Monads and Continuations, Part 2

Kenneth Lai

Brandeis University

November 2, 2022

Monads and Continuations, Part 2

Kenneth Lai

Brandeis University

November 2, 2022

Source

Monads and Continuations, Part 2

Kenneth Lai

Brandeis University

November 2, 2022

Source

Announcements

- Please continue to fill out the Mid-course Feedback!
- By 11:59pm today
  - HW3 due
- For 11/9
  - Final Project Idea due
- For 11/16
  - HW4 due

Today's Plan

- Final Project Idea: VP Ellipsis as Anaphora
- Monads
- Continuations in Language

Today’s Plan

- Final Project Idea: VP Ellipsis as Anaphora
- Monads
- Continuations in Language
- (we’ll see how far we get...)

VP Ellipsis as Anaphora

- "Bill laughed. Mary did too."
- Johnson: "VP ellipsis is the name given to instances of anaphora in which a missing predicate...is able to find an antecedent in the surrounding discourse"
- Bill PAST [$_{VP}$ laugh]. Mary did [$_{VP}$ $\varnothing$] too.

VP Ellipsis as Anaphora

- Provide an interpretation of VP ellipsis in the model, and determine if it is satisfied
  - Can be very similar to pronoun anaphora; see HW4 for details

VP Ellipsis as Anaphora

- Some things to think about, if you can
  - Unsaturated predicates
    - "Bill raised his hand. Mary did too."
    - Bill_i PAST $[_{VP}$ raise i's hand]. Mary_i did $[_{VP}\ \varnothing]$ too.
    - (not #Bill PAST $[_{VP}$ raise Bill's hand]. Mary did $[_{VP}\ \varnothing]$ too.)

VP Ellipsis as Anaphora

- Some things to look at
  - Ronnie Cann, Ruth Kempson, and Eleni Gregoromichelaki (2009), Semantics: An Introduction to Meaning in Language, Chapter 7
  - Kyle Johnson (2001), "What VP Ellipsis Can Do, and What it Can't, But Not Why", In The Handbook of Contemporary Syntactic Theory, Mark Baltin and Chris Collins (eds.)

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

- `return` is just like `pure` for applicative functors

Monads

- To understand `(>>=)` (pronounced bind), it may help to think in terms of its flipped version, `(=<<)`

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare (=<<) with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

Monads

- To understand `(>>=)` (pronounced bind), it may help to think in terms of its flipped version, `(=<<)`

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare `(=<<)` with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

- `(=<<)` (and `(>>=)`) are maps for monadic functions
  - Functions that create their own boxes

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare (=<<) with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

- (=<<) (and (>>=)) are maps for monadic functions
  - Functions that create their own context

Monads

- To understand `(>>=)`, it may also help to think in terms of `join`

```
join :: (Monad M) => M (M a) -> M a
```

- If you have two nested boxes, `join`, well, "joins" them together

Monads

`g >>= f = join (fmap f g) :: M a -> (a -> M b) -> M b`

- `f :: a -> M b` is a monadic function
- `fmap f` lifts it to type `M a -> M (M b)`
- `g :: M a` is a value of type a in a box
- `fmap f g :: M (M b)` outputs a value of type `b` in two nested boxes
- `join (fmap f g)` extracts a monadic value of type `M b` from the outermost box

Monads

```
g >>= f = join (fmap f g) :: M a -> (a -> M b) -> M b
```

- `f :: a -> M b` is a monadic function
- `fmap f` lifts it to type `M a -> M (M b)`
- `g :: M a` is a value of type a in a box
- `fmap f g :: M (M b)` outputs a value of type `b` in two nested boxes
- `join (fmap f g)` extracts a monadic value of type `M b` from the outermost box

- `g >>= f` extracts a value of type a from g and feeds it to `f` to get a monadic value of type `M b`

Monads

- Examples of monadic functions
  - `putStrLn :: String -> IO ()`
  - `readFile :: FilePath -> IO String`

Monads

- Examples of monadic functions
  - `putStrLn :: String -> IO ()`
  - `readFile :: FilePath -> IO String`
- `getLine >>= putStrLn` extracts a `String` from `getLine` and feeds it to `putStrLn`
- `getLine >>= readFile` extracts a `FilePath` (i.e., `String`) from `getLine` and feeds it to `readFile`, which reads the file and puts its contents in a box

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

- (>>) is shorthand for when we don't need to bind the value inside `x` to evaluate `y`

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

- (>>) is shorthand for when we don't need to bind the value inside `x` to evaluate `y`
- `fail` is an error handler for pattern matching in do expressions

do notation

```
do {f}             = f
do {g; f}          = g >> do {f}
do {x <- g; f} = g >>= \ x -> do {f}
```

do notation

```
action = getLine >>= putStrLn
       = getLine >>= \ x -> putStrLn x
       = getLine >>= \ x -> do {putStrLn x}
```

do notation

```
action = getLine >>= putStrLn
       = getLine >>= \ x -> putStrLn x
       = getLine >>= \ x -> do {putStrLn x}

action = do
    x <- getLine
    putStrLn x
```

Monads

- Lists are monads

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

Monads

- Lists are monads

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

- `return` makes a singleton list
- `fail` makes the empty list
- What about `(>>=)`?

Monads

```
[1,2,3,4] == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             return (a + b)
```

Monads

```
[1,2,3,4] == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             return (a + b)

          == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             [a + b]
          == [0,2] >>= \ a ->
             concat (map (\ b -> [a + b]) [1,2])
          == [0,2] >>= \ a ->
             concat [[a+1], [a+2]]
          == [0,2] >>= \ a ->
             [a+1, a+2]
```

Monads

```
[1,2,3,4] == [0,2] >>= \ a -> [a+1, a+2]
          == concat (map (\ a -> [a+1, a+2]) [0,2])
          == concat [[0+1, 0+2], [2+1, 2+2]]
          == [1,2,3,4]
```

Monads

```
[1,2,3,4] == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             return (a + b)
```

Monads

```
[1,2,3,4] == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             return (a + b)

          == do
             a <- [0,2]
             b <- [1,2]
             return (a + b)
```

Monads

```
[1,2,3,4] == [0,2] >>= \ a ->
             [1,2] >>= \ b ->
             return (a + b)

          == do
             a <- [0,2]
             b <- [1,2]
             return (a + b)

          == [a + b |
             a <- [0,2]
             b <- [1,2]]
```

- List comprehensions are syntactic sugar for monadic computations!

Monads

- Monad laws:
- Left Identity: `return x >>= f = f x`
- Right Identity: `m >>= return = m`
- Associativity: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Monads

- 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`)

Monads

- 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`)
- Monads are boxes that support functions that create their own boxes
  - If you have a monadic function (`a -> M b`), you can apply it to a value (`a`) in a box (`M a`) to get another box (`M b`)

Monads

- 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`)
- Monads represent contexts that support functions that create their own contexts
  - If you have a monadic function (`a -> M b`), you can apply it to a value (`a`) in a context (`M a`) to get another context (`M b`)

Monads

- 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`)
- Monads represent contexts that can be joined together
  - If you have a context in another context (`M (M a)`), you can join the two contexts into one (`M a`)

Continuations in Language

- First, let us define some type synonyms
  - Note that `type Comp a r = (a -> r) -> r`

```
type Cont a r = a -> r
type Comp a r = Cont a r -> r
```

Continuations in Language

- What are these functions?

```
cpsConst :: a -> Comp a r
cpsConst c = \ k -> k c

cpsApply :: Comp (a -> b) r -> Comp a r -> Comp b r
cpsApply m n = \ k -> n (\ b -> m (\ a -> k (a b)))
```

Continuations in Language

- What are these functions?

```
cpsConst :: a -> Comp a r
cpsConst c = \ k -> k c

cpsApply :: Comp (a -> b) r -> Comp a r -> Comp b r
cpsApply m n = \ k -> n (\ b -> m (\ a -> k (a b)))
```

- Let `(Comp _ r)` be an applicative functor
  - `cpsConst = pure`
  - `cpsApply = (<*>)`

Continuations in Language

- We use cpsConst to lift values to computations

```
intNP_CPS :: NP -> Comp Entity Bool
intNP_CPS SnowWhite  = cpsConst snowWhite

intVP_CPS :: VP -> Comp (Entity -> Bool) Bool
intVP_CPS Laughed     = cpsConst laugh

intTV_CPS :: TV -> Comp (Entity -> Entity -> Bool) Bool
intTV_CPS Loved     = cpsConst love

intCN_CPS :: CN -> Comp (Entity -> Bool) Bool
intCN_CPS Girl      = cpsConst girl
```

Continuations in Language

- We use cpsApply to do function application within computations

```
intSent_CPS :: Sent -> Comp Bool Bool
intSent_CPS (Sent np vp) =
   cpsApply (intVP_CPS vp) (intNP_CPS np)

intVP_CPS (VP1 tv np) =
   cpsApply (intTV_CPS tv) (intNP_CPS np)
```

Continuations in Language

- We use cpsApply to do function application within computations

```
intSent_CPS :: Sent -> Comp Bool Bool
intSent_CPS (Sent np vp) =
   cpsApply (intVP_CPS vp) (intNP_CPS np)

intVP_CPS (VP1 tv np) =
   cpsApply (intTV_CPS tv) (intNP_CPS np)
```

- So far, so good!
    - No monads yet, though...

Continuations in Language

- van Eijck and Unger define special continuized determiner interpretations

```
intDET_CPS :: DET -> (Comp (Entity -> Bool) Bool)
                                -> (Comp Entity Bool)
intDET_CPS Some  = \ k p -> k (\ q ->
                     any p (filter q entities))
intDET_CPS Every = \ k p -> k (\ q ->
                     all p (filter q entities))
intDET_CPS No    = \ k p -> k (\ q ->
                     not (any p (filter q entities)))
intDET_CPS The   = \ k p -> k (\ q ->
                     singleton (filter q entities)
                     && p (head (filter q entities)))
                   where
                     singleton [x] = True
                     singleton _   = False
```

Continuations in Language

- We don't need them, though!
  - We will use our determiner interpretations from before

```
intDET :: DET ->
          (Entity -> Bool) -> (Entity -> Bool) -> Bool

intDET Some p q = any q (filter p entities)

intDET Every p q = all q (filter p entities)

intDET The p q = singleton plist && q (head plist)
          where
              plist = filter p entities
              singleton [x] = True
              singleton  _  = False

intDET No p q = not (intDET Some p q)
```

Continuations in Language

- Note that
  `(Entity -> Bool) -> (Entity -> Bool) -> Bool =`
  `(Entity -> Bool) -> Comp Entity Bool`
  - Determiner interpretations are monadic functions!

Continuations in Language

- Note that
  `(Entity -> Bool) -> (Entity -> Bool) -> Bool =`
  `(Entity -> Bool) -> Comp Entity Bool`
  - Determiner interpretations are monadic functions!

```
cpsBind :: Comp a r -> (a -> Comp b r) -> Comp b r
cpsBind x y = \ k -> x (\ a -> (y a) k)
```

- Let `(Comp _ r)` be a monad
  - `cpsBind = (>>=)`

Continuations in Language

- Note that
  `(Entity -> Bool) -> (Entity -> Bool) -> Bool =`
  `(Entity -> Bool) -> Comp Entity Bool`
  - Determiner interpretations are monadic functions!

```
cpsBind :: Comp a r -> (a -> Comp b r) -> Comp b r
cpsBind x y = \ k -> x (\ a -> (y a) k)
```

- Let `(Comp _ r)` be a monad
  - `cpsBind = (>>=)`

```
intNP_CPS (NP1 det cn)
   = cpsBind (intCN_CPS cn) (intDET det)
```

Continuations in Language

```
compSent s = intSent_CPS s id
```

- "We interpret sentences using the function `intSent_CPS`.
  - The result of that function is a sentence computation, i.e. a function of type `(Bool -> Bool) -> Bool`, that takes a sentence continuation (representing the linguistic context of the sentence) and returns a result value of type `Bool`.
  - An example of a possible sentence continuation is negation: if we had a negated sentence, we could apply the computation of the unnegated sentence to the negation function `neg`.
  - But here we do not want to bother about the linguistic context of sentences and instead want the sentence computation to return a result value of type `Bool`.
  - Therefore we apply the sentence computation to the trivial continuation, the identity function `id`."