02157 Functional programming

Michael R. Hansen
DTU Compute
November 14, 2023

Exercises: Week 11

This exercise set consists of 2 problems:

Problem 1 is the second problem from the exam set from May, 2022.

Problem 2 is the fourth problem from the exam set from May, 2022.

Problem 1

The functions `skipWhile` and `takeWhile` from the `List` library could have the following declarations:

```
let rec skipWhile p = function
                      | x::xs when p x -> skipWhile p xs
                      | xs             -> xs;;
val skipWhile: ('a -> bool) -> 'a list -> 'a list

let rec takeWhile p = function
                      | x::xs when p x -> x::takeWhile p xs
                      | _              -> [];;
val takeWhile: ('a -> bool) -> 'a list -> 'a list
```

Notice that the F# system automatically infers the types of these functions.

1. Give an argument showing that `('a -> bool) -> 'a list -> 'a list` is the most general type of `takeWhile`. That is, any other type for `takeWhile` is an instance of `('a -> bool) -> 'a list -> 'a list`.

Let `diff5` be declared by:

```
let diff5 n = n<>5;;
```

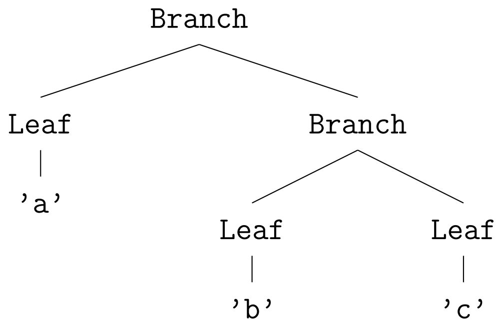
2. Give an evaluation of the expression `skipWhile diff5 [2;6;5;1;5;6]`. Use the notation $e_1 \rightsquigarrow e_2$ from the textbook and include at least as many steps as there are recursive calls.
3. Describe what `takeWhile` and `skipWhile` compute. Your descriptions should focus on *what* they compute, rather than on individual computation steps.
4. Consider each of the above declarations and explain briefly whether the considered function is tail recursive or not. If you encounter a function that is not tail recursive, then provide a declaration of a tail-recursive variant with an accumulating parameter for that function.

Problem 2

Consider now binary trees where leaf nodes (constructor `Leaf`) carry characters:

```
type T = Leaf of char | Branch of T*T
```

The figure below shows a tree `t0` of type `T` containing three characters: `’a’`, `’b’` and `’c’`.



A tree t is called *legal* if any character occurs at most once in t and t contains at least 2 characters. Thus, `t0` is a legal tree.

1. Make an F# value for the tree `t0` shown above and declare a function

   ```
   toList: T -> char list
   ```

   that gives the list of characters occurring in a tree. The sequence in which the characters occur in the list is of no significance.
2. Declare a function `legal` t that can check whether a tree t is legal.

We assume from now on that trees are legal and consider the so-called Huffman coding for characters in a given tree t, where a code $ds = [d_1; d_2; \ldots ; d_n]$ (type `Code`) is a list of directions denoting a path from the root to a leaf in t.

```
type Dir  = | L       // go left
            | R       // go right
type Code = Dir list
type CodingTable = Map<char, Code>
```

For example, the codes for `’a’`, `’b’` and `’c’` in `t0` are `[L]` `[R;L]` `[R;R]`, respectively.

Furthermore, a *coding table* (for a given tree) is a map from characters to their codes. The coding table for `t0`, for example, has the entries (`’a’`, `[L]`), (`’b’`, `[R;L]`) and (`’c’`, `[R;R]`).

The code for a list of characters $cs = [c_1; \ldots ; c_m]$, given a coding table, is obtained by appending the codes for the individual characters of cs. For example, the code for `[’c’;’a’;’a’;’b’]` is `[R;R;L;L;R;L]`.

3. Declare a function `encode: CodingTable -> char list -> Code` that gives the code for a list of characters for a given coding table. The function should raise an exception if the coding table does not contain a code for some character in the list.

4. Declare a function `ofT: T -> CodingTable` that gives the coding table for a tree.

We now consider a function to reproduce the character list *cs* from a code *ds* on the basis of the underlying tree *t*. This function is called *decode*:

```
decode: T -> Code -> char list
```

For example, `decode t0 [R;R;L;L;R;L]` = `['c';'a';'a';'b']`.

It is convenient to use a helper function

```
firstCharOf: T -> Code -> char * Code
```

in the declaration of `decode`.

This helper function decodes the first character of the code and returns that character and the remaining code. For example,

```
firstCharOf t0 [R;R;L;L;R;L] = ('c', [L;L;R;L])
firstCharOf t0     [L;L;R;L] = ('a', [L;R;L])
firstCharOf t0       [L;R;L] = ('a', [R;L])
firstCharOf t0         [R;L] = ('b', [])
```

5. Give declarations for the functions `firstCharOf` and `decode`.