Laws!

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

28th August 2018
class Monoid m where
    mempty :: m
    (<>)   :: m -> m -> m

Left identity: mempty <> y = y
Right identity: x <> mempty = x
Associativity: (x <> y) <> z = x <> (y <> z)
class Monoid m where
  mempty :: m
  (<>) :: m -> m -> m

  Left identity:    mempty <> y = y
  Right identity:  x <> mempty = x
  Associativity:   (x <> y) <> z = x <> (y <> z)
data Sum = Sum Int

instance Monoid Sum where
  mempty = Sum 0
  Sum x <> Sum y = Sum (x + y)
Left identity: \[ 0 + y = y \]

Right identity: \[ x + 0 = x \]
Left identity:  
\[ 0 + 5 = 5 \]

Right identity:  
\[ x + 0 = x \]
Left identity:
\[ 0 + 5 = 5 \]

Right identity:
\[ 7 + 0 = 7 \]
Associativity:

\[ 3 + 4 + 5 \]

\[ (3 + 4) + 5 \]

\[ 3 + (4 + 5) \]
$$\frac{3}{4} + \frac{5}{4} + \frac{5}{3}$$

$$\frac{3}{4} + \frac{5}{4} + \frac{5}{3}$$
12

3 + 4

+ 5

+
So?
(1 + (2 + (3 + 4) + 5)) + 6 + 7
\texttt{mconcat} :: \texttt{Monoid} \ m \Rightarrow \ [m] \rightarrow \ m
mconcat :: Monoid m => [m] -> m

mconcat list =
  case list of
    [] -> mempty
    (h:t) -> h <> mconcat t
\texttt{mconcat [\texttt{Sum} 1, \texttt{Sum} 2, \texttt{Sum} 3, \texttt{Sum} 4]}
mconcat [\textbf{Sum} 1, \textbf{Sum} 2, \textbf{Sum} 3, \textbf{Sum} 4]

\textbf{Sum} 1 \texttt{<>} (\textbf{Sum} 2 \texttt{<>} (\textbf{Sum} 3 \texttt{<>} (\textbf{Sum} 4 \texttt{<> mempty}))))
mconcat \[\text{Sum } 1, \text{Sum } 2, \text{Sum } 3, \text{Sum } 4\]

\[
\text{Sum } 1 \langle\rangle (\text{Sum } 2 \langle\rangle (\text{Sum } 3 \langle\rangle (\text{Sum } 4 \langle\rangle \text{mempty})))
\]

\[\Rightarrow \text{Sum } 10\]
mconcatR :: NotMonoid m => [m] -> m

mconcatR list =
  case list of
    []    -> mempty
    (h:t) -> h <> mconcatR t
mconcatR :: NotMonoid m => [m] -> m

mconcatR list =
    case list of
      [] -> mempty
      (h:t) -> h <> mconcatR t

mconcatL :: NotMonoid m => [m] -> m

mconcatL list =
    helper mempty list
    where
      helper acc xs =
        case xs of
          [] -> acc
          (h:t) -> helper (acc <> h) t
foldr :: (a -> b -> b) -> b -> [a] -> b

foldl :: (b -> a -> b) -> b -> [a] -> b
Laws give us **freedom** when working **in terms of** our abstractions
instance Monoid [a] where
    mempty = []
    left <> right =
        case left of
            [] -> right
            (h:t) -> h : (t <> right)
instance Monoid [a] where
  mempty   = []
  left <> right =
    case left of
      []    -> right
      (h:t) -> h : (t <> right)

Left identity:  [] ++ y  = y

Right identity:  x ++ []  = x

Associativity:  (x ++ y) ++ z = x ++ (y ++ z)
greeting :: [Char] -> [Char]

greeting name =
  "(" <> "Hello, " <> name <> ", how are you?" <> ")"
greeting :: [Char] -> [Char]
greeting name =
   "(" <> "Hello, " <> name <> ", how are you?" <> ")"

between op cl x =
   op <> x <> cl
greeting :: [Char] -> [Char]

greeting name =
  between "(" ")" $
    "Hello, " <> name <> " , how are you?"

between op cl x =
  op <> x <> cl
greeting :: [Char] -> [Char]

greeting name =
    between "(" ")" $
    between "Hello, " ", how are you?"
    name

between op cl x =
    op <> x <> cl
Hello, name, how are you?
Hello, name, how are you?
Laws let us **refactor** and **reuse** more.
([1, 2, 3] <> [4, 5, 6]) <> [7, 8, 9]
([1, 2, 3] <> [4, 5, 6]) <> [7, 8, 9]

( :(
\[
1 : 2 : 3 : \text{Nil} \quad 4 : 5 : 6 : \text{Nil} \quad 7 : 8 : 9 : \text{Nil}
\]
\[
1 : 2 : 3
\]
1 : 2 : 3 :

4 : 5 : 6 : Nil

7 : 8 : 9 : Nil
data DList α

instance Monoid (DList α) -- O(1) append

fromList :: [α] -> DList α -- O(1)

toList :: DList α -> [α] -- O(n)
data DList a

instance Monoid (DList a) -- O(1) append
data DList a

instance Monoid (DList a)  -- O(1) append

fromList :: [a]    -> DList a  -- O(1)
toList    :: DList a    -> [a]  -- O(n)
result :: [a]
result = ((((((x <> y) <> z) <> ...
result :: [a]
result = ((((((x <> y) <> z) <> ...  

appended :: DList a
appended = ((((fromList x <> fromList y) <> fromList z) <> ...  

result' :: [a]
result' = toList appended
$O(n^2)$

left-associated appends

list $\rightarrow$ list
\[ O(n^2) \]

left-associated appends

\[ \text{list} \rightarrow \text{list} \]

\[ O(n) \]

fromList

\[ \text{DList} \]
\( O(n^2) \)

\[
\text{list} \quad \rightarrow \quad \text{list}
\]

left-associated appends

\( O(n) \)

\[
\text{fromList}
\]

\[
\text{DList} \quad \rightarrow \quad \text{DList}
\]

left-associated appends

\( O(n) \)
Optimisation is altering the program to get **the same answer** more efficiently.
toList is the left inverse of fromList

toList (fromList x) = x
fromList is a monoid homomorphism

\[ \text{fromList} :: [a] \rightarrow \text{DList} \ a \]
fromList is a monoid homomorphism

fromList :: [a] -> DList a

fromList mempty = mempty

fromList (x <> y) = fromList x <> fromList y
x <> y <> z
\[
x \leftrightarrow y \leftrightarrow z
\]

**Left inverse:** \( \text{toList} \left( \text{fromList} \left( x \right) \right) = x \)
toList (fromList (x <> y <> z))

Left inverse: \[ \text{toList} \ (\text{fromList} \ (x)) = x \]
toList (fromList (x <> y <> z))

Monoid homomorphism: fromList (x <> y <> z)

= fromList x <> fromList y <> fromList z
toList (fromList x <> fromList y <> fromList z)

Monoid homomorphism:  

\[
\text{fromList } (x <> y <> z) = \text{fromList } x <> \text{fromList } y <> \text{fromList } z
\]
What about a world without laws?
class Default a where
  def :: a
class Default a where
  def :: a

instance Default [a] where
  def = []
class Default a where
    def :: a

instance Default [a] where
    def = []

instance Default Int where
    def = 0
orElse :: a -> Maybe a -> Maybe a
orElse d ma =
case ma of
    Just a  -> a
    Nothing -> d
orElse :: a -> Maybe a -> a
orElse d ma =
case ma of
  Just a  -> a
  Nothing -> d
data-default: A class for types with a default value

Versions
0.2, 0.2.0.1, 0.3.0, 0.4.0, 0.5.0, 0.5.1, 0.5.2, 0.5.3, 0.6.0, 0.7.0, 0.7.1, 0.7.1.1

Dependencies
base (>=2 && <5), data-default-class (>=0.1.2.0), data-default-instances-containers, data-default-instances-dlist, data-default-instances-old-locale [details]

License
BSD-3-Clause
acme-default: A class for types with a distinguished aesthetically pleasing value

This package defines a type class for types with certain distinguished values that someone considers to be aesthetically pleasing. Such a value is commonly referred to as a default value.

This package exists to introduce artistic variety regarding the aesthetics of Haskell’s base types, but is otherwise identical in purpose to data-default.
instance Default Int64 where 
def = -1
-- | Current default -1 chosen by ertes, the largest negative number.

_instance Default Int64 where_
  def = -1

-- | Current default 'False' chosen by ertes, the answer to the question whether mniip has a favourite 'Bool'.

_instance Default Bool where_
  def = False
instance Default Int64 where
  def = -1

instance Default Bool where
  def = False

instance Default String where
  def = "Call me Ishmael. Some years ago - never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world."
How do I know whether I obey the laws?
QuickCheck + checkers

Property-based testing for laws!
monoid :: (Monoid a, Show a, Arbitrary a, EqProp a) => a -> TestBatch
monoid :: (Monoid a, Show a, Arbitrary a, EqProp a) => a -> TestBatch

functor :: (Functor t, Arbitrary a, Arbitrary b, Arbitrary c, CoArbitrary a, CoArbitrary b, Show (t a), Arbitrary (t a), EqProp (t a), EqProp (t c)) => t (a, b, c) -> TestBatch
data Subtraction = Subt Int

-- totally dodgy
instance Monoid Subtraction where
    mempty = Subt 0
    Subt x <> Subt y = Subt (x - y)
data Subtraction = Subt Int

-- totally dodgy
instance Monoid Subtraction where
  mempty = Subt 0
  Subt x <> Subt y = Subt (x - y)

main :: IO ()
main = do
  quickBatch (monoid (Sum 0))
  quickBatch (monoid (Subt 0))
Sum monoid:
  left identity: +++ OK, passed 500 tests.
  right identity: +++ OK, passed 500 tests.
  associativity: +++ OK, passed 500 tests.

Subtraction "monoid",
  left identity: *** Failed! Falsifiable (after 2 tests)
  right identity: +++ OK, passed 500 tests.
  associativity: *** Failed! Falsifiable (after 2 tests)
Laws give rise to useful functions

Laws allow us to refactor more

Laws help us to optimise
Laws are the difference between an **overloaded name** and an **abstraction**
Thanks for listening!
References

- Daniel J. Velleman “How To Prove It”
- Tom Ellis “Demystifying DList”
  http://h2.jaguarpaw.co.uk/posts/demystifying-dlist/
- Edward Kmett “Why not Pointed?”
  https://wiki.haskell.org/Why_not_Pointed%3F
- Tim Humphries “Continuations All The Way Down”
- Edward Kmett “The Free Theorem for fmap”
  https://www.schoolofhaskell.com/user/edwardk/snippets/fmap
What’s up with Foldable?

It sort of has laws.

- Gershom Bazerman wrote a paper:
- Then started a mailing list discussion:
  https://mail.haskell.org/pipermail/libraries/2015-February/024943.html
- ...and then another one:
  https://mail.haskell.org/pipermail/libraries/2018-May/028761.html
Are there reasonable cases of law breakage?
Are there reasonable cases of law breakage?

Yes! Both QuickCheck and hedgehog break the Applicative and Monad laws.