Weekly Assignments 7

To be submitted: Tuesday, 27 February 2018

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious. Please also pay attention to stylistic issues. The goal is always to submit code that does not just correctly do what was asked for, but also could be committed without further changes to an imaginary company codebase.

W7.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

Please do *NOT* try to upload your package to Hackage.

W7.2 FunLists (*)

This exercise is quite difficult and – apart from W7.2.5 – pretty abstract. Consider it optional and do not waste too much time on it!

For lenses, prisms and isos, we have seen both concrete descriptions and abstract van-Laarhoven-style definitions; for traversals however, we have only seen the latter.

In this exercise, we want to present a concrete description for traversals and use that description to implement interesting operations on traversals.

W7.2.1

Consider the nested(!) datatype

```
data FunList a b t = Done t | More a (FunList a b (b -> t))
```

Define Functor- and Applicative instances for FunList a b.

Defining (<*>) may be tricky – let the types guide you!

W7.2.2

To get a feeling for FunLists, define functions

```
toList    :: FunList a b t -> [a]
fromList  :: [a] -> FunList a b [b]
```

such that toList . fromList = id :: [a] -> [a].

W7.2.3

Define functions

```
singleton  :: a -> FunList a b b
fuse       :: FunList a a t -> t
```

satisfying fuse . singleton = id :: a -> a.

W7.2.4

Using W7.2.1 and W7.2.3, define mutually inverse functions

```
toFunList   :: Traversal s t a b -> (s -> FunList a b t)
fromFunList :: (s -> FunList a b t) -> Traversal s t a b
```

The first should be very easy with W7.2.1 and W7.2.3, for the second it might be helpful to first define a helper function

```
fromFunList' :: Applicative f => (a -> f b) -> FunList a b t -> f t
```

We thus arrive at a concrete description of traversals!

W7.2.5

To make the FunList-description of traversals derived in the last exercises more convenient, define a function

```
transform  :: (FunList a b t -> FunList a b t)
           -> (Traversal s t a b -> Traversal s t a b)`
```

which uses toFunList and fromFunList to transform a transformation on FunLists into a transformation on Traversals.

Using transform, write the following transformations of traversals:

```
heading    :: Traversal' s a -> Traversal' s a
tailing    :: Traversal' s a -> Traversal' s a
taking     :: Int -> Traversal' s a -> Traversal' s a
dropping   :: Int -> Traversal' s a -> Traversal' s a
filtering  :: (a -> Bool) -> Traversal' s a -> Traversal' s a
element    :: Int -> Traversal' s a -> Traversal' s a
```

In case the names are not suggestive enough – here are the expected result when using the various transformations:

```
set (heading          each) "Athens" 'x' -- "xthens"
set (tailing          each) "Athens" 'x' -- "Axxxxx"
set (taking 3         each) "Athens" 'x' -- "xxxens"
set (dropping 3       each) "Athens" 'x' -- "Athxxx"
set (filtering (< 'm') each) "Athens" 'x' -- "xtxxns"
set (element 1        each) "Athens" 'x' -- "Axhens"
```

W7.3 Some prisms

In this exercise, we want to explore some non-standard prisms.

W7.3.1

Define a prism

```
_Natural :: Prism' Integer Natural
```

(You can find the Natural type of arbitrary-precision natural numbers in module Numeric.Natural in the base libraries.)

```
preview _Natural 42   -- Just 42
preview _Natural (-7) -- Nothing
```

W7.3.2

Define a function of type

```
_TheOne :: Eq a => a -> Prism' a ()
```

Given an a, the resulting prism's focus should be the given element:

```
preview (_TheOne 'x') 'x' -- Just ()
preview (_TheOne 'x') 'y' -- Nothing
review  (_TheOne 'x') ()  -- 'x'
```

W7.3.3

Let's define the following wrapper type:

```
newtype Checked a = Checked { unChecked :: a } deriving Show
```

Define a function

```
_Check :: (a -> Bool) -> Prism' a (Checked a)
```

The idea is that the prism finds only elements that fulfill the given predicate. (This will only be a law-abiding prism if we agree to never put an `a` into the `Checked`-wrapper which does not satisfy the predicate.)

```
preview (_Check odd) 42          -- Nothing
preview (_Check odd) 17          -- Just (Checked {unChecked = 17})
review (_Check odd) (Checked 3) -- 3
```

W7.3.4

Using `_Check`, we can provide a much simpler implementation of the `filtering` transformer from exercise W7.2.5, one which does not involve `FunLists` at all and also does not require you to write a new traversal or prism by hand.

Give this simpler implementation using `_Check`.

W7.4

Consider the following tree type:

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

W7.4.1

Define three traversals

```
inorder, preorder, postorder :: Traversal (Tree a) (Tree b) a b
```

which traverse the nodes in *inorder* (left, value, right), *preorder* (value, left, right) and *postorder* (left, right, value), respectively.

W7.4.2

Define two functions

```
printNodes  :: Show a => Traversal' (Tree a) a
            -> Tree a -> IO ()

labelNodes  :: Traversal (Tree a) (Tree (a, Int)) a (a, Int)
            -> Tree a -> Tree (a, Int)
```

Given a traversal, `printNodes` should print all values stored in the tree *in order of the traversal*, whereas `labelNodes` should label all nodes, starting at 1, again in the order of the given traversal.

Test your functions on `inorder`, `preorder` and `postorder` from W7.4.1 on at least the following example tree:

```
tree :: Tree Char                  --         c
tree = Node                        --        / \
            (Node                  --       /   \
                (Node Tip 'a' Tip) --      b     d
                'b'                --     / \   / \
                Tip)               --    /
            'c'                    --   a
            (Node Tip 'd' Tip)     --  / \
```