L6: Even More Func. Programming

Rachata Ausavarungnirun

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

January 23rd, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Let’s Pick Up from Tuesday

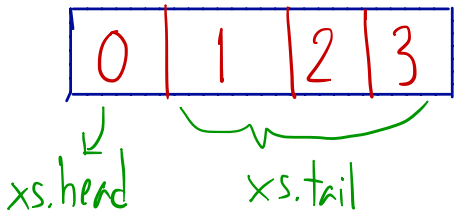
- We talked about recursion

- We talked about a tuple and a list

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)
    }
```



* Call badMin twice

- 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 expressing emptiness (Nothing in here)
  - Some(v: T) keeping 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

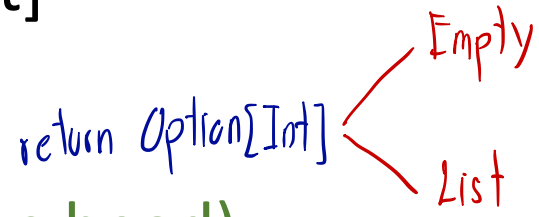
```
def addOne(x: Option[Int]): Option[Int] = x match {
    case None => None
    case Some(value) => Some(1+value)
}
```

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

- 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 - Take memories
- **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)
  }

Accumulator

f(4) -> tailFac(4,1) -> if (4==0) 1 else tailFac(4-1, 1·4)

tailFac(3,4) -> if(3==0) 4 else tailFac(3-1, 4·3)

tailFac(2,12) -> if(2==0) 12 else tailFac(2-1, 12·2)

tailFac(1,24) -> if(1==0) 24 else tailFac(1-1, 24·1)

tailFac(0,24) -> if(0==0) 24 else tailFac(-1,0)

-> 24

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) — Base Case
        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

Before We Leave Today

In-class Exercise 6

- 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.