Programming Languages: Recitation 8

Goktug Saatcioglu

04.04.2019

1 HW6 Problem 1

1. Last recitation we went over how to convert any general recursive list algorithm into a fold left or fold right. So we could simply use that knowledge here to take whatever solution we had for HW5. Another approach is to consider a function that given an element in the original list splits it into two and adds the result to two accumulator variables. We know that the original list is of pairs so we have `fun (x, y)`. Furthermore, we need to produce two lists so then the next parameter should be two lists giving us `fun (x, y) (xs, ys)`. Since we return a pair of lists we then get the function `fun (x, y) (xs, ys) -> (x :: xs, y :: ys)`. Getting the full answer then simply involves filling out the remaining bits of a fold right.

```
let unzip xys =
  List.fold_right (fun (x, y) (xs, ys) -> (x :: xs, y :: ys)) xys ([], [])
```

2. We wish to implement a fold right using a fold left. A simple way to think about this answer is to notice that a fold left will always reverse a list (why is this the case?). Since we operate on individual elements of a list the results of a fold left will be in the reverse order of the results of a fold right. So we can then simply reverse the input list first using a fold left where the fold function is `cons` and then carry out the operation on the individual elements of this reversed list. This then gives us the answer that a fold right would give.

```
let fold_right op xs z =
  List.fold_left (fun xs x -> x :: xs) [] xs |>
  List.fold_left (fun acc x -> op x acc) z
```

3. The idea behind this question is that we need to keep track of what we saw last as we traverse a list so that we can compare it against the current value. So we can use an accumulator that holds onto the last seen value and check if the relationship between the last seen value by doing `p acc x`. If `p acc x` is true we continue with `x` now becoming `acc` and otherwise we can return false as there is no point to checking the rest of the list. This gives us our answer except there is a slight complication as we need an initial value for our traversal. So we need to consider the case when the list is empty and return if so and otherwise we split the list into its head and tail and call the function as follows: `in_relation_tr head tail`. The complete solution is given below.

```
let in_relation p xs =
  let rec in_relation_tr acc = function
  | [] -> true
  | x :: xs -> if p acc x then in_relation_tr x xs
               else false
  in
  match xs with
  | [] -> true
  | x :: xs -> in_relation_tr x xs
```

2 HW6 Problem 2

1. To get a tail recursive implementation we know that an accumulator variable is necessary. Furthermore, we know that we wish to flatten lists of arbitrary depth. One way to do this is by appending the contents of the nested list to the rest of the list everytime we see a nesting. For example, `[[1, 2], 3, 4]` would then become `[1, 2, 3, 4]` and `[[1], 2], 3, 4]` would become `[[1], 2, 3, 4]` in the first step and then become `[1, 2, 3, 4]` in the second step. Since a nested list is just a list of `NList`'s (via the ADT definition) the standard OCaml append operator `@` will work. The only problem is that we we wish to append a nested list to the "shallowest" level of the `nlist` which cannot be done due to the ADT definition. This can be overcome by placing the initial list `xs` into a list to get `[xs]` so we can now freely append any two `nlist`'s. Overall, the idea is to only update the accumulator when we see an `Atom` as this is the flattest a list can be and other wise everytime we see `NList xs :: tl` we append `xs` and `tl` to remove one level of depth from the nested tree. Recursive applications of this algorithm will eventually flatten the list and add every element to the head of our accumulator variable. As a final note, we again have to reverse the result. Can you see why? The reason is we are building the list in the reverse order (which is another hint in the question). The answer is given below.

```
type 'a nlist =
  | NList of ('a nlist) list
  | Atom of 'a

let flatten xs =
  let rec flatten_tr acc = function
  | NList hd :: tl -> flatten_tr acc (hd @ tl)
  | Atom hd :: tl -> flatten_tr (hd :: acc) tl
  | [] -> List.rev acc
  in
  flatten_tr [] [xs]
```

As a challenge you can also try implementing this function without the append operation. Your implementation must still be tail-recursive. As a hint, the way to think about this is to declare a two accumulator variables where one keeps track of further work that needs to be done while the other keeps track of the current work being done. That is, we wish to keep flattening a sub-list until it is completely flat and then continue with other sub-lists in the nested list to flatten all sub-lists to one big list. The solution to the challenge question is given below.

```
let flatten xs =
  let rec flattener todo acc = function
    | NList hd :: tl -> flattener (tl :: todo) acc hd
    | Atom hd :: tl -> flattener todo (hd :: acc) tl
    | [] -> match todo with
            | hd :: tl -> flattener tl acc hd
            | [] -> List.rev acc
  in
  flattener [] [] [xs]
```

a. An in-order traversal fold proceeds as follows: apply operation on all the elements of the left sub-tree, then apply operation on current node in tree and then apply operation on all the elements of the right sub-tree. Thus, we get a natural solution that involves following the definition of an in-order traversal and using the ADT.

```
type tree =
  | Leaf
  | Node of int * tree * tree

let rec fold op z t =
  match t with
  | Leaf -> z
  | Node (x, l, r) -> let l' = fold op z l in
                      fold op (op l' x) r
```

b. The question text gives us a hint. We need to get a list of elements of the tree that is equivalent to the in-order traversal of tree using our fold function from above. So we basically need to specify what `op` our function gets. What could this be? It is, of course, going to be `cons`. Are we done? No, since the output from consing every element while we do a in-order traversal will lead to the reverse of the list we want. This is because `cons` always adds an element to the head of a list meaning the element we want to appear first will actually be at the end of the list. Thus, we reverse the result we obtain from the `fold` to get the correct answer.

```
let list_of_tree t =
  fold (fun xs x -> x :: xs) [] t |> List.rev
```

Bonus: Could we have also reversed the tree first and then done the `fold`? Yes, and as a bonus exercise implement this function `reverse` that reverse the in-order traversal of a tree. (This was in midterm.)

c. This time we can again use fold and compare our current value to an accumulator variable. However, we also need to keep track of whether at some point the sortedness property is violated. Thus, we have an accumulator variable that is actually the pair `(prev, p)` where `p` is true as long as it was true and `prev < x` where `x` is the current value we are considering. This gives us the `op` for our `fold` where the idea is to update `p` such that if the previous value indicated to us that the tree was not sorted then `p` should also indicate so. Otherwise, if `p` was `true` then we need to compare `prev` and `x` to make sure the current element in the tree is sorted. Finally, we need an initial value for the accumulator and this will be the tuple `(min_int, true)` as we know that the empty tree is sorted and its lower bound is the minimum possible machine integer. The solution to the question is given below.

```
let is_sorted t =
  let res, _ =
    fold (fun (prev, p) x ->
      (x, p && prev < x))
    (min_int, true)
    t
  in
  res
```

3 Memory Management

- Done in class.

4 Lambda Calculus

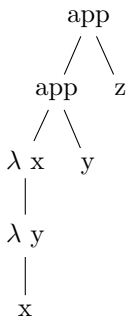
- In lecture 7 the Lambda Calculus was introduced along with Church encodings. Below are a few exercises to practice important concepts introduced from that lecture.

- Question: Consider the lambda expression

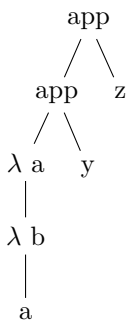
$$\texttt{((}\lambda\ \texttt{x y. x) y) z.}$$

What are the set of free variables of this expression? Perfom alpha-renaming to avoid variable capturing.

- Answer: We look at the AST of the expression.



From the AST, we see that the set of free variables are $\{y, z\}$ while the only bound variable if $\{x\}$. We can then rename the AST appropriately to get an alpha-renaming which look as follows.



Thus, the alpha-renaming of the lambda expression is

$$\texttt{((}\lambda\ \texttt{x y. x) y) z} \xrightarrow{\alpha} \texttt{((}\lambda\ \texttt{a b. a) y) z.}$$

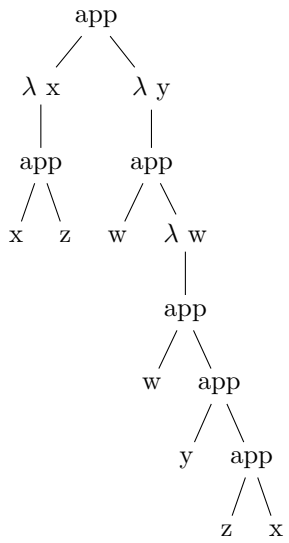
Note that the lambda expression given here was the definition of the Church encoding `true`.

- Question: This is a more involved alpha-renaming question but the principles to solve it are the same as above. Given the lambda expression

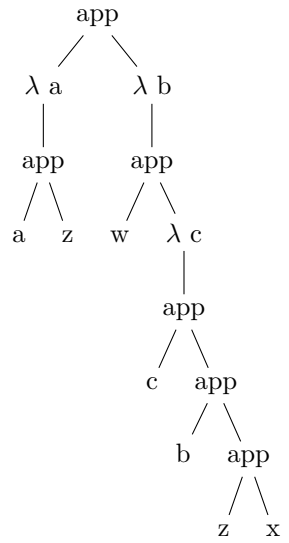
$$\texttt{(}\lambda\ \texttt{x. x z) (}\lambda\ \texttt{y. w (}\lambda\ \texttt{w. w (y (z x))))}$$

identify the set of free variables of this expression and perfom alpha-renaming to avoid variable capturing.

- Answer: We again look at the AST of this expression.



From the AST, we see that the set of free variables are {the first z, the first w, the second z, the second x} and the bound variables are {the first x, the second w, y}. We can then rename the AST appropriately to get an alpha-renaming which look as follows.



Thus, the alpha-renaming of the lambda expression is

`(λ x.  x z) (λ y.  w (λ w.  w( y (z x)))` $\xrightarrow{\alpha}$ `(λ a.  x a) (λ b.  w (λ c.  c (b (z x)))).`

- Question: We already have encodings for the logical operations `true`, `false`, `and`, `or`, `not` and `if-then-else`. How can we define a new lambda term that is an encoding for the logical operation `xor`? That is `xor` should evaluate to `true` only if one of its arguments is `true`.
- Answer: For a moment let's step away from the lambda calculus and think about how we would define `xor` in OCaml. We know that if the first argument is true then the whole expression is only true if the second argument is false. Similarly, if the first argument is false then the whole expression is only true if the second argument is true. So we split the code into two cases: when the first argument is true we

return the negation of the second argument so that if the second argument is false then `xor` evaluates to true and if the first argument is false then we return the second argument so that `xor` evaluates to true if the second argument is false. The implementation is given below.

```
let xor a b = if a then not b else b
```

This gives us an easy solution to the question which can then be translated into the lambda calculus. Thus, the Church encoding for `xor` is

$$\texttt{xor}=(\lambda\ \texttt{a b.}\ \ \texttt{ite a (not b) b}).$$

Note that we can simplify this expression so as to not use `ite` (if-then-else). Since `true` and `false` already behaive like `ite` how could we simplify this expression? The idea is, if `a` is true we evaluate `(not b)` and if `a` is false then we evaluate `b`. So by definition, our `xor` encoding becomes

$$\texttt{xor}=(\lambda\ \texttt{a b.}\ \ \texttt{a (not b) b}).$$

- Question: Using any of the two definitions of `xor` given above show that `xor (not false) false` evaluates to `true` in either normal-order or applicative-order.
- Answer: We use the simplified definition of `xor`, so `xor` is defined as

$$\texttt{xor}=(\lambda\ \texttt{a b.}\ \ \texttt{a (not b) b}).$$

Then proceeding in normal-order we get

$$\begin{aligned}
\texttt{xor (not false) false} &= (\lambda\ \texttt{a b.}\ \ \texttt{a (not b) b})\ \texttt{(not false) false} && \text{def. of } \texttt{xor}\\
&\xrightarrow{\beta} (\lambda\ \texttt{b.}\ \ \texttt{(not false) (not b) b})\ \texttt{false} && \text{reduce}\\
&\xrightarrow{\beta} \texttt{((not false) (not false) false)} && \text{reduce}\\
&= (((\lambda\ \texttt{b x y.}\ \ \texttt{b y x})\ \texttt{false})\ \texttt{(not false) false}) && \text{def. of } \texttt{not}\\
&\xrightarrow{\beta} ((\lambda\ \texttt{x y.}\ \ \texttt{false y x})\ \texttt{(not false) false}) && \text{reduce}\\
&\xrightarrow{\beta} ((\lambda\ \texttt{x.}\ \ \texttt{false y (not false)})\ \texttt{false}) && \text{reduce}\\
&\xrightarrow{\beta} \texttt{(false false (not false))} && \text{reduce}\\
&= ((\lambda\ \texttt{x y.}\ \ \texttt{y})\ \texttt{false (not false)}) && \text{def. of } \texttt{false}\\
&\xrightarrow{\beta} ((\lambda\ \texttt{y.}\ \ \texttt{y})\ \texttt{(not false)}) && \text{reduce}\\
&\xrightarrow{\beta} \texttt{(not false)} && \text{reduce}\\
&= ((\lambda\ \texttt{b x y.}\ \ \texttt{b y x})\ \texttt{false} && \text{def. of } \texttt{not}\\
&\xrightarrow{\beta} (\lambda\ \texttt{x y.}\ \ \texttt{false y x}) && \text{reduce}\\
&= (\lambda\ \texttt{x y.}\ \ (\lambda\ \texttt{x y.}\ \ \texttt{y})\ \texttt{y x}) && \text{def. of } \texttt{false}\\
&\xrightarrow{\beta} (\lambda\ \texttt{x y.}\ \ (\lambda\ \texttt{y.}\ \ \texttt{y})\ \texttt{x}) && \text{reduce}\\
&\xrightarrow{\beta} (\lambda\ \texttt{x y.}\ \ \texttt{x}) && \text{reduce}\\
&= \texttt{true} && \text{def. of } \texttt{true}
\end{aligned}$$

$\therefore$ `xor (not false) false` $=$ `true`.

□

We could have also proceeded in applicative order to get

$$
\begin{aligned}
\texttt{xor (not false) false} &= \texttt{xor ((}\lambda\texttt{ b x y. b y x) false) false} && \text{def. of } \texttt{not} \\
&\xrightarrow{\beta} \texttt{xor (}\lambda\texttt{ x y. false y x) false} && \text{reduce} \\
&= \rightarrow\texttt{xor (}\lambda\texttt{ x y. (}\lambda\texttt{ x y. y) y x) false} && \text{def. of } \texttt{false} \\
&\xrightarrow{\beta} \texttt{xor (}\lambda\texttt{ x y. (}\lambda\texttt{ y. y) x) false} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{xor (}\lambda\texttt{ x y. x) false} && \text{reduce} \\
&= \texttt{xor true false} && \text{def. of } \texttt{true} \\
&= \texttt{(}\lambda\texttt{ a b. a (not b) b) true false} && \text{def. of } \texttt{xor} \\
&\xrightarrow{\beta} \texttt{(}\lambda\texttt{ b. true (not b) b) false} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{(true (not false) false)} && \text{reduce} \\
&= \texttt{(true ((}\lambda\texttt{ b x y. b y x) false) false)} && \text{def. of } \texttt{not} \\
&\xrightarrow{\beta} \texttt{(true (}\lambda\texttt{ x y. false y x) false)} && \text{reduce} \\
&= \texttt{(true (}\lambda\texttt{ x y. (}\lambda\texttt{ x y. y) y x) false)} && \text{def. of } \texttt{false} \\
&\xrightarrow{\beta} \texttt{(true (}\lambda\texttt{ x y. (}\lambda\texttt{ y. y) x) false)} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{(true (}\lambda\texttt{ x y. x) false)} && \text{reduce} \\
&= \texttt{(true true false)} && \text{def. of } \texttt{true} \\
&= \texttt{((}\lambda\texttt{ x y. x) true true)} && \text{def. of } \texttt{true} \\
&\xrightarrow{\beta} \texttt{(}\lambda\texttt{ y. true) true)} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{true} && \text{reduce} \\
\therefore \texttt{xor (not false) false} &= \texttt{true}. \\
& && \square
\end{aligned}
$$