Recursion schemes

Using recursion-schemes to showcase BK-trees

BK-trees

BK-trees are a recursive tree structures that can be used to index into metric space. Each node is an item, each edge is the distance between the two items.

- Edit-distance similarities
- Hamming-distance similarities

Constructing BK-trees

I'm going over the insertion quickly, we'll get back to how the tree looks later

```
data BKTree a = Empty
              | Node !a [Index (BKTree a)]
              deriving (Functor)
```

I'm going over the insertion quickly, we'll get back to how the tree looks later

```haskell
insert :: Distance a -> a -> BKTree a -> BKTree a
insert distance a = \case
  Empty -> Node a []
  Node b children ->
    let newDistance = distance a b
    in Node b (addChild newDistance children)
  where
    addChild d = \case
      [] -> [Index d (insert distance a Empty)]
      Index d' child:children
        | d == d' -> Index d' (insert distance a child)
                   : children
        | otherwise -> Index d' child
                   : addChild d children
```

Querying from the BK-tree

We want to go through the tree and find all the elements within a distance

- If the current node is within range add it to the results
- If the child is outside the extended range, don't recurse to it

```haskell
search :: Distance a -> Int -> a -> BKTree a -> [a]
search distance range target = \case
  Empty -> []
  Node current children ->
    let currentDistance = distance current target
        upperBound = currentDistance + range
        lowerBound = currentDistance - range
        includedChildren =
          concat [ search distance range target x
                 | Index dist x <- children
                 , dist <= upperBound
                 , dist >= lowerBound ]
    in if currentDistance < range
         then current : includedChildren
         else includedChildren
```

Introducing algebras

Recursion schemes abstract recursive structures with the help of f-algebras and f-coalgebras

```
type FAlgebra f a = f a -> a

type FCoAlgebra f a = a -> f a
```

The f-algebras represent recursion (folds) and f-coalgebras represent corecursion (unfolds)

To bring the algebras back to our original type, we need a way to represent our tree as this `f` type. This can be done by replacing the recursive step with a functor

```haskell
data BKTree a = Empty
              | Node !a [Index (BKTree a)]
              deriving (Functor)

data BKTreeF a f = EmptyF
                 | NodeF !a [Index f]
                 deriving (Functor)
```

Introducing Fix

Typing the functored tree turns out to be difficult

```
level1 :: BKTreeF String a
level1 = ...

level2 :: BKTreeF String (BKTreeF String a)
level2 = ...

level3 :: BKTreeF String (BKTreeF String (BKTreeF String a)
level3 = ...
```

Fix is the fixpoint of functors

```haskell
newtype Fix f = Fix { unFix :: f ( Fix f ) }

-- f (BKTreeF a (f (BKTree a (f ...))))
type BKTree a = Fix (BKTreeF a)
```

Introducing folds

With algebras and `Fix` in place, we can finally see what `cata` and `ana` look like

```
cata :: (f a -> a) -> Fix f -> a
cata algebra = go
  where go = algebra . fmap algebra . unFix

ana :: (a -> f a) -> a -> Fix f
ana coalgebra = go
  where go = Fix . fmap go . coalgebra
```

Introducing recursion-schemes

The `recursion-schemes` library has some template haskell, type families and type classes to do the conversions between types like `BKTree a` and `BKTreeF a`, no need to use Fix

```
type family Base t :: * -> *

type instance Base [a] = ListF a
type instance Base (BKTree a) = BKTreeF a

class Functor (Base t) => Recursive t where
  project :: t -> Base t t
  cata :: (Base t a -> a)

class Functor (Base t) => Corecursive t where
  embed :: Base t t -> t
  ana :: (a -> Base t a) -> a -> t
```

The `Base t` types are difficult to read. Fully qualifying them makes them easier to understand

```
project :: BKTree a -> BKTreeF a (BKTree a)
cata :: (BKTreeF a b -> b) -> BKTree String -> b

embed :: BKTreeF a (BKTree a) -> BKTree a
ana :: (b -> BKTreeF a b) -> b -> BKTree a
```

Recursing the query

For our use case, a regular `cata` should be fine. For that we need to find the F-algebra for the algorithm.

```haskell
search :: Distance a -> Int -> a -> BKTree a -> [a]
search distance range target = \case
  Empty -> []
  Node current children ->
    let currentDistance = distance current target
        upperBound = currentDistance + range
        lowerBound = currentDistance - range
        includedChildren =
          concat [ search distance range target x
                 | Index dist x <- children
                 , dist <= upperBound
                 , dist >= lowerBound ]
    in if currentDistance < range
          then current : includedChildren
          else includedChildren
```

The relevant part of the original signature is the `BKTree a -> [a]`

Remember that an f-algebra is of the form `f a -> a`

```
search :: Distance a -> Int -> a -> BKTreeF a [a] -> [a]
search = ...
```

The empty case is trivial

```
search :: Distance a -> Int -> a -> BKTreeF a [a] -> [a]
search distance range target = \case
  EmptyF -> []
```

The node case is almost the same as the original, but the type of `children` is different

```haskell
search :: Distance a -> Int -> a -> BKTreeF a [a] -> [a]
search distance range target = \case
  EmptyF -> []
  NodeF current (children :: [Index [a]]) ->
    let currentDistance = distance current target
        upperBound = currentDistance + range
        lowerBound = currentDistance - range
        includedChildren = [ xs
                           | Index dist xs <- children
                           , dist <= upperBound
                           , dist >= lowerBound
                           ]
    in if currentDistance < range
         then current : concat includedChildren
         else concat includedChildren
```

What are we querying

I want to show you guys what the tree looks like

We need to fold the tree into a graphviz file. This sounds like a good use for recursion schemes

```
dot :: BKTreeF String String -> String
```

But to have our edges pointing to correct nodes, we need *unique* node names. An auto-incrementing id?

```
dot :: BKTreeF String String -> State Int String
```

We also need to keep track of the child node names, let's add that as well

```
dot
  :: BKTreeF String (Maybe String, String)
  -> State Int (Maybe String, String)
```

The base case is simple. No node, nothing rendered

```haskell
dot'
  :: BKTreeF String (Maybe String, String)
  -> State Int (Maybe String, String)
dot' = \case
  EmptyF -> pure (Nothing, "")
```

Some helper functions. Edges are labeled by their distance, nodes are labeled by the element. Edges flow from parent to children

```
dot' :: BKTreeF String (Maybe String, String) -> State Int
dot' = \case
  EmptyF -> pure (Nothing, "")
  where
    mkEdge :: String -> String -> Int -> String
    mkEdge = printf "%s -> %s [label=\"%d\"];\n"
    mkLabel :: String -> String
    mkLabel = printf " [label=\"%s\"];\n"
    nodeKey :: State Int String
    nodeKey = printf "node_%d" <$> newIdx
```

In the node element we just create the current node and edges and concatenate these with whatever the children have rendered.

```haskell
dot'
  :: BKTreeF String (Maybe String, String)
  -> State Int (Maybe String, String)
dot' = \case
  EmptyF -> pure (Nothing, "")
  NodeF current children -> do
    key <- nodeKey
    let currentNode = key <> mkLabel current
        childNodes
          = [ mkEdge key childKey dist <> child
            | Index dist (Just childKey, child)
              <- children
            ]
    pure (Just key, currentNode <> concat childNodes )
  where
    ...
```

Diversion cataM

So how do we invoke this algebra? The type of `cata` was `(BKTreeF a b -> b) -> BKTree a -> b` right? We have the `State Int` there.

```
cata :: (BKTreeF a b -> b) -> BKTree a -> b
```

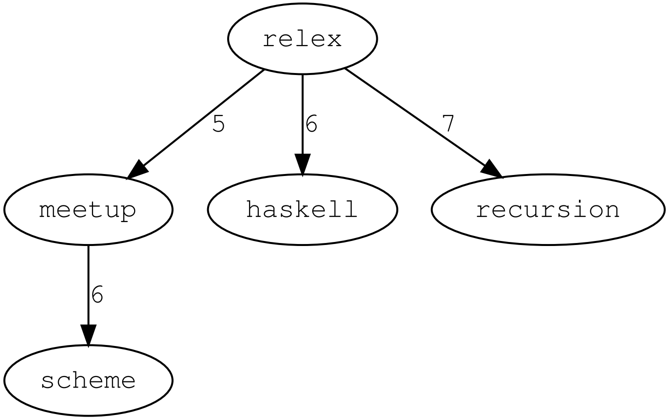
We need something that has a monadic layer

```
cataM
  :: Monad m
  => (BKTreeF a b -> m b)
  -> BKTree a
  -> m b
cataM f = cata (sequence >=> f)
```

What does the BKTree look like

Let's find out what the tree looks like

```
main :: IO ()
main = do
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (cataM dot' tree) 0
  putStrLn "}"
```


relex
5
6
7
meetup
haskell
recursion
6
scheme

Diversion Cofree

The `Cofree` is really similar to the `Fix` type, but it has an extra `a` in the record. You can use this to annotate each layer of the type

```haskell
newtype Fix f = Fix (f (Fix f))

data Cofree f a = a :< (f (Cofree f a))
```

What are we querying

So let's see what we're actually querying for in the search tree. I want to build a graph that shows which children are culled from consideration.

1. Modify our graph to take an annotated tree
2. Create annotations to create our original graph
3. Create annotations to describe the query plan

Render the annotations

Use a simple `Bool` for the annotations, with the semantics that `True` is considered, `False` is culled.

Culled elements are rendered with a greyed out color.

The only difference to our previous implementation is the addition of the CofreeF wrapper and the label color (omitted).

```
type Out = (Maybe String, String)
dot
  :: CF.CofreeF (BKTreeF String) Bool Out
  -> State Int (Maybe String, String)
dot = \case
  _ CF.:< EmptyF -> pure (Nothing, "")
  visible CF.:< NodeF current children -> do
    key <- nodeKey
    let currentNode = key <> mkLabel current visible
        childNodes =
          [ mkEdge key childKey dist <> child
          | Index dist (Just childKey, child)
            <- children
          ]
    pure (Just key, currentNode <> concat childNodes )
  where
    ...
```

Foray into annotation

We want to first of all annotate each element with a constant value. If we annotate each element with a constant `True` our renderer will behave like the original renderer we wrote earlier

For annotating the data we want to go the other way. F-algebras were `f a -> a` or `BKTreeF String a -> a`.

We want to do an f-*co*algebra which is `a -> f a` or in this case `BKTree String -> CofreeF (BKTreeF String) Bool (BKTree String)`.

```
constAnnotate
  :: x
  -> BKTree a
  -> CF.CofreeF (BKTreeF a) x (BKTree a)
```

The implementation is really simple. For each node, just annotate with the constant value

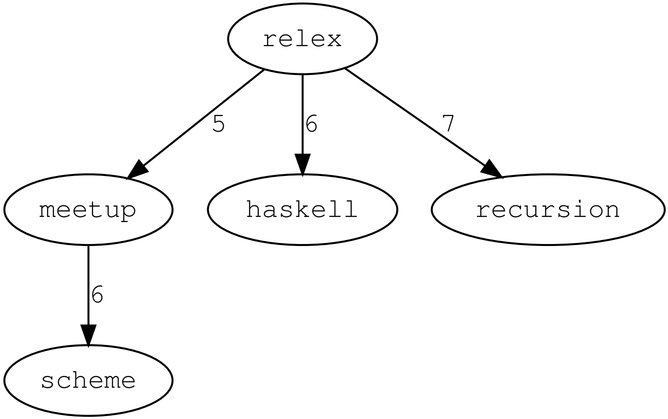
```haskell
data Cofree f a = a :< (f (Cofree f a))

data CofreeF f a b = a :< (f b)

constAnnotate
  :: x
  -> BKTree a
  -> CF.CofreeF (BKTreeF a) x (BKTree a)
constAnnotate x = \case
  Empty ->
    x CF.:< EmptyF
  Node current children ->
    x CF.:< NodeF current children
```

Let there be dot

```haskell
main :: IO ()
main = do
  let annotated = ana (constAnnotate True) tree
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (cataM dot annotated) 0
  putStrLn "}"
```


relex
5
6
7
meetup
haskell
recursion
6
scheme

Apoc are we online

We have so far only seen only the basic algebras, but there are plenty more.

Destruct	Construct
cata	ana
para	apo

- `para` has access to the full subtree
- `apo` can return the full subtree

```haskell
type Input = BKTree String
type Output =  Cofree BKTreeString Bool

para
  :: (BKTreeF String (Input, Output) -> Output)
  -> BKTree String
  -> a
```

```haskell
type BKTreeFun = BKTreeF String
type Input = BKTree String
type Output =  Cofree BKTreeFun Bool
apo
  :: (
       Input
    -> CofreeF BKTreeFun Bool (Either Output Input)
    )
  -> Input
  -> Output
```

Annotating the queries

We will be using the `constAnnotate` function to handle the culling case, which is why we need the ability to return the entire subtree at once.

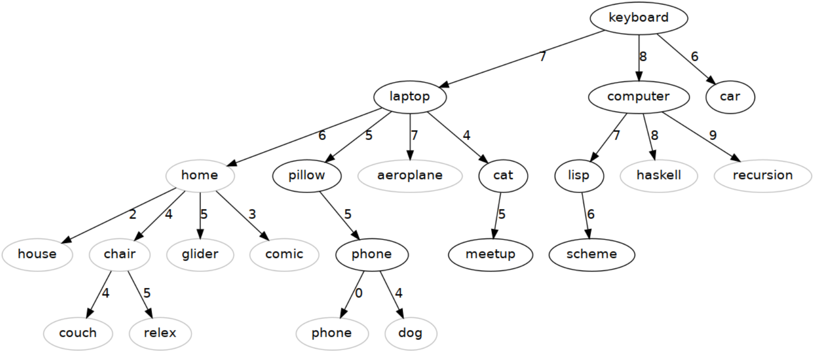
```
type Algebra a b = CF.CofreeF (BKTreeF a) Bool b
type Input a = BKTree a
type Output a = Cofree (BKTreeF a) Bool

annotate
  :: Distance a
  -> Int
  -> a
  -> Input
  -> Algebra a (Either Output Input)
```

```haskell
annotate distance range target = \case
  Empty ->
    False CF.:< EmptyF
  Node current children ->
    let currentDistance = distance current target
        upperBound = currentDistance + range
        lowerBound = currentDistance - range
        within d = d >= lowerBound && d <= upperBound
        accepted = within currentDistance
        mkFalse = ana (constAnnotate False)
        cull child = Left (mkFalse child)
        mkChild child = bool (cull child) Right
        culledChildren =
          [ Index d (mkChild child (within d))
          | Index d child <- children ]
    in accepted CF.:< NodeF current culledChildren
```

Cull the trees

```haskell
main :: IO ()
main = do
  let annotated = apo
                    (annotateF editDistance 1 "meetup")
                    tree
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (cataM dot annotated) 0
  putStrLn "}"
```


keyboard
7
8
6
laptop
computer
car
6
5
7
4
7
8
9
home
pillow
aeroplane
cat
lisp
haskell
recursion
2
4
5
3
5
5
6
house
chair
glider
comic
phone
meetup
scheme
4
5
0
4
couch
relex
phone
dog

Thank you

Questions?

Combining folds and unfolds

Where cata and friends are folds from a recursive structure into a value, ana and friends are unfolds from a value into recursive structures.

Some times you need to both unfold and fold, for this we have the `refold` or `hylo`.

```
main :: IO ()
main = do
  let annotated = ana (constAnnotate True) tree
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (cataM dot annotated) 0
  putStrLn "}"
```

Combining folds and unfolds

Where cata and friends are folds from a recursive structure into a value, ana and friends are unfolds from a value into recursive structures.

Some times you need to both unfold and fold, for this we have the `refold` or `hylo`.

```
main :: IO ()
main = do
  let render = hylo
                 algebra
                 coalgebra
      algebra = sequence >=> dot
      coalgebra = constAnnotate True
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (render tree) 0
  putStrLn "}"
```

Generics

The `g` family of functions takes a distributive function and modifies the `fold`/`unfold`/`refold` to work based the distributive.

```
para = gcata distPara
```

This is really useful when you have a refold and you need to use something other than a `cata` and `ana` as your algebra/coalgebra.

```
main :: IO ()
main = do
  let annotated = apo
                    (annotateF editDistance 1 "meetup")
                    tree
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (cataM dot annotated) 0
  putStrLn "}"
```

Generics

The `g` family of functions takes a distributive function and modifies the `fold`/`unfold`/`refold` to work based the distributive.

```
para = gcata distPara
```

This is really useful when you have a refold and you need to use something other than a `cata` and `ana` as your algebra/coalgebra.

```haskell
main :: IO ()
main = do
  let customRefold alg = ghylo
                           distCata
                           distApo
                           (alg . fmap runIdentity)
      algebra = sequence >=> dot
      coalgebra = annotateF editDistance 1 "meetup"
      render = customRefold algebra coalgebra
  putStrLn "digraph g {"
  putStrLn $ snd $ evalState (render tree) 0
  putStrLn "}"
```