CMPT 383: Vitamin #4

Anders Miltner
`miltner@cs.sfu.ca`

Due Oct 12

Introduction

This Vitamin is here to get you accustomed to writing Functor and Applicative instances.

This submission will be autograded. There are some portions of the assignment that are ungraded, and some that will be graded. We provide a (partial) test suite for partial validation. You can run these tests by opening a terminal in the v4 directory, and running `stack test`.

We have not imported any functions. You are welcome to use or import anything the the Haskell standard library that may help. You do not need to import anything or use any built-in functions to finish this Vitamin.

1 `AssocList`

An AssocList is a list-like data structure. They have the same base case (Nil), the only difference lies in the Cons case. Where the Cons case of a List contains two components: a data value and a List, the Cons case of an AssocList contains three: two data values (of possibly different types) and an AssocList.

Intuitively, AssocLists can be thought of as a form of (inefficient) dictionaries. They store associations between keys (the first element of the Cons case) and data (the second element of the cons case).

In this Vitamin, we will build the functor for AssocLists. This functor should transform the data of the AssocList, not the keys. For example, `fmap (+1) Cons(1,1,Nil) = Cons(1,2,Nil)`. More examples are provided in the tests.

However, sometimes we would like to transform the data and the keys. For this type of situation, we have the doubleMap function. The doubleMap takes a function as input of type `k -> a -> (k',a')`. In other words, for each key/value pair “k” and “a”, it produces a new pair “k' ” and “a' ”. A partial test suite is provided in the test folder.

2 `ErrJst`

The `ErrJst` data structure contains either a piece of data, or some data representing an error. In this way, it is similar to the Maybe data structure. The key difference lies in the way errors are represented. In Maybe, errors are simply the null value Nothing. In ErrJst, the errors contain a piece of data – perhaps a string to go to the message or perhaps an integer error code. The `ErrJst e a` data structure can either have an error value `Err e` or it can have a data value `Jst a`.

ErrJst can be both a functor and an applicative. You will be providing these type class instantiations. These type class instantiations should look similar to those of the Maybe data structure – you simply need to maintain information on the error state.

When performing an fmap on a Jst, simply apply the value of the fmap to the data of the Jst. When performing an fmap on a Err, simply return the Err.

The pure function of ErrJst should simply encapsulate the provided function in a Jst. The application function <*> should only return Jst when both the provided function and argument are Jsts themselves – otherwise they should propogate errors. Furthermore, if both the function and the argument are errors – when “choosing” which error to propogate – between the function error and the value error – the function error is the one that persists.

3 ZipTree

A Tree can be a functor and applicative as well. When acting as an applicative, the applicative acts similar to ZipList – hence why this section is the ZipTree section.

The pure function for the ZipTree has been provided. Similarly to ZipList, the pure function creates an infinitary data structure, corresponding to a tree with no leaves.

Similarly to ZipList, when a ZipTree encounters a smaller ZipTree, it simply prunes down the size. This is a bit more complex for Trees, because some trees are incomparable in size – the “function” tree can lack a left subtree, and the “argument” can lack a right subtree. In such a case, one should not try to “even out” the two trees. Rather, the largest shared structure should persist, with leaves occuring if a Leaf appears. As a concrete example:

```
Node(Node(Leaf,f1,Leaf),f2,Leaf) <*> Node(Leaf,x2,Node(Leaf,x3,Leaf)) =
  Node(Leaf,f2 x2,Leaf)
```

Additional examples are provided in the test directory.