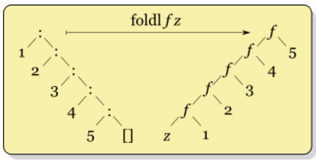
Base – Folds and Typeclasses

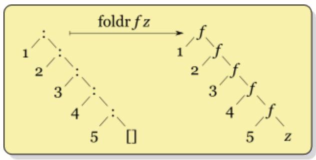
Left associative (foldl)

If you are **reducing to a single value,** then you may get more performance from a strict left **foldl'**.



Right associative (foldr)

If what you are reducing is **potentially infinite**, or you are **building a structure**, use **foldr**.




```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

toList :: Foldable t => t a -> [a]

and, or               :: Foldable t => t Bool -> Bool
any, all              :: Foldable t => (a -> Bool) -> t a -> Bool
sum, product          :: (Foldable t, Num a) => t a -> a
minimum, maximum      :: (Foldable t, Ord a) => t a -> a
minimumBy, maximumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a

elem, notElem :: (Foldable t, Eq a) => a -> t a -> Bool
find          :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

```
> foldl' (flip (:)) [0] [1,2,3]
[3,2,1,0]

> foldr (:) [5] [1,2,3,4]
[1,2,3,4,5]

> take 5 $ foldr (:) [] [1..]
[1,2,3,4,5]
```

```
> all even [1,2,3]
False

> any even [1,2,3,undefined]
True

> find (> 42) [1..]
Just 43
```

Applicative Traversals/Folds

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
for      :: (Traversable t, Applicative f) => t a -> (a -> f b) -> f (t b)
sequenceA :: Applicative f => t (f a) -> f (t a)

> for [1000000, 2000000] $ \t -> threadDelay t >> getCurrentTime
[2015-04-29 05:24:30.040399 UTC,2015-04-29 05:24:32.042665 UTC]
```

Functor

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

Control.Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

Control.Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a

(>=>) :: Monad m
      => (a -> m b) -> (b -> m c) -> a -> m c

join  :: Monad m => m (m a) -> m a
ap    :: Monad m => m (a -> b) -> m a -> m b
```

Base – Lists and misc

Data.List

```haskell
intersperse :: a -> [a] -> [a]
> intersperse ',' "abcde" == "a,b,c,d,e"

intercalate :: a -> [a] -> [a]
> intercalate " love " ["ponies", "ducks"]
"ponies love ducks"

subsequences, permutations :: [a] -> [[a]]
> subsequences "abc"
["","a","b","ab","c","ac","bc","abc"]
> permutations "abc"
["abc","bac","cba","bca","cab","acb"]

scanl :: (b -> a -> b) -> b -> [a] -> [b]
> scanl f z [x1, x2, ...]
[z, z `f` x1, (z `f` x1) `f` x2, ...]
> take 8 $ fix (\fib -> scanl (+) 1 (0:fib))
[1,1,2,3,5,8,13,21]

iterate :: (a -> a) -> a -> [a]
> iterate f x
[x, f x, f (f x), ...]
> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]

replicate :: Int -> a -> [a]
repeat :: a -> [a]
cycle :: [a] -> [a]

 :: Int -> [a] -> ([a], [a])
> splitAt n xs
(take n xs, drop n xs)

takeWhile, dropWhile
  :: (a -> Bool) -> [a] -> [a]
> takeWhile (< 3) [1,2,3,4,1,2,3,4]
[1,2]

isPrefixof, isSuffixOf, isInfixOf
  :: Eq a => [a] -> [a] -> Bool

lines, words :: String -> [String]
words, lines :: [String] -> String

nub :: Eq a => [a] -> [a]
> nub [1,2,2,3,2]
[1,2,3]

delete :: Eq a => a -> [a] -> [a]
> delete 'a' "banana"
"bnana"

(\\), union, intersect
  :: Eq a => [a] -> [a] -> [a]
> (xs ++ ys) \\ xs -- difference
ys
```

Commonly re-defined

```haskell
strip :: String -> String
strip =
  join fmap (reverse . dropWhile isSpace)

> strip "  a "
"a"

pairs :: [a] -> [(a, a)]
pairs = zip <*> tail

> pairs [1]
[]
> pairs [1,2,3]
[(1,2),(2,3)]
```

Data.Function

```haskell
fix :: (a -> a) -> a
> fix $ \f n ->
>   if n < 0 then [] else n : f (n - 1)) 5
[5,4,3,2,1,0]

on :: (b -> b -> c)
   -> (a -> b) -> a -> a -> c
> (*) `on` f
\x y -> f x * f y.
> sortBy (compare `on` length) ["bb", "a"]
["a","bb"]
```

Debug.Trace

The usual output stream is **stderr**.

```haskell
trace       :: String -> a -> a
traceShow   :: Show a => a -> b -> b
traceShowId :: Show a => a -> a
traceStack  :: String -> a -> a
traceIO     :: String -> IO ()
traceM      :: Monad m => String -> m ()
traceShowM  :: (Show a, Monad m) => a -> m ()

> trace ("call f with x = " ++ show x) (f x)
> g x y = traceShow (x, y) (x + y)
```

Control.Arrow

```haskell
class Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b,d) (c,d)
  second :: a b c -> a (d,b) (d,c)
  (***) :: a b c -> a b' c' -> a (b,b') (c,c')
  (&&&) :: a b c -> a b c' -> a b (c,c')
```

Control.Lens (viewing)

Getting with Getters

Any function $(s \to a)$ can be flipped into continuation passing style, $(a \to r) \to s \to r$ and decorated with **Const** to obtain:

```
type Getting r s a =
  (a -> Const r a) -> s -> Const r s
```

A **Getter** describes how to retrieve a single value in a way that can be composed with other **LensLike** constructions.

When you see this in a type signature it indicates that you can pass the function a **Lens**, **Getter**, **Traversal**, **Fold**, **Prism**, **Iso**, or one of the indexed variants, and it will just "do the right thing".

Safe head

Perform a safe head of a **Fold** or **Traversal** or retrieve **Just** the result from a **Getter** or **Lens**.

$(\hat{}?) \equiv \textit{flip preview}$

```
(^?) :: s -> Getting (First a) s a -> Maybe a

> Right 4 ^?_Left
Nothing
> "world" ^? ix 3
Just 'l'
```

Viewing lenses

View the value pointed to by a **Getter** or **Lens** or the result of folding over all the results of a **Fold** or **Traversal** that points at a monoidal values.

This is the same operation as **view** with the arguments flipped.

```
(^.) :: s -> Getting a s a -> a

> (0, -5)^._2.to abs
5
> ["a", "b", "c"] ^. traversed
"abc"
```

Using MonadState

Use the target of a **Lens**, **Iso**, or **Getter** in the current state, or use a summary of a **Fold** or **Traversal** that points to a monoidal value.

```
use :: MonadState s m => Getting a s a -> m a

> evalState (use _1) (1,2)
1
> evalState (uses _1 length) ("hello","")
5
```

Folding Foldables

```
type Fold s a =
  forall m. Monoid m => Getting m s a
```

A **Fold s a** is a generalization of something **Foldable**. It allows you to extract multiple results from a container. Every **Getter** is a valid **Fold** that simply doesn't use the Monoid it is passed.

If there exists a **foo** method that expects a **Foldable (f a)**, then there should be a **fooOf** method that takes a **Fold s a** and a value of type **s**.

Extracting lists from Folds

Extract a list of the targets of a **Fold**, an infix version of **toListOf**.

$toList\ xs \equiv xs\hat{}..folded$

```
(^..) :: s -> Getting (Endo [a]) s a -> [a]

> [[1,2],[3]] ^.. traverse . traverse
[1,2,3]
> (1,2) ^.. both
[1,2]
```

Checking for matches

Check to see if this **Fold** or **Traversal** matches 1 or more entries. For the opposite, use **hasn't**.

```
has :: Getting Any s a -> s -> Bool

> has (element 0) []
False
> has _Right (Left 12)
False
> hasn't _Right (Left 12)
True
```

Indexed Getters

For most operations, there is an indexed variant which will work as expected if the underlying target supports a notion of **Indexing**.

```
> ["ab", "c"] ^@.. itraversed <.> itraversed
[((0,0),'a'),((0,1),'b'),((1,0),'c')]
> "hello" ^@.. itraversed . indices even
[(0,'h'),(2,'l'),(4,'o')]

> ifind (\i k -> i > k) [1,2,2,2]
Just (3,2)
```

Control.Lens (setting)

Modifying records with Setters

A **Setter s t a b** is a generalization of fmap from **Functor**. It allows you to map into a structure and change out the contents, but it isn't strong enough to allow you to enumerate those contents. Starting with $fmap :: Functor f \Rightarrow (a \to b) \to f a \to f b$ we monomorphize the type to obtain $(a \to b) \to s \to t$ and then decorate it with Identity to obtain:

```
type Setter s t a b =
  (a -> Identity b) -> s -> Identity t
```

Every **Traversal** is a valid **Setter**, since **Identity** is **Applicative**.

Modifying with a function

Modifies the target of a **Lens** or all of the targets of a **Setter** or **Traversal** with a user supplied function.

This is an infix version of **over**.

```
(%~) :: Profunctor p
     => Setting p s t a b -> p a b -> s -> t

> traverse %~ even $ [1,2,3]
[False,True,False]
```

Modifying with a constant value

Replace the target of a **Lens** or all of the targets of a **Setter** or **Traversal** with a constant value.

```
(.~) :: ASetter s t a b -> b -> s -> t

> [1,2,3] & element 0 .~ 3
[3,2,3]
> [1,2,3] & traversed . filtered odd .~ 0
[0,2,0]
```

Prisms and Isos

An **Iso** is a pair of inverse functions. You can invert an **Iso** with **from**.

Prisms can be thought of as **Iso**s that can fail in one direction. You can invert a **Prism** with **re**.

```
type Prism s t a b
  forall p f. (Choice p, Applicative f) =>
              p a (f b) -> p s (f t)
type Prism' s a = Prism s s a a

prism  :: (b -> t)
       -> (s -> Either t a)
       -> Prism s t a b
prism' :: (a -> s)
       -> (s -> Maybe a)
       -> Prism' s a

> 5^.re _Left ^?! _Left
5
> _Left # 1
Left 1

type Iso s t a b =
  forall p f. (Profunctor p, Functor f) =>
              p a (f b) -> p s (f t)
type Iso' s a = Iso s s a a

iso :: (s -> a) -> (b -> t) -> Iso s t a b
from :: AnIso s t a b -> Iso b a t s

> 'a' ^. from enum
97
> 97 ^. enum :: Char
'a'

> Map.empty & at "hi"
>           . non Map.empty
>           . at "world" ?~ "!"
fromList [("hi",fromList [("world","!")])]
```

Some setting operators

<table>
<tr><th>Operator</th><th>W/result</th><th>W/state</th><th>W/both</th><th>Action</th></tr>
<tr><td>+~</td><td><+~</td><td>+=</td><td><+=</td><td>Add to target(s)</td></tr>
<tr><td>-~</td><td><-~</td><td>-=</td><td><-=</td><td>Subtract from target(s)</td></tr>
<tr><td>*~</td><td><*~</td><td>*=</td><td><*=</td><td>Multiply target(s)</td></tr>
<tr><td>//~</td><td><//~</td><td>//=</td><td><//=</td><td>Divide target(s)</td></tr>
<tr><td>^~</td><td><^~</td><td>^=</td><td><^=</td><td>Raise target(s) to a non-negative Integral power</td></tr>
<tr><td>^^~</td><td><^^~</td><td>^^=</td><td><^^=</td><td>Raise target(s) to an Integral power</td></tr>
<tr><td>**~</td><td><**~</td><td>**=</td><td><**=</td><td>Raise target(s) to an arbitrary power</td></tr>
<tr><td>||~</td><td><||~</td><td>||=</td><td><||=</td><td>Logically or target(s)</td></tr>
<tr><td>&&~</td><td><&&~</td><td>&&=</td><td><&&=</td><td>Logically and target(s)</td></tr>
<tr><td><>~</td><td><<>~</td><td><>=</td><td><<>=</td><td>mappend to the target monoidal value(s)</td></tr>
</table>