L4: Basic Evaluation

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

January 16th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Recap

- Iterator:
  - Iterate group of elements
- Decorator:
  - A function that modify features of that function

Let’s Dive into Func. Programming

Make Sure You Have Scala

- Please install it right now

- Try to run this following code:

```
object HelloWorld extends App {
println("Hello, World!")
}
```

- The code should print Hello, World!

REPL

- REPL
  - **R**epeat
  - **E**valuate
  - **P**rint
  - **L**oop

- Expression can be entered directly into the REPL

Expressions

- What are expressions?
  - 10
    - Expression that evaluate to 10, has type Int
  - 12 + 13 → 25
    - Expression that evaluate to 25, type Int
- You can bind a name to expression
  - def number = 10
    - This gives magic: Int
- You can combine expressions
  - number * 10
- Expression does not always have a type
  - 3*”Hello”

Expressions Definition

- This is called named expression
- You can think of this as a math function

- Example:
  - def cube(x: Double) = x*x*x
  - def ssc(x: Double, y: Double) = cube(x) + cube(y)

Substitution Model

- When evaluating an expression, you can use substitution
- Example: Assume def f(n:Int) = n*n
  - We want to evaluate f(2+1)

- First, 2+1 is evaluated to 3
  - Then, every time we see f as its expression
    - We replace it with 3
- f(2+1) → f(3)
  → {n*n} [n← 3]
  → 3*3
  → 9

Termination

- If everything is a function, when does the evaluation of an expression reduces to a value
- Question:
  - Does every expression reduce to a value in finite step?
- Let's look at this seemingly confused example:
  - def loop: Int = loop
- loop has a type int, but never terminate
- Our substitution model replaces loop with loop …
  - And this goes on indefinitely
- So, not every expression reduce to a value in finite step

Another Evaluation Strategy

- So far, we use the substitution model to evaluate exp.
- Let's experiment with a different strategy:
  - Idea: Pass the arguments into the function w/o reducing them

f(2+1) → {n*n} [n← 2+1]
→ (2+1)*(2+1)
→ 9

- This evaluation strategy yields the same result
- Why? Because our computation has no side effect!
  - I.e., the order of substitute vs. reduce does not affect the final result

Different Function-calling Style

- Call by value (CBV)
  - Reduce first, then substitute
- Call by name (CBN)
  - Substitute first, then reduce

- Both strategies should evaluate to the same final value

Theorem 4.1

- Both strategies reduce to the same final values as long as
  - All expressions involved are pure functions (i.e., no side effect)
  - Both evaluation terminates
- Furthermore:
  - If CBV of expression e terminates, then CBN of e terminates
  - Does not true for the other direction!

- CBV → every function's argument is evaluated once
- CBN → no evaluation if unused in the function body
- Scala is CBV by default
  - You can invoke CBN by annotating input param with the type
    - Def addTwo(x: => Int) = x+2

Let's Play Around

- Consider
  - def leftCBV(x:Int, y:Int) = x
  - def leftCBN(x:=> Int, y: =>Int) = x

- Try to call the two version with
  - left(1+1,loop) and left(loop, 1+1)

  - What happen?

Conditional Expressions

- Scala offers the if-then-else construct
  - It tell which ***expression*** to step to next
    - Vs. which statement/commands to proceed with

- Example
  - def abs(x:Int) =
    if (x<=0) –x else x

- Using the construct, we can say if (e1) e2 else e3 behave:
  - e1 => true [if(e1) e2 else e3] → e2
  - e1 => false [if(e1) e2 else e3] → e3

Example

- Let's evaluate abs(-40)

→ [if(x<=0) –x else x]/[x = - 40]
→ if(-40 <=0) – (-40) else - 40
→ if(true) –(-40)else -40
→ - (-40)
→ 40

- Let's try abs(5)

→ [if(x<=0) –x else x]/[x=5]
→ if(5 <=0) – (5) else 5
→ if(false) –(5)else 5
→ 5

More Complex Example

- def loop: Int = loop
- def goof(x:Int) = if (x<0) loop else 10

- What happen if we run goof (1) vs. good (-1)

Reduction on Boolean Expression

- Takes two basic values: True and False

- Evaluating the expression following normal logic op.

true → false

true && e → e

true || e → true

!false → true

false && e → false

false || e → e

What Does “def” Do?

- def binds and expression to a name

- So, fundamentally, def is a “by-name” type
  - The right-hand expression is not evaluated until used

- If we want to use a by-value form, use “val”

```
def foo1 = 11
val foo2 = 11
```

Example

- Suppose x:Boolean and y:Boolean
- We want to simulate && and ||
  - Remember that **they are short circuit**: false && loop = false

- Answer:
  - def and(x: Boolean, y => Boolean) = if(x) y else false
  - def or(x: Boolean, y => Boolean) = if(x) true else y

Nested Functions

- Example
  - def sumOfSquares(x:Int, y:Int) = {
    def sqr(t:Int) = t*t
    sqr(x) + sqr(y)
    }

- This helps namespace pollution
  - sqr only seen inside sumOfSquare
  - Also notice the last statement of {…} is the return value
    - I.e., it determine what sumOfSquare evaluates to

Blocks

- The following is valid

```
{
        val number = 10
        number+1
}
```

- Extendind this idea, we can do

```
def foo = {
        val number = 10
        number+1
}
```

- This binds foo to the expression inside the brace

Visibility

- Definition inside a block is only visible inside
- Definition inside shadows things defined outside the block

- Example: What is the outcome of

```
val x=5
val result = {
        val x = 6
        x+1
}
println(x)
println(result)
```

Before We Leave Today

In-class Exercise 4

- Implement sqrt using the Newton's method