Principles of (Functional) Programming

(4190.306)

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering

Seoul National University

Syllabus

- Lecture
  - Mon & Tue, 9:00 ~ 10:50 (302-208)
  - https://github.com/snu-sf-class/pp201602
- Instructor
  - Chung-Kil Hur
  - http://sf.snu.ac.kr/gil.hur/
- Teaching Assistant
  - Youngju Song
  - http://sf.snu.ac.kr/youngju.song/
- Grading
  - Attendance: 5%
  - Assignments: 25%
  - Midterm exam: 30%
  - Final exam: 40%

Introduction

Imperative vs. Functional Programming

- Imperative Programming
  - Computation by memory reads/writes
  - Sequence of read/write operations
  - Repetition by loop
  - More procedural
  - Easier to write efficient code

```
sum = 0;
i = n;
while (i > 0) {
  sum = sum + i;
  i = i - 1;
}
```

- Functional Programming
  - Computation by function application
  - Composition of function applications
  - Repetition by recursion
  - More declarative
  - Easier to write safe code

```
def sum(n) =
  if (n <= 0)
    0
  else
    n + sum(n-1)
```

Both Imperative & Functional Style Supported

- Many languages support both imperative & functional style
  - More imperative: Java, Javascript, C++, Python, …
  - More functional: OCaml, SML, Lisp, Scheme, …
  - Middle: Scala
  - Purely functional: Haskell

- Why Scala?
  - Equally well support both imperative & functional style
  - A lot of advanced features
  - Compatible with Java

Names, Functions and Evaluations

Values, Expressions, Names

- Types and Values
  - A type is a set of values
  - Int: {-2147483648,…,-1,0,1, …,2147483647} //32-bit integers
  - Double: 64-bit floating point numbers // real numbers in practice
  - Boolean: {true, false}
  - …
- Expressions
  - Composition of
    values, names, primitive operations
- Name Binding (= Programming)
  - Binding expressions to names
- Examples

```
def a = 1 + (2 + 3)
def b = 3 + a * 4
```

Evaluation

- Evaluation
  - Reducing an expression into a value
  - Strategy

1. Take a name or an operator (outer to inner)
2. (name) Replace the name with its associated expression
3. (name) Evaluate the expression
4. (operator) Evaluate its operands (left to right)
5. (operator) Apply the operator to its operands

- Examples

```
5+b ~ 5+(3+a*4) ~ … ~ 32
```

Functions and Substitution

- Functions
  - Expressions with Parameters
  - Binding functions to names

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

- Evaluation by substitution
  - …
  - (function) Evaluate its operands (left to right)
  - (function)
    Replace the function application by the expression of the function
    Replace its parameters with the operands

```
5+f(f(3)+1) ~ 5+f((3+a)+1) ~ … ~ 5+f(10) ~
5+(10+a) ~ … ~ 21
```

Simple Recursion

- Recursion
  - Use X in the definition of X
  - Powerful mechanism for repetition
  - Nothing special but just rewriting

```
def sum(n) =
  if (n <= 0)
     0
  else
     n + sum(n-1)
```

```
sum(2) ~ if (2<=0) 0 else (2+sum(2-1)) ~
2+sum(1) ~ 2+(if (1<=0) 0 else (1+sum(1-1))) ~
2+(1+sum(0)) ~ 2+(1+(if (0<=0) 0 else (0+sum(0-1))))
~ 2+(1+0) ~ 3
```

Termination/Divergence

Evaluation may not terminate

- Termination
  - An expression may reduce to a value

- Divergence
  - An expression may reduce forever

```
def loop: Int = loop

loop ~ loop ~ loop ~ …
```

Evaluation strategy: Call-by-value, Call-by-name

`f(e1,e2)`

- Call-by-value
  - Evaluate the arguments first, then apply the function to them
- Call-by-name
  - Just apply the function to its arguments, without evaluating them.

```
def square (x: Int) = x * x

[cbv]square(1+1) ~ square(2) ~ 2*2 ~ 4

[cbn]square(1+1) ~ (1+1)*(1+1) ~ 2*(1+1) ~ 2*2 ~ 4
```

CBV, CBN: Differences

- Call-by-value
  - Evaluates arguments once
- Call-by-name
  - Do not evaluate unused arguments

- Question
  - Do both always result in the same value?

Scala's evaluation strategy

- Call-by-value
  - By default
- Call-by-name
  - Use "=>"

```
def one(x: Int, y: =>Int) = 1

one(1+2, loop)

one(loop, 1+2)
```

Scala's name binding strategy

- Call-by-value
  - Use "val" (also called "field") e.g. `val x = e`
  - Evaluate the expression first, then binding the name to it
- Call-by-name
  - Use "def" (also called "method") e.g. `def x = e`
  - Just bind the name to the expression, without evaluating it
  - Mostly used to define functions

```
def a = 1 + 2 + 3
val a = 1 + 2 + 3 // 6
def b = loop
val b = loop

def f(a: Int, b: Int): Int = a*b - 2
```

Conditional Expressions

- If-else
  - `if (`b`)` e_1 `else` e_2
  - b : Boolean expression
  - e_1, e_2: expressions of the same type
- Rewrite rules:
  - `if (true)` e_1 `else` e_2 $\rightarrow$ e_1
  - `if (false)` e_1 `else` e_2 $\rightarrow$ e_2

```
def abs(x: Int) = if (x >= 0) x else -x
```

Boolean Expressions

- Boolean expression
  - `true, false`
  - `!b`
  - `b && b`
  - `b || b`
  - `e <= e, e >= e, e < e, e > e, e == e, e != e`
- Rewrite rules:
  - `!true → false`
  - `!false → true`
  - `true && b → b`
  - `false && b → false`
  - `true || b → true`
  - `false || b → b`

```
true && (loop == 1) ~ loop == 1 ~ loop == 1
```

Exercise: and, or

- Write two functions
  - `and(x,y) == x && y`
  - `or(x,y) == x || y`
  - Do not use `&&`, `||`

```
and(false,loop==1)
~ if (false) loop==1 else false
~ false

and(true,loop==1)
~ if (true) loop==1 else false
~ loop==1 ~ loop==1 …
```

Exercise: square root calculation

- Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  ??? // guess*guess is 99% close to x
def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2
def sqrtIter(guess: Double, x: Double): Double =
  ??? // repeat improving guess until it is good
enough
def sqrt(x: Double) =
  sqrtIter(1, x)
sqrt(2)
```

Blocks and Name Scoping

Blocks in Scala

- Block
  - ```
    { val x1 = e1
      def x2 = e2
      e
    }
    ```
  - Is an expression
  - Allow nested name binding
  - Allow arbitrary order of “def”s, but not “val”s (think about why)
    ```

Scope of names

- Block

```
val t = 0
def square(x: Int) = t + x * x
val x = square(5)
val r = {
  val t = 10
  val s = square(5)
  t + s
}
val y = t + r
```

- A definition inside a block is only accessible within the block
- A definition inside a block shadows definitions of the same name outside the block
- A definition inside a block is accessible unless it is shadowed
- A function is evaluated under the environment where it is defined, not the environment where it is invoked.

Rewriting for blocks

```
1: val t = 0
2: def f(x: Int) = t + g(x)
3: def g(x: Int) = x*x
4: val x = f(5)
5: val r = {
6:   val t = 10
7:   val s = f(5)
8:   t + s }
9: val y = t + r
```

- Evaluation by rewriting

```
[f=(x)t+g(x),g=(x)x*x],1 ~ […,t=0],2 ~ […],3 ~ […],4
~ […,x=25],5 ~ […]:[],6 ~ […]:[t=10],7
~ […]:[…,s=25],8 ~ […,r=35],9 ~ […,y=35],10
4: [f=…,g=…,t=0]:[x=5],t+g(x) ~…~ […]:[…],25
7: [f=…,g=…,t=0]:[x=5],t+g(x) ~…~ […]:[…],25
```

Semi-colons and Parenthesis

- Block
  - Can write two definitions/expressions in a single line using ;
  - Can write one definition/expression in two lines using (), but can omit () when clear

```
// ok
val r = {
  val t = 10; val s = square(5); t +
  s }
// Not ok
val r = {
  val t = 10; val s = square(5); t
  + s }
// ok
val r = {
  val t = 10; val s = square(5); (t
  + s) }
```

Exercise: Writing Better Code using Blocks

- Make the following code better

```
def isGoodEnough(guess: Double, x: Double) =
  guess*guess/x > 0.999 && guess*guess/x < 1.001
def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2
def sqrtIter(guess: Double, x: Double): Double = {
  if (isGoodEnough(guess,x)) guess
  else sqrtIter(improve(guess,x),x)
}
def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Solution

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double = {
    if (isGoodEnough(guess,x)) guess
    else sqrtIter(improve(guess,x),x)
  }
  def isGoodEnough(guess: Double, x: Double) = {
    val ratio = guess * guess / x
    ratio > 0.999 && ratio < 1.001
  }
  def improve(guess: Double, x: Double) =
    (guess + x/guess) / 2

  sqrtIter(1, x)
}

sqrt(2)
```

Recursion

Recursion needs care

- Summation function
  - Write a summation function `sum` such that `sum(n) = 1+2+…+n`
  - Test `sum(10),sum(100),sum(1000),sum(10000), sum(100000), sum(1000000)`
  - What’s wrong? (Think about evaluation)

Recursion: Try 1

```
def sum(n: Int): Int =
  if (n <= 0) 0 else (n+sum(n-1))
```

Recursion: Tail Recursion

```scala
import scala.annotation.tailrec

def sum(n: Int): Int = {
  @tailrec def sumItr(res: Int, m: Int): Int =
    if (m <= 0) res else sumItr(m+res,m-1)
  sumItr(0,n)
}
```

Higher-Order Functions

Functions as Values

- Functions
  - Functions are normal values of function types $(A_1,\ldots,A_n\text{=>}B)$.
  - They can be copied, passed and returned.
  - Functions that take functions as arguments or return functions are called higher-order functions.
  - Higher-order functions increase code reusability.

Examples

```
def sumLinear(n: Int): Int =
  if (n <= 0) 0 else n + sumLinear(n-1)

def sumSquare(n: Int): Int =
  if (n <= 0) 0 else n*n + sumSquare(n-1)

def sumCubes(n: Int): Int =
  if (n <= 0) 0 else n*n*n + sumCubes(n-1)
```

Q: How to write reusable code?

Examples

```
def sum(f: Int=>Int, n: Int): Int =
  if (n <= 0) 0 else f(n) + sum(f, n-1)

def linear(n: Int) = n
def square(n: Int) = n * n
def cube(n: Int) = n * n * n

def sumLinear(n: Int) = sum(linear, n)
def sumSquare(n: Int) = sum(square, n)
def sumCubes(n: Int) = sum(cube, n)
```

Anonymous Functions

- Anonymous Functions
  - Syntax

    $(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow e$

    or

    $(x_1, \ldots, x_n) \Rightarrow e$

```
def sumLinear(n: Int) = sum((x:Int)=>x, n)
def sumSquare(n: Int) = sum((x:Int)=>x*x, n)
def sumCubes(n: Int) = sum((x:Int)=>x*x*x, n)
```

Or simply

```
def sumLinear(n: Int) = sum((x)=>x, n)
def sumSquare(n: Int) = sum((x)=>x*x, n)
def sumCubes(n: Int) = sum((x)=>x*x*x, n)
```

Exercise

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0

def product(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) * product(f, a+1, b) else 1
```

DRY (Do not Repeat Yourself) using a higher-order function, called “mapReduce”.

Exercise

```
def mapReduce(combine:(Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,inival,f,a+1,b))
  else inival
}

def sum(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x+y,0,f,a,b)

def product(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x*y,1,f,a,b)
```

Parameterized expression vs. values

- Functions defined using “def” are not values but parameterized expressions.
- Anonymous functions are values.
- But, parameterized expressions are implicitly converted to values.
- Explicit conversion: f _
- Anonymous functions can be seen as syntactic sugar:
  ```
  (x:T)=>e
  ```
  is equivalent to
  ```
  { def __noname(x:T)=>e; __noname _ }
  ```
- One can even write a recursive anonymous function in this way.

- Q: what’s the difference between param. exps and function values?
  A: functions values are “closures” (ie, param. exp. + env.)
- Q: how to implement call-by-name?
  A: The argument expression is converted to a closure.

Currying

Motivation

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0
def linear(n: Int) = n
def square(n: Int) = n * n
def cube(n: Int) = n * n * n
def sumLinear(a: Int, b: Int) = sum(linear, a, b)
def sumSquare(a: Int, b: Int) = sum(square, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
```

We want the following. How?

```
def sumLinear = sum(linear)
def sumSquare = sum(square)
def sumCubes = sum(cube)
```

Solution

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF
}

def sumLinear = sum(linear)
def sumSquare = sum(square)
def sumCubes = sum(cube)
```

Benefits

```
def sumLinear = sum(linear)
def sumSquare = sum(square)
def sumCubes = sum(cube)

sumSquare(3,10) + sumCubes(5,20)
```

We don't need to define the wrapper functions.

```
sum(square)(3,10) + sum(cube)(5,20)
```

Multiple Parameter List

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF
}
```

We can also write as follows.

```
def sum(f: Int=>Int): (Int,Int)=>Int =
  (a,b) => if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Or more simply:

```
def sum(f: Int=>Int)(a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Currying and Uncurrying

- A function of type

$$(T_1,T_2,\ldots,T_n)\text{=>}T$$

can be turned into one of type

$$T_1\text{=>}T_2\text{=>}\ldots\text{=>}T_n\text{=>}T$$

- This is called "currying" named after Haskell Brooks Curry.
- The opposite direction is called "uncurrying".

Exercise

Curry the mapReduce function.

Solution

```
def mapReduce(combine:(Int,Int)=>Int,inival: Int)
             (f: Int=>Int) (a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,inival)(f)(a+1,b))
  else inival
}

// need to make a closure since mapReduce is param. code.
def sum = mapReduce((x,y)=>x+y,0) _

// val is better than def. Think about why.
val product = mapReduce((x,y)=>x*y,1) _
```