02157 Functional programming

Michael R. Hansen
DTU Compute
November 1, 2023

Exercises: Week 10

This exercise set consists of 3 problems:

Problem 1 is the second problem from the exam set from May, 2021.

Problems 2 and 3 are the third problem and fourth problem from the exam set from December, 2021.

Problem 1

The function `countBy` from the `List` library could have the following declaration:

```
let rec ins x = function
                | (y,n) :: ys when x=y -> (y,n+1)::ys
                | pair  :: ys          -> pair::ins x ys
                | []                   -> [(x,1)];;
   ins: 'a -> ('a * int) list -> ('a * int) list when 'a : equality

let rec cntBy f xs acc = match xs with
                         | []      -> acc
                         | x::rest -> cntBy f rest (ins (f x) acc);;
   cntBy: ('a -> 'b) -> 'a list -> ('b * int) list -> ('b * int) list
          when 'b : equality

let countBy f xs = cntBy f xs [];;
   countBy: ('a -> 'b) -> 'a list -> ('b * int) list when 'b : equality
```

where `ins` and `cntBy` are helper functions. Notice that the F# system automatically infers the types of `ins`, `cntBy` and `countBy`.

1. Give an argument showing that

   ```
   'a -> ('a * int) list -> ('a * int) list when 'a : equality
   ```

   is the most general type of `ins` and that

   ```
   ('a -> 'b) -> 'a list -> ('b * int) list -> ('b * int) list
   when 'b : equality
   ```

   is the most general type of `cntBy`. That is, any other type for `ins` is an instance of `'a -> ('a * int) list -> ('a * int) list when 'a : equality`. Similarly for `cntBy`.

An example using `countBy` is:

```
countBy (fun x -> x%2) [1 .. 3];;
   val it : (int * int) list = [(1, 2); (0, 1)]
```

2. Give an evaluation showing that `countBy (fun x -> x%2) [1 .. 3]` evaluates to `[(1,2); (0,1)]`. Present your evaluation using the notation $e_1 \rightsquigarrow e_2$ from the textbook. You should include at least as many evaluation steps as there are calls of `ins`, `cntBy` and `countBy`.

Problem 2

Consider the following declarations:

```
type T = | One of int | Two of int * T * int * T

let rec f p t =
   match t with
   | One v when p v           -> [v]            (* C1 *)
   | Two(v1,t1,_,_) when p v1 -> v1::f p t1     (* C2 *)
   | Two(_,_,v2,t2)           -> v2::f p t2     (* C3 *)
   | _                        -> [];;           (* C4 *)
```

1. Give the type for `f` and describe what `f` computes. Your description should focus on what it computes, rather than on individual computation steps.

Notice that the declaration of `f` has a match expression with 4 clauses marked `C1` to `C4` in comments.

A *test description* for `f` consists of

- a value p_v for argument `p`,
- a value t_v for argument `t`,
- the expected value of `f` p_v t_v, and
- an enumeration of the clauses that are selected during evaluation of `f` p_v t_v. The order in which clauses are enumerated is not significant. Repeated enumeration of a clause is not necessary.

2. Give a small number (≤ 4) of test descriptions for `f`. Together they should ensure that every clause of `f` is selected during an evaluation.

Problem 3

A type for so-called *tries* is defined as a tree type Trie<'a>, where a node carries a value of type 'a, a truth value, and an arbitrary number of child tries:

```
type Trie<'a>    = N of 'a * bool * Children<'a>
and Children<'a> = Trie<'a> list
```

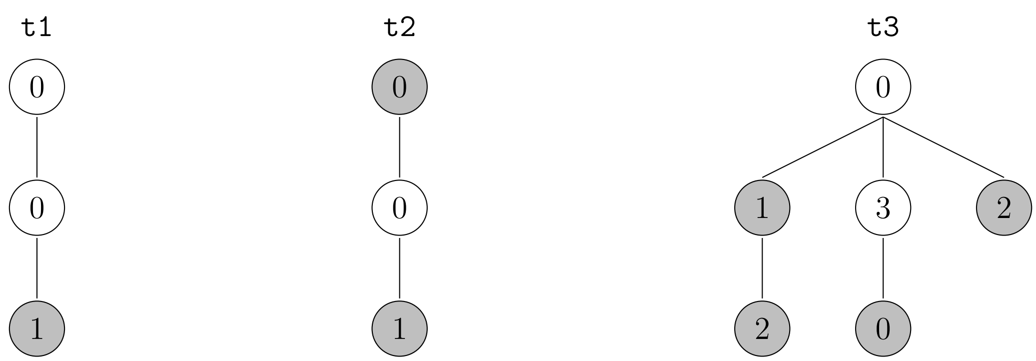
Consider the three values t1, t2 and t3 of type Trie<int>:

```
let t1 = N(0, false, [N(0, false, [N(1,true,[])])]);;

let t2 = N(0, true, [N(0, false, [N(1,true,[])])]);;

let ta = N(1,true,[N(2,true,[])]);;
let tb = N(3,false,[N(0,true,[])]);;
let tc = N(2,true,[]);;
let t3 = N(0,false, [ta;tb;tc]);;
```

The three values are illustrated as trees in the following figure, where each node carry an integer value, and a shaded node indicates that the truth value associated with the node is true. Shaded nodes are also called *accepting nodes*.



t1 accepts [0;0;1]   t2 accepts [0] and [0;0;1]   t3 accepts [0;1], [0;1;2], [0;3;0] and [0;2]

A value in a node of a trie is called a *letter*. For example, trie t3 contains four letters: $0, 1, 2, 3$.

A *word* is a list of letters. Furthermore, a word w is *accepted by* a trie t if there is a path from the root of t to an accepting node, so that w equals the list of letters of the nodes of the path. For example, $[0;1;2]$ is accepted by `t3` and the tries `t1`, `t2` and `t3` accept 1, 2 and 4 words, respectively, as shown in the figure.

1. Declare a function that counts the number of nodes of a trie. For example, `t3` has 6 nodes.
2. Declare a function `accept` w t that can check whether word w is accepted by trie t. Give the type of `accept`.
3. Declare a function `wordsOf: Trie<'a> -> Set<'a list>` that gives the set of words accepted by a trie t.

Leaves of tries have the form $\mathtt{N}(v, b, [])$. Leaves where $b =$ `false` do not contribute to the words accepted by a trie and such leaves are called *useless*.

4. Declare a function that can check whether a trie contains useless leaves.

The *degree of a node* $\mathtt{N}(v, b, ts)$ is the length of the list of children ts. The maximum degree of all nodes in a trie is called the *degree of a trie*.

5. Declare a function that computes the degree of a trie.

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;;
```

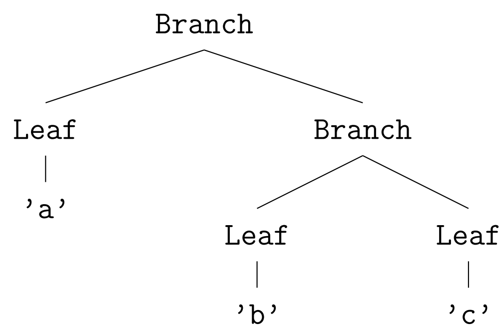
2. Give an evaluation of the expression `skipWhile diff5 [2;6;5;1;5;6]`. Use the notation $e_1 \rightsquigarrow e_2$ from the textbook and include at least as many steps as there are recursive calls.
3. Describe what `takeWhile` and `skipWhile` compute. Your descriptions should focus on *what* they compute, 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`.