Lab 4

Functional Programming

2024-02-16

This week we learned about function literals and higher-order functions. A *function literal* is a free-standing expression representing a value of a function type. We write the function with formal parameter `x` and body `t` using the (ASCIIfied) λ notation "`\ x => t`". For example, the generic identity function with type `a -> a` can be written as "`\ x => x`". For convenience, we can use *section notation* for infix operators, leaving away either or both operands.

A *higher-order function* is a function that traffics in other functions, either taking them as arguments or returning them as results. We saw how the `map`, `filter`, and `zip` functions for `List` types allow us to perform tasks that would typically be done using loops in imperative programming languages, and how the `fold` function for an inductive type reifies its recursion principle as an ordinary function.

Task 1
Work out for yourself the types and values of the following expressions involving `Lecture2.is_even`.

```
(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.

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

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

so that:

```
Lab4> map_maybe (2 * ) Nothing
Nothing
Lab4> map_maybe (2 * ) (Just 21)
Just 42
```

Task 3
Use a function literal (λ-expression) to complete the following function that returns the numbers in a list that are multiples of 10:

```
round_numbers  :  List Integer -> List Integer
round_numbers  =  filter ?p
```

For example:

```
Lab4> round_numbers [5,10,15,20]
[10, 20]
```

Hint: the functions `mod` and `(==)` will be helpful.

Task 4
Write the generic higher-order function,

```
iterate  :  Nat -> (a -> a) -> a -> a
```

that composes the given function with itself the given number of times.

For example:

```
Lab4> iterate 3 (2 * ) 1
8
Lab4> iterate 8 ("Na" ++ ) " Batman!"
"NaNaNaNaNaNaNaNa Batman!"
```

Task 5
Use recursion to write a function that adds together all the numbers in a list:

```
sum_list  :  List Integer -> Integer
```

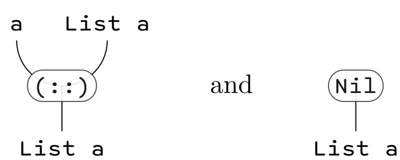
For example,

```
Lab4> sum_list [1, 2, 3]
6
Lab4> sum_list []
0
```

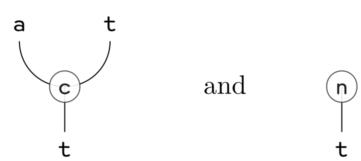
Task 6
Recall that the *fold* function for list types reifies the pattern of list-recursion as a function:

```
fold_list  :  (c : a -> t -> t) -> (n : t) -> List a -> t
fold_list c n []  =  n
fold_list c n (x :: xs)  =  c x (fold_list c n xs)
```

The idea is that the element constructors for list types, `(::) : a -> List a -> List a` and `Nil : List a`, have the "shapes":



So that in any diagram made up of `(::)`s and `Nil`s, representing an element of type `List a`, if we uniformly replace them with respectively:



then we obtain a diagram with the same "shape" representing an element of type `t`.

Use the fold for list types to rewrite the list-summing function as a one-liner:

```
sum_list'  :  List Integer -> Integer
sum_list'  =  fold_list ?c ?n
```

Task 7
Write the `fold` function for the `Bool` type, `fold_bool`.

- First determine the type of this function using the algorithm described in class.
- 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.