Introduction to Functional Programming

Practicals

Tom Harper
Ralf Hinze
Nicolas Wu

0 Getting started

We will be using GHCi for the practicals. To run GHCi, simply open a terminal window and type '`ghci`'. One typically uses a text editor to write or edit a Haskell script, saves that to disk, and loads it into GHCi. To load a script, it is helpful if you run GHCi from the directory containing the script. You can simply give the name of the script file as a parameter to the command `ghci`. Or, within GHCi, you can type '`:l`' followed by the name of the script to load, and '`:r`' with no parameter to reload the file previously loaded.

1 Basic evaluation

1. Here is a script of function definitions:

```
add :: Integer -> Integer -> Integer
add x y = x + y

double :: Integer -> Integer
double x = x + x

first :: Integer -> Integer -> Integer
first x y = x

cond :: Bool -> Integer -> Integer -> Integer
cond x y z = if x then y else z

twice :: (Integer -> Integer) -> Integer -> Integer
twice f x = f (f x)

infinity :: Integer
infinity = infinity + 1
```

(The function `twice` is an example of a higher-order function, which takes another function as one of its arguments.)

These first exercises are designed to help you understand how expressions are evaluated in Haskell. Evaluate the following expressions by hand, by giving both the applicative and normal-order reductions:

- `first 42 (double (add 1 2))`
- `first 42 (double (add 1 infinity))`
- `first infinity (double (add 1 2))`
- `add (cond True 42 (1+infinity)) 4`
- `twice double (add 1 2)`
- `twice (add 1) 0`

NOTE: There is not a mistake in the last expression; all you need to know for the time being is that a function application that doesn't have enough arguments is already in normal form. Just follow the rules when reducing the expression.

2. Give a reduction sequence for `fact 3`, where the factorial function `fact` is defined as:

```
fact :: Integer -> Integer
fact 0  =  1
fact n  =  n * fact (n - 1)
```

2 Basic definitions

1. Define the following numeric functions:

   - a function `square` that squares its argument, then a function `quad` that raises its argument to the fourth power using `square` (try using function composition to define `quad`);
   - a function `larger` that returns the larger of its two arguments;
   - a function for computing the area of a circle with a given radius (use the type `Double`). (Hint: the formula for calculating the area A of a circle with a radius r is $A = \pi r^2$, where π is called `pi` in Haskell.)
   - Define `(&&)` and `(||)` once using *conditional expressions* (i.e. `if-then-else` statements), and again using *conditional definitions* (i.e. guarded equations). In each case, there is more than one plausible way to do it, but only one correct way (Hint: strictness).

2. Define a function `showDate` that takes three integers representing the day, month and year, and returns them formatted as a string (Hint: The ++ operator to appends one string to another, and the `show` function converts a number to a string). For example:

   ```
   showDate 2 8 2004 = "2 August 2004"
   ```

   If that was too easy, make the day number an ordinal:

   ```
   showDate 2 8 2004 = "2nd August 2004"
   ```

3 Lists

3.1 Types

Try to answer the following questions without the help of `ghci`:

1. What is the type of `[1,2,3]`?
2. What is the type of `[[1,2],[3,4]]`?
3. What is the type of `[(1, 2), (3, 4)]`?
   What about `[(1,'a'), (2,'b')]`?
4. What is the type of `['a','b','c']`?
   What about `"abc"`?
5. What is the type of `["abc","xyz"]`?
6. What is the type of `[]`? What about `[[]]`?
7. What is the type of `([1,2],['a','b'])`?
   What about `[[1, 2],['a','b']]`?

3.2 Definitions

Using the *list design pattern* given in the lectures, give recursive definitions of

1. a function `prod :: [Int] -> Int` that calculates the product of a list of integers;
2. a function `allTrue :: [Bool] -> Bool` that determines whether every element of a list of booleans is true;
3. a function `allFalse` that similarly determines whether every element of a list of booleans is false;
4. a function `decAll :: [Int] -> [Int]` that decrements each integer element of a list by one;
5. a function `convertIntBool :: [Int] -> [Bool]` that, given a list of integers, converts any zero to `False`, and any other number to `True`;

6. a function `pairUp :: [a] -> [b] -> [(a,b)]` that pairs up the corresponding elements of two lists, stopping when either list runs out. For example:

```
pairUp [1,2,3]  "abc"  = [ (1,'a'), (2,'b'), (3,'c') ]
pairUp [1,2]    "abc"  = [ (1,'a'), (2,'b') ]
pairUp [1,2,3]  "ab"   = [ (1,'a'), (2,'b') ]
```

7. a function `takePrefix :: Int -> [a] -> [a]` that returns the prefix of the specified length of the given list (or the whole list, if it is too short);

8. a function `dropPrefix :: Int -> [a] -> [a]` that similarly drops such a prefix (or the whole list, if it is too short);

9. a function `member :: Eq a => [a] -> a -> Bool` that determines whether a given list contains a specified element.

The definitions of the following functions deviate slightly from the list design pattern. Specifically, they may not be able to return a valid answer for all inputs. In order to deal with this, you can use the `Maybe` datatype:

```
data Maybe a = Nothing | Just a
```

This datatype allows a function to return a value wrapped in a `Just` constructor or `Nothing` if there is no value to return. This datatype is one common form of error-handling. Using the `Maybe` type, give recursive definitions for:

1. a function `select :: [a] -> Int -> Maybe a` that selects the element of the list at the given position using 0-based indexing;

2. a function `largest :: [Int] -> Maybe Int` that calculates the largest value in a list of integers;

3. a function `smallest :: [Int] -> Maybe Int` that similarly calculates the smallest value in a list of integers;

The `Maybe` datatype is useful for signalling that there was an error; sometimes it is also useful to have data about the error that is raised. Which datatype would be more suitable for this?

When we have covered the corresponding material in the lectures, you may want to return to consider which of these functions can be written more simply using *list comprehensions* or standard *higher-order operators* like `map` and `foldr`.

4 Composition

The function `loremIpsum :: String` returns a large block of text. The following exercises ask you to define functions that will process this text to extract various statistics.

You might find the following functions from the Prelude useful:

```
words :: String -> [String]
unwords :: [String] -> String
```

The `words` function takes a block of text as a `String`, and outputs a list of `String`s, where each one corresponds to a word in the original text. The `unwords` function reverses the process.

Similarly, the following functions might come handy:

```
lines :: String -> [String]
unlines :: [String] -> String
```

These return the number of lines in a block of text.

Your task is to compute the following statistics, using a combination of the functions above composed with higher order functions. Wherever possible, try to make use of function composition.

- Word count
- Line count
- Sentence count
- Average words per line
- Average words per sentence
- Average words per paragraph
- Average characters per word

5 Trees

The datatype declaration

```
data Tree a = Empty | Node (Tree a) a (Tree a)
  deriving (Show)
```

defines a binary tree. Identify the analogous design pattern for this structure and use it or the list design pattern to give recursive definitions of:

1. a function `size :: Tree a -> Integer` that calculates the number of elements in a tree;
2. a function `tree :: [a] -> Tree a` that converts a list into a tree;
3. a function `member :: (Eq a) => a -> Tree a -> Bool` that determines whether a given tree contains a specified element;
4. a function `searchTree :: (Ord a) => [a] -> Tree a` that converts a list into a search tree (a search tree is a tree in which, for a given node, all the values in the left subtree are smaller and all the values in the right subtree are larger);
5. a function `memberS :: (Ord a) => a -> Tree a -> Bool` that determines whether a given search tree contains a specified element (this should be more efficient than your definition of `member`);
6. a function `inOrder :: Tree a -> [a]` that produces the list of elements from an in-order tree traversal (an in-order tree traversal is one in which the left subtree is traversed, then the root node, and then the right subtree).

6 Higher-Order Functions

1. Use folds and unfolds to define the functions from Exercise 3.

2. The Fibonacci function can be written recursively as:

   ```
   fib :: Integer -> Integer
   fib 0 = 1
   fib 1 = 1
   fib n = fib (n-1) + fib (n-2)
   ```

   However, there is a much more efficient version that can be written as an unfold. This is done by producing an infinite list of all the Fibonacci numbers using the unfold, and picking out the index of the number required. Define such a function.

3. The factorial of a number can be calculated using the following list comprehension:

   ```
   fac :: Integer -> Integer
   fac n = product [1 .. n]
   ```

   Write this as an `unfold` followed by a `foldr`.

7 Type classes

1. Define type class `Functor`, `Eq`, and `Ord` instances for the `Tree` datatype.

2. Define the type class `Sizeable`, which provides a function

   ```
   size :: a -> Integer
   ```

   that calculates the number of elements in a value. Give instances for primitive types such as `Int`, `Char`, and `Bool`, as well as for container types such as `Maybe`, `List`, and `Tree`.

3. *Optional:* Define a type class `Hashable`, which provides a function `hash :: a -> Integer`. As with the previous exercise, give instances for both primitive types and container types.

8 Monads

The `State` monad is used to model stateful computations. The type `State s a` represents a stateful computation where the state is of type `s` and the return value is of type `a`. Its key functions are

```
put :: s -> State s ()
get :: State s s
```

where `put x` stores the value `x` as the current state, and `get` retrieves the current state. After a stateful computation has been run, we want to be able to extract the result from the monad, which is accomplised with

```
evalState :: State s a -> s -> a
```

The effect of `evalState st s` is to evaluate the stateful computation `st` with the initial state `s` and return the result.

1. As a warm-up, use the State monad to write a function

   ```
   sumS :: Num a => [a] -> State a a
   ```

   that sums a list by accumulating the running sum in the state. Extract the final result with the function

   ```
   evalSumS :: Num a => [a] -> a
   evalSumS xs = evalState (sumS xs) 0
   ```

2. Write a function

   ```
   decorate :: Tree a -> State Int (Tree (Int,a))
   ```

   that decorates the nodes of a `Tree` using an in-order traversal. Use the function

   ```
   evalDecorate :: Tree a -> Tree (Int, a)
   evalDecorate t = evalState (decorate t) 1
   ```

   to execute `decorate t` and extract the result. For example, given the tree

```
Node
  (Node Empty 'b' Empty)
  'a'
  (Node
    (Node Empty 'd' Empty)
    'c'
    Empty)
```

the result of decorating it should be

```
Node
 (Node Empty (1,'b') Empty)
 (2,'a')
 (Node
   (Node Empty (3,'d') Empty)
   (4,'c')
   Empty)
```

3. *Challenge:* Write an algorithm using the `State` monad that finds the least natural number that does not occur in a list of length `n` using a single pass.