Functors and Applicative Functors

Kenneth Lai

Brandeis University

October 19, 2022

Functors and Applicative Functors

Kenneth Lai

Brandeis University

October 19, 2022

Source

Announcements

- By 11:59pm tonight
  - HW2 due
  - Paper Presentation Ideas due
    - Don't worry if you're still waiting for a group–I'm still matching people up
- For next Monday
  - Read Huth and Ryan Chapter 5.1, 5.2, 3.2
- For 11/2
  - HW3 due
- For 11/9
  - Final Project Ideas due
    - More details next week

Today's Plan

- Extension and Intension: Two Ideas
- Functors
- Applicative Functors

Extension and Intension

- Idea 1: If the interpretation of something in an extensional model has type α, then its intensional interpretation has type $s \to \alpha$, where s is the type of possible worlds (`World`)

Extension and Intension

- Idea 1: If the interpretation of something in an extensional model has type α, then its intensional interpretation has type $s \to \alpha$, where s is the type of possible worlds (`World`)
  - Introduce abbreviations for types `World -> Entity` and `World -> Bool`

```
type IEntity = World -> Entity
type IBool   = World -> Bool
```

Extension and Intension

```
iSnowWhite :: IEntity
iSnowWhite W1 = snowWhite
iSnowWhite W2 = snowWhite'
iSnowWhite W3 = snowWhite'

iGirl, iPrincess, iPerson :: World -> Entity -> Bool
iGirl     W1 = girl
iGirl     W2 = girl'
iGirl     W3 = girl'
iPrincess W1 = princess
iPrincess W2 = princess'
iPrincess W3 = girl'
iPerson   W1 = person
iPerson   W2 = person'
iPerson   W3 = person'
```

Extension and Intension

```
iLaugh, iShudder :: World -> Entity -> Bool
iLaugh W1 =  laugh
iLaugh W2 =  laugh'
iLaugh W3 =  laugh'
iShudder W1 =  shudder
iShudder W2 =  shudder'
iShudder W3 =  shudder'

iCatch :: World -> Entity -> Entity -> Bool
iCatch W1 = \ x y -> False
iCatch W2 = \ x y -> False
iCatch W3 = \ x y -> elem x [B,R,T] && girl' y
```

Extension and Intension

- Idea 2: If the extensional interpretation of a linguistic expression has some type, then its intensional interpretation has the type that replaces all instances of e with $s \to e$ (`IEntity`) and all instances of t with $s \to t$ (`IBool`)

Extension and Intension

- Some are easier than others...

```
iSent :: Sent -> IBool
iSent (Sent np vp) = iNP np (iVP vp)

iNP :: NP -> (IEntity -> IBool) -> IBool
iNP SnowWhite = \ p -> p iSnowWhite
iNP (NP1 det cn)  = iDET det (iCN cn)

iVP :: VP -> IEntity -> IBool
iVP Laughed   = \ x i -> iLaugh i (x i)
iVP Shuddered = \ x i -> iShudder i (x i)

iCN :: CN -> IEntity -> IBool
iCN Girl = \ x i -> iGirl i (x i)
iCN Princess = \ x i -> iPrincess i (x i)
```

Extension and Intension

- Some are easier than others...

```
iNP Everyone  = \ p i -> all (\x -> p (\j -> x) i)
      (filter (\y -> iPerson i y) entities)
iNP Someone  = \ p i -> any (\x -> p (\j -> x) i)
      (filter (\y -> iPerson i y) entities)

iDET :: DET -> (IEntity -> IBool)
            -> (IEntity -> IBool) -> IBool
iDET Some p q = \ i -> any (\x -> q (\j -> x) i)
      (filter (\x -> p (\j -> x) i) entities)
iDET Every p q = \ i -> all (\x -> q (\j -> x) i)
      (filter (\x -> p (\j -> x) i) entities)
iDET No p q = \ i -> not (any (\x -> q (\j -> x) i)
      (filter (\x -> p (\j -> x) i) entities))
```

Extension and Intension

- There is method to this madness!
  - We can express the intensionalization process in terms of functors, in particular, applicative functors

Computing with Boxes

- I/O types are boxes
  - `IO a` is the type of a function that performs an I/O action and returns an object of type a in a box

Computing with Boxes

- I/O types are boxes
  - `IO a` is the type of a function that performs an I/O action and returns an object of type a in a box
- Lists are boxes
  - Learn You a Haskell book: "You can think of a list as a box that has an infinite amount of little compartments and they can all be empty, one can be full and the others empty or a number of them can be full."

Computing with Boxes

- Suppose we have a function of type `(a -> b)`, and (an) object(s) of type `a` in a box. How can we apply the function to the object(s)?

Computing with Boxes

- Suppose we have a function of type `(a -> b)`, and (an) object(s) of type `a` in a box. How can we apply the function to the object(s)?
- Lists: `map :: (a -> b) -> [a] -> [b]`
  - "The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members."

Computing with Boxes

- I/O: something like this

```
iomap :: (a -> b) -> IO a -> IO b
iomap f action = do
    result <- action
    return (f result)
```

Computing with Boxes

- I/O: something like this

```
iomap :: (a -> b) -> IO a -> IO b
iomap f action = do
    result <- action
    return (f result)
```

- Bind the result of `action` to `result`
- Apply `f` to `result` and put it in a box

Functors

- Functors are boxes

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

Functors

- Functors are boxes

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

- `Functor` is a type class that contains types that can be "mapped" over
  - `F` is a polymorphic type (i.e., type constructor)

Functors

- Functors are boxes

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

- `Functor` is a type class that contains types that can be "mapped" over
  - `F` is a polymorphic type (i.e., type constructor)

```
instance Functor [] where
    fmap = map

instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

Functors

- Functors are boxes

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

- `fmap` takes a function (of type `a -> b`) and a box of `a` and outputs a box of `b`

Functors

- Functors are boxes

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

- `fmap` takes a function (of type `a -> b`) and a box of `a` and outputs a box of `b`
- Alternatively, `fmap` takes a function (of type `a -> b`) and lifts it to a function over boxes (of type `F a -> F b`)

Functors

- Functor laws (from the Learn You a Haskell book):
  - “All functors are expected to exhibit certain kinds of functor-like properties and behaviors.
    - They should reliably behave as things that can be mapped over.
    - Calling `fmap` on a functor should just map a function over the functor, nothing more.
  - This behavior is described in the functor laws.”

Functors

- Functor laws (from the Learn You a Haskell book):
  - "All functors are expected to exhibit certain kinds of functor-like properties and behaviors.
    - They should reliably behave as things that can be mapped over.
    - Calling `fmap` on a functor should just map a function over the functor, nothing more.
  - This behavior is described in the functor laws."
- Identity: `fmap id = id`
- Composition: `fmap (g . f) = fmap g . fmap f`

Functors

- Functor laws (from the Learn You a Haskell book):
  - "All functors are expected to exhibit certain kinds of functor-like properties and behaviors.
    - They should reliably behave as things that can be mapped over.
    - Calling `fmap` on a functor should just map a function over the functor, nothing more.
  - This behavior is described in the functor laws."
- Identity: `fmap id = id`
- Composition: `fmap (g . f) = fmap g . fmap f`
- That is, functors must preserve identity and composition of functions
  - Haskell will not enforce this for you–you have to do it yourself

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

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions of objects in context (of type `F a -> F b`)

Functors

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions of objects in context (of type `F a -> F b`)
    - `IO`: input/output
    - `[]`: nondeterminism

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?

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]`

Lists as Nondeterminism

- We have a function of numbers and a box of numbers, let's `map` the function over the list

```
map (+) [0,2] = [(0+),(2+)]
```

Lists as Nondeterminism

- We have a function of numbers and a box of numbers, let's `map` the function over the list

```
map (+) [0,2] = [(0+),(2+)]
```

- Now we have a box of functions
  - How can we extract the functions and apply them to the second box of numbers?

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 (of type `a`) and puts it in a box (of type `F a`)
- `(<*>)` takes a box of functions (of type `F (a -> b)`) and returns a function of boxes (of type `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 (of type `a`) and puts it in a default context (of type `F a`)
- `(<*>)` takes a function in a context (of type `F (a -> b)`) and returns a function of objects in context (of type `F a -> F b`)

Applicative Functors

- Lists are applicative functors

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

Applicative Functors

- Lists are applicative functors

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

- `pure` makes a singleton list
- `<*>` takes each function `f` in `fs` and each argument `x` in `xs`, applies `f` to `x`, and puts it in a list

Applicative Functors

- Lists are applicative functors

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

- `pure` makes a singleton list
- `<*>` takes each function `f` in `fs` and each argument `x` in `xs`, applies `f` to `x`, and puts it in a list

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

- Can also be written `(+) <$> [0,2] <*> [1,2] = [1,2,3,4]`, where `<$>` is an infix version of `fmap`

Applicative Functors

- I/O types are applicative functors

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

Applicative Functors

- I/O types are applicative functors

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

- `pure` puts its argument in an `IO` box
- `<*>` binds the contents of `a` and `b` to `f` and `x` respectively, applies `f` to `x`, and puts it in an `IO` box

Applicative Functors

- 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 Functors

- 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

- 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
  - If you have a function in a box (`F (a -> b)`), you can 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 function in a context (`F (a -> b)`), you can apply it to an object in context (`F a`) to get another object in context (`F b`)

Functions as Functors

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- (Technically, functions that take arguments of type `r` are functors, where `r` is any type)

Functions as Functors

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- (Technically, functions that take arguments of type `r` are functors, where `r` is any type)
- A function of type `r -> a` can be seen as an object (of type `a`) that depends on the context (of type `r`)
  - Can also be seen as a box containing the eventual result of running the function

Functions as Functors

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- (Technically, functions that take arguments of type `r` are functors, where `r` is any type)
- A function of type `r -> a` can be seen as an object (of type `a`) that depends on the context (of type `r`)
  - Can also be seen as a box containing the eventual result of running the function
- Note that `fmap` is just function composition
  - `fmap = (.)`

Functions as Applicative Functors

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

Functions as Applicative Functors

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

- `pure` takes a value (of type `a`) and makes a "default" function (of type `r -> a`)
    - The most "default" function is the one that, no matter the argument, always outputs that value

Functions as Applicative Functors

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

- `pure` takes a value (of type `a`) and makes a "default" function (of type `r -> a`)
  - The most "default" function is the one that, no matter the argument, always outputs that value
- `<*>` is a function that
  - Takes functions `f :: r -> a -> b` and `g :: r -> a`, and a context `x :: r`
  - Applies both `f` and `g` to `x` (to get `(f x) :: a -> b` and `(g x) :: a`)
  - Applies `(f x)` to `(g x)` to get a result of type `b`
- `<*> :: (r -> a -> b) -> (r -> a) -> r -> b`