Functional Thinking, Part 1

Christopher R. Genovese

Department of Statistics & Data Science

26 Sep 2024
Session #10

Plan

Finish No Loops

Plan

Finish No Loops

Categories and Functors, Part 2/n

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Design Activity

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Design Activity

Activity (cont'd)

We talked about the no-loops solutions, but let's implement one or two. Challenge yourself.

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Design Activity

Functors (cont'd)

A **Functor** is a "trait", a type `f : Type -> Type` with an associated `map` function:

```
trait Functor f where
    map : (a -> b) -> f a -> f b
```

We saw two instances of this trait

```
instance Functor Maybe where
    map : (a -> b) -> Maybe a -> Maybe b
    map f None = None
    map f (Some x) = Some (f x)
```

And

```
instance Functor List where
    map : (a -> b) -> List a -> List b
    map f Nil = Nil
    map f (Cons x rest) = Cons (f x) (map f rest)
```

Functors (cont'd)

A **Functor** is a "trait", a type `f : Type -> Type` with an associated `map` function:

```
trait Functor f where
    map : (a -> b) -> f a -> f b
```

Similarly `newtype Identity a = Identity { runIdentity : a }`

```
instance Functor Identity where
    map : (a -> b) -> Identity a -> Identity b
    map f (Identity x) = Identity (f x)
```

Consider also `newtype Const a b = Const { runConst : a }`

```
instance Functor (Const a) where
    map : (b -> c) -> Const b -> Const c
    map _ (Const x) = Const x
```

Functors (cont'd)

```
instance Functor ((->) r) where
    map : (a -> b) -> (r -> a) -> (r -> b)

    map f g =

    -- or equivalently
    map f g x =
```

Functors (cont'd)

```
instance Functor ((->) r) where
    map : (a -> b) -> (r -> a) -> (r -> b)

    map f g = f . g        -- (.) is composition

    -- or equivalently
    map f g x = f (g x)
```

Functors (cont'd)

```
type BinaryTree a = Node (BinaryTree a) a (BinaryTree a)

instance Functor BinaryTree where
    map : (a -> b) -> BinaryTree a -> BinaryTree b

    map f (Node left x right) =
```

Functors (cont'd)

```
type BinaryTree a = Node (BinaryTree a) a (BinaryTree a)

instance Functor BinaryTree where
    map : (a -> b) -> BinaryTree a -> BinaryTree b

    map f (Node left x right) = Node (map f left) (f x) (map f right)
```

Rose Trees

We will frequently use a more general tree structure, *rose trees*:

```
type Tree a = Node { value : a
                   , children : List (Tree a)
                   }

instance Functor Tree where
    map : (a -> b) -> Tree a -> Tree b

    map f tree =
```

Rose Trees

We will frequently use a more general tree structure, *rose trees*:

```
type Tree a = Node { value : a
                   , children : List (Tree a)
                   }

instance Functor Tree where
    map : (a -> b) -> Tree a -> Tree b

    map f tree = Node { value = f(tree.value)
                      , children = map f (tree.children)
                      }
```

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Design Activity

Minimax Analysis of Game Trees

Consider a traditional, two-player perfect-information game like tic-tac-toe or chess. We can analyze a game by looking at the "game tree" and scoring positions heuristically.

Assume we have types `Player` and `Position`. For instance,

```
type Player = X | O
type Position = TopLeft | TopMid | TopRight | ... | BotRight
```

We will let these be generic types as they can apply to any game.

To keep things simple, we'll start by ignoring the player, assuming that player can be determined from a position.

Minimax Analysis of Game Trees (cont'd)

We have a function

```
moves : Position -> List Position
```

and define (using rose trees)

```
propagate : (a -> List a) -> a -> Tree a
propagate f x = Node { value = x
                     , children = map (propagate f) (f x)
                     }
gameTree : Position -> Tree Position
gameTree = propagate moves
```

Minimax Analysis of Game Trees (cont'd)

We could handle the player as follows.

```
next : Player -> Player
moves : Player -> Position -> List Position
```

defining

```
propagate : (a -> List a) -> (a -> List a) -> a -> Tree a
propagate f1 f2 x = Node { value = x
                         , children = map (propagate f2 f1) (f1 x)
                         }
gameTree : Player -> Position -> Tree Position
gameTree player = propagate (moves player) (moves (next player))
```

But we'll keep to the simple version in what follows.

Minimax Analysis of Game Trees (cont'd)

Now imagine that we have some heuristic static evaluator for some position:

```
static : Position -> Number
```

Assume that the results are negative when they favor one player and positive when they favor another. This is a local guess that we will refine by analyzing the game tree. Note that `map static : Tree Position -> Tree Number`.

To extend our static analyzer, we lookahead in the tree, taking account of the best (greedy) move at each stage.

```
maximize : Tree Number -> Number
maximize (Node v Nil) = v
maximize (Node v sub) = max (map minimize sub)

minimize : Tree Number -> Number
minimize (Node v Nil) = v
minimize (Node v sub) = min (map maximize sub)

evaluate : Position -> Number
evaluate = maximize . map static . gameTree
```

This is fine, but it might not terminate. (Why?) And it can take a long time in any case.

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) =
prune n (Node v cs) =
```

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v Nil
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v Nil
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Now we have a more realistic evaluation

```
evaluate : Position -> Number
evaluate = maximize . map static . prune 4 . gameTree
```

which gives a lookahead of 4 moves.

We are using *lazy evaluation* here. This evaluates positions only as *demanded* by `maximize` so the whole tree is never in memory.

We can optimize this further.

Newton-Raphson Square Roots

Consider the recurrence relation $a_{n+1} = (a_n + x/a_n)/2$ for $x > 0$ and $a_0 = x$. As n increases, $a_n \to \sqrt{x}$.

We might compute with this typically with

```
def sqrt(x, tolerance=1e-7):
    u = x
    v = x + 2.0 * tolerance
    while abs(u - v) > tolerance:
        v = u
        u = 0.5 * (u + x / u)
    return u
```

We will put this in a more modular style with ingredients that can be reused for other problems.

Newton-Raphson Square Roots (cont'd)

```
next : Real -> Real -> Real
next x a = (a + x / a) / 2

iterate f x = Cons x (iterate f (f x))     -- a lazy sequence

iterate (next x) init   -- lazy sequence of sqrt approximations

within : Real -> List Real -> Real
within tol (Cons a0 (Cons a1 rest))
   | (abs(a0 - a1) <= tol) = a1
   | otherwise             = within tol (Cons a1 rest)

relative : Real -> List Real -> Real
relative tol (Cons a0 (Cons a1 rest))
   | (abs(a0/a1 - 1) <= tol) = a1
   | otherwise               = relative tol (Cons a1 rest)

a_sqrt init tol   = within tol (iterate (next x) init)
r_sqrt init tol x = relative tol (iterate (next x) init)
```

These same primitives apply to give us other approximations, e.g., numerical differentiation, integration, ...

Plan

Finish No Loops

Categories and Functors, Part 2/n

More Examples

Design Activity

Design Activity: Dominoes

See **dominoes** activity in problem-bank.

What are the layers of responsibility here?

What are the entities we need to manage/track?

What are the data? the actions? the calculations?

Sketch out the structure of the task.

For consideration later:

```
type Algebra f a = f a -> a
type CoAlgebra f a = a -> f a

-- Lazily build and reduce the tree
-- >>> is chained composition
hylo : Functor f => Algebra f a -> CoAlgebra f b -> b -> a
hylo alg coalg = coalg >>> map (hylo alg coalg) >>> alg
```

A Quick Tour of Zippers

See **zippers** in problem bank.

THE END