CS 350 2024-25 Sem I Lecture 9

Satyadev Nandakumar

August 31, 2024

Outline

Outline

Map over lists

map

```
map f [] = []
map f (x:xr) = (f x) : (map f xr)
```

Map over binary trees

mapTree (from previous lecture)

```
mapTree f Nil = Nil
mapTree f (Node n l r) = (Node (f n) (mapTree f l) (mapTree f r
```

Map over `Maybe`

Often computations may not succeed, but it is not a fatal error.
e.g. trying to find an occurrence of an element in a list which does not contain the element. We would like to return a value which means "Not found"
`Maybe` is used in computations which may either return a value `Just x`, or may return `Nothing`.

Map over Maybe

Often computations may not succeed, but it is not a fatal error.
e.g. trying to find an occurrence of an element in a list which does not contain the element. We would like to return a value which means "Not found"
`Maybe` is used in computations which may either return a value `Just x`, or may return `Nothing`.

Maybe basic definition and example

Maybe

```haskell
data Maybe a = Nothing | Just a

-- example : find value of a key in a dictionary,
-- otherwise return Nothing
occursDict k [] = Nothing
occursDict k (x:xr) = if (fst x)==k then Just (snd x)
                                    else occursDict k xr

instance Functor Maybe where
 fmap = mapMaybe where
   mapMaybe Nothing = Nothing
   mapMaybe (Just v) = Just (f v)
```

NestedList and map over nested lists

Nested Lists

```
data NestedList a  = Nil |
  LL1 a (NestedList a) |
  LL2 (NestedList a) (NestedList a) deriving Show



nlmap f Nil          = Nil
nlmap f (LL1 x ys)  = LL1 (f x) (nlmap f ys)
nlmap f (LL2 xs ys) = LL2 (nlmap f xs) (nlmap f ys)
```

Laws

- `fmap` preserves the structure (shape and number of elements)

Laws

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

Note about laws

- Prove this (on paper) for each `fmap` implementation.
- Haskell compiler does not enforce this.
- What's wrong with `mapDestroy f xs = []` as an `fmap` for lists? Which law does it violate? Does it obey any law?
- Try a similar function for binary trees, and verify that it will compile. This shows that these laws are properties that we have to ensure manually, and are beyond the type-checker or the compiler.

Outline

Motivation

- Functors are for one-argument functions
- can we generalize for multi-argument functions?
- use currying

Motivation, continued

- e.g. `fmap2 (+) (Just 1) (Just 2)` operates with addition, which requires two arguments.

Currying

converting a multi-argument function into a sequence of partially-evaluated single argument functions

Currying

converting a multi-argument function into a sequence of partially-evaluated single argument functions

Example of curried addition

```
add = (\l x -> (\l y -> x+y))
```

Explanation

1. on one argument x, it returns a function
2. this function takes an argument y and returns x+y
3. the second function has access to x because of lexical scoping
4. uses the concept of closure.

Applicatives

Applicative

```
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Maybe as Applicative

An implementation of Maybe as Applicative

```
data MyMaybe a = MyNothing | MyJust a
  deriving Show

instance Functor MyMaybe where
  fmap = mapMyMaybe where
    mapMyMaybe f MyNothing = MyNothing
    mapMyMaybe f (MyJust x) = MyJust (f x)

instance Applicative MyMaybe where
  pure = MyJust
  (MyJust foo) <*> mx = fmap foo mx
```

List as Applicative

List as Applicative

```
instance Applicative [] where
  pure x = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Example usage

```
add3 x y z = x+y+z
pure add3 <*> (MyJust 3) <*> (MyJust 4) <*> (MyJust 5)
add3 <$> [1,2,3] <*> [4,5,6] <*> [7,8,9]
```

Applicative Laws

Outline