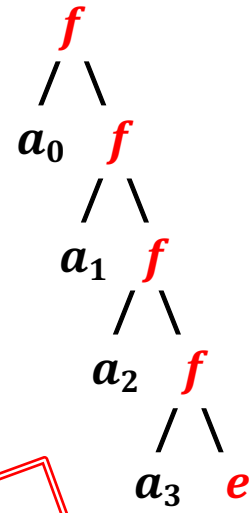
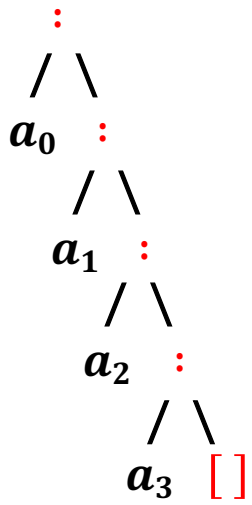


Folding



CHEAT-SHEET
#9

List Unfolding
unfold as the Computational Dual of *fold*
and how *unfold* relates to *iterate*

slides by



@philip_schwarz

FP  uminated

<https://fpilluminated.org/>



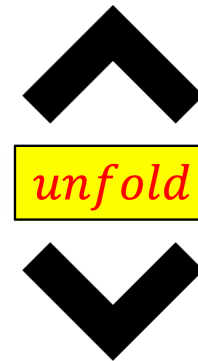
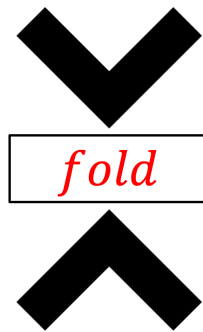
 [@philip_schwarz](#)

This **cheat sheet** is based on the following **deck**, which it aims to **condense**, partly by focusing on a **single running example**, and partly by choosing, rather than to provide **excerpts** from **referenced sources**, to simply **present salient points** made by the sources.

List Unfolding

unfold as the Computational Dual of *fold*
and how *unfold* relates to *iterate*

 Haskell



 Scala

slides by



[@philip_schwarz](#)

FP Illuminated

<https://fpilluminated.org/>



Here is a function called **digits** which takes an **integer number**, and returns a **list** of **integers** corresponding to the **number's digits** (yes, we are not handling cases like n=0 or negative n).

The **digits** function makes use of a function called **iterate**. See the next two slides for the **Haskell** and **Scala** definitions of **iterate**.

The **Haskell** version of **digits** is defined in **point-free style**, uses the **function composition** function, and uses **backticks** to specify the **infix** version of functions **mod** and **div**. If you find it at all **cryptic**, see the slide after the next two for **alternative definitions** that are less **succinct**.

$$\text{iterate} :: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$$
$$\text{iterate } f \ x = x : \text{iterate } f \ (f \ x)$$


```
digits :: Int -> [Int]
digits =
  reverse
    . map (`mod` 10)
    . takeWhile (/= 0)
    . iterate (`div` 10)
```

```
> digits 2718
[2,7,1,8]
```

```
def digits(n: Int): List[Int] =
  LazyList iterate(n)(_ / 10 )
    .takeWhile(_ != 0)
    .map(_ % 10 )
    .toList
    .reverse
```



```
scala> digits(2718)
val res0: List[Int] = List(2, 7, 1, 8)
```

base-4.21.0.0: Core data structures and operations

List operations

Building lists

Copyright	(c) The University of Glasgow 2001
License	BSD-style (see the file libraries/base/LICENSE)
Maintainer	libraries@haskell.org
Stability	stable
Portability	portable
Safe Haskell	Safe
Language	Haskell2010

Infinite lists

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

[# Source](#)

`iterate` `f` `x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

```
iterate :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow \alpha \rightarrow [\alpha]$   
iterate f x = x : iterate f (f x)
```



scala.collection.immutable.LazyList



Scala 3 3.7.0



▼ `def iterate[A](start: => A)(f: A => A): LazyList[A]`

An infinite LazyList that repeatedly applies a given function to a start value.

Value parameters

f

the function that's repeatedly applied

start

the start value of the LazyList

Attributes

Returns

the LazyList returning the infinite sequence of values `start, f(start), f(f(start)), ...`

Source

[LazyList.scala](#)

$iterate :: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
 $iterate\ f\ x = x : iterate\ f\ (f\ x)$



```
digits :: Int -> [Int]
digits n =
  reverse (
    map (`mod` 10) (
      takeWhile (/= 0) (
        iterate (`div` 10)
          n
      )
    )
  )
```

```
digits :: Int -> [Int]
digits n =
  reverse (
    map (\x -> mod x 10) (
      takeWhile (\x -> x /= 0) (
        iterate (\x -> div x 10)
          n
      )
    )
  )
```

```
digits :: Int -> [Int]
digits =
  reverse
    . map (`mod` 10)
    . takeWhile (/= 0)
    . iterate (`div` 10)
```

```
digits :: Int -> [Int]
digits n =
  reverse $ map (`mod` 10)
           $ takeWhile (/= 0)
           $ iterate (`div` 10)
           $ n
```

```
digits :: Int -> [Int]
digits n =
  reverse $ map (\x -> mod x 10)
           $ takeWhile (\x -> x > 0)
           $ iterate (\x -> div x 10)
           $ n
```

```
digits :: Int -> [Int]
digits n =
  n & iterate (`div` 10)
  & takeWhile (/= 0)
  & map (`mod` 10)
  & reverse
```

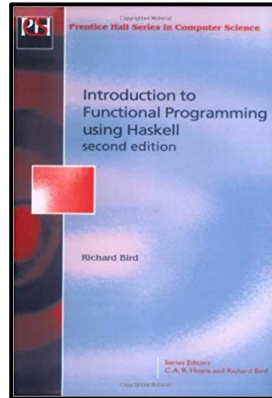
```
digits :: Int -> [Int]
digits n =
  n & iterate (\x -> div x 10)
  & takeWhile (\x -> x > 0)
  & map (\x -> mod x 10)
  & reverse
```

```
digits :: Int -> [Int]
digits =
  reverse
    . map (`mod` 10)
    . takeWhile (/= 0)
    . iterate (`div` 10)
```

```
digits :: Int -> [Int]
digits =
  reverse
    . map (\x -> mod x 10)
    . takeWhile (\x -> x > 0)
    . iterate (\x -> div x 10)
```



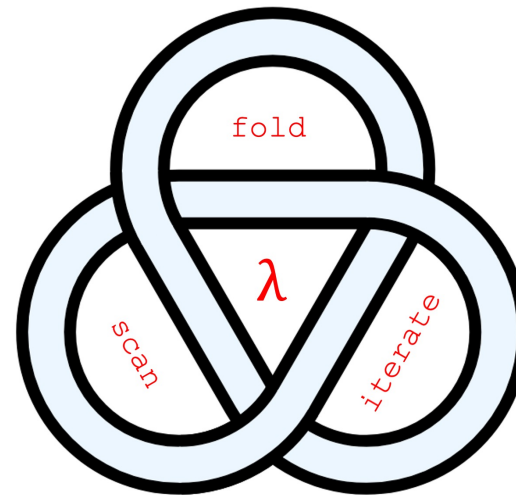
If you would like to know more about how the **digits** function works then see either the following deck, or the one mentioned at the beginning of this deck.



Richard Bird

<http://www.cs.ox.ac.uk/people/richard.bird/>

The **Functional Programming Triad** of **Folding**, **Scanning** and **Iteration**
a first example in **Scala** and **Haskell**
Polyglot **FP** for **Fun** and **Profit**



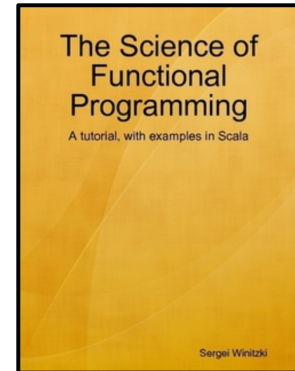
slides by



[@philip_schwarz](https://twitter.com/philip_schwarz)



<https://www.slideshare.net/pjschwarz>



Sergei Winitzki

[in sergei-winitzki-11a6431](https://www.linkedin.com/in/sergei-winitzki-11a6431)



```
digits :: Int -> [Int]
digits =
  reverse
    . map (`mod` 10)
    . takeWhile (/= 0)
    . iterate (`div` 10)
```

```
def digits(n: Int): List[Int] =
  LazyList iterate(n)(_ / 10)
    .takeWhile (_ != 0)
    .map (_ % 10)
    .toList
    .reverse
```



The **digits** function composes **map**, **takeWhile** and **iterate** in **sequence**.

One often finds such a **pattern of computation**, which may be **captured** as a **generic function** called **unfold**.

unfold :: $(\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold h p t = map h ◦ takeWhile p ◦ iterate t



```
unfold :: (b -> a) -> (b -> Bool) -> (b -> b) -> b -> [a]
unfold h p t = map h . takeWhile p . iterate t
```



```
def unfold[A,B](h: B => A, p: B => Boolean, t: B => B, b: B): LazyList[A] =
  LazyList.iterate(b)(t).takeWhile(p).map(h)
```




Let's simplify the implementation of **digits** using the **unfold** function.

unfold $:: (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold $h\ p\ t = \text{map } h \circ \text{takeWhile } p \circ \text{iterate } t$



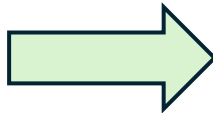
```
digits :: Int -> [Int]
digits =
  reverse
    . map (`mod` 10)
    . takeWhile (/= 0)
    . iterate (`div` 10)
```



```
digits :: Int -> [Int]
digits = reverse . unfold (`mod` 10) (/= 0) (`div` 10)
```



```
def digits(n: Int): List[Int] =
  LazyList.iterate(n)(_ / 10)
    .takeWhile (_ != 0)
    .map (_ % 10)
    .toList
    .reverse
```



```
def digits(n: Int): List[Int] =
  unfold[Int, Int](_ % 10, _ != 0, _ / 10, n).toList.reverse
```



Here is a **more efficient** variant of **unfold** that fuses together the **map**, **takewhile** and **iterate**.

The signature is slightly different in that there is some **renaming** and **reordering** of **parameters**, and the **condition** tested by **predicate function** *p* is **inverted**.

```
unfold :: (β → α) → (β → Bool) → (β → β) → β → List α
unfold h p t = map h ∘ takewhile p ∘ iterate t
```

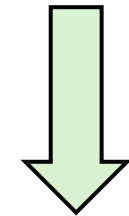
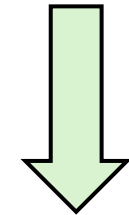
```
unfold p f g b = if p b then Nil
                  else Cons (f b) (unfold p f g (g b))
```

```
digits :: Int -> [Int]
digits = reverse . unfold (`mod` 10) (/= 0) (`div` 10)
```

```
digits :: Int -> [Int]
digits = reverse . unfold (== 0) (`mod` 10) (`div` 10)
```

```
def digits(n: Int): List[Int] =
  unfold[Int,Int](_ % 10, _ != 0, _ / 10, n).toList.reverse
```

```
def digits(n: Int): List[Int] =
  unfold[Int,Int](_ == 0, _ % 10, _ / 10, n).toList.reverse
```





Just for completeness, here is how **iterate** can be defined in terms of **unfold**.

$iterate :: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
 $iterate\ f = unfold\ (const\ False)\ id\ f$



$iterate :: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
 $iterate\ f\ x = x : iterate\ f\ (f\ x)$

$unfold :: (\beta \rightarrow Bool) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow List\ \alpha$
 $unfold\ p\ f\ g\ b = \text{if } p\ b \text{ then Nil}$
 $\quad \text{else Cons } (f\ b) (unfold\ p\ f\ g\ (g\ b))$



Let's reimplement **digits** using the **unfold'** function.


$$\begin{aligned} \text{unfold} & :: (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfold } p \ f \ g \ b &= \text{if } p \ b \text{ then Nil} \\ &\quad \text{else Cons } (f \ b) \ (\text{unfold } p \ f \ g \ (g \ b)) \end{aligned}$$

 **unfoldr**  **unfold**

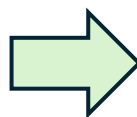
$$\begin{aligned} \text{unfold}' &:: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfold}' f u &= \text{case } f u \text{ of} \\ &\quad \text{Nothing} \rightarrow \text{Nil} \\ &\quad \text{Just } (x, v) \rightarrow \text{Cons } x (\text{unfold}' f v) \end{aligned}$$


```
digits :: Int -> [Int]
digits = reverse . unfold (== 0) (`mod` 10) (`div` 10)
```

```
def digits(n: Int): List[Int] =
  unfold[Int,Int](n == 0, _ % 10, _ / 10, n).toList.reverse
```



```
digits :: Int -> [Int]
digits = reverse . unfoldr gen
  where gen 0 = Nothing
        gen d = Just (d `mod` 10, d `div` 10)
```





```
def digits(n: Int): List[Int] =
  LazyList.unfold(n):
    case 0 => None
    case x => Some(x % 10, x / 10)
    .toList.reverse
```



Just for completeness, here is how **iterate** can be defined in terms of **unfold'**.

iterate $:: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
iterate $f = \text{unfold}' (\lambda x. \text{Just } (x, f\ x))$

iterate $:: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
iterate $f\ x = x : \text{iterate } f\ (f\ x)$

 **unfoldr**  **unfold**

unfold' $:: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold' $f\ u = \text{case } f\ u \text{ of}$
 Nothing $\rightarrow \text{Nil}$
 Just $(x, v) \rightarrow \text{Cons } x\ (\text{unfold}'\ f\ v)$



The **dual** of the **digits** function is the **decimal** function, which given a **list** of **digits**, returns the **integer number** with such **digits**.

As seen below, the **decimal** function is neatly and efficiently implemented using a **left fold** (if you want to know more then see [folding cheat-sheet #4](#)).



```
decimal :: [Int] -> Int
decimal = foldl ( $\oplus$ ) 0

( $\oplus$ ) :: Int -> Int -> Int
n  $\oplus$  d = 10 * n + d
```

```
> decimal [2,7,1,8]
2718
```

```
def decimal(ds: List[Int]): Int =
  ds.foldLeft(0)(_ $\oplus$ _)

extension (n: Int)
  def  $\oplus$ (d: Int): Int = 10 * n + d
```



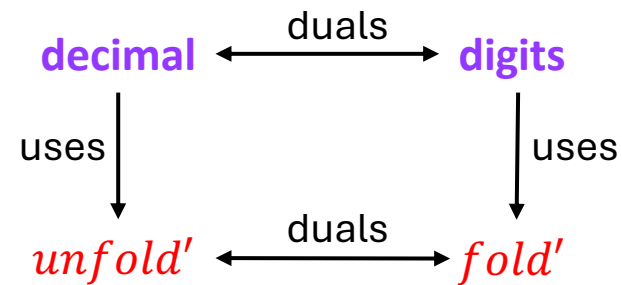
```
> decimal(List(2,7,1,8))
val res0: Int = 2718
```



Given the following...

- **decimal** is the computational **dual** of **digits**
 - **decimal** can be implemented using **unfold'** (a **right unfold**)
 - the computational **dual** of **unfold'** is **fold'** (a **right fold**)
- ...let's see if **decimal** can be implemented using **fold'**.

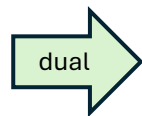
Because **fold'** is not available in **Haskell** and **Scala**, we implement it ourselves.



`unfoldr` `unfold`

```

unfold' :: (β → Maybe (α, β)) → β → List α
unfold' f u = case f u of
    Nothing → Nil
    Just (x, v) → Cons x (unfold' f v)
  
```

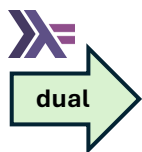


```

fold' :: (Maybe (α, β) → β) → List α → β
fold' f Nil = f Nothing
fold' f (Cons x xs) = f (Just (x, fold' f xs))
  
```

```

digits :: Int -> [Int]
digits = reverse . unfoldr gen
  where gen 0 = Nothing
        gen d = Just (d `mod` 10, d `div` 10)
  
```

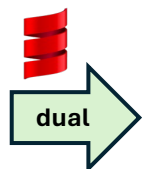


```

decimal :: [Int] -> Int
decimal = fst . fold' f
  where f Nothing = (0, 0)
        f (Just (d, (n, e))) = (d * (10 ^ e) + n, e + 1)
  
```

```

def digits(n: Int): List[Int] =
  LazyList.unfold(n):
    case 0 => None
    case x => Some(x % 10, x / 10)
  .toList.reverse
  
```



```

def decimal(ds: List[Int]): Int =
  foldp[Int, (Int, Int)](ds){
    case None => (0, 0)
    case Some(d, (n, e)) => (d * (pow(10, e)).toInt + n, e + 1)
  }(0)
  
```



```

fold' :: (Maybe (a, b) -> b) -> [a] -> b
fold' f [] = f Nothing
fold' f (x:xs) = f (Just (x, fold' f xs))
  
```



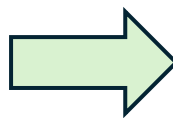
```

def foldp[A,B](as: List[A])(f: Option[(A,B)] => B): B = as match
  case Nil => f(None)
  case x :: xs => f(Option(x, foldp(xs)(f)))
  
```



Let's now switch from **fold'** to the **more customary** right **fold**, which *is* provided by **Haskell** and **Scala**.

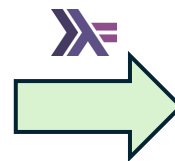
```
fold' :: (Maybe ( $\alpha, \beta$ )  $\rightarrow \beta$ )  $\rightarrow$  List  $\alpha \rightarrow \beta$   
fold' f Nil = f Nothing  
fold' f (Cons x xs) = f (Just (x, fold' f xs))
```



```
fold :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow$  List  $\alpha \rightarrow \beta$   
fold f e Nil = e  
fold f e (Cons x xs) = f x (fold f e xs)
```

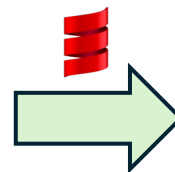
 foldr  foldRight

```
decimal :: [Int] -> Int  
decimal = fst . fold' f  
  where f Nothing = (0, 0)  
        f (Just (d, (n, e))) = (d * (10 ^ e) + n, e + 1)
```



```
decimal :: [Int] -> Int  
decimal = fst . foldr f (0, 0)  
  where f d (n, e) = (d * (10 ^ e) + n, e + 1)
```

```
def decimal(ds: List[Int]): Int =  
  foldp[Int, (Int, Int)](ds){  
    case None => (0, 0)  
    case Some(d, (n, e)) => (d * (pow(10, e)).toInt + n, e + 1)  
  }(0)
```



```
def decimal(ds: List[Int]): Int =  
  ds.foldRight(0, 0){ case (d, (n, e)) =>  
    (d * (pow(10, e)).toInt + n, e + 1)  
  }(0)
```




Here are the **four different implementations** of **digits** that we have considered.

iterate

```
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)
```

unfold v1

```
digits = reverse . unfold (`mod` 10) (/= 0) (`div` 10)
```

unfold v2

```
digits = reverse . unfold (== 0) (`mod` 10) (`div` 10)
```

unfold'

```
digits = reverse . unfoldr gen
  where gen 0 = Nothing
        gen d = Just (d `mod` 10, d `div` 10)
```



iterate

```
def digits(n: Int): List[Int] = LazyList iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList.reverse
```

unfold v1

```
def digits(n: Int): List[Int] = unfold[Int, Int](_ % 10, _ != 0, _ / 10, n).toList.reverse
```

unfold v2

```
def digits(n: Int): List[Int] = unfold[Int, Int](_ == 0, _ % 10, _ / 10, n).toList.reverse
```

unfold'

```
def digits(n: Int): List[Int] =
  LazyList.unfold(n):
    case 0 => None
    case x => Some(x % 10, x / 10)
  .toList.reverse
```

```
def digits(n: Int): List[Int] =
  LazyList.unfold(n): x =>
    Option.unless(x==0)(x % 10, x / 10)
  .toList.reverse
```





And here are the **three different implementations** of **decimal** that we have considered.

left fold

```
decimal = foldl1 ( $\oplus$ ) 0
```

$n \oplus d = 10 * n + d$

fold'

```
decimal = fst . fold' f
  where f Nothing      = (0, 0)
        f (Just (d, (n,e))) = (d * (10 ^ e) + n, e + 1)
```

fold

```
decimal = fst . foldr f (0, 0)
  where f d (n, e) = (d * (10 ^ e) + n, e + 1)
```



left fold

```
def decimal(ds: List[Int]): Int = ds.foldLeft(0)(_ $\oplus$ _)
```

```
extension (n: Int)
  def  $\oplus$ (d: Int): Int = 10 * n + d
```

fold'

```
def decimal(ds: List[Int]): Int =
  foldp[Int, (Int, Int)](ds){
    case None      => (0, 0)
    case Some(d, (n, e)) => (d * (pow(10, e)).toInt + n, e + 1)
  }(0)
```

fold

```
def decimal(ds: List[Int]): Int =
  ds.foldRight(0, 0){ case (d, (n, e)) =>
    (d * (pow(10, e)).toInt + n, e + 1)
  }(0)
```





The next slide is the penultimate one and suggests that

- the introduction of **fold'** and **unfold'** makes the duality between **folding** and **unfolding** very clear
- the names of **fold'** and **unfold'** are **primed** because the two functions are **less convenient** for programming than **fold** and **unfold**.

The slide after that is the last one and introduces the terms **catamorphism** and **anamorphism**.

While they may sometimes be less convenient for programming with, *fold'* and *unfold'*, the **primed** versions of *fold* and *unfold*, make the **duality** between the **fold** and the **unfold** very clear



 foldr  foldRight

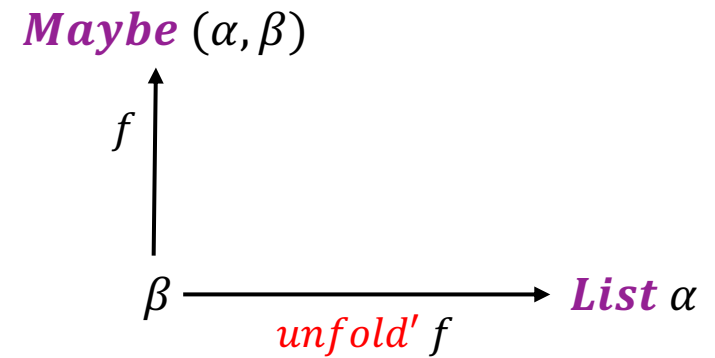
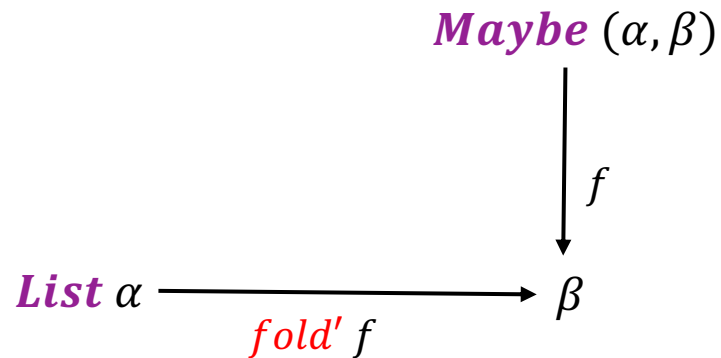
fold :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta$
fold *f* *e Nil* = *e*
fold *f* *e (Cons x xs)* = *f* *x (fold f e xs)*

fold' :: $(\text{Maybe } (\alpha, \beta) \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \beta$
fold' *f Nil* = *f Nothing*
fold' *f (Cons x xs)* = *f (Just (x, fold' f xs))*

unfold :: $(\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold *p f g b* = **if** *p b* **then Nil**
 else Cons (*f b*) (*unfold p f g (g b)*)

unfold' :: $(\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold' *f u* = **case** *f u* **of**
 Nothing → *Nil*
 Just (x, v) → **Cons** *x (unfold' f v)*

 unfoldr  unfold



 foldr  foldRight

$fold :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow List\ \alpha \rightarrow \beta$



Functions that ‘**destruct**’ a **list** – they have been called **catamorphisms**, from the **Greek** proposition **κατά**, meaning ‘**downwards**’ as in ‘**catastrophe**’.

$fold' :: (Maybe\ (\alpha, \beta) \rightarrow \beta) \rightarrow List\ \alpha \rightarrow \beta$

$unfold :: (\beta \rightarrow Bool) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow List\ \alpha$

Functions that ‘**generate**’ a **list** of **items** of type α from a **seed** of type β – they have been called **anamorphisms**, from the **Greek** proposition **ἀνά**, meaning ‘**upwards**’ as in ‘**anabolism**’.

$unfold' :: (\beta \rightarrow Maybe\ (\alpha, \beta)) \rightarrow \beta \rightarrow List\ \alpha$

 unfoldr  unfold