COMP3141
Software System Design and Implementation

Functors, Applicatives, and Monads Practice

Christine Rizkallah
CSE, UNSW (and Data61)
Term 2 2019

Recall: Functors, Applicatives, Monads

- Functors are types for containers where we can `map` pure functions on their contents.
- Applicative Functors are types where we can combine n containers with an n-ary function.
- Monads are types `m` where we can *sequentially compose* functions of the form `a -> m b`.

Recall: Functors

```
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)`

Recall: 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`
2. `pure f <*>  pure  x = pure (f x)`
3. `u <*> pure y = pure ($ y) <*> u`
4. `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`

Alternative Applicative

It is possible to express `Applicative` equivalently as:

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

Example (Alternative Applicative)

1. Using , 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.

Recall: Monads

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

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

```
(<=<) :: (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:

```
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

```haskell
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
               | Plus (Formula v) (Formula v)
               | Times (Formula v) (Formula v)
               | Constant Int
               deriving (Eq,Show)
```

Example (Formulas Example)

Show that `Formula` is a `Monad` instance.
done in Haskell.

9