EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 8

Hand in this assignment until **Thursday, July 15, 12pm** at the latest.

> 🤔 Running out of ideas?
> Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` 🌐 channel on our Discord server—maybe you'll find just the help you need.

> 🚨 FAQ: How many assignments are there?
> This is the last assignment sheet in this semester.

> Ⓔxam-style Exercises
> Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Task 1: Functor Instances Ⓔ

A Make the following data type for rose trees an instance of `Functor`:

```haskell
data RoseTree a = RoseTree a [RoseTree a]
```

B Consider a data type for simple key-value maps:

```haskell
data Map k v = Map [(k, v)]
```

The following instance cannot be defined:

```
> instance Functor Map where
error:
    * Expecting one more argument to 'Map'
      Expected kind '* -> *', but 'Map' has kind '* -> * -> *'
    * In the first argument of 'Functor', namely 'Map'
      In the instance declaration for 'Functor Map'
```

Explain the problem. Define a slightly different but reasonable `Functor` instance for `Map`, instead.

Task 2: Functor Laws

A 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. 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.

```haskell
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)
```

Your proof should follow the usual structure of inductive proofs: First, show 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, assume that the property holds for some trees `t1`, `t2` (the *inductive hypothesis*). Use this assumption to show that the property holds for `Node x t1 t2` (where `x` is arbitrary). This is the *inductive step*. This problem does not require any fancy proof techniques, just simple equational reasoning (**i.e.**, replacing function names by their definitions).

B 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.

C Ⓔ 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 do, or give a counterexample to show that they do not.

Task 3: Applicative Zip-Lists

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.

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

B 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[1] 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.

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

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

[1] https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#t:Applicative

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

E Ⓔ 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]
```