

# (Introduction to) Recursion Schemes (in Haskell)<sup>1</sup>

Traian Florin Șerbănuță

University of Bucharest and Runtime Verification

---

<sup>1</sup>Edward Kmett's recursion-schemes package serves as inspiration for this talk

# Summary

- ▶ Identify / recall basic recursion patterns on lists
- ▶ A little bit of universal algebra
- ▶ Generalize recursion to arbitrary datastructures

# GHC language extensions used

- ▶ Automatic derivation of Functor instances:

```
{-# LANGUAGE DeriveFunctor      #-}
```

- ▶ Writing type signatures in where declarations:

```
{-# LANGUAGE ScopedTypeVariables #-}
```

- ▶ Declaring type families

```
{-# LANGUAGE TypeFamilies      #-}
```

- ▶ Using non-type variable arguments in type constraints

```
{-# LANGUAGE FlexibleContexts   #-}
```

## Basic recursion on lists

# Definition of lists

```
data [a]  
  = []  
  | a : [a]
```

- ▶ A constructive (initial) view

A list of elements of type `a` is

- ▶ either empty (`[]`), or
- ▶ constructed by adding an element to an existing list

```
[]  :: [a]  
(:) :: a -> [a] -> [a]
```

- ▶ A destructive (final) view

A (potentially infinite) list of elements of type `a` can *maybe* be decomposed into its *head* and its *tail*:

```
uncons :: [a] -> Maybe (a, [a])
```

# Fold (reduce, bananas)

- ▶ Definition

`foldr :: (a -> b -> b) -> b -> [a] -> b`

Right-associative fold of a list. Given

- ▶ `f :: a -> b -> b`, a binary operator and
- ▶ `z :: b`, a starting value

reduce a list, from right to left, as follows:

$$\text{foldr } f \ z \ (x_1 : x_2 : \dots : x_n : []) \\ == x_1 \ `f` \ (x_2 \ `f` \ \dots \ (x_n \ `f` \ z) \ \dots)$$

- ▶ It matches the constructive view of the list

- ▶ `(:) :: a -> [a] -> [a]` constructor for the list
- ▶ `[] :: [a]`

we have that `foldr (:) [] 1 == 1`

## Fold examples

```
sumF, productF :: Num a => [a] -> a
```

```
sumF = foldr (+) 0
```

```
productF = foldr (*) 1
```

```
mconcatF :: Monoid a => [a] -> a
```

```
mconcatF = foldr (<)> mempty -- for any monoid
```

```
mapF :: (a -> b) -> [a] -> [b]
```

```
mapF f = foldr (\x xs -> f x : xs) []
```

```
lengthF :: Num n => [a] -> n
```

```
lengthF = foldr (\_ n -> 1 + n) 0
```

```
partitionF :: (a -> Bool) -> [a] -> ([a], [a])
```

```
partitionF p = foldr op ([], [])
```

```
  where x `op` (ps, nps) | p x      = (x:ps, nps)  
                        | otherwise = (ps, x:nps)
```

# Unfold

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b = case f b of
  Nothing    -> []
  Just (a,b') -> a : unfoldr f b'
```

- ▶ The unfoldr function is a *dual* to foldr:
  - ▶ foldr *reduces* a list to a summary value
  - ▶ unfoldr *builds* a list from a seed value.
- ▶ It matches the destructive view on lists

```
uncons :: [a] -> Maybe (a, [a])
```

```
unfoldr uncons l == l
```



## Unfold examples

```
repeatU :: a -> [a]
repeatU = unfoldr (\a -> Just (a,a))

replicateU :: (Num n, Ord n) => n -> a -> [a]
replicateU n a = unfoldr g n
  where g n = if n <= 0 then Nothing else Just (a, n-1)

iterateU :: (a -> a) -> a -> [a]
iterateU f = unfoldr (\a -> Just (a, f a))

mapU :: (a -> b) -> [a] -> [b]
mapU f = unfoldr g
  where g []      = Nothing
        g (x:xs) = Just (f x, xs)

zipU :: [a] -> [b] -> [(a,b)]
zipU as bs = unfoldr g (as,bs)
  where g ([],_) = Nothing
        g (_,[]) = Nothing
        g (a:as, b:bs) = Just ((a,b), (as,bs))
```

## Refolds: Combining folds and unfolds

A refold is an algorithm whose recursion is shaped like a list.

```
factorialR :: (Num n, Ord n) => n -> n
factorialR = foldr (*) 1 . unfoldr g
  where g n = if n <= 0 then Nothing else Just (n, n-1)

sumOfSquaresR :: (Num n, Ord n) => n -> n
sumOfSquaresR = foldr (+) 0 . unfoldr g
  where g n = if n <= 0 then Nothing else Just (n*n, n-1)
```

## Refolds: Combining folds and unfolds

A refold is an algorithm whose recursion is shaped like a list.

```
factorialR :: (Num n, Ord n) => n -> n
factorialR = foldr (*) 1 . unfoldr g
  where g n = if n <= 0 then Nothing else Just (n, n-1)

sumOfSquaresR :: (Num n, Ord n) => n -> n
sumOfSquaresR = foldr (+) 0 . unfoldr g
  where g n = if n <= 0 then Nothing else Just (n*n, n-1)

filterR :: forall a . (a -> Bool) -> [a] -> [a]
filterR p = foldr f [] . unfoldr g
  where g :: [a] -> Maybe (Maybe a, [a])
        g [] = Nothing
        g (x:xs) | p x = Just (Just x, xs)
                  | otherwise = Just (Nothing, xs)
  f :: Maybe a -> [a] -> [a]
  f Nothing xs = xs
  f (Just x) xs = x:xs
```

Algebras: initial, final, and in-between

# Universal algebra

Universal algebra is the field of mathematics that studies algebraic structures themselves, not examples (“models”) of algebraic structures.

For instance, rather than take particular groups as the object of study, in universal algebra one takes the class of groups as an object of study.

## Ingredients

- ▶ Signatures – describing the type of algebras under study
  - ▶ symbols for operations and their arities
  - ▶ symbols for sorts, too, if multisorted
- ▶ Algebras - concrete models interpreting
  - ▶ sorts as sets
  - ▶ operation symbols as functions

## Signatures as algebraic types

A (non-recursive) type for list-like structures

```
data ListF a list
  = Nil
  | Cons a list
deriving (Functor)
```

## Signatures as algebraic types

A (non-recursive) type for list-like structures

```
data ListF a list
  = Nil
  | Cons a list
deriving (Functor)
```

A way to view lists as the *canonical (initial)* structure for this type:

```
projectL :: [a] -> ListF a [a]
projectL [] = Nil
projectL (a:as) = Cons a as
```

```
embedL :: ListF a [a] -> [a]
embedL Nil = []
embedL (Cons a as) = a:as
```

Note that projectL/embedL form an isomorphism.

## What is a list-like algebra?

```
data ListF a list = Nil | Cons a list deriving (Functor)
```

A list-like structure is given by a carrier type, and an interpretation of the list operations in it:

```
algebra :: ListF a carrier -> carrier
```

## Examples

- ▶ lists themselves: `embedL :: ListF a [a] -> [a]`
- ▶ monoid operations

```
monoidAlg :: Monoid a => ListF a a -> a
```

```
monoidAlg Nil = mempty
```

```
monoidAlg (Cons a b) = a <> b
```

- ▶ and even ways for partitioning a list

```
partitionAlg :: (a -> Bool) -> ListF a ([a],[a]) -> ([a],[a])
```

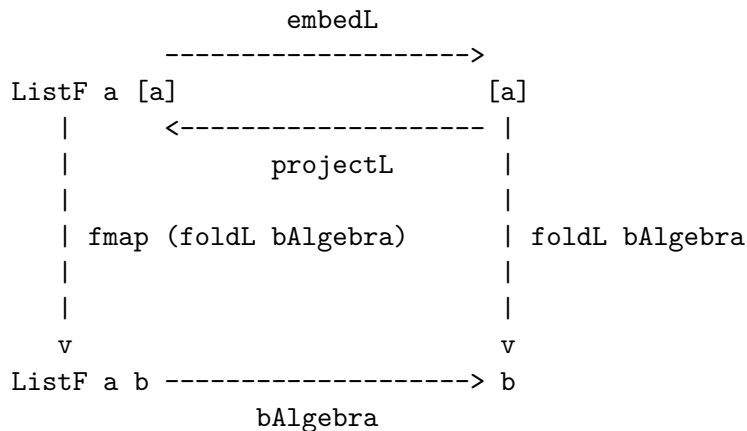
```
partitionAlg Nil = ([], [])
```

$$\text{partitionAlg } p \text{ (Cons } a \text{ (as,bs))} \mid p \ a = (a:\text{as},\text{bs})$$

```
| otherwise = (as,a:bs)
```

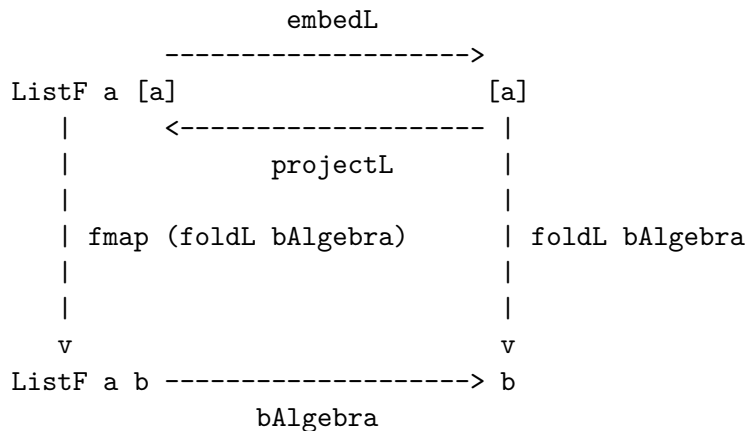


## Folds on list-like algebras revisited



`foldL bAlgebra.embedL = bAlgebra.fmap (foldL bAlgebra)`

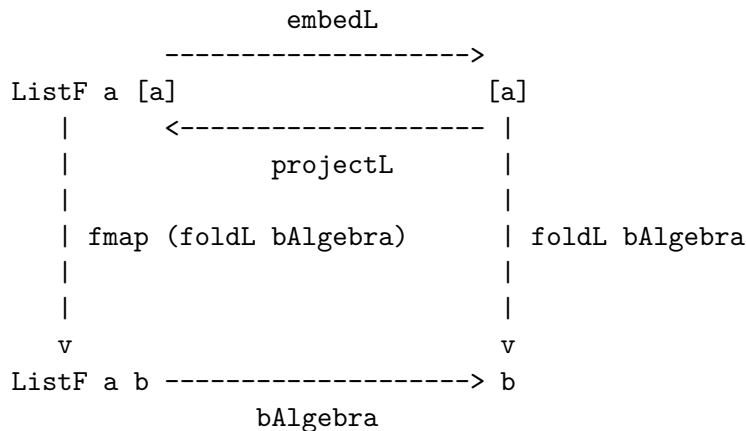
## Folds on list-like algebras revisited



`foldL bAlgebra.embedL = bAlgebra.fmap (foldL bAlgebra)`

`foldL bAlgebra = bAlgebra.fmap (foldL bAlgebra).projectL`

## Folds on list-like algebras revisited



```
foldL bAlgebra.embedL = bAlgebra.fmap (foldL bAlgebra)
```

```
foldL bAlgebra = bAlgebra.fmap (foldL bAlgebra).projectL
```

```
foldL bAlgebra = go
```

```
  where go = bAlgebra . fmap go . projectL
```

## Folds on (list-like) algebras

```
foldL :: (ListF a b -> b) -> [a] -> b
foldL algebra = go
  where go = algebra . fmap go . projectL
```

*Note:* definition does not depend on the structure

### Examples

```
mconcatAF :: Monoid a => [a] -> a
mconcatAF = foldL monoidAlg

partitionAF :: (a -> Bool) -> [a] -> ([a], [a])
partitionAF p = foldL (partitionAlg p)
```

## What is a list-like co-algebra?

```
data ListF a list = Nil | Cons a list    deriving (Functor)
```

A list-like co-structure is given by a carrier type, and a way to deconstruct an element into applications of operations:

```
coalgebra :: carrier -> ListF a carrier
```

### Examples

► lists themselves: `projectL :: [a] -> ListF a [a]`

► iterate

```
iterateCoAlg :: (a -> a) -> (a -> ListF a a)
```

```
iterateCoAlg f a = Cons a (f a)
```

► and even zip

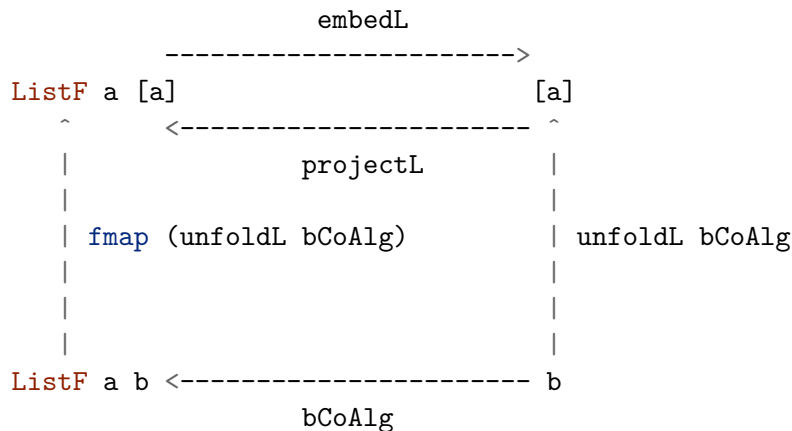
```
zipCoAlg :: ([a], [b]) -> ListF (a,b) ([a],[b])
```

```
zipCoAlg ([], _) = Nil
```

```
zipCoAlg (_, []) = Nil
```

```
zipCoAlg (a:as, b:bs) = Cons (a,b) (as,bs)
```

## Folds on list-like algebras revisited



`projectL . unfoldL bCoAlg = fmap (unfoldL bCoAlg) . bCoAlg`

`unfoldL bCoAlg = embedL . fmap (unfoldL bCoAlg) . bCoAlg`

`unfoldL bCoAlg = go`

`where go = embedL . fmap go . bCoAlg`

## Unfolds on (list-like) algebras

```
unfoldL :: (b -> ListF a b) -> b -> [a]
unfoldL bCoAlg = go
  where go = embedL . fmap go . bCoAlg
```

*Note:* definition does not depend on the structure

### Examples

```
iterateCU = unfoldL . iterateCoAlg
-- iterateCU f = unfoldL (iterateCoAlg f)

zipCU = curry (unfoldL zipCoAlg)
-- zipCU as bs = unfoldL zipCoAlg (as,bs)
```

Recursion schemes : folds and unfolds on  
arbitrary (recursive) structures



## F-algebras

Given a Functor  $f$ , the type of  $F$ -(co)algebras induced by  $f$  is:

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

### Examples of such (base) functors

- ▶ List-like structures

```
data ListF a carr = Nil | Cons a carr    deriving Functor
```

- ▶ Binary tree-like structures

```
data TreeF a carrier = Empty | Node a carrier carrier
  deriving Functor
```

- ▶ Arithmetic expression-like structures

```
data ExpF carrier    = Num Int | Var String
                    | carrier :+: carrier
                    | carrier :*: carrier
  deriving Functor
```

## Base Functor and Fix

**The Base functor** helps us establish an initial F-algebra / final F-co-algebra for a type

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

For example,

```
type instance Base [a] = ListF a
```

## Base Functor and Fix

**The Base functor** helps us establish an initial F-algebra / final F-co-algebra for a type

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

For example,

```
type instance Base [a] = ListF a
```

**Fixpoint constructions** allow us to build initial F-algebras / final F-co-algebras for a (base) functor

```
newtype Fix f = Fix { unfix :: f (Fix f) }
```

```
type instance Base (Fix f) = f      -- the base functor of F
```

For example,

```
Fix (ListF a) ~= ListF a (Fix (ListF a))  ~~ [a]
```

```
Fix (TreeF a) -- binary trees with labeled nodes
```

```
Fix ExpF -- expressions with + and * over ints and vars
```

# Generalized folds

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

  cata :: (Base t a -> a) -- ^ a (Base t)-algebra
    -> t                  -- ^ fixed point
    -> a                  -- ^ result
  cata alg = go           -- a cata(morphism) is a fold
    where go = alg . fmap go . project
```

## Examples

```
instance Recursive [a] where
  project [] = Nil
  project (x:xs) = Cons x xs

instance Functor f => Recursive (Fix f) where
  project = unfix
```

## Generalized unfolds

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

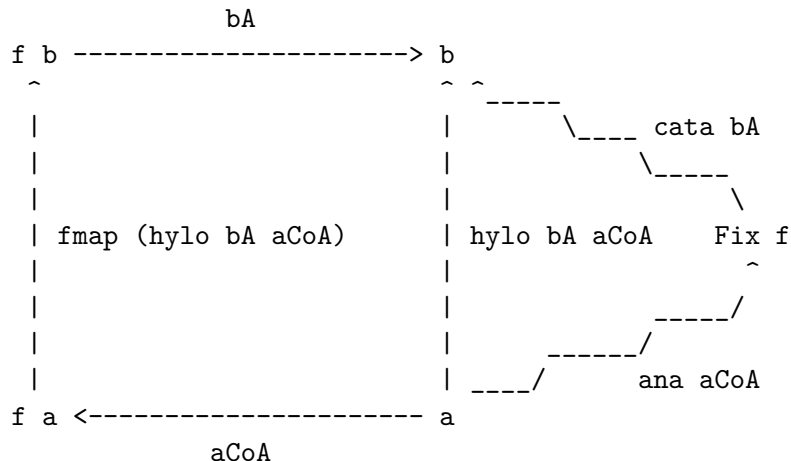
ana
  :: (a -> Base t a) -- ^ a (Base t)-coalgebra
  -> a               -- ^ seed
  -> t               -- ^ resulting fixed point
ana coalg = go      -- an ana(morphism) is an unfold
  where go = embed . fmap go . coalg
```

### Examples

```
instance Corecursive [a] where
  embed Nil = []
  embed (Cons x xs) = x:xs

instance Functor f => Corecursive (Fix f) where
  embed = Fix
```

## Generalized refolds



```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo bA aCoA = r           -- a hylo(morphism) is a refold
  where r = bA . fmap r . aCoA
```

In particular `hylo bA aCoA == cata bA . ana aCoA`

## Examples

# Fold

Build an interpreter without caring for recursion

```
interp :: [(String,Int)] -> Fix ExpF -> Maybe Int
interp env = cata interpA
  where
    interpA :: ExpF (Maybe Int) -> Maybe Int
    interpA e = case e of
      Num i -> Just i
      Var x -> lookup x env
      v1 :+: v2 -> pure (+) <*> v1 <*> v2
      v1 **: v2 -> pure (*) <*> v1 <*> v2
```



# Unfold

Perfectly ballanced tree

```
toBalancedTree :: [a] -> Fix (TreeF a)
toBalancedTree = ana toBalancedCoalg
  where
    toBalancedCoalg :: [a] -> TreeF a [a]
    toBalancedCoalg [] = Empty
    toBalancedCoalg list = Node a begin end
      where
        len = length list `div` 2
        (begin, a:end) = splitAt len list
```

## Refold

Quick-Sort uses a binary tree as an intermediary structure to split the list, then recombines results.

```
qsort :: Ord a => [a] -> [a]
qsort = hylo qSortAlg qSortCoalg
  where
    qSortCoalg :: Ord a => [a] -> TreeF a [a]
    qSortCoalg []      = Empty
    qSortCoalg (a:as) = Node a lta gta
      where (lta, gta) = partitionAF (< a) as

    qSortAlg :: TreeF a [a] -> [a]
    qSortAlg Empty      = []
    qSortAlg (Node a lta gta) = lta ++ a:gta
```

All is well when it ends

# What now?

- ▶ Go and implement everything as folds/unfolds/refolds! :-)
  - ▶ visitors
  - ▶ transformers
  - ▶ ...
- ▶ Read some more about the other amazing recursion schemes
  - ▶ para, zigo, histo, apo, hoist, lambek, elgot, ...
  - ▶ In this talk we barely scratched the surface
- ▶ What about mutually-recursive datatypes?

## References / Further readings

- ▶ Edward A. Kmett (2008) recursion-schemes: Representing common recursion patterns as higher-order functions
- ▶ Bartosz Milewski (2013) Understanding F-Algebras
- ▶ Erik Meijer, Maarten Fokkinga, Ross Paterson (1991) Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire
- ▶ Philip Wadler (1990) Recursive types for free!
- ▶ Daniel Fischer (2009) GHC/Type families
- ▶ Edward A. Kmett (2009) Recursion Schemes: A Field Guide