Monoids

Sept. 20 2024

Functions

- Consider the addition function:

Functions

- Consider the addition function:
  - `1 + 1 = 2`

Functions

- Consider the addition function:
  - `1 + 1 = 2`
  - `2 + 2 = 4`

Functions

- Consider the addition function:
  - `1 + 1 = 2`
  - `2 + 2 = 4`
- `(+) :: Num a => a -> a -> a`

Functions

- Consider the addition function:
  - `1 + 1 = 2`
  - `2 + 2 = 4`
- `(+) :: Int -> Int -> Int`

Functions

- Addition by 0:

Functions

- Addition by 0:
  - `0 + x = x`

Functions

- Addition by 0:
  - `0 + x = x`
  - `x + 0 = x`

Functions

- Addition by 0:
  - `0 + x = x`
  - `x + 0 = x`
- Addition of three numbers:

Functions

- Addition by 0:
  - `0 + x = x`
  - `x + 0 = x`
- Addition of three numbers:
  - `(x + y) + z = x + (y + z)`

Functions

- Addition by 0:
  - Right Identity: `0 + x = x`
  - Left Identity: `x + 0 = x`
- Addition of three numbers:
  - Associativity: `(x + y) + z = x + (y + z)`

Monoids

- Wikipedia: Suppose that S is a set and $\bullet$ is some binary operation $S \times S \to S$, then S with $\bullet$ is a monoid if it satisfies the following two axioms:
  - Associativity: For all a, b and c in S, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds.
  - Identity element: There exists an element e in S such that for every element a in S, the equations $e \bullet a = a \bullet e = a$ hold.

Monoids

- Suppose that `m` is a type and `mappend` is some binary function `m -> m -> m`, then `m` with `mappend` is a monoid if it satisfies the following two axioms:
  - Associativity: For all `x`, `y` and `z` in `m`, the equation `(x ‘mappend‘ y) ‘mappend‘ z = x ‘mappend‘ (y ‘mappend‘ z)` holds.
  - Identity element: There exists an element `mempty` in `m` such that for every element `x` in `m`, the equations `mempty ‘mappend‘ x = x ‘mappend‘ mempty = x` hold.

Monoids

- ```
  class Monoid m where
      mempty ::  m
      mappend ::  m -> m -> m
      mconcat ::  [m] -> m
      mconcat = foldr mappend mempty
  ```
  ```

Monoids

- ```
  class Monoid m where
      mempty ::  m                          (zero)
      mappend ::  m -> m -> m               (plus)
      mconcat ::  [m] -> m                  (sum)
      mconcat = foldr mappend mempty
  ```
  ```

Lists are Monoids

- ```
  instance Monoid [a] where
      mempty = []
      mappend = (++)
  ```
  ```

Lists are Monoids

- ```
  instance Monoid [a] where
      mempty = []
      mappend = (++)
  ```
  - `mconcat = concat`
  ```

Lists of a are Monoids

- ```
  instance Monoid [a] where
      mempty = []
      mappend = (++)
  ```
  - `mconcat = concat`
  ```

Strings are Monoids

- `instance Monoid String where`
  ```
  mempty = ""
  mappend = (++)
  ```
  - `mconcat = concat`

Languages are Monoids

- ```
  instance Monoid String where
      mempty = ""
      mappend = (++)
  ```
  - `mconcat = concat`
  ```

Strings are Monoids

- `"" ++ "the" = "the"`

Strings are Monoids

- `"" ++ "the" = "the"`
- `"the" ++ "" = "the"`

Strings are Monoids

- `"" ++ "the" = "the"`
- `"the" ++ "" = "the"`
- `("the␣" ++ "dog␣") ++ "barked" = "the␣" ++ ("dog␣" ++ "barked")`

Monoids

- Other examples of monoids:

Monoids

- Other examples of monoids:
    - Numbers (`Product`, `Sum`)

Monoids

- Other examples of monoids:
  - Numbers (`Product`, `Sum`)
  - `Bool` (`Any`, `All`)

Monoids

- Other examples of monoids:
  - Numbers (`Product`, `Sum`)
  - `Bool` (`Any`, `All`)
  - `Ordering`

Monoids

- Other examples of monoids:
  - Numbers (`Product`, `Sum`)
  - `Bool` (`Any`, `All`)
  - `Ordering`
  - `Maybe`

Monoids

- Other examples of monoids:
  - Numbers (`Product`, `Sum`)
  - `Bool` (`Any`, `All`)
  - `Ordering`
  - `Maybe`
  - Functions (`r -> r`) (`Endo`)

Functions are Monoids

- ```
  instance Monoid (a -> a) where
      mempty = id
      mappend = (.)
  ```
  ```

Functions are Monoids

- ```
  instance Monoid (Endo a) where
      mempty = Endo id
      Endo g `mappend` Endo f = Endo (g .  f)
  ```
  ```

Functions are Monoids

- ```
  instance Monoid (Endo a) where
      mempty = Endo id
      Endo g `mappend` Endo f = Endo (g . f)
  ```
- `newtype Endo a = Endo { appEndo :: a -> a }`
  ```

Functions are Monoids

- `id . f = f`

Functions are Monoids

- `id . f = f`
- `f . id = f`

Functions are Monoids

- `id . f = f`
- `f . id = f`
- `(f . g) . h = f . (g . h)`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat = foldr mappend mempty`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappending` them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappending` them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap = foldr (mappend . f) mempty`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
  - What if we don't have a conversion function?

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
    - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
    - Sequence (e.g. list, tree)
    - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
    - What if we don't have a conversion function?
- `:: Foldable t => (a -> Endo b) -> t a -> Endo b`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
  - What if we don't have a conversion function?
- `:: Foldable t => (a -> b -> b) -> t a -> b -> b`

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
    - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
    - Sequence (e.g. list, tree)
    - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
    - What if we don't have a conversion function?
- `:: Foldable t => (a -> b -> b) -> t a -> b -> b`
    - Accumulator value

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
  - What if we don't have a conversion function?
- `:: Foldable t => (a -> b -> b) -> t a -> b -> b`
  - Accumulator value
  - Function that updates the accumulator with the arbitrary value

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
  - What if we don't have a conversion function?
- `:: Foldable t => (a -> b -> b) -> t a -> b -> b`
  - Accumulator value
  - Function that updates the accumulator with the arbitrary value
  - Starting accumulator value

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
    - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
    - Sequence (e.g. list, tree)
    - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
    - What if we don't have a conversion function?
- `:: Foldable t => (a -> b -> b) -> t a -> b -> b`
    - Accumulator value
    - Function that updates the accumulator with the arbitrary value
    - Starting accumulator value
    - Result value

Folds

- `mconcat :: Monoid m => [m] -> m`
- `mconcat` combines a list of monoid values by `mappend`ing them
  - What if we want to combine a sequence of arbitrary values?
- `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`
  - Sequence (e.g. list, tree)
  - Function that converts arbitrary values to monoid values
- `foldMap` combines a list of arbitrary values by converting them into monoid values and `mappend`ing them
  - What if we don't have a conversion function?
- `foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b`
  - Accumulator value
  - Function that updates the accumulator with the arbitrary value
  - Starting accumulator value
  - Result value

List Comprehensions

- ```
  [1,2,3,4] = [a + b |
                a <- [0,2],
                b <- [1,2]]
  ```
  ```

List Comprehensions

- ```
  [1,2,3,4] = [a + b |
               a <- [0,2],
               b <- [1,2]]
  ```

  ```
            = do
               a <- [0,2]
               b <- [1,2]
               return $ a + b
  ```
  ```

List Comprehensions

- ```
  [1,2,3,4] = [a + b |
                a <- [0,2],
                b <- [1,2],
                even $ a + b]

            = do
                a <- [0,2]
                b <- [1,2]
                return $ a + b
  ```
  ```

List Comprehensions

- ```
  [  2,   4] = [a + b |
                 a <- [0,2],
                 b <- [1,2],
                 even $ a + b]

             = do
                 a <- [0,2]
                 b <- [1,2]
                 return $ a + b
  ```
  ```

List Comprehensions

- ```
  [   2,   4] = [a + b |
                  a <- [0,2],
                  b <- [1,2],
                  even $ a + b]

                = do
                  a <- [0,2]
                  b <- [1,2]
                  guard (even $ a + b)
                  return $ a + b
  ```
  ```

MonadPluses

- ```
  class Monad m => MonadPlus m where
      mzero :: m a
      mplus :: m a -> m a -> m a
  ```
  ```

MonadPluses

- ```
  class Monad m => MonadPlus m where
        mzero ::  m a                 (mempty)
        mplus ::  m a -> m a -> m a (mappend)
  ```
  ```

MonadPluses

- ```
  class (Monad m, Monoid m a) => MonadPlus m where
      mzero ::  m a                   (mempty)
      mplus ::  m a -> m a -> m a (mappend)
  ```
  ```

MonadPluses

- ```
  class Monad m => MonadPlus m where
      mzero ::  m a                (mempty)
      mplus ::  m a -> m a -> m a  (mappend)
  ```
  ```

Lists are MonadPluses

- `instance MonadPlus [] where`
  ```
  mzero = []
  mplus = (++)
  ```

Guards

- `guard :: (MonadPlus m) => Bool -> m ()`

```
guard True = return ()
guard False = mzero
```

Guards

- ```
  [2,4] = do
          a <- [0,2]
          b <- [1,2]
          guard (even $ a + b)
          return $ a + b
  ```
  ```

Guards

- [2,4] =

```
[0,2] >>= \a -> do
b <- [1,2]
guard (even $ a + b)
return $ a + b
```

Guards

- `[2,4] =`

```
[0,2] >>= \a ->
[1,2] >>= \b -> do
guard (even $ a + b)
return $ a + b
```

Guards

- `[2,4] =`

```
[0,2] >>= \a ->
[1,2] >>= \b ->
guard (even $ a + b) >>= \_ -> do
return $ a + b
```

Guards

- `[2,4] =`

```
[0,2] >>= \a ->
[1,2] >>= \b ->
guard (even $ a + b) >>= \_ ->
return $ a + b
```

Guards

- `[2,4] =`

```
concat (map (\a ->
[1,2] >>= \b ->
guard (even $ a + b) >>= \_ ->
return $ a + b) [0,2])
```

Guards

- `[2,4] = concat [`

```
([1,2] >>= \b ->
 guard (even $ 0 + b) >>= \_ ->
 return $ 0 + b),
([1,2] >>= \b ->
 guard (even $ 2 + b) >>= \_ ->
 return $ 2 + b)
]
```

Guards

- ```
  [2,4] = concat [
            concat (map (\b ->
             guard (even $ 0 + b) >>= _ ->
             return $ 0 + b) [1,2]),
            concat (map (\b ->
             guard (even $ 2 + b) >>= _ ->
             return $ 2 + b) [1,2])
           ]
  ```
  ```

Guards

- [2,4] = concat [

```
              concat [(guard (even $ 0 + 1) >>= \_ ->
                        return $ 0 + 1),
                       (guard (even $ 0 + 2) >>= \_ ->
                        return $ 0 + 2)],
              concat [(guard (even $ 2 + 1) >>= \_ ->
                        return $ 2 + 1),
                       (guard (even $ 2 + 2) >>= \_ ->
                        return $ 2 + 2)]
             ]
```

Guards

- ```
  [2,4] = concat [
            concat [(guard (even $ 1) >>= _ ->
                      return $ 1),
                     (guard (even $ 2) >>= _ ->
                      return $ 2)],
            concat [(guard (even $ 3) >>= _ ->
                      return $ 3),
                     (guard (even $ 4) >>= _ ->
                      return $ 4)]
          ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
            concat [(guard False >>= _ ->
                      [1]),
                     (guard True >>= _ ->
                      [2])],
            concat [(guard False >>= _ ->
                      [3]),
                     (guard True >>= _ ->
                      [4])]
           ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
             concat [(mzero >>= _ ->
                      [1]),
                     (return () >>= _ ->
                      [2])],
             concat [(mzero >>= _ ->
                      [3]),
                     (return () >>= _ ->
                      [4])]
            ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
           concat [([] >>= _ ->
                     [1]),
                    ([()] >>= _ ->
                     [2])],
           concat [([] >>= _ ->
                     [3]),
                    ([()] >>= _ ->
                     [4])]
          ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
           concat [concat (map (_ -> [1]) []),
                   concat (map (_ -> [2]) [()])],
           concat [concat (map (_ -> [3]) []),
                   concat (map (_ -> [4]) [()])]
          ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
             concat [concat [],
                     concat [[2]]],
             concat [concat [],
                     concat [[4]]]
            ]
  ```
  ```

Guards

- ```
  [2,4] = concat [
           concat [[],
                   [2]],
           concat [[],
                   [4]]
          ]
  ```
  ```

Guards

- `[2,4] = concat [[2],[4]]`

Guards

- `[2,4] = [2,4]`