Applicative Functors

October 22, 2019

Functions

- Consider the addition function:

Functions

- Consider the addition function:
  - `1 + 1 = 2`

Functions

- Consider the addition function:
  - `1 + 1 = 2`
  - `2 + 2 = 4`

Functions

- Consider the addition function:
  - `1 + 1 = 2`
  - `2 + 2 = 4`
- `(+) :: Num a => a -> a -> a`

Maps

- Remember maps:

Maps

- Remember maps:
  - `map succ [1,2] = [2,3]`

Maps

- Remember maps:
  - `map succ [1,2] = [2,3]`
  - `map (+) [0,2] [1,2] =`

Maps

- Remember maps:
  - `map succ [1,2] = [2,3]`
  - `map (+) [0,2] [1,2] =`

    ```
    Couldn't match expected type '[Integer] -> t'
    with actual type '[Integer -> Integer]'
    Relevant bindings include it :: t (bound at
    <interactive>:2:1)
    The function 'map' is applied to three arguments,
    but its type '(Integer -> Integer -> Integer)
    -> [Integer] -> [Integer -> Integer]'
    has only two
    In the expression: map (+) [0, 2] [1, 2]
    In an equation for 'it': it = map (+) [0, 2] [1, 2]
    ```

Maps

- First of all, what do we expect `map (+) [0,2] [1,2]` to be?

Maps

- First of all, what do we expect `map (+) [0,2] [1,2]` to be?
  - Python: `[0,2] + [1,2] = [0,2,1,2]`

Maps

- Lists as nondeterminism:

Maps

- Lists as nondeterminism:
  - We want to add two numbers, but we don't know what they are

Maps

- Lists as nondeterminism:
  - We want to add two numbers, but we don't know what they are
  - All we know is that we have two boxes of numbers, `[0,2]` and `[1,2]`

Maps

- Lists as nondeterminism:
  - We want to add two numbers, but we don't know what they are
  - All we know is that we have two boxes of numbers, `[0,2]` and `[1,2]`
  - We pick a number from the first box and a number from the second box, and add them

Maps

- Lists as nondeterminism:
    - We want to add two numbers, but we don't know what they are
    - All we know is that we have two boxes of numbers, `[0,2]` and `[1,2]`
    - We pick a number from the first box and a number from the second box, and add them
    - What are our possible results?

Maps

- Lists as nondeterminism:
    - We want to add two numbers, but we don't know what they are
    - All we know is that we have two boxes of numbers, `[0,2]` and `[1,2]`
    - We pick a number from the first box and a number from the second box, and add them
    - What are our possible results?
        - `[0+1,0+2,2+1,2+2] = [1,2,3,4]`

Maps

- The function ‘map’ is applied to three arguments, but its type ‘(Integer -> Integer -> Integer) -> [Integer] -> [Integer -> Integer]’

  has only two

Maps

- `The function ‘map’ is applied to three arguments, but its type ‘(Integer -> Integer -> Integer) -> [Integer] -> [Integer -> Integer]’`

  `has only two`

  - Let's give it two arguments!

Maps

- `The function 'map' is applied to three arguments, but its type '(Integer -> Integer -> Integer) -> [Integer] -> [Integer -> Integer]'`

  `has only two`

  - Let's give it two arguments!
  - `(map (+) [0,2]) [1,2] = ([(0+),(2+)]) [1,2] =`

Maps

- The function ‘map’ is applied to three arguments,
  but its type ‘(Integer -> Integer -> Integer)
  -> [Integer] -> [Integer -> Integer]’

  has only two

  - Let's give it two arguments!
  - (map (+) [0,2]) [1,2] = ([(0+),(2+)]) [1,2] =

    Couldn't match expected type ‘[Integer] -> t’
    with actual type ‘[Integer -> Integer]’
    Relevant bindings include it :: t (bound at
    <interactive>:3:1)
    The function ‘[(0 +), (2 +)]’ is applied to one
    argument,
    but its type ‘[Integer -> Integer]’ has none
    In the expression: ([(0 +), (2 +)]) [1, 2]

    In an equation for ‘it’: it = ([(0 +), (2 +)]) [1, 2]

Functors

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)

Functors

- Lists are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `[a] -> [b]`)

Functors

- Lists are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `[a] -> [b]`)
- But now we have functions inside of boxes (of type `[a -> b]`)

Functors

- Lists are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `[a] -> [b]`)
- But now we have functions inside of boxes (of type `[a -> b]`)
  - How do we extract these functions and apply them to a box of type `[a]` to get a box of type `[b]`?

Applicative Functors

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

Applicative Functors

- `class (Functor f) => Applicative f where`
```
      pure ::  a -> f a
     (<*>) ::  f (a -> b) -> f a -> f b
```
  - pure takes a value and puts it in a box

Applicative Functors

- `class (Functor f) => Applicative f where`
```
     pure ::  a -> f a
    (<*>) ::  f (a -> b) -> f a -> f b
```
  - pure takes a value and puts it in a default context

Applicative Functors

- class (Functor f) => Applicative f where

```
      pure ::  a -> f a
     (<*>) ::  f (a -> b) -> f a -> f b
```

  - pure takes a value and puts it in a default context
  - (<*>) takes a box of functions and returns a function over boxes

Applicative Functors

- ```
  class (Functor f) => Applicative f where
      pure ::  a -> f a
     (<*>) ::  f (a -> b) -> f a -> f b
  ```
  - pure takes a value and puts it in a default context
  - (<*>) takes a function in a context and returns a function over contexts
  ```

Lists are Applicative Functors

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

Lists are Applicative Functors

- `[(0+),(2+)] <*> [1,2] = [f x | f <- [(0+),(2+)], x <- [1,2]]`

Lists are Applicative Functors

```
• [(0+),(2+)] <*> [1,2] = [f x | f <- [(0+),(2+)], x <- [1,2]]
                        = [(0+) 1, (0+) 2, (2+) 1, (2+) 2]
```

Lists are Applicative Functors

- ```
  [(0+),(2+)] <*> [1,2] = [f x | f <- [(0+),(2+)], x <- [1,2]]
                        = [(0+) 1, (0+) 2, (2+) 1, (2+) 2]
                        = [1,2,3,4]
  ```
  ```

Applicative Style

- `[1,2,3,4] = [(0+),(2+)] <*> [1,2]`

Applicative Style

- ```
  [1,2,3,4] = [(0+),(2+)] <*> [1,2]
            = (fmap (+) [0,2]) <*> [1,2]
  ```
  ```

Applicative Style

-
```
[1,2,3,4] = [(0+),(2+)] <*> [1,2]
          = (fmap (+) [0,2]) <*> [1,2]
          = (+) <$> [0,2] <*> [1,2]
```

Applicative Style

- ```
  [1,2,3,4] = [(0+),(2+)] <*> [1,2]
            = (fmap (+) [0,2]) <*> [1,2]
            = (+) <$> [0,2] <*> [1,2]
  ```
  - `f <$> x = fmap f x`
  ```

Applicative Style

- ```
  [1,2,3,4] = [(0+),(2+)] <*> [1,2]
            = (fmap (+) [0,2]) <*> [1,2]
            = (+) <$> [0,2] <*> [1,2]
  ```
  - `f <$> x = fmap f x`
    - Does this remind you of anything?
  ```

Applicative Style

- `1 + 1 = 2`

Applicative Style

- `1 + 1 = 2`

  `(+)        1        1   =   2`

Applicative Style

- `1 + 1 = 2`

```
 (+)      1       1  =  2
((+)  $   1   )   1  =  2
```

Applicative Style

- `1 + 1 = 2`

```
 (+)        1        1  =  2
((+)  $     1   )    1  =  2
 (+) <$>   [1] <*>  [1] = [2]
```

Applicative Style

- `1 + 1 = 2`

```
 (+)        1         1   =   2
((+)   $    1    )    1   =   2
 (+) <$>  [1]  <*>  [1]   =  [2]
```

  - `$` is function application, `<$>` is lifted function application

Applicative Style

- `1 + 1 = 2`

```
 (+)       1       1  =  2
((+)  $    1   )   1  =  2
 (+) <$> [1] <*> [1] = [2]
```

  - `$` is function application, `<$>` is lifted function application
  - `liftA2 f a b = f <$> a <*> b` (imported from `Control.Applicative`)

IO is an Applicative Functor

- `instance Applicative IO where`

```
pure = return
a <*> b = do
    f <- a
    x <- b
    return (f x)
```

IO is an Applicative Functor

- `instance Applicative IO where`

```
pure = return

a <*> b = do
    f <- a
    x <- b
    return (f x)
```

- `instance Functor IO where`

```
f <$> b = do

    x <- b
    return (f x)
```

IO is an Applicative Functor

- ```
  instance Applicative IO where
      pure = return
      a <*> b = do
          f <- a
          x <- b
          return (f x)
  ```

- ```
  instance Functor IO where

      f <$> b = do

          x <- b
          return (f x)
  ```

- Get an `x` from the outside world, apply `f` to `x`, and wrap it up in an `IO` box

IO is an Applicative Functor

- `instance Applicative IO where`

```
pure = return

a <*> b = do
    f <- a
    x <- b
    return (f x)
```

- `instance Functor IO where`

```
f <$> b = do

    x <- b
    return (f x)
```

- Get an `x` from the outside world, apply `f` to `x`, and wrap it up in an `IO` box
- Get both an `f` and an `x` from the outside world, apply `f` to `x`, and wrap it up in an `IO` box

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b
  ```
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b

          = (++) <$> getLine <*> getLine
  ```
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b
  ```

  ```
          = (++) <$> getLine <*> getLine
  ```

  - Get a line `a`, apply `(++)` to `a` (to get `((++) a)`), and wrap it up in an `IO` box
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b
  ```

  ```
          = (++) <$> getLine <*> getLine
  ```

  - Get a line `a`, apply `(++)` to `a` (to get `((++) a)`), and wrap it up in an `IO` box
  - Take `((++) a)` out of the box, get another line `b`, apply `((++) a)` to `b` (to get `a ++ b`), and wrap it up in another `IO` box
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
        a <- getLine
        b <- getLine
        return $ a ++ b

                = (++) <$> getLine <*> getLine
  ```
  - Actions
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. "Return" the lines concatenated together

- ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b
  ```

  ```
          = (++) <$> getLine <*> getLine
  ```

    - What to do with the results
  ```

Sequencing Actions

- Sequencing more actions

Sequencing Actions

- Sequencing more actions
  - ```
    (\x y z -> x ++ y ++ z)
      <$> getLine <*> getLine <*> getLine
    ```
    ```

Sequencing Actions

- Sequencing more actions
  - ```
    (\x y z -> x ++ y ++ z)
     <$> getLine <*> getLine <*> getLine
    = liftA3 (\x y z -> x ++ y ++ z)
        getLine getLine getLine
    ```
    ```

Sequencing Actions

- Sequencing more actions
  - ```
    (\x y z -> x ++ y ++ z)
     <$> getLine <*> getLine <*> getLine
    = liftA3 (\x y z -> x ++ y ++ z)
        getLine getLine getLine
    ```
  - ```
    (\w x y z -> w ++ x ++ y ++ z)
     <$> getLine <*> getLine <*> getLine <*> getLine
    ```

Sequencing Actions

- Sequencing more actions
  - ```
    (\x y z -> x ++ y ++ z)
     <$> getLine <*> getLine <*> getLine
    = liftA3 (\x y z -> x ++ y ++ z)
        getLine getLine getLine
    ```
  - ```
    (\w x y z -> w ++ x ++ y ++ z)
     <$> getLine <*> getLine <*> getLine <*> getLine
    ≠ liftA4 (\w x y z -> w ++ x ++ y ++ z)
        getLine getLine getLine getLine
    = <interactive>:4:1:  Not in scope:  ‘liftA4’
    ```

Sequencing Actions

- Sequencing an arbitrary number of actions

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`
    - `sequenceA :: (Applicative f) => [f a] -> f [a]`

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`
    - `sequenceA :: (Applicative f) => [f a] -> f [a]`
    - Takes a list of actions and returns an action that contains a list of results

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`
    - `sequenceA :: (Applicative f) => [f a] -> f [a]`
    - Takes a list of actions and returns an action that contains a list of results
  - What to do with the results

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`
    - `sequenceA :: (Applicative f) => [f a] -> f [a]`
    - Takes a list of actions and returns an action that contains a list of results
  - What to do with the results
    - ```
      (foldr (++) "")
        <$> sequenceA [getLine, getLine, getLine]
      ```
      ```

Sequencing Actions

- Sequencing an arbitrary number of actions
  - `sequenceA [getLine, getLine, getLine]`
    - `sequenceA :: (Applicative f) => [f a] -> f [a]`
    - Takes a list of actions and returns an action that contains a list of results
  - What to do with the results
    - ```
      (foldr (++) "")
        <$> sequenceA [getLine, getLine, getLine]
      ```
    - See Chapter 6.5 for folds
      ```

Applicative Laws

- Identity: `pure id <*> v = v`

Applicative Laws

- Identity: `pure id <*> v = v`
- Composition: `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`

Applicative Laws

- Identity: `pure id <*> v = v`
- Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Compare to functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`

Applicative Laws

- Identity: `pure id <*> v = v`
- Composition: `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`
- Compare to functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.)  u v <$> w = u <$> (v <$> w)`
- Compare to definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.)  u v $ w = u $ (v $ w)`

Applicative Laws

- Identity: `pure id <*> v = v`
- Composition: `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`
- Homomorphism: `pure f <*> pure x = pure (f x)`

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`

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`

- Bonus: `pure f <*> x = fmap f x = f <$> x`

Applicative Functors

- Other examples of applicative functors:

Applicative Functors

- Other examples of applicative functors:
    - `Maybe`

Applicative Functors

- Other examples of applicative functors:
  - `Maybe`
  - Functions (`(->) r`)

Applicative Functors

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)

Applicative Functors

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)
- Applicative functors are boxes that support function application

Applicative Functors

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)
- Applicative functors are boxes that support function application
  - If you have a normal function (`a -> b`), you can put it in a box (`F (a -> b)`), and apply it to a box (`F a`) to get another box (`F b`)

Applicative Functors

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions over context (of type `F a -> F b`)
- Applicative functors represent contexts that support function application
  - If you have a normal function (`a -> b`), you can put it in a context (`F (a -> b)`), and apply it to a context (`F a`) to get another context (`F b`)