COMP3141

Software System Design and Implementation

Functors, Applicatives, and Monads Practice

Curtis Millar
CSE, UNSW (and Data61)

15 July 2020

Exercise 4

- Capitalise all characters in the input file.
- Sum all the numbers in the input file.
- Implement a guessing game AI.

State & IO

Week 5 covered `State` and `IO` and you've had a few weeks to work with them.
Do you have any questions?

Functors, Applicatives, Monads

- Consider *higher-kinded* types of kind `* -> *` that *contain* or *produce* their argument type.
- *Functor* lets us use a pure function to map between the higher-kinded type applied to different concrete types.
- *Applicative* lets us apply a n-ary function in the context of the higher-kinded type.
- *Monad* lets us *sequentially compose* functions that return values in the higher-kinded type.

Functors

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The functor type class must obey two laws:

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`

Applicatives

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The functor type class must obey four additional laws:

Applicative Laws

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

Alternative Applicative

It is possible to express `Applicative` equivalently as:

```haskell
class Functor f => App f where
  pure :: a -> f a
  tuple :: f a -> f b -> f (a,b)
```

Example (Alternative Applicative)

1. Using `tuple`, `fmap` and `pure`, let's implement `<*>`.
2. And, using `<*>`, `fmap` and `pure`, let's implement `tuple`.

done in Haskell.

Proof exercise: Prove that `tuple` obeys the applicative laws.

Monads

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

We can define a composition operator with `(>>=)`:

```haskell
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```

The monad type class must obey three additional laws:

Monad Laws

1. `f <=< (g <=< x) == (f <=< g) <=< x` (associativity)
2. `pure <=< f == f` (left identity)
3. `f <=< pure == f` (right identity)

Alternative Monad

It is possible to express `Monad` equivalently as:

```haskell
class Applicative m => Mon m where
  join :: m (m a) -> m a
```

Example (Alternative Monad)

1. Using `join` and `fmap`, let's implement >>=.
2. And, using >>= let's implement `join`.

done in Haskell.

Tree Example

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

Example (Tree Example)

Show that `Tree` is an `Applicative` instance.
done in Haskell.

Note that `Tree` is not a `Monad` instance.

Formulas Example

```haskell
data Formula v = Var v
               | And (Formula v) (Formula v)
               | Or  (Formula v) (Formula v)
               | Not (Formula v)
               | Constant Bool
               deriving (Eq,Show)
```

Example (Formulas Example)

Show that `Formula` is a `Monad` instance.

done in Haskell.

Homework

1. Week 5's quiz is due on Friday. Make sure you submit your answers.
2. The fifth programming exercise is due by the start of my next lecture (in 7 days).
3. This week's quiz is also up, it's due Friday week (in 9 days).

Consultations

- Consultations will be made on request. Ask on piazza or email `cs3141@cse.unsw.edu.au`.
- If there is a consultation it will be announced on Piazza with a link a room number for Hopper.
- Will be in the Thursday lecture slot, 9am to 11am on Blackboard Collaborate.
- Make sure to join the queue on Hopper. Be ready to share your screen with REPL (`ghci` or `stack repl`) and editor set up.