Programming Languages: Recitation 7

Goktug Saatcioglu

03.14.2019

1 HW5 Problem 1

1. For this question we need to expand on the multiplicative identity of the binomial coefficients given in the hint.

$$\binom{n}{k} = \frac{n\binom{n-1}{k-1}}{k} = \frac{n\frac{(n-1\binom{n-2}{k-2})}{k-1}}{k} = \frac{n(n-1)\binom{n-2}{k-2}}{k(k-1)} = \dots$$

Which hints to us that

$$\binom{n}{k} = \frac{n(n-1)(n-2)\dots}{k(k-1)(k-2)\dots}$$

until we reach the base cases of $\binom{n}{n}$ or $\binom{n}{0}$. This pretty much shows us what our answer should look like. We need 2 accumulator variable `n_acc` and `k_acc` which we multiply by `n` and `k` until we reach one of the base cases which at that point we can return `n_acc / k_acc`. Note that we could have also used the other recursive definition for the binomial coefficient but then we would need to use `Double` types which are suscpetible to rounding error. The above implementation allows us to keep everything as `Int` and then do one integer division that also gaurantees us an `Int` making it a better implementation. As a last note, we require that $0 \leq k \leq n$ since otherwise our function is undefined.

```
def binom(n: Int, k: Int): Int = {
  require (0 <= k && k <= n)
  @tailrec
  def binomTail(n: Int, k: Int, n_acc: Int, k_acc: Int): Int = {
    if (n == k || k == 0) n_acc / k_acc
    else binomTail(n - 1, k - 1, n_acc * n, k_acc * k)
  }

  binomTail(n, k, 1, 1)
}
```

2. For this question it is best to think about what the base cases are then build the answer up from there. We know that if we have 0 or 1 we get 0, if we have 2 we get 1. So these are our base cases for our pattern matching. Next, we have to figure out what to do with $T(n-1) + T(n-2) + T(n-3)$ such that it is tail recursive. Consider using three auxillary variables: `t1`, `t2` and `t3`. Then to get the n-th tribonnaci number we need to add the first three tribonnaci numbers, then the next three and so forth until we get to the n-th number. So we can use our counter variable to count down while using our auxillaries to "bottom-up" build the n-th tribonnaci number. The solution is given below. As a last note, we require that $n \geq 0$ since otherwise our function is undefined.

```
def trib(n: Int): Int = {
  require (0 <= n)
  @tailrec
  def tribTail(n: Int, t1: Int, t2: Int, t3: Int): Int = {
    n match {
      case 0 | 1 => 0
      case 2 => t1
```

```
      case _ => tribTail(n - 1, t1 + t2 + t3, t1, t2)
    }
  }

  tribTail(n, 1, 0, 0)
}
```

2 HW5 Problem 2

1. For this problem we see that we take a list of pairs and return a pair of ists where the first list contains the first element of the pairs and the second list contains the second element of the pairs. The non tail recursive implementation is given below.

```
let rec unzip (xys: ('a * 'b) list) : 'a list * 'b list = match xys with
| [] -> [], []
| (x, y) :: tls -> let (xs, ys) = unzip tls
                   in
                   x :: xs, y :: ys
```

   We can also turn this into a tail recursive implementation by utilizing the `reverse` function given in part 3.

```
let reverse (x: 'a list) : 'a list =
  let rec rev acc = function
    | [] -> acc
    | x :: xs -> rev (x :: acc) xs
  in rev [] x

let rec unzip (xys: ('a * 'b) list) : 'a list * 'b list =
  let rec unzip_tr acc1 acc2 = function
  | [] -> reverse acc1, reverse acc2
  | (x, y) :: tls -> unzip_tr (x :: acc1) (y :: acc2) tks
  in
  unzip_tr [] [] xys
```

2. This question is straightforward to think about. All we need to do is use a counter variable and start counting as we build the answer list by considering candidate elements from the original list. If at any point the counter equals n, i.e. the number we need to drop, then we don't pick up the element we are considering, reset the counter to 1 and continue building the list. If the counter does not equal n then we can pick up the current element we are considering, increment the counter and continue building the list. At the end of the algorithm we will have the answer to the question. The non tail recursive implementation is given below.

```
let drop (n: int) (xs: 'a list) : 'a list =
  let rec dropper c = function
  | [] -> []
  | x :: xs -> if c = n then dropper 1 xs else x :: dropper (c + 1) xs
  in
  if n <= 0 then xs else dropper 1 xs
```

We can also turn this into a tail recursive implementation by utilizing the `reverse` function given in part 3.

```
let reverse (x: 'a list) : 'a list =
  let rec rev acc = function
    | [] -> acc
    | x :: xs -> rev (x :: acc) xs
  in rev [] x

let drop (n: int) (xs: 'a list) : 'a list =
  let rec dropper c acc = function
  | [] -> reverse acc
  | x :: xs -> if c = n then dropper 1 acc xs else dropper (c + 1) (x :: acc) xs
  in
  if n <= 0 then xs else dropper 1 [] xs
```

3. There are a few approaches to this question. The easiest way to get a tail recursive implementation is to reverse the first list and then traverse it while concatenating elements from it to the second list. The reason this works is intuitive. The first element to be concatenated to the second list must be the last element of the first list. The same reasoning applies to the second element to be concatenated and so forth until we put all elements of the first list in their correct place. Since the `cons` operation, i.e. `::`, takes constant time we can simply traverse the first list and use the second list as the initial value of our accumulator. This way we get a tail recursive implementation. Note that we could have also declared a new empty accumulator and done two list traversals, one on the reverse of the second list and one on the reverse of the first list, too but the solution given here is a simplification of this approach. The OCaml code is given below.

```
let reverse (x: 'a list) : 'a list =
  let rec rev acc = function
    | [] -> acc
    | x :: xs -> rev (x :: acc) xs
  in rev [] x

let concat (xs: 'a list) (ys: 'a list) : 'a list =
  let rec concat_tr acc = function
  | [] -> acc
  | hd :: tl -> concat_tr (hd :: ans) tl
  in
  concat_tr ys (reverse xs)
```

3 From Recusion to Folding

- In the lecture notes we are presented with a way to go from a recursive implementation on lists to a fold left/fold right implementation on lists. If we have

```
let rec f xs =
  match xs with
  | [] -> (* xs is empty *)
    (* result for base case *)
    val_for_empty_list
  | hd :: tl -> (* xs is non-empty *)
```

```
    (* solve the problem for tl recursively and store result in res *)
    let res = f tl in
    (* compute actual result for whole list from hd and res *)
    work_using_hd_and_res
```

then, in general, we will have

```
let f xs =
  List.fold_right
    (fun hd res -> work_using_hd_and_res)
    xs
    val_for_empty_list
```

as a fold right implementation.

- Similarly, if we have

```
let f xs =
  let rec f_tr xs acc =
    match xs with
    | [] -> (* xs is empty *)
      (* result for base case *)
      acc
    | hd :: tl -> (* list is non-empty *)
      (* do work with hd and acc *)
      let acc_new = work_using_hd_and_acc in
      (* call the helper recursively on tl
      with the updated accumulator acc_new *)
      f_tr tl acc_new
  in
  f_tr xs val_for_empty_list
```

then, in general, we will have

```
let f xs =
  List.fold_left (fun acc hd -> work_using_hd_and_acc)
  val_for_empty_list
  xs
```

as a fold left implementation.

- Let's practice this approach on the last question in homework 5. The non tail recursive version concatenating two-lists is as follows:

```
let rec concat xs ys = match xs with
| [] -> ys
| x :: xs1 -> x :: concat xs1 ys
```

which can be written as

```
let concatter xs ys =
  let rec concat xs = match xs with
  | [] -> ys
  | x :: xs1 -> x :: concat xs1
  in
  concat xs
```

Here we see that `val_for_empty_list=ys` and `work_using_hd_res=hd :: res` where `res=f tl=concat xs1`. The trick is to notice that `ys` can be viewed as a constant so that we can realize we are only operating on a single list. We can then write this function using a fold right which is shown below.

```
let concat xs ys =  List.fold_right
  (fun hd res -> hd :: res)
  xs
  ys
```

- We take the same approach and apply it to our tail recursive solution. Recall that the tail recursive solution to the question was

```
let reverse x =
  let rec rev acc = function
    | [] -> acc
    | x :: xs -> rev (x :: acc) xs
  in rev [] x

let concat xs ys =
  let rec concat_tr ans = function
  | [] -> ans
  | hd :: tl -> concat_tr (hd :: ans) tl
  in
  concat_tr ys (reverse xs)
```

which can then be written as

```
let reverse x =
  let rec rev acc = function
    | [] -> acc
    | x :: xs -> rev (x :: acc) xs
  in rev [] x

let concatter xs ys =
  let concat xs =
    let rec concat_tr xs acc = match xs with
    | [] -> acc
    | hd :: tl -> concat_tr tl (hd :: acc)
    in
    concat_tr (reverse xs) ys
  in
  concat xs
```

Here we see that `val_for_empty_list=acc=ys` and `work_using_hd_and_res=hd :: acc` where, again, `acc=ys`. Also, note that we operate on `reverse xs` so we now get `xs = reverse xs`. The trick is to, again, notice that `ys` can be viewed as a constant so that we can realize we are only operating on a single list. Pulling out the `ys` then allows us to get into the form we would like to reason about. We can then write this function using a fold left which is shown below.

```
let concat xs ys = List.fold_left
  (fun acc hd -> hd :: acc)
  ys
  (reverse xs)
```

We can also have a left fold implementation for reverse which is left to you as an exercise. The solution is given below.

```
let reverse xs =
  List.fold_left
  (fun acc hd -> hd :: acc)
  []
  xs
```

4 Algebraic Data Types

- ADTs provide type constructors for building user-defined immutable tree-like data structures.
- They are known as variant types and (disjoint) sum types. Intuitively, you can think of ADTs as a mix of enumeration types and record types that you'll find in most imperative languages.
- Together with pattern matching, ADTs enable very powerful programming techniques.
- The lecture notes provide the example for the tree ADT for binary search trees, a polymorphic version of the ADT and how we can pattern match on the ADT to do operations such as checking whether the tree is sorted.
- Another nice aspect of ADTs is that equality = is defined structurally on ADT values, similar to lists and tuples. This means that two ADTs are equal if they both have the same structure and contents.
- Let's use ADTs to solve the following problem: Implement an ADT to represent arithmetic expressions containing polynomials of arbitrary degrees. That is we will have expressions of the form $2\times x^2+3\times x^{12}$ where $2 \times x^2$ and $3 \times x^{12}$ are sub-expressions and 2 and 3 are sub-sub expressions and x^2 and x^{12} are also sub-sub-expressions. We proceed as follows.

```
type expr =
      | Num of float         (* a constant: c *)
      | Var                  (* a variable: x *)
      | Mult of expr * expr  (* multiplcation: *)
                             (* e1 * e2 *)
      | Add of expr * expr   (* addition: *)
                             (* e1 + e2 *)
      | Pow of expr * int    (* exponentiation: *)
                             (* e1^n *)
```

We see that our expressions consist of constants, variables, multiplcation of expressions, addition of expression and the exponentiation of expressions to some power. Using this ADT, how can we compute the derivative of such expressions? The derivative of numbes are 0, the derivative of variables are a constant 1.0, the derivative of addition is the derivative of the left and right sub-expressions, the derivative of multiplication is $(fg)' = f'g + fg'$ by the product rule and the derivative of exponentiation is $(f^n)' = n(f^{n-1})f'$ by the chain rule and the power rule. So we can combine all of this together and actually compute the derivative of such arithmetic expression expressed in our ADT. We get the following implementation.

```
let rec derivative = function
| Num _ -> Num 0.0
| Var -> Num 1.0
| Add (e1, e2) -> Add (derivative e1, derivative e2)
| Mult (e1, e2) -> Add (Mult (derivative e1, e2), Mult(e1, derivative e2))
| Pow (e1, n) ->
    Mult (Mult (Num (float_of_int n), Pow (e1,  n - 1)),
          derivative e1)
```

There may be one problem with our implementation, what could it be? After evaluating the derivative we could end up with expressions that can be simplified such as $f \times 0$ or $f + 0$ and we would like to simplify these expressions. How can this be done? We know that there are a limited amount of cases we can simplify and they all revolve around multiplication or addition. If we multiply by zero we get zero, if we multiply by one we get the left/right expression, if we multiply two numbers we can simplify, if we add by zero we get the left/right expression, if we add two numbers we can simplify and any other expression we cannot simplify. Using this knowledge we implement the function `simplify_top` which is given below.

```
let simplify_top = function
| Mult (Num 0.0 as zero, _) | Mult(_, (Num 0.0 as zero)) -> zero
| Mult (Num 1.0, e2) | Mult (e2, Num 1.0) -> e2
| Mult (Num c1, Num c2) -> Num (c1 *. c2)
| Add (Num 0.0, e2) | Add (e2, Num 0.0) -> e2
| Add (Num c1, Num c2) -> Num (c1 +. c2)
| e -> e
```

Now all we have to do is keep calling `simplify_top` on an expression `e` and all of its sub-expression until we reach a fix-point. That is, we keep simplifying `e` until simplifying it further just yields the same expression `e'`. The implementation of this is given below.

```
(** Simplify expression [e0] by applying [simplify_top]
  * recursively until a fixpoint is reached *)
let rec simplify e0 =
  let e = simplify_top e0 in
  let e = match e with
  | Add (e1, e2) -> Add (simplify e1, simplify e2)
  | Mult (e1, e2) -> Mult (simplify e1, simplify e2)
  | Pow (e1, n) -> Pow (simplify e1, n)
  | e -> e
  in
  if e = e0 then e else simplify e
```

Finally, we compute the derivative of an expression by calling `derivative` followed by `simplify` to get a nice expression. This looks like as follows.

```
(** Compute derivative [e'] of an expression [e]
  * and simplify [e'] *)
let derivative e =
  e |> derivative |> simplify
```

The `|>` isthe reverse application operator in OCaml which can be read as take `e` and use it in the function `derivative` and then take the output of that and use it in the function `simplify`.