e`, `e1`, `e2` and `e3` denote expressions.

Scala Functional Programming

Example

```
def fac(x:Int):Int = {
    if (x == 0) { 1 } else { x*fac(x-1) }
}

val result = fac(10)
```

Recall Strict and Lazy Evaluation

Common Rules

$$\texttt{(ifI)} \quad \frac{t_1 \longrightarrow t_1'}{\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3 \longrightarrow \mathit{if}\ t_1'\ \mathit{then}\ t_2\ \mathit{else}\ t_3}$$

$$\texttt{(ifT)} \quad \mathit{if}\ \mathit{true}\ \mathit{then}\ t_2\ \mathit{else}\ t_3 \longrightarrow t_2$$

$$\texttt{(ifF)} \quad \mathit{if}\ \mathit{false}\ \mathit{then}\ t_2\ \mathit{else}\ t_3 \longrightarrow t_3$$

$$\texttt{(OpI1)} \quad \frac{t_1 \longrightarrow t_1'}{t_1\ \mathit{op}\ t_2 \longrightarrow t_1'\ \mathit{op}\ t_2}$$

$$\texttt{(OpI2)} \quad \frac{t_2 \longrightarrow t_2'}{c_1\ \mathit{op}\ t_2 \longrightarrow c_1\ \mathit{op}\ t_2'}$$

$$\texttt{(OpC)} \quad \frac{\mathit{invoke\ low\ level\ call\ op}(c_1, c_2) = c_3}{c_1\ \mathit{op}\ c_2 \longrightarrow c_3}$$

$$\texttt{(Let)} \quad \mathit{let}\ x = t_1\ \mathit{in}\ t_2 \longrightarrow [t_1/x]t_2$$

$$\texttt{(unfold)} \quad \mu f.t \longrightarrow [(\mu f.t)/f]t$$

Recall Strict and Lazy Evaluation

- Lazy Evaluation

(β reduction) $(\lambda x.t_1)\ t_2 \longrightarrow [t_2/x]t_1$

(NOR) $$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

- Strict Evaluation

(β reduction) $(\lambda x.t_1)\ v_2 \longrightarrow [v_2/x]t_1$

(App1) $$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

(App2) $$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}$$

Where

(Values) $v ::= \lambda x.t \mid c$

Scala Lazy and Strict Evaluation

Let `f` be a non-terminating function

```
def f(x:Int):Int = f(x)
```

The following shows that the function application in Scala is using strict evaluation.

```
def g(x:Int):Int = 1
g(f(1)) // it does not terminate
```

On the other hand, the following code is terminating.

```
def h(x: => Int):Int = 1
h(f(1)) // it terminates!
```

Scala List

- `Nil` - an empty list.
- `List()` - an empty list.
- `List(1,2)` - an integer list contains two values.
- `List("a")` - an string list contains one value.
- `1::List(2,3)` - prepends a value `1` to a list containing `2` and `3`.
- `List("hello") ++ List("world")` - concatenating two string lists.
- Given a list `l`, and a val `x`, `x::l` takes $O(1)$.
- Given two lists `l1` and `l2`, `l1 ++ l2` takes $O(N)$ time where N is the size of `l1`.

Scala List is immutable

```
scala> val l = List(1,2,3)
val l: List[Int] = List(1, 2, 3)
scala> l(1)
val res0: Int = 2
scala> l(1) = 3
-- [E008] Not Found Error: ----------------------------------------------------
1 |l(1) = 3
  |^
  |value update is not a member of List[Int] - did you mean l.updated?
1 error found
```

We can iterate the list via for loop

```
scala> for (i <- l) { println(i) }
1
2
3
```

But it is not **fun** (pun intended)!

Scala List Pattern Matching

- `Nil` and `::` are not just value constructors, they are also known as the pattern constructors, used in pattern matching.
- We use pattern constructors to match a structured data.

```
scala> l match {
     | case Nil => "empty"
     | case (x::xs) => "not empty" }
val res0: String = not empty
```

- `match` keyword defines a pattern matching expression. `|`s are inserted by scala REPL for multiline expression.
- `l` is matched against two pattern alternatives
  - `case Nil => "empty"` is matched when `l` is an empty list, the RHS `"empty"` is the result.
  - `case (x::xs)) =>  "non empty"` is matched when `l` is not an empty list, the RHS `"not empty"` is the result.

Scala List Pattern Matching

```
def sum(l:List[Int]):Int = l match {
        case Nil => 0
        case (x::xs) => x + sum(xs)
    }
sum(List(1,2,3)) // gives us 6
```

- Patterns are tried from top to bottom.
- The second pattern captures sub parts of the `l` into pattern variables `x` and `xs`.

```
def first(l:List[Int]):Int = l match {
        case (x::xs) => x
    }
```

- Scala warns us that the pattern is not exhaustive.

Lambda Calculus with List and Pattern Matching

$$
\begin{array}{rrcl}
\texttt{(Lambda Terms)} & t & ::= & x \mid \lambda x.t \mid t\ t \mid \mathit{let}\ x = \ t\ \mathit{in}\ t \mid \mu f.t \mid \\
 & & & \mathit{Nil} \mid \mathit{Cons}\ t\ t \mid t\ \mathit{match}\ \{\overline{\mathit{case}\ p \Rightarrow t}\} \\
\texttt{(Values)} & v & ::= & c \mid \lambda x.t \mid \mathit{Nil} \mid \mathit{Cons}\ t\ t \\
\texttt{(Patterns)} & p & ::= & \mathit{Nil} \mid \mathit{Cons}\ p\ p \mid x \\
\texttt{(Substitutions)} & \theta & ::= & \overline{[t/x]}
\end{array}
$$

- variables appearing in patterns must be linear, i.e. $\mathit{Cons}\ x\ x$ is not allowed.
- $\overline{o}$ denotes a sequence of items $o_1, ..., o_n$.

Additional evaluation rules

$$
\texttt{(Match1)} \qquad \frac{t \longrightarrow t'}{t\ \mathit{match}\ \{\overline{p \Rightarrow t}\} \longrightarrow t'\ \mathit{match}\ \{\overline{\mathit{case}\ p \Rightarrow t}\}}
$$

$$
\texttt{(Match2)} \qquad \frac{\begin{array}{c} pm(v, p_i) = \theta \\ \forall j \in [1, i-1]\ pm(v, p_j)\ \texttt{fails to generate a substitution.} \end{array}}{v\ \mathit{match}\ \{\mathit{case}\ p_1 \Rightarrow t_1; ...; \mathit{case}\ p_n \Rightarrow t_n\} \longrightarrow \theta t_i}
$$

Substitution is extended

$$
\begin{array}{rcl}
[]t & = & t \\
[t_1/y, \overline{t/x}]t_2 & = & \overline{[t/x]}([t_1/y]t_2)
\end{array}
$$

Pattern matching function

$$
\begin{array}{rcl}
pm(\mathit{Nil}, \mathit{Nil}) & = & [] \\
pm(\mathit{Cons}\ t_1\ t_2, \mathit{Cons}\ p_1\ p_2) & = & pm(t_1, p_1) \cup pm(t_2, p_2) \\
pm(t, x) & = & [t/x]
\end{array}
$$

Lambda Calculus with List and Pattern Matching

$$
\begin{array}{ll}
(\mu sum.\lambda l.l\ match\{case\ Nil \Rightarrow 0; case\ (Cons\ x\ xs) \Rightarrow x + (sum\ xs)\})\ (Cons\ 1\ Nil) & \longrightarrow_{\texttt{unfold}+\alpha+\texttt{subst}} \\
(\lambda l.l\ match\{case\ Nil \Rightarrow 0; case\ (Cons\ x\ xs) \Rightarrow x + (sum_1\ xs)\})\ (Cons\ 1\ Nil) & \longrightarrow_{\beta} \\
[Cons\ 1\ Nil/l](l\ match\{case\ Nil \Rightarrow 0; case\ (Cons\ x\ xs) \Rightarrow x + (sum_1\ xs)\}) & \longrightarrow_{\texttt{subst}} \\
(Cons\ 1\ Nil)\ match\{case\ Nil \Rightarrow 0; case\ (Cons\ x\ xs) \Rightarrow x + (sum_1\ xs)\} & \longrightarrow_{\texttt{Match2}} \\
[1/x, Nil/xs](x + (sum_1\ xs)) & \longrightarrow_{\texttt{subst}} \\
1 + (sum_1\ Nil) & \\
\ldots & \\
1 + 0 &
\end{array}
$$

Where $sum_1 = (\mu sum.\lambda m.m\ match\{case\ Nil \Rightarrow 0; case\ (Cons\ x\ xs) \Rightarrow x + (sum\ xs)\})$

Quick Summary

We've covered

1. Scala Imperative and OOP
2. Scala Syntax
3. Scala FP vs Lambda Calculus
4. Scala Evaluation
5. Scala List and Pattern Matching