L2: Intro to Functional Prog.

Rachata Ausavarungnirun

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

September 18th, 2020

Architecture Research Group

SSE, TGGS

Recap

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 number: Int
- You can combine expressions
  - number * 10
- Expression does not always have a type
  - 3*"Hello"

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

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 on CBV/CBN

- 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
  - def loop:int = loop
- Try to call the two version with
  - leftCBV(1+1,loop) and leftCBV(loop, 1+1)
  - leftCBN(1+1,loop) and leftCBN(loop, 1+1)

  - What happen?

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

Intro to Functional Language

Functional Language

- Now that you installed Scala
  - Let’s formally define what is a functional language

- Program is created by applying functions

- Function definitions are tree of expressions
  - Eventually reduce to a value in most cases

- Function is a first-class citizen
  - We will discuss this in a few weeks

Repetion (Instead of a Loop)

- Recursion is the answer to looping
  - Calling the function itself again → next iteration
- You can write
  def func_name(v1: Type, …, vN: Type): RetType = {
  …
  }
  - The return type is optional unless your function is recursive

Example

- Let’s say you want to write pow(x: Int, y: Int): Int
  - For x!=0 and y>=0, and evaluates to x^y
- You can write
  def pow(x: Int, y: Int): Int = if(y ==0) 1 else pow(x, y-1)*x
- While Scala has a loop, recursion is a more powerful construct than a loop
  - So, use recursion for now
- Functions are values → defining a function just binds the expression to a name
  - No actual execution happen during the binding process

Compound Data

- So far, we have talked about a single data item
  - Number
  - Boolean
  - Conditionals
  - Variables
  - Functions
- Let’s look at a way to build up data with multiple parts

Tuple

- A fixed number of items, each can have different type
- Example:
  - (“hello”, 1) will results in a value of type (String, Int)

Tuple – A Pair

- For a pair, the rule is simple
  - Value: if e1 → v1 and e2 → v2, then (e1,e2) → (v1,v2)
  - Type: if e1 : t1 and e2 → t2, then (e1, e2) : (t1, t2)

- You can bind a pair to a name
  - Val t = (“hello”, 1)
- Accessing each component by
  - t._1 for the first item
  - t._2 for the second item
- Generally, you can use ._k to get the k^{th} item
- You can also unpack the pair using val
  - val (a, b) = t

Examples

- def swap(p: (String, Int)) = (p._2, p._1)
- def swapInts(p: (Int, Int)) = (p._2, p._1)
- def sum(p: (int, int)) = (p._1 + p._2)
- def order(p: (int, Int)) = if (p._1 < p._2) p else swapInts(p)

- Then, you can have k-tuple by using this same concept
- You can also have nested tuples (tuples within a tuple)

List

- Lists in Scala and most functional language are front-access lists
- List() makes an empty list
  - Type List[Nothing]
  - We can force a type by saying List[Type]
    - Example: List(): List[Int]
- You can also make a list with elements in it
  - List(1,2,3,1,2)
- You can stick element to the front of the list
  - You will get a brand new list
- Specifically
  - If e1→v, e2→l = [v1, v2, … vn], where e1: T and e2: List[T] then e1::e2 has the type List[T] and represent [v, v1, v2, … vn]

Accessing a List

- Let's discuss some standard functions for a list
- Check if a list is empty
  - If L is a list, then L.isEmpty is true if L is empty
- Access the head
  - If L is non-empty, L.head evaluates to the head element of L
  - Else, you get an exception
- Access the tail
  - If L = [v1, v2, … ,vn], then L.tail is the list [v2, …, vn]
  - Notice how you got the remaining elements

Examples

- How can I summation items in my list xs?
  - def sumList(xs: List[Int]): Int = if (xs.isEmpty) 0 else xs.head + sumList(xs.tail)
- How can I create a descending list of range n?
  - def descRange(n: Int): List[Int] = if(n==0) Nil else n::descRange(n-1)
- How can I concatenate two lists together?
  - Def concat(xs: List[Int], ys: List[Int]): List[Int] = if(xs.isEmpty) ys else (xs.head)::concat(xs.tail, ys)

Pattern Matching

- Can I do switch-case in functional programming?
- Yes! Use
  - selector match { alternative }

- Example
  - Def sumList(xs: List[Int]): Int = xs match {
    case Nil => 0
    case h::t → h + sumList(t)
    }
- This pattern-match your list with each cases

Pattern Matching

- Benefits:
  - Gets a warning if you are missing any cases
  - Gets a warning if you have duplicate cases
  - Most concise, and hopefully more readable
    - Compared to tons of functions …
- Example: You can also use pattern matching to break down tuples in a list
  - Let's say you have a list of (Int, String)
    ```
    xs match {
    case (number, name)::t => …
    …. // Other cases here
    }
    ```
  - This will break down to the numbers and names for you

Styles

- For the following code

```
def countUpFrom1(n: Int): List[Int] = {
  def count(from: Int, to: Int): List[Int] =
    if (from == to) Nil else from :: count(from+1, to)
  count(1, n)
}
```

- In this case, you definitely know to = n so you can write

```
def countUpFrom1(n: Int): List[Int] = {
  def count(from: Int): List[Int] =
    if (from == n) Nil else from :: count(from+1)
  count(1)
}
```

In-class Exercise 2

- Implement sqrt using the Newton’s method

More Pattern Matching

Why Is This Code Bad?

- ```
  def badMin(xs: List[Int]): Int =
    if (xs.isEmpty) {
      2147483647 // really bad idea but what can i do?
    }
    else if (xs.tail.isEmpty) {
      xs.head
    } else if (xs.head < badMin(xs.tail)) {
      xs.head
    } else {
      badMin(xs.tail)
    }
  ```
- What if we do val x = badMin(List(1,2,3,…,30))
  vs. val x = badMin(List(30,29,…,1))
  ```

Better Code

- def betterMin(xs: List[Int]): Int =
```
if (xs.isEmpty) {
  2147483647 // really bad idea but what can i do?
} else if (xs.tail.isEmpty) {
  xs.head
} else {
  val tailMin = betterMin(xs.tail)
  if (xs.head < tailMin) xs.head else tailMin
}
```
- This code call the function once, instead of twice
- Still, we have not handled the empty list case
  - Options come to the rescue!

Options

- Option is a type
  - Option[T]
- Think of it as Option[T] is either
  - ***None*** that expresses emptiness
  - ***Some(v: T)*** that keeps a value v of type T

Options – Usage and Examples

- val x: Option[String] = None
- val y: Option[String] = Some("hi")
- val z: Option[(Double, String)] = Some((3.14, "Pi"))
- val q: Option[List[Double]] = Some(List(3.1, 2.5, 9.0))
- val r: Option[List[Double]] = None

Options – Accessing Options' Values

- If t: Option[T] then
- t.isEmpty: Boolean and t.nonempty: Boolean indicates whether t is empty or non-empty
- If t is non-empty and Some(v: T) then t.get evaluates to v
  - Throws an exception otherwise
- To avoid the exception, you can use t.getOrElse(whenEmpty: T): T
  - This is similar to t.get, but if empty the expression evaluates to whenEmpty
- Pattern matching also works with options

```
def addOne(x: Option[Int]): Option[Int] = x match {
    case None => None
    case Some(value) => Some(1+value)
}
```

Let’s Fix BetterMin

- Let’s get rid of that one weird case when xs is empty

- def betterMin(xs: List[Int]): Option[Int] =

```
  if (xs.isEmpty) None
  else { val tlAns = betterMin(xs.tail)
  if (tlAns.nonEmpty && tlAns.get < xs.head)
    tlAns
  else
    Some(xs.head)
  }
```

- We can also separate the empty and non-empty cases

Let’s Fix BetterMin

- def betterMin(xs: List[Int]): Option[Int] =

```
    if (xs.isEmpty) None else {
    def minNonEmpty(ys: List[Int]): Int =
        if (ys.tail.isEmpty)
            ys.head
        else {
            val tlAns = minNonEmpty(ys.tail)
            if (tlAns < ys.head) tlAns else ys.head
        }
    Some(minNonEmpty(xs))
  }
```

Tail Recursion

Tail Recursion

- Let's see how to evaluate fac(4) from
  def fac(n: Int): Int =
  if (n==0) 1 else n * fac(n – 1)

- See how the expression keeps growing?
- During this time, Scala needs to remember all these values
- **Q:** Can we rewrite the code so that it does not grow?

Tail Recursion

- Let's use tail recursion

- def facTail(n:Int) = {
    def tailFac(n: Int, prod: Int): Int =
      if (n==0) prod else tailFac(n-1, prod*n)
    tailFac(n, 1)
  }

```
def facTail(n:Int) = {
    def tailFac(n: Int, prod: Int): Int =
        if (n==0) prod else tailFac(n-1, prod*n)
    tailFac(n, 1)
}
```

Tail Recursion

- The difference here is that the last line of the function is a call to a function
  - Not to itself but to a tail call
  - This is call *tail recursive*
- Benefits:
  - Stack frames can be recycled
  - Compile to a very nice iterative program with no additional state on each stack call
    - Reduce burden on the compiler
- In the previous example, *prod* is the accumulator
  - Accumulate the answer we have so far instead of waiting for the call to return

More Example

- How can I make a tail recursive out of
```
def sum(xs: List[Int]): Int =
   if (xs.isEmpty) 0 else xs.head + sum(xs.tail)
```

-
```
def sum(xs: List[Int]): Int = {
   def tailSum(ys: List[Int], acc: Int): Int =
      if (ys.isEmpty)
         acc
      else
         tailSum(ys.tail, acc + ys.head)
   tailSum(xs, 0)
}
```

Common Things in Tail Recursion

- When we are at the base case, the helper function returns the accumulator
- Accumulator stores the partial computation we have seen so far

- This tail-call is very similar but more general to a while loop
  - Why? Tail call can actually call to other functions

In-class Exercise 3

- Write the following functions:
  - sumPairList(xs: List[(Int, Int)]): Int adds up all the numbers (both in the first and second coordinates).
  - firsts(xs: List[(Int, Int)]): List[Int] returns a list that extracts the first coordiate.
  - seconds(xs: List[(Int, Int)]): List[Int] returns a list that extracts the second coordiate.
  - pairSumList(xs: List[(Int, Int)]): (Int, Int) returns a pair where the first number is the sum of the first coordiates, and the second number is the sum of the second coordiates

- Submit them to in-class exercise 3

Before We Leave Today

In-class Exercise 4

- Write a function ***def find(xs: List[(Int, String)], key: Int): Option[String]*** that takes in a list of key-value (Int,String)-pairs and returns the string value matching the given integer key. It should return None if nothing matches it.
- Write a function ***def rev(xs: List[Int]): List[Int]*** that takes a list and produces the reverse of the input list. Can you write it as a tail-recursive function?
- Write a function ***def fib(n: Int): Long*** that computes the n-th Fibonacci number in a tail-recursive manner.