Foundations of Computer Science

Supervision 3

7. Dictionaries and functional arrays

7.1. Conceptual questions

1. Draw the binary search tree that arises from successively inserting the following pairs into the empty tree: `(Alice, 6)`, `(Tobias, 2)`, `(Gerald, 8)`, `(Lucy, 9)`. Then repeat this task using the order `(Gerald, 8)`, `(Alice, 6)`, `(Lucy, 9)`, `(Tobias, 2)`. Why are results different? How could we avoid the issue encountered in the first case?

7.2. Exercises

2. Code an insertion function for binary search trees (with keys of `string` type, and values of polymorphic `'a` type). It should resemble the existing `update` function except that it should raise the exception `Collision` if the item to be inserted is already present. Now try modifying your function so that `Collision` returns the value previously stored in the dictionary at the given key. What problems do you encounter and why?

3. Describe and code an algorithm for deleting an entry from a binary search tree. Comment on the suitability of your approach. There are two reasonable methods – one is simple, the other is efficient but a bit more tricky.

4. Write a function to remove the first element from a functional array. All the other elements are to have their subscripts reduced by one. The cost of this operation should be linear in the size of the array.

7.3. Optional questions

5. Show that the functions `preorder`, `inorder` and `postorder` all require $O(n^2)$ time in the worst case, where n is the size of the tree.

6. Show that the functions `preord`, `inord` and `postord` take linear time in the size of the tree.

8. Functions as values

8.1. Conceptual questions

1. Consider the following polymorphic functions. Infer the types of `sw`, `co` and `cr` (without asking OCaml) and the give the definitions of `id`, `ap` and `ucr` based on their types. What do these functions do and what are their uses?

```
let sw f x y = f y x
let co g f x = g (f x)
let cr f a b = f (a,b)
val id  : 'a -> 'a = <fun>
val ap  : ('a -> 'b) -> 'a -> 'b = <fun>
val ucr : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

8.2. Exercises

2. *Ordered types* are OCaml types `T` with a comparison operator `< : T -> T -> bool` such that `a < b` returns `true` if `a : T` is "smaller than" `b : T`. Many OCaml types – such as `string` and `int` – can be ordered and compared with the `<` operator in the obvious way. We often want to combine two such orderings to get comparison operators for compound types such as `string * int`. Two ways of doing this are *pairwise ordering*, which compares elements of the pair individually:

$$(x, y) <_p (x', y') \iff x < x' \land y < y'$$

and *lexicographic ordering*, which orders by the first elements, and if they are equal, by the second element (for an arbitrary number of elements we get the familiar word ordering used in dictionaries):

$$(x, y) <_\ell (x', y') \iff x < x' \ \lor \ (x = x' \ \land \ y < y')$$

   a) Write OCaml functions implementing pairwise and lexicographic ordering for the type of pairs `string * int`.

   b) Hardcoding the comparison operator `<` makes these functions a bit inflexible: for example, we cannot order a list of pairs in increasing order on the first element, but decreasing order on the second. We can make the functions more abstract by taking the comparison operators as higher-order arguments, and using them instead of `<`. Write two higher-order OCaml functions to perform pairwise and lexicographic comparison of values of type `'a * 'b`, where the comparison operators for types `'a` and `'b` are passed as arguments.

   c) Explain how you would use your functions in the previous part, and the higher-order sorting function `insort`, to sort a list of type `(string * (int * string)) list` according to the following specification:

$$(s_1, (m, s_2)) < \left(s_1', (n, s_2')\right) \iff s_1 \le s_1' \ \land \ \left(m > n \ \lor \ (m = n \ \land \ s_2 < s_2')\right)$$

3. Without using `map`, write a function `map2` such that `map2 f` is equivalent to the composition `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with one by exploiting nested pattern-matching.

4. The built-in type `option`, shown below, can be viewed as a type of lists having at most one element. (It is typically used as an alternative to exceptions.) Declare an analogue of the function `map` for type `option`.

```
type 'a option = None | Some of 'a
```

8.3. Optional questions

5. Recall the making change function of Lecture 4:

```
let rec change till amt = match till, amt with
  | _, 0  -> [ [] ]
  | [], _ -> []
  | c::till, amt ->
     if amt < c then change till amt else
       let rec allc = function
         | [] -> []
         | (cs::css) -> (c::cs) :: allc css
       in allc (change (c::till) (amt-c))
          @ change till amt
```

The function `allc` applies the function "cons a `c`" to every element of a list. Eliminate it by declaring a curried cons function and applying `map`.

9. Sequences and laziness

9.1. Conceptual questions

1. Consider the list function `concat`, which concatenates a list of lists to form a single list. Can it be generalised to concatenate a sequence of sequences? What can go wrong?

```
let rec concat = function
  | []    -> []
  | l::ls -> l @ concat ls
```

2. Why are lazy lists (sequences) useful and why are they not "natively" supported by OCaml? How do we simulate lazy lists in OCaml and why does that not have the issues you described above?

9.2. Exercises

3. Code an analogue of `map` for sequences.

4. A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an OCaml datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. The order of the elements in the lazy list does not matter, as long as it contains all potential tree nodes.

9.3. Optional questions

5. Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint*: to benefit from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

6. Code the lazy list whose elements are all ordinary lists of zeroes and ones, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;1]`, `[1;0]`, `[1;1]`, `[0;0;0]`, `....`

7. (Continuing the previous exercise.) A palindrome is a list that equals its own reverse. Code the lazy list whose elements are all palindromes of 0s and 1s, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;0;0]`, `[0;1;0]`, `[1;1]`, `[1;0;1]`, `[1;1;1]`, `[0;0;0;0]`, `...` You can use the list reversal function `List.rev`.

8. With some exceptions (such as appending or concatenation), we can adapt many common list operations to work on lazy lists. In addition, lazy lists can be used to generate infinite sequences without the risk of nontermination. This exercise explores some interesting examples using the `seq` type.

   a) Define the analogues of `filter` and `zipWith` for sequences. The `zipWith` list functional is similar to `zip` but it applies a binary function to the pair it constructs. For example, `zipWith (+) [1;2;3;4] [5;6;7;8] = [6;8;10;12]` where `(+)` is the function version of the `+` operator.

   ```
   val filterS  : ('a -> bool) -> 'a seq -> 'a seq = <fun>
   val zipWithS : ('a -> 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq
                = <fun>
   ```

   b) Consider the following two definitions. What do they represent and how do they work?

   ```
   let s = let rec s_aux (Cons (x, xf)) =
                     Cons (x * x, fun () -> s_aux (xf()))
           in s_aux (from 0)

   let rec f = Cons (0, fun () ->
               Cons (1, fun () -> zipWithS (+) f (tail f)))
   ```

   c) Define the lazy list `sieve : int seq` that uses the Sieve of Eratosthenes to calculate the infinite sequence of prime integers. *Hint*: Define an auxiliary function `sieve_aux` such that `sieve = sieve_aux (from 2)`. You may want to use `filterS` for the "sieving".

Very optional question

Make sure that you complete Question §8.1.1 before reading this exercise! Don't worry if you can't finish it, but do give it a try sometime – it shows you the real power of functional programming.

Pointfree (or *tacit*) programming is a style of writing functional programs by composing and combining smaller functions instead of defining a function by giving its value at every point (argument). In practice, point-free functions do not mention all of their arguments before the `=` so the expression after the `=` will be a function of the hidden arguments. The basic example is simplifying a function that calls another function on its argument:

```
let firstElem xs = List.hd xs
> val firstElem : 'a list -> 'a = <fun>
```

The value of the function firstElem on each of its points (arguments) xs is the head of xs. The property of *function extensionality* states that two functions are equal if their values are equal at every point. That is, with the definition above, firstElem has exactly the same behaviour as List.hd and it can therefore be simply defined as a value that equals List.hd. The types do not change, as firstElem simply inherits the type of List.hd.

```
let firstElem = List.hd
> val firstElem : 'a list -> 'a = <fun>
```

Similarly, pointfree style can be combined with partial application to create specialised functions from more general ones. A special case of this are the auxiliary functions we define for tail recursion: to get a function of the required type we need to specify the initial value of the accumulator in the auxiliary function. The most idiomatic way of doing this would be with partial application (as long as the accumulator is the first argument):

```
let rec sum_aux acc = function
  | [] -> acc
  | x::xs -> sum_aux (acc + x) xs
> val sum_aux : int -> int list -> int = <fun>
let sum = sum_aux 0
> val sum : int list -> int = <fun>
```

That is, the sum function is equal to sum_aux when partially applied to 0. Note that we do not mention the list argument on either side, just like we didn't always mention the list argument of insort on Page 69.

Before you move on, I would recommend that you go through these and similar examples to make sure you understand how partial application and pointfree programming follows from currying. Feel free to write some notes about this.

The functions in Question §8.1.1 are all utilities for combining and transforming smaller functions. For example, co g f is the composition of two functions f and g, mathematically defined as

$$(g \circ f)(x) = g(f(x))$$

There is no built-in composition operator in OCaml, but to simplify writing pointfree code, it's worth defining it ourselves as an *infix* operator (so instead of co g f we can write g << f).

```
let (<<) g f x = g (f x)
> val (<<) : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Composition is one of the fundamental ways of building larger functions out of smaller ones, the crux of functional programming. Notice that using composition brings the function application to the "top level" instead of nested in several levels of parentheses, which means it plays well with pointfree style programming.

```
let last xs = List.hd (List.rev xs)
let last xs = (List.hd << List.rev) xs
let last    = List.hd << List.rev
```

That is, getting the last element of a list is the same as reversing it first and then getting the head element. (Remember, this is not an efficient implementation of this function!)

Your task will be to transform the functions given below into pointfree style. You may (and should!) use the combinators from §8.1.1, various list functionals and list processing functions from lectures and exercises such as `map` and `sum`. You may also want to remove pattern matching if it becomes redundant due to your definition of choice, or rewrite the function entirely. Basically, make it as simple and as elegant as possible – all of the functions can be made into concise almost-one-liners.

1. Apply the function twice (you can leave the `f` argument).

   ```
   let applyTwice f x = f (f x)
   ```

   Remove the first and last elements of a list.

   ```
   let peel xs = List.rev (List.tl (List.rev (List.tl xs)))
   ```

   As a side-note, you can use `List.(...)` to open the `List` module locally in the parentheses to avoid having to write `List.` in front of every list function:

   ```
   let peel xs = List.(rev (tl (rev (tl xs))))
   ```

2. Count the number of vowels in a sentence represented as a list of strings. The following declarations can be used freely, no need to transform them.

   ```
   let vowels = ['a'; 'e'; 'i'; 'o'; 'u']
   let strToCharList s = List.init (String.length s) (String.get s)
   let rec sum = function | [] -> 0 | x::xs -> x + sum xs
   ```

   The functions `isVowel`, `getVowels` and `countVowels` can be combined into one short expression – try transforming them individually first, then write a single function that does the same thing as `countVowels`.

   ```
   let isVowel ch = List.mem ch vowels

   let rec getVowels = function
   ```

```
  | [] -> []
  | x::xs -> if isVowel x then x :: getVowels xs
                          else      getVowels xs

let rec countVowels = function
  | [] -> 0
  | w::ws -> List.length (getVowels (strToCharList w)) +
             countVowels ws
```

3. Quite contrived, but also quite neat. In this case you can keep the x argument, but change the function so that x only appears once in the body!

```
let calc x = [x; x +. 1.0; 2.0 *. x; x *. x; x /. 2.0;
              Float.pow 2.0 x; Float.sin x; Float.cosh x]
```