Programming Languages: Recitation 4

Goktug Saatcioglu

02.21.2019

1 HW3 Problem 1

- We know that for the and operator if the first argument is false then the whole expression is false. Furthermore, for the and operator if the first argument is true then the whole expression is equivalent to the truth value of the second argument. This gets us the following solution.

```
def and(e1: Boolean, e2: Boolean): Boolean = {
  if (e1) e2 else false
}
```

- We know that for the or operator if the first argument is true then the whole expression is true. Furthermore, for the or operator if the first argument is false then the whole expression is equivalent to the truth value of the second argument. This gets us the following solution.

```
def or(e1: Boolean, e2: Boolean): Boolean = {
  if (e1) true else e2
}
```

2 HW3 Problem 2

- The idea is simple: calculate the next value in the sequence until we fall under a certain tolerance value specified by the user. So we calculate the next value in the sequence by doing

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n}$$

and the condition of our while loop should check for

$$\left|x_n^2 - c\right| < \epsilon$$

meaning we put the condition in the `...` section of our `while(...) +++` and then put the update statement in the `+++` section of the loop. This gets us the following solution.

```
def squareRoot(c: Double, epsilon: Double): Double = {
    require (c >= 0) // makes sure that we don't call squareRoot with
                     // negative values
    require (epsilon >= 0) // the error bound should also be positive

    var xn = 1.0

    // Newton's method given below
    while (Math.abs(xn * xn - c) < epsilon) {  // while not in tolerance
      xn = xn - (xn * xn - c) / (2 * xn)       // calculate next step
    }

    xn
  }
```

3 HW3 Problem 3

- Let's start with `isStrictlySorted`. To know whether a list a is strictly sorted we have to loop through every element and check whether $a[i] < a[i+1]$. We firstly note that if a list has length 1 then by definition it must be strictly sorted. Next, if a list has length greater than 1 then we must compare every element to the next element except for the last element in the list (since the last element has no next element). So there are two ways to approach the problem. The first is to consider the corner case and if its true then return true else continue and compare every element. If at any point $a[i] > a[i+1]$ then return false. Else at the end of the loop return true. The second way is to create a fresh variable $i = 0$ and use a while loop to increment this variable as long as the strictly sorted condition is met. Thus, if at the end of the loop we have i equal to the list's length we will say its strictly sorted and otherwise its not. Both approaches work and we give the second approach below.

```
def isStrictlySorted(a: Array[Int]): Boolean = {
    require (a != null) // make sure that 'a' is non-null

    var i = 0
    var aLen = a.length

    while (i < aLen && (i == aLen - 1 || a(i) < a(i + 1))) {
      i = i + 1
    }

    i == a.length
  }
```

- Before beginning to implement our `binarySearch` method let's consider what invariants there are based on the description of the assignment. Binary search requires that we have two indicies `l` (for left) and `r` (for right) which we move based on some conditions. What is clear is that `l` is initialized to 0 and `r` to the size of the list. Furthermore, `l` can only increase, `r` can only decrease and `l` is always less than or equal to `r`. Thus, we get the following first invariant

$$0 \leq l \leq r \leq a.length.$$

Now let's consider what we know for `l`. Since we want the smallest index i such that $a(i) \geq x$ we know that at any point either all value to the left of l are less than x or l is at its starting point such that nothing to its left is less than x. Thus, we get the following second invariant

$$l = 0 \vee a(l-1) < x.$$

Now we move onto `r`. The argument here is similar to that of `l`. Either `r` is equal to its starting point such that nothing to its right is greater than x or we have moved `r` such that everything to its right is greater than x. Thus, we get the following third invariant

$$r = a.length \vee a(r+1) > x.$$

Question: Why is it only enough to check for $l-1$ and $r+1$? Answer: Because the array is strictly sorted. If $a(l-1) < x$ then $a(l-2) < x$ and $a(l-2) < x$ until $a(0) < x$.
We have three invariants that are standard for binary search. However, we see that we want the smallest index i such that $a(i) \geq x$ and if such an index does not exist then we return the length of the list. So we must only set the index of `l` to `l=m+1` if $a(m) < x$ and otherwise set the index of `r` to `r=m`. This way we can get the smallest index i such that $a(i) \geq x$ because instead of searching for the index i by doing `r=m-1` we are considering the points `r=m-1` and its neighbor `r=m` where the latter is

the smallest index greater than `r=m-1`. So we can now derive the final invariant. We know that at any given point we either have not found any such index meaning `r` is equal to the list's length, or we have found $a(i) = x$ which can only be achieved via index `l` (because we use the left to search and the right to keep track of the smallest index greater than what we want) or `r` is an index of the list such that $x < a(r)$ (the reasoning behind this was discussed in the previous sentence). This, we the following final invariant

$$r = a.length \vee x = a(l) \vee x < a(r).$$

Bringing everything together, we get the following invariant using answer to the question.

```
def binarySearch(x: Int, a: Array[Int]): Int = {
  require (a != null && isStrictlySorted(a)) // make sure that 'a' is non-null
                                             // and sorted

  var l: Int = 0         // left
  var r: Int = a.length  // right

  // the (inductive) loop invariant
  def invariant: Unit = {
    assert (0 <= l && l <= r && r <= a.length)       // invariant 1
    assert (0 >= l || a(l - 1) < x)                  // invariant 2
    assert (r >= a.length - 1 || x < a(r + 1))       // invariant 3
    assert (r == a.length || x == a(l) || x < a(r))  // invariant 4
                                                     // i.e. how we find the result
  }

  while (l < r) {
    invariant                // check invariant for every iteration of the loop

    val m = l + (r - l) / 2  // find a middle index safely

    if (a(m) < x) {          // if we know that everything to the left is < x
      l = m + 1
    } else if (a(m) > x) {   // if we know that everything to the right is > x
      r = m                  // we find the next smallest index
                             // i.e. a[r+1] > a[r] >= x
    } else {                 // we found our solution
      return m
    }

  }

  invariant // the invariant must hold upon exiting the loop

  l         // this is our answer
}
```

4 Recursion and Tail Recursion

- Why recursion? Conceptually it is easier to reason about compared to loops. However, there is overhead everytime a recursive call is made and sometimes we get stack overflow.
- A tail-recursive subroutine is one in which no additional computation ever follows a recursive call.

- For tail-recursive subroutines, the compiler can reuse the current activation record at the time of the recursive call, eliminating the need to allocate a new one. This optimization is called tail call elimination.
- Most compilers in modern languages will do this optimization.
- Example: Consider adding up numbers in an interval [x, y].

```
def addInterval(x: Int, y: Int): Int = {
  if (x == y) x else x + addInterval(x+1, y)
}
```

- Is this function tail recursive? No, each call will create a new activation record on the stack. We will not stop popping off the stack until the last call is made. As can be seen, we will easily get stack overflow as the interval we wish to consider grows.
- Tail recursive version is given below.

```
def addInterval(x: Int, y: Int): Int = {
  def addIntervalTr(x: Int, y: Int, acc: Int): Int = {
    if (x == y) x + acc else addIntervalTr(x+1, y, x + acc)
  }
  addIntervalTr(x, y, 0)
}
```

- Now we have a tail recursive implementation. We can check if this is so using the `@tailrec` annotation.
- The Scala compiler will detect the tail recursion and performs tail-call elimination. Effectively, tail call elimination yields an implementation that is equivalent to one that uses a loop.

5 OCaml Programming

- Lists: `1 :: 2 :: 3 :: [] ;;`, `[1; 2; 3] ;;`
- Tuples: `let p = (1, 2) ;;`, `(1, 2, 3) ;;`, `1, 2 ;;`, `fst p ;;`, `snd p ;;`
- Function types: `let plus x y = x + y ;;` gets us `val plus : int -> int -> int`
- Most of the time we don't have side effects. However, they are sometimes necessary such as in printing or I/O.
- Recursion: `let rec fac = fun x -> fun acc -> if x = 0 then acc else fac (x - 1) (acc * x)`, equivalently we get

```
let rec fac x =
  let rec fac_tail x acc =
    if x = 0 then acc else fac_tail (x - 1) (x * acc)
  in
  fac_tail x 1
```

- Currying and partial function application: `let plus = fun x -> fun y -> x + y` and `let plusOne = plus 1`

- Pattern matching: See exercise.
- Exercise: Write a recursive OCaml function that gives the product of a list of numbers.

```
let rec product xs = match xs with
| [] -> 0
| hd :: tl -> hd * product tl
```

- Is this function tail recursive? No. What else is wrong with it? The correct solution is given below.

```
let product xs =
  let rec product_tr xs acc = match xs with
  | [] -> 0
  | [a] -> a * acc
  | hd :: tl -> product_tr tl (hd * acc)
  in
  product_t xs 1
```

- Bonus question: What is the type signature of this function? Hint: `*` expects an integer on its left and right operands.
- Harder exercise: Write a tail recursive OCaml function that gives the product of the positive numbers of a list.

```
let positive_product xs =
  let rec product_tr acc = function
  | [] -> 0
  | [a] -> if a > 0 then a * acc else acc
  | hd :: tl -> if hd > 0 then product_tr (hd * acc) tl else product_tr acc tl
  in
  product_tr 1 xs
```

- One more bonus: Write the same program above but now use pattern gaurds.

```
let positive_product xs =
  let rec product_tr acc = function
  | [] -> 0
  | [a] when a > 0 -> a * acc
  | [a] when a < 0 -> acc
  | hd :: tl when hd > 0 -> product (hd * acc) tl
  | _ :: tl -> product_tr acc tl
  in
  product_tr 1 xs
```

- _ indicates the wildcard pattern.