Lab 4

Functional Programming (ITI0212)

2023-02-24

This week we learned about function literals and higher-order functions. Function literals are free-standing expressions that represent values for the function types. We can refer to the function with formal parameter `x` and body `t` using the (ASCIIfied) λ notation `\ x => t`. For example, the generic identity function can be written as `\ x => x`.

A *higher-order function* is a function that can take other functions as arguments or return them as results. We saw how the `map` and `filter` functions for `List` types allow us to perform tasks that would typically be done in imperative programming languages using loops, and how the `fold` function for an inductive type lets us encode its recursion principle inside a function and allows us to define other functions without using pattern matching or recursion.

You can use the `filter` function for `List`s in the standard library by writing `import Data.List` at the beginning of your script file.

Task 1
Before consulting Idris, work out for yourself the types and values of the following two expressions. (Don't forget to copy the definition for `is_even` from lecture 2).

```
(map S . filter is_even)[0, 1, 2, 3]
```

```
(filter is_even . map S)[0, 1, 2, 3]
```

Then check your understanding by asking Idris to evaluate them for you.

Note: Since Idris overloads the syntax for `List`, you should either add the following `%hide Prelude.SnocList.filter` after the import statement or call `List.filter`.

Task 2
Write the `map` function for `Maybe` types:

```
map_maybe  :  (a -> b) -> Maybe a -> Maybe b
```

so that

```
Lab4> map_maybe S Nothing
Nothing
Lab4> map_maybe S (Just 41)
Just 42
```

Task 3
Write a higher-order function that uses a given function to transform the element at the specified index of a list:

```
transform : (f : a -> a) -> (index : Nat) -> List a -> List a
```

If the index is out-of-bounds for the list then your function should behave like the identity function. For example:

```
> transform S 0 [1, 2, 3]
[2, 2, 3]
> transform S 1 [1, 2, 3]
[1, 3, 3]
>transform S 2 [1, 2, 3]
[1, 2, 4]
>transform S 3 [1, 2, 3]
[1, 2, 3]
```

Task 4
Use a function literal (λ-expression) and the `filter` function for lists to write the following function:

```
ignore_lowerCaseVowels : String -> String
```

which behaves in the following way: it takes a string as an input and returns the string in which the lowercase vowels were removed. For example:

```
>ignore_lowerCaseVowels ”the cat who saw the moon.”
”th ct wh sw th mn.”
>ignore_lowerCaseVowels ”the cat who sAw the moon.”
”th ct wh sAw th mn.”
```

Hint: the functions `pack` and `unpack` from the standard library will be helpful. You should use `:doc` to find out their types and how to use them. They are using the type `Char` which represents strings of length 1 that has the following literal expression: one character between single quotation marks, such as ’a’ or ’f’.

Challenge: Can you define the function using the following helper function: `elem : a -> List a -> Bool` which returns `True` if an element is in a list and `False` otherwise?

Task 5
Write the following functions using `fold` for `Nat` or `fold` for `List`:

- Rewrite the multiplication (lab 2) function for natural numbers:

  ```
  mult’ : Nat -> Nat -> Nat
  mult’ m = fold_nat ?n ?c
  ```

- Rewrite the functions `n_to_lu` and `lu_to_n` from lab 3, such that they define a type isomorphism between the types `Nat` and `List Unit`. I.e., rewrite the following:

  – ```
    n_to_lu : Nat -> List Unit
    n_to_lu = fold_nat ?n ?c
    ```

  – ```
    lu_to_n : List Unit -> Nat
    lu_to_n  = fold_list ?n ?c
    ```

  such that

  ```
  >n_to_lu (lu_to_n [(), (), ()])
  [(), (), ()]
  >lu_to_n (n_to_lu 2)
  2
  ```

Task 6
Write the `fold` function for the `Bool` type, `fold_bool`.

- First determine the type of this function using the algorithm described in the lecture.
- Then write the function definition using the algorithm for that.

Up to argument order, you should recognize this function as a construct present in nearly every programming language, what is it? Idris also supports the conventional syntax for this construct, try it out.