EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming
Wintersemester 2021/2022

Prof. Dr. T. Grust, D. Hirn

Assignment 10 (21.01.2022)

Handin until: Friday, 28.01.2022, 00:00

Exercise 1: Functor Laws **[6 Points]**

1. Any Functor is expected to adhere to the two *functor laws*:

   (a) `fmap id ≡ id`

   (b) `fmap f . fmap g ≡ fmap (f . g)`

   Consider the following `Functor` instance for simple binary trees.

```
data BTree a = Node a (BTree a) (BTree a) | Leaf

instance Functor BTree where
    fmap f Leaf = Leaf
    fmap f (Node a t1 t2) = Node (f a) (fmap f t1) (fmap f t2)
```

   Our intuition tells us that the functor laws hold for this type and instance. And indeed, they do. However, rather than just hoping for the best, we can actually prove this. Please give a **formal proof** that the two functor laws hold for the `Functor BTree` instance by structural induction on the `BTree` data type.

   Your proof should have the usual structure of inductive proofs: Start by showing that the property in question (here: one of the functor laws) holds for a *base case*. Here, we choose a `Leaf` tree as the base case. Next, we assume that the property holds for some trees `t1`, `t2` (the *inductive hypothesis*) and use this assumption to show that the property holds for a tree `Node a t1 t2` (with `a` being arbitrary). This is the *inductive step*.

   Note that this task requires no fancy proof techniques, just simple equational reasoning (*i.e.* replacing function names by their definitions).

2. Let's prove that a second `Functor` instance is well-behaved (or *lawful*). As discussed in the lecture, regular functions admit a `Functor` instance in which `fmap` is just function composition:

```
instance Functor ((->) a) where
    fmap f g = f . g
```

   This time, there is no structure to work with inductively. Apply simple equational reasoning.

3. Finally, we look at the following type and instance:

```
data AppCount a = AC Int a

instance Functor AppCount where
    fmap f (AC c a) = AC (c + 1) (f a)
```

   This instance is type-correct, but do the functor laws hold here? Either show that they hold or give a counter-example to demonstrate that they do not hold.

Exercise 2: Monoids **[7 Points]**

Let's talk about *monoids* and trees, more specifically the following type of rose trees with labels of type `a`:

```haskell
data Tree a = Node a [Tree a]
```

Please hand in a file `MyMonoids.hs`. Import the module `Data.Monoid` and have a look at the documentation for that module[1].

1. Write a function `sumTree :: Num a => Tree a -> a` that computes the *sum* of all node labels in a tree.
2. Write a function `treeLabels :: Tree a -> [a]` that computes the *list* of all node labels in a tree.
3. We continue our hunt for common patterns in computations that we might abstract over. `sumTree` and `treeLabels` are suspiciously similar: The label of the current node is combined with the results for all subtrees. We abstract over this pattern in a function

   ```haskell
   foldTree :: Monoid m => (a -> m) -> Tree a -> m
   ```

   Given a function that maps a node label to some element of monoid `m`, `foldTree` combines the monoidal result for the node label and the results for all subtrees.

   Implement `foldTree`.
4. First, use `foldTree` to implement a function `treeLabels' :: Tree a -> [a]` that behaves like `treeLabels`.

   Then, do the same for `sumTree`. Remember that there is no single `Monoid` instance for numeric types. Instead, we have `newtype` wrappers `Sum` and `Product` whose `Monoid` instances implement the additive and multiplicative monoids for numeric type `a`, respectively.
5. Finally, implement functions

   ```haskell
   allNodes :: (a -> Bool) -> Tree a -> Bool
   someNode :: (a -> Bool) -> Tree a -> Bool
   ```

   that check whether *all* or *some* node labels in a tree satisfy a predicate.

   Again, we do not have a single `Monoid` instance for `Bool`, but wrappers `Any` and `All` that implement monoids with different behaviours.

[1] http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html

Exercise 3: Applicative Zip-Lists **[7 Points]**

In the lectures we discussed the `Applicative` instance for lists as implemented in `Prelude`; for `<*>` each function of the left argument list is applied to *each* value of the right argument list:

```
Prelude> [(*2),(+2)] <*> [21,40,7]
[42,80,14,23,42,9]
```

However, this is not the only possible way to define a useful instance of `Applicative` for lists. As with `Product` and `Sum` for alternative instances of `Monoid`, we can define a `newtype Zip a` to implement an alternative instance of `Applicative` for lists.
Consider the following behavior of `<*>` for these `Zip`-lists:

```
Prelude> Zip [(*2),(+2)] <*> Zip [21,40,7]
Zip [42,42]
```

On the level of values, each function of the left argument list is applied to the *one* value that is on the same position of the right argument list. On the level of structures, the lists are combined to a list with the length of the shorter input list.

1. Define a `newtype Zip a` for lists of values of type `a`.

2. Make `Zip a` an instance of `Applicative`. As for now, assume `pure x = Zip [x]` and define the *tie-fighter* operator `<*>` as described above.

The implementation of `pure` as suggested above violates a law. The Haskell compiler does not prevent us from defining this instance, but all `Applicative` instances must satisfy the laws. The *identity* rule documented for the `Applicative` class[2] requires for all possible inputs `v` that:

```
pure id <*> v ≡ v
```

In other words: A computation which neither touches the structure (`pure`) nor affects the inner value (`id`) must not have any effect at all.

3. Give an example which shows that the *identity* rule is violated.

4. Implement a definition of `pure` that does not violate the *identity* rule for any possible Zip-list `v`.

The `Zip` applicative instance can now be used as an alternative to all `zip`*N* and `zipWith`*N* functions in `Data.List`.

5. Reformulate the following expression to use only `<$>` and `<*>` instead of `zipWith3`:

   ```
   zipWith3 (\a b c -> a + b * c) [1,2,3] [4,5,6] [7,8]
   ```

[2]https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#t:Applicative