Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2022-12-19
Discussion on 2023-01-13

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 9

On this sheet we will explore the difference between `Applicative` and `Monad`. More specifically, we will investigate one data type which admits two different `Applicative` instances. One of those can be extended to a law-abiding `Monad` instance. The other one, however, does not admit such an instance. The exercises on this sheet are designed to guide you through that process.

Start from the two datatypes below. They are isomorphic to each other but we will give their `Applicative` instances different semantics.

```
data Validation e a = VFail e | VOk a
data Error      e a = EFail e | EOk a
```

Exercise 1 (`Functor` instances)

First, define `Functor` instances for both `Validation` and `Error`.

Test your implementation by writing properties for the functor laws:

Identity	`fmap id == id`
Composition	`fmap (f . g) == fmap f . fmap g`

Exercise 2 (`Validation` applicative)

Implement the `Applicative` instance for `Validation`. It should accumulate all the errors according to a `Semigroup` instance. For example, this property should hold:

```
prop_ValidationAccumulatesErrors es a es' =
  (VFail es <*> VOk a <*> VFail es') == VFail (es ++ es')
```

Test your implementation by writing properties for the `Applicative` laws:

Identity	`pure id <*> v == v`
Composition	`pure (.) <*> u <*> v <*> w == u <*> (v <*> w)`
Homomorphism	`pure f <*> pure x == pure (f x)`
Interchange	`u <*> pure y == pure ($ y) <*> u`

Exercise 3 (`Error` applicative)

Implement the `Applicative` instance for `Error`. The `(<*>)` operator should return `EFail` if any of the two branches evaluate to `EFail`. More specifically, evaluation should short-circuit if evaluation of the first branch results in a failure:

```
prop_ErrorShortCircuits e m = (EFail e <*> m) == EFail e
```

Again, test your implementation by writing properties for the `Applicative` laws.

Note If your properties from Exercise 2 are general enough you should be able to adapt them by changing only the names and replacing `Validation` in the type signatures with `Error`.

Exercise 4 (A `Monad` instance)

Try to write **`Monad`** instances for **`Validation`** and **`Error`**. (Because we are now using the **`Monad`** class from the **`Prelude`** only a definition of the bind operator `(>>=)` is required. By default **`return`** is the same as `pure`—as is required by a law.)

Write properties to test that

(a) the instance satisfies the **`Monad`** laws

Left identity	`return a >>= k == k a`
Right identity	`m >>= return == m`
Associativity	`m >>= (\x -> k x >>= h) == (m >>= k) >>= h`

(b) the **`Monad`** and **`Applicative`** operations relate correctly

```
m1 <*> m2 == m1 >>= (\x1 -> m2 >>= (\x2 -> return (x1 x2)))
```

Describe, why one of the **`Monad`** instances is not law-abiding. But can you give an argument why that non-conforming instance might be considered a valid instance as well?