Applicative Functors in Language and Intensional Constructs

James Pustejovsky

Brandeis University

September 27, 2024

Today's Plan

- Applicative Functors in Language
- Intensional Constructs
- Modal Logic
  - Syntax
  - Semantics

Today's Plan

- Applicative Functors in Language
- Intensional Constructs
- Modal Logic
  - Syntax
  - Semantics
- (we'll see how far we get...)

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`

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
  - Alternatively, takes a function `f :: r -> a -> b` and lifts it to a function `(<*>) f :: (r -> a) -> r -> b`

- `<*> :: (r -> a -> b) -> (r -> a) -> r -> b`

Applicative Functors in Language

- Let `r` = `World`
    - Worlds are contexts, both in the linguistic sense and in the computational sense

Applicative Functors in Language

- Let `r` = `World`
    - Worlds are contexts, both in the linguistic sense and in the computational sense
- Consider intensional verb phrases/common nouns, which have type `(World -> Entity) -> World -> Bool` (or `IEntity -> IBool`)
    - `(iVP Laughed)`, `(iCN Girl)`, etc.
- Our model contains intensional relations of type `World -> Entity -> Bool`
    - `iLaugh`, `iGirl`, etc.

Applicative Functors in Language

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

- Given functions `iLaugh :: World -> Entity -> Bool` and `x :: World -> Entity`, apply both `iLaugh` and `x` to the same context `i :: World`
- Then apply `(iLaugh i)` to `(x i)`

Applicative Functors in Language

```
iProp :: (World -> Entity -> Bool) -> IEntity -> IBool
iProp x = \ y i -> x i (y i)
```

- "This function can be used for automating the lift of extensional CN and VP denotations to intensional ones."

Applicative Functors in Language

```
iProp :: (World -> Entity -> Bool) -> IEntity -> IBool
iProp x = \ y i -> x i (y i)
```

- "This function can be used for automating the lift of extensional CN and VP denotations to intensional ones."

```
vpINT :: VP -> World -> Entity -> Bool
vpINT Laughed   = iLaugh
vpINT Shuddered = iShudder

intensVP :: VP -> IEntity -> IBool
intensVP = iProp . vpINT
```

- "This defines the same function as `iVP`."

Applicative Functors in Language

- But note that `iProp = (<*>)`
- This means that we can write

  ```
  iVP Laughed    = \ x -> iLaugh <*> x
  ```

  - Alternatively, we can write

    ```
    iVP Laughed    = iProp iLaugh
    ```

Applicative Functors in Language

- `<*>` (or `iProp`) takes a function `f :: r -> a -> b` and lifts it to a function `iProp f :: (r -> a) -> r -> b`
  - Can we do the reverse, i.e., can we take a function `iProp f :: (r -> a) -> r -> b` and lower it to a function `f :: r -> a -> b`?

Applicative Functors in Language

```
eProp :: (IEntity -> IBool) -> World -> Entity -> Bool
eProp y = \ j x -> y (\k -> x) j
```

- `eProp` takes a function `y :: (IEntity -> IBool)`, and returns a function that takes a world `j :: World` and an entity `x :: Entity`, and applies `y` to "`x`" and `j`
  - `y` takes an `IEntity` as input, while `x` is an `Entity`
    - We lift `x` to the type `IEntity`, by making a "default" function `(\k -> x)`
    - Note that `(\k -> x) = pure x`

Applicative Functors in Language

```
eProp :: (IEntity -> IBool) -> World -> Entity -> Bool
eProp y j = \ x -> y (pure x) j
```

- eProp takes a function `y :: (IEntity -> IBool)` and a world `j :: World`, and returns a function that takes an entity `x :: Entity`, and applies `y` to `pure x` and `j`

Applicative Functors in Language

- Consider intensional determiners, which have type `(IEntity -> IBool) -> (IEntity -> IBool) -> IBool`
- But any and `filter` take relations of type `Entity -> Bool`

Applicative Functors in Language

- Consider intensional determiners, which have type `(IEntity -> IBool) -> (IEntity -> IBool) -> IBool`
- But `any` and `filter` take relations of type `Entity -> Bool`

```
iDET Some p q = \ i -> any (\x -> q (\j -> x) i)
      (filter (\x -> p (\j -> x) i) entities)
```

- We lower `p` and `q` from type `IEntity -> IBool` to type `Entity -> Bool`

Applicative Functors in Language

- But note that `(\x -> q (\j -> x) i) = eProp q i`
- This means that we can write

```
iDET Some p q = \ i -> any (eProp q i)
      (filter (eProp p i) entities)
```

Applicative Functors in Language

- Warning! Not every function can be “extensionalized” in this manner
  - Basically, `eProp` loses information
    - `(World -> Entity) -> World -> Bool` contains two instances of `World`, while `World -> Entity -> Bool` contains one

Applicative Functors in Language

- Warning! Not every function can be “extensionalized” in this manner
  - Basically, `eProp` loses information
    - `(World -> Entity) -> World -> Bool` contains two instances of `World`, while `World -> Entity -> Bool` contains one
- We were able to get away with it because all of our intensional predicates were of the form `iProp p` (or `\ x -> p <*> x`)
  - `eProp (iProp p) = p`, but it is not necessarily the case that `iProp (eProp p) = p`
    - See van Eijck and Unger Chapter 8.4 for more details

Intensional Constructs

- "A fake princess is someone who in actual fact is not a princess, but pretends to be one. How does one model such pretense? Let us say that in some other world she is a princess."

Intensional Constructs

- “A fake princess is someone who in actual fact is not a princess, but pretends to be one. How does one model such pretense? Let us say that in some other world she is a princess.”

```
iADJ :: ADJ -> (IEntity -> IBool) -> IEntity -> IBool
iADJ Fake = \ p x i ->
  not (p x i) && any (\ j -> p x j) worlds
```

Intensional Constructs

- "Attitude verbs like *want* and *hope* also give rise to intensional constructs. Such verbs combine with infinitives to form complex VPs."

Intensional Constructs

- "Attitude verbs like *want* and *hope* also give rise to intensional constructs. Such verbs combine with infinitives to form complex VPs."

```
iINF :: INF -> IEntity -> IBool
iINF Laugh   = \ x i -> iLaugh i (x i)
iINF Shudder = \ x i -> iShudder i (x i)
iINF (INF tinf np) = \ s -> iNP np (\ o -> iTINF tinf s o)

iTINF :: TINF -> IEntity -> IEntity -> IBool
iTINF Catch = \x y w -> iCatch w (x w) (y w)
```

- Note that, e.g., `iINF Laugh = \ x -> iLaugh <*> x` and `iTINF Catch = \x y -> iCatch <*> x <*> y`

Intensional Constructs

- "An attitude towards an intensional property should map that property to a property that holds in all worlds where the attitude is realized. So for each agent in the model that can hold attitudes, we have to identify the set of worlds for that attitude: desired worlds or hoped-for worlds. Let us assume that in all worlds everyone wants w_2 or w_3 and everyone hopes for w_3."

Intensional Constructs

- "An attitude towards an intensional property should map that property to a property that holds in all worlds where the attitude is realized. So for each agent in the model that can hold attitudes, we have to identify the set of worlds for that attitude: desired worlds or hoped-for worlds. Let us assume that in all worlds everyone wants w_2 or w_3 and everyone hopes for w_3."

```
iAttit :: AV -> IEntity -> IBool
iAttit Wanted x = \i -> elem i [W2,W3]
iAttit Hoped  x = \i -> i == W3
```

- Note that this is greatly simplified!

Intensional Constructs

- “To check whether a property holds under an attitude we check whether the property holds in all of the designated attitude worlds.”

Intensional Constructs

- “To check whether a property holds under an attitude we check whether the property holds in all of the designated attitude worlds.”

```
iAV :: AV -> (IEntity -> IBool) -> (IEntity -> IBool)
iAV Wanted p = \ x i ->
  and [ p x j | j <- worlds, iAttit Wanted x j ]
iAV Hoped  p = \ x i ->
  and [ p x j | j <- worlds, iAttit Hoped  x j ]
```

Intensional Constructs

- “Whether or not a statement is true in some model depends on what is the actual world in that model.
  - The actual world is the world where we evaluate.
- To check whether a statement is necessarily true in an intensional model, we have to check whether it is true in all possible worlds.
- A statement is contingently true if it is true, but it ain’t necessarily so: there exists a world where that statement is false.”

Intensional Constructs

- "Whether or not a statement is true in some model depends on what is the actual world in that model.
  - The actual world is the world where we evaluate.
- To check whether a statement is necessarily true in an intensional model, we have to check whether it is true in all possible worlds.
- A statement is contingently true if it is true, but it ain't necessarily so: there exists a world where that statement is false."

```
iJudgement :: Judgement -> IBool
iJudgement (IsTrue s) = \ i -> iSent s i
iJudgement (IsNec s) = \ i ->
  all (\j -> iSent s j) worlds
iJudgement (IsCont s) = \ i ->
  iSent s i && not (all (\j -> iSent s j) worlds)
```