

Императивное программирование на Haskell. Шаг-2. Изменяемые переменные

Монада State и монадные трансформеры. Пример

В будущем, чтобы научиться работать с изменяемыми переменными, нам необходимо изучить тему работы с монадами и монадными трансформерами. Это тема — отдельная и трудная для беглого понимания. Вот небольшой иллюстративный пример, а саму тему оставим «на потом».

```
import Control.Monad.State
import System.IO

code :: StateT Int IO ()
code = do
    x <- get
    liftIO $ print x
    liftIO $ putStr "Input number: "
    y <- liftIO $ (readLn :: IO Int)
    let z = x + y
    put z
    liftIO $ print z
    return ()

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    runStateT code 1
    return ()
```

Сам пример ничего интересного не представляет, на Паскале он выглядел бы примерно так:

```
var
    x,y,z,datastorage: integer;
begin
    datastorage := 1;
    x := datastorage;
    writeln(x);
    write('Input number: ');
    readln(y);
    z := x+y;
    datastorage := z;
    writeln(z);
end.
```

В Haskell у нас нет возможности изменять переменные, таких переменных попросту не существует. То, что в коде на Haskell выглядит как переменные `x,y,z`, на самом деле — неизменяемые константы (в этом случае монад — всё ещё сложнее). А для того, чтобы у нас была возможность использовать изменяемые переменные (в терминах Haskell — изменяемые состояния), нам требуется специальная монада `State` с утилитами `get` и `put`. Эту ситуацию можно мыслить как обращение к базе данных, когда `x<-get` позволяет получить некоторое значение и связать его с «переменной» `x`, а `put x` отправляет значение `x` в эту базу.

Кроме того, мы не можем просто так в одном блоке `main` организовать работу и с вводом-выводом, и с сохранением состояний, как в обычных языках. Поэтому нам приходится организовывать два блока кода. Один из которых `main` отвечает традиционно за ввод-вывод, а другой, в нашем примере `code`, отвечает за работу с состояниями. При этом, блок `code` теперь организован не как обычная монада состояния `State`, а как монада-трансформер `StateT` над монадой ввода-вывода. Об этом мы будем позже говорить подробнее, а сейчас нам важно видеть, что в блоке `code` мы можем вызывать утилиты ввода-вывода с помощью `liftIO`.

Несколько полезных примеров по работе с трансформером-монадой состояния `StateT` в связке с монадой ввода-вывода можно найти здесь [Simple StateT use](#).

Отметим, что по своей сути, работа со `State` это очень виртуозная эмуляция состояний средствами функционального программирования с привлечением мощного математического аппарата *теории категорий*. Весь смысл здесь направлен не на эффективность кода (который по сути так и остается передачей параметров вместо работы с глобальными переменными), а на удобство представления с точки зрения императивного программирования.

Реально изменяемые состояния

В Haskell'е есть то, чего там нет и быть не должно — реально изменяемые переменные. Особого удобства для работы с ними Haskell не предоставляет, да и не должен. Они не слишком рекомендуются в обычном Haskell-программировании, хотя и не запрещаются. Применяются они исключительно внутри **IO**-монад и получаемый код не является «чистым», таким образом теряя все преимущества чистого Haskell-программирования.

Data.IORef

Модуль [Data.IORef](#) позволяет нам в рамках стандарта Haskell-98 использовать ссылки на реально изменяемые переменные в рамках использования монады **IO**, т.е. как мы ранее договорились, по крайней мере, внутри действия `main`.

```
import Data.IORef

main :: IO ()
main = do
    a <- newIORef (10 :: Int)

    x <- readIORef a
    print (x+2)

    writeIORef a (x+2)
    modifyIORef a (*4)

    readIORef a >>= print
```

Действие `newIORef` создает ссылку на «контейнер», содержащее число 10, тип которого мы указали как `Int`. Чтение осуществляется действием `x <- readIORef a`, при этом, `x` будет «чистой одноразовой переменной» типа `Int`.

Этот контейнер не может быть пустым, но мы можем легко изменять его значения. С помощью действий `writeIORef a (x+2)` или `writeIORef a (12)` мы можем записать в него любое чистое значение.

А с помощью действия `modifyIORef a (*4)`, которое вторым параметром получает чистую функцию, мы можем «на месте» с ее помощью изменить значение в этом контейнере.

Эта операция ленивая, и она может приводить к неприятным эффектам типа `stack overflow`:

```
import Data.IORef
import Control.Monad

main :: IO ()
main = do

    ref <- newIORef 0
    replicateM_ 100000000 $ modifyIORef ref (+1)
    readIORef ref >>= print
```

Вот этот тест [согласно документации](#) уже на счетчике 1000000 должен давать `stack overflow`. У меня он сработал, но при 100000000 повторах зависал. Рекомендуют использовать энергичную (строгую) версию `modifyIORef'`. Она в тест сильнее грузит процессор, но доходит до успешного конца.

В любом случае, этот метод считается «потоко-небезопасным» и рекомендуется использовать более изощренные варианты.

MVar

Следующая техника, предоставляемая модулем [Control.Concurrent](#), как раз является потокобезопасной, рассчитанной на многопоточное использование, и её применение в обычных программах для сохранения состояния является «как бы побочным эффектом». Мы не будем разбирать основы и детали конкурентного программирования, разберем только простой пример, который дает нам возможность работы с сохранением состояния.

```
-- import Control.Concurrent
import Control.Concurrent.MVar

main :: IO ()
main = do
    a <- newEmptyMVar
    putMVar a (1 :: Int)
    m <- takeMVar a
    putStrLn $ show (m+1)
    putMVar a 10
    n <- takeMVar a
    putStrLn $ show n

    b <- newMVar (100 :: Double)
    takeMVar b >>= print
```

Рассмотрим код. Импорт модуля `Control.Concurrent` как раз показывает, что мы используем возможности конкурентного написания программ. Действие `a <- newEmptyMVar` создает новый пустой контейнер и ссылку на него. С `MVar` он может быть пустым. Далее мы помещаем в него значение типа `Int`, одновременно задавая его тип (уже навсегда). В этот момент доступ на запись блокируется (для всех потоков), до тех пор, пока значение не будет изъято с помощью действия `m <- takeMVar a`, которое по ссылке извлекает из контейнера чистое значение в «неизменяемую переменную» `m` и блокирует *пустой контейнер* на чтение, оставляя возможность только на запись. Таким образом, гарантируется, что два потока не изменят значение переменной в одно и то же время.

Также возможна работа в стиле `newIORef (100 :: Double)`, когда мы создаём контейнер и сразу его инициализируем:

```
b <- newMVar (100 :: Double)
```

[A Guide to Mutable References](#)

[Mutable State in Haskell](#)

[Data.IORef](#)

[Control.Concurrent.MVar](#)

[S.Marlow. Parallel and Concurrent Programming in Haskell. 3.2 Communication: MVars](#)

На этом, знакомство с императивным программированием в Haskell пока закончим.

Работа с текстом и текстовыми файлами. Шаг-2

Рассмотрим более подробно работу с `Data.Text`. Итак, как нам говорят руководства, существует как минимум 5 различных типов данных для обработки строк в Haskell: `String`, `Text` и `ByteString` (последние два в ленивой и строгой версии).

Алехандро Серано Мена, «Изучаем Haskell», С.282

[Data.Text](#)

из пакета

[text](#)

[Data.ByteString](#)

из пакета

[bytestring](#)

Тип `String` на базе полиморфного типа списка — универсален во многих смыслах: так, любая функция, работающая со списком, будет уметь работать со строками `String`.

Но уже много раз обсужденный тип данных `String` имеет проблемы, связанные прежде всего со своей списочной организацией:

Тип данных `ByteString` в общем-то для текста не предназначен, он рассматривает потоки байт как массивы (8-битных значений или символов). Этот тип подходит, если нам не

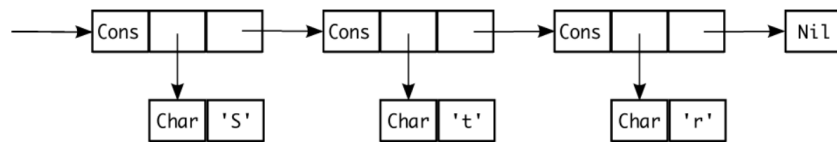


Рис. 1: Низкоуровневая структура String

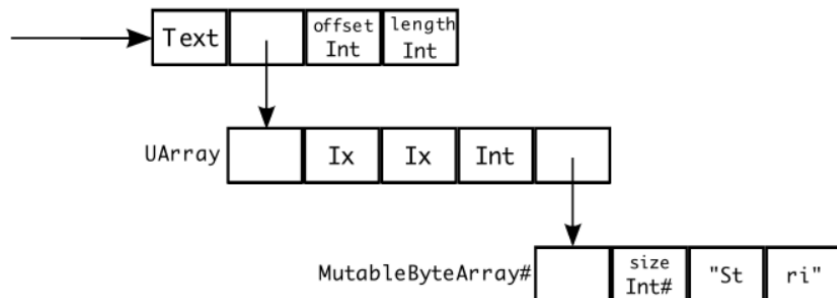


Рис. 2: Низкоуровневая структура Text

нужен Unicode или какой-то дополнительный смысл, но нужна быстрая низкоуровневая обработка в бинарном представлении.

Тип данных `Text` предназначен узко для работы с текстом, он не универсален, организован на базе массивов, поэтому, за исключением операции `cons` (как `(:)` в списках, т.е. добавление символа в начало строки) и операции конкатенации (слияние двух строк), которые в списках осуществляются за константное время беспрецедентно быстро, — все остальные операции обработки в типе данных `Text` будут в несколько раз быстрее и эффективнее по памяти.

Картинки ниже иллюстрируют эффективность различных текстовых типов для разных задач:

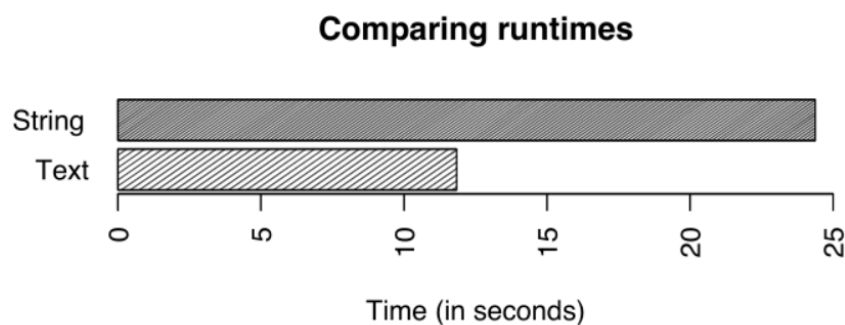


Рис. 3: Сравнение времени обработки

[T.Harper. Stream Fusion on Haskell Unicode Strings](#)

Давайте вновь рассмотрим типовую обработку строк с помощью типа данных `Text`:

```

import qualified Data.Text as T

test :: T.Text
test = 'c' `T.cons` (T.toLower ((T.pack "Hello ")))
  
```

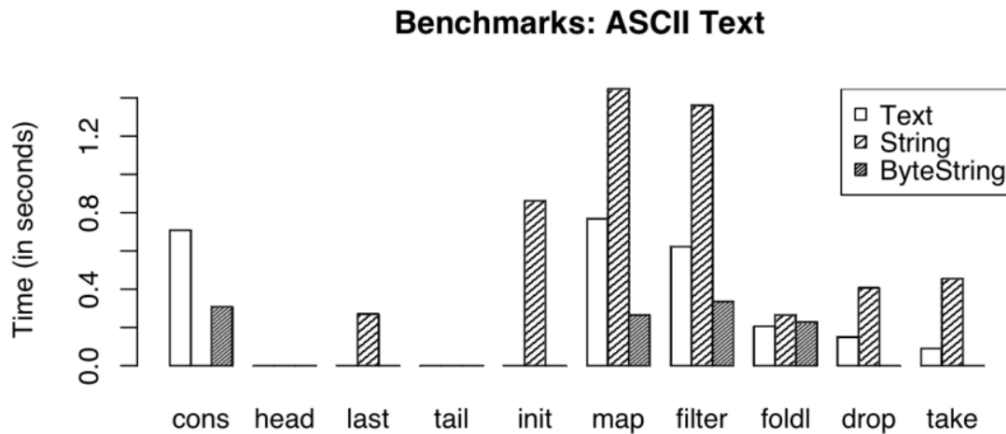


Рис. 4: Эффективность операций обработки текста

```
`T.append` (T.pack " Person!"))
```

Усовершенствуем, разрешим использование строковых литералов как текстовых строк:

```
{-# LANGUAGE OverloadedStrings #-}
```

```
import qualified Data.Text as T
```

```
test :: T.Text
test = 'c' `T.cons`
      (T.toLower $ "Hello " `T.append` " Person!")
```

Используем удобный оператор `<>` из пакета `Data.Monoid`, который мы в дальнейшем часто будем использовать вместо `++`:

```
{-# LANGUAGE OverloadedStrings #-}
```

```
import Data.Text
import Data.Monoid ((<>))
```

```
test :: Text
test = 'c' `cons` (toLower $ "Hello " <> " Person!")
```

Стоит отметить, что есть модуль [Data.Text.Lazy.Builder](#) в составе пакета [text](#) и отдельно развиваемый модуль [Text.Builder](#) специально оптимизированные для работы с конкатенацией строк (при установке последний, правда, тащит собой половину хранилища Hackage).

```
{-# LANGUAGE OverloadedStrings #-}
```

```
import Data.Text.Lazy.Builder
import Data.Monoid ((<>))
```

```
test :: Builder
test = singleton 'c' <> ("Hello " <> " Person!")
```

и

```
{-# LANGUAGE OverloadedStrings #-}

import Text.Builder
import Data.Monoid ((<>))

test = run $ char 'c' <>
      (text "Hello " <> text " Person!")
```

Отметим, что в современных версиях ghc нет необходимости делать импорт

```
import Data.Monoid ((<>))
```

операция (<>) импортируется теперь по умолчанию, а указанный импорт можно оставить из соображений совместимости с более ранними версиями компилятора.

WorkSample with Data.Text

Рассмотрим почти полностью (пока нет обработки исключений) сформированную программу по обработке текстового файла:

```
{-# LANGUAGE OverloadedStrings #-}

import qualified System.IO as S
import qualified Data.Text as T

import qualified Data.Text.IO as TIO
import Data.Monoid
import Data.IRef

transform n s = n <> T.replace "@gmail.com" "@yahoo.com" s

main = do
  openhandle <- S.openFile "test.txt" S.ReadMode
  writehandle <- S.openFile "test2.txt" S.WriteMode

  a <- newIORef (1 :: Int)

  let
    readByLn = do
      eof <- S.hIsEOF openhandle
      if eof
      then return ()
      else do
        str <- TIO.hGetLine openhandle
        n <- readIORef a
        TIO.hPutStrLn writehandle
          (transform ((T.pack $ show n) <> ": ") str)
        writeIORef a (n+1)
        readByLn

  readByLn
```

```
S.hClose openhandle
S.hClose writehandle
```

Вместо цикла с условием `while`, мы используем рекурсивно заданную функцию `readByLn`, которая построчно (и буфер по умолчанию построчный) считывает текстовый файл, обрабатывает его с помощью чистой функции `transform`, кроме того, отслеживает счетчик итераций и его значение использует для нумерации строк. Обработанные строки выводятся в другой файл. При достижении конца файла

```
eof <- S.hIsEOF openhandle
if eof
  then return ()
```

выходим из выполнения рекурсивных итераций.

А если мы установим специализированный пакет для «показа» текстовых строк [TextShow](#)

```
cabal install text-show
```

то мы можем избежать промежуточного преобразования числа в тип **String** во фрагменте (`T.pack $ show n`) и получим такой более короткий и эффективный код:

```
...
import TextShow
...
TIO.hPutStrLn writehandle (transform ((showt n) <> ": ") str)
```

Программа получилась вполне императивной. :)

Моноиды

Моноиды в математике

Выше, мы столкнулись с оператором `<>`, который оказался удобным. Откуда он взялся? Оказывается, это бинарная функция, заданная в классе `Monoid`, и даже точнее, в классе — `Semigroup`. Давайте разберемся, что и откуда.

Определение 1. Алгебраическую систему $\mathcal{G} = \langle G, \cdot \rangle$ назовем *группой*, если на ней выполнены следующие аксиомы:

1. ассоциативность: $\forall a, b, c \in G \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$;
2. существование нейтрального элемента: $\exists e \in G \forall a \in G \quad (e \cdot a = a \cdot e = a)$ (часто называемого *единицей*);
3. существование обратного элемента: $\forall a \exists a^{-1} \in G \quad (a \cdot a^{-1} = a^{-1} \cdot a = e)$.

Иногда рассматривают другую сигнатуру $\sigma = \langle \cdot, {}^{-1}, e \rangle$.

[wikipedia: Группа](#)

Определение 2.

Алгебраическую систему $\mathcal{G} = \langle G, \cdot \rangle$ назовем *моноидом*, если на ней выполнены следующие аксиомы:

1. ассоциативность: $\forall a, b, c \in G \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$;
2. существование нейтрального элемента: $\exists e \in G \forall a \in G \quad (e \cdot a = a \cdot e = a)$;

[wikipedia: Моноид](#)

Определение 3.

Алгебраическую систему $\mathcal{G} = \langle G, \cdot \rangle$ назовем *полугруппой*, если на ней выполнена следующая аксиома:

1. ассоциативность: $\forall a, b, c \in G \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$;

[wikipedia: Полугруппа](#)

Примеры

Группы в математике: целые числа (\mathbb{Z}) по сложению, ненулевые рациональные числа ($\mathbb{Q} \setminus \{0\}$) по умножению. В др. областях: преобразования кубика Рубика.

Моноиды в математике: все группы, натуральные числа с нулем ($\mathbb{N} \cup \{0\}$) по сложению. В др. областях: списки с операцией конкатенации (нейтральный элемент — пустой список).

Полугруппы в математике: все группы и моноиды, натуральные числа без нуля (\mathbb{N}) по сложению, целые числа с операцией $\min(x_1, x_2)$ или $\max(x_1, x_2)$ (включение бесконечности превратит систему в моноид). В др. областях: непустые списки с операцией конкатенации.

Полугруппы и моноиды в Haskell

Data.Semigroup

Разберем сначала наиболее общую структуру — полугруппы [Data.Semigroup](#).

```
class Semigroup a where
    (<>) :: a -> a -> a
    sconcat :: NonEmpty a -> a
    stimes :: Integral b => b -> a -> a
```

```
infixr 6 (<>)
```

Минимальной базовой операцией является `(<>)`. Отметим, что если для какой-то структуры указано, что она является воплощением класса полугрупп, то Haskell не может проверить ассоциативность воплощения операции `(<>)`:

```
x <> (y <> z) == (x <> y) <> z
```

Это — обязанность программиста! Уникальная особенность Haskell, с которой мы будем далее встречаться ещё не один раз.

Кстати, отметим, что при этом в классе `Semigroup` операция `(<>)` вводится как правоассоциативная. Это означает, что при разборе выражения вида:

```
x <> y <> z
```

компилятор расставит скобки так:

```
x <> y <> z == x <> (y <> z)
```

Но а равенство (ассоциативность)

```
x <> (y <> z) == (x <> y) <> z
```

означает, что независимо от явного расположения скобок, результат вычисления обязан быть одинаковым. Это должен гарантировать программист, и этим могут пользоваться как разработчики, использующие такой код и, значит, использование ассоциативности может содержаться в модулях и в другом коде, предполагающем это свойство.

Функция `stimes` является многократным повторением операции (`<>`), а `sconcat` — конкатенацией (фактически, свёрткой) элементов непустого списка (за это отвечает особый полиморфный тип `NonEmpty` а из модуля [Data.List.NonEmpty](#)).

Рассмотрим пример полугруппы на булевой алгебре.

```
import Data.Semigroup
import Data.List.NonEmpty

instance Semigroup Bool where
  (<>) = (&&)
```

Тогда, в `ghci`:

```
*Main> True <> True
True
*Main> stimes 3 True
True
*Main> sconcat $ fromList [True, False, True]
False
```

(о проверке поговорим чуть позже...)

А следующее — полугруппа с операцией $\max(x_1, x_2)$. Причём, задаётся полиморфно, а не только на целых числах (для всех упорядоченных типов)!

Так как пример учебный, чтобы задать собственную реализацию, то необходимо сокрытие конструктора `Max` и деструктора `getMax` из модуля `Data.Semigroup`:

```
import Data.Semigroup hiding (Max, getMax)
import Data.List.NonEmpty

newtype Max a = Max { getMax :: a }
  deriving (Eq, Ord, Show)

instance Ord a => Semigroup (Max a) where
  Max x <> Max y = Max (x `max` y)
```

ВЫВОД:

```
*Main> Max 1 <> Max 2
Max {getMax = 2}
*Main> getMax $ Max 1 <> Max 2
```

2

```
*Main> sconcat $ fromList [Max 1, Max 2]
Max {getMax = 2}
```

```
*Main> sconcat $ fromList [Max True, Max False]
Max {getMax = True}
```

Отметим, что если бы нам была нужна только операция `<>`, то в импорте модуля

```
import Data.Semigroup
```

в современной версии `ghc` уже нет необходимости. Но для других методов класса это необходимо, иначе:

```
*Main> sconcat $ fromList [True]
```

```
<interactive>:1:1: error:
  * Variable not in scope: sconcat :: NonEmpty Bool -> t
  * Perhaps you meant one of these:
    `mconcat' (imported from Prelude), `concat' (imported from Prelude)
```

```
<interactive>:1:1: error:
  Variable not in scope: stimes :: Integer -> Bool -> t
```

Отдельно, про метод `fromList` из модуля [Data.List.NonEmpty](#), позволяющего работать с гарантированно непустыми списками:

```
*Main> fromList [True, False, True]
True :| [False, True]
*Main> fromList [False, True]
False :| [True]
*Main> fromList [True]
True :| []
*Main> fromList []
*** Exception: NonEmpty.fromList: empty list
```

[wiki.haskell: Data.Semigroup](#)

Соотношения между полугруппами и моноидами были не так давно изменены, ранее — это были два независимых класса, теперь у них подчиненное положение. Подробнее читать тут:

[Semigroup \(as superclass of\) Monoid Proposal](#)

[Proposal: Make Semigroup as a superclass of Monoid](#)

[Implement Semigroup as a superclass of Monoid Proposal \(Phase 1\)](#)

Data.Monoid

Класс типов `Monoid` определён в модуле [Data.Monoid](#). Вот его формальное определение:

```
class Semigroup a => Monoid a where
  mempty :: a
```

```
mappend :: a -> a -> a
mconcat :: [a] -> a
```

Так как с примерно 2015 года класс типов `Semigroup` стал суперклассом по отношению к классу типов `Monoid` (до этого они существовали сами по себе):

[Semigroup \(as superclass of\) Monoid Proposal](#)

[Compatible code for Semigroup Monoid Proposal](#)

[Make Semigroup a superclass of Monoid?](#)

то операция `mappend` становится излишней (осталась для некоторой совместимости), и определяется просто:

```
mappend :: Semigroup a => a -> a -> a
mappend = (<>)
```

Таким образом, минимально необходимой операцией для реализации воплощения класса становится `mempty`.

Операцию `mconcat` обычно определяют в этом же классе:

```
mconcat = foldr mappend mempty
```

И её целью является применение базовой операции `<>` между всеми элементами списка. Хотя, как указывают в документации, она может быть переопределена при необходимости для более эффективного представления.

Расширим наш пример на булевой алгебре до моноида:

```
import Data.Semigroup
import Data.Monoid
import Data.List.NonEmpty

instance Semigroup Bool where
    (<>) = (&&)

instance Monoid Bool where
    mempty = True

*Main> True <> mempty
True
*Main> True <> mempty <> False
False
*Main> True <> mempty `mappend` False
False
*Main> mconcat [True, mempty, False]
False
```

Но в старых компиляторах (напр., в 8.0.2) или при необходимости, мы можем создавать свои собственные воплощения `mappend`.

Пример с дуальным определением БА ...

```
import Data.Semigroup
import Data.Monoid
import Data.List.NonEmpty

instance Semigroup Bool where
  (<>) = (||)

instance Monoid Bool where
  mempty = False
```

Отдельно, удивила ситуация с пустыми списками:

```
*Main> mconcat []
()
*Main> sconcat []
<interactive>:3:9: error:
    * Couldn't match expected type `NonEmpty a' with actual type `[a0]'
    * In the first argument of `sconcat', namely `[]'
      In the expression: sconcat []
      In an equation for `it': it = sconcat []
    * Relevant bindings include it :: a (bound at <interactive>:3:1)
*Main> sconcat $ fromList []
()
*Main> fromList []
*** Exception: NonEmpty.fromList: empty list
*Main> sconcat (fromList [])
()
*Main> sconcat $! fromList []
*** Exception: NonEmpty.fromList: empty list
*Main> mconcat ([]::[Bool])
True
*Main> sconcat ([]::[Bool])
<interactive>:5:10: error:
    * Couldn't match expected type `GHC.Base.NonEmpty a'
*Main> sconcat $ fromList ([]::[Bool])
*** Exception: NonEmpty.fromList: empty list
```

Пока тут сложно понять, почему `sconcat $ fromList []` не выдает ошибку!

Кроме того, и это важно, — проверка законов моноидов для вновь созданных структур вновь ложится на программиста!

```
mempty <> a == a
a <> mempty == a
(a <> b) <> c == a <> (b <> c)
```

В нашем случае, это математически следует из того, что

```
True && x == x
x && True == x
```

и

```
(x && y) && z == x && (y && z)
```

для любых $x, y, z :: \text{Bool}$.

Теперь, когда более-менее разобрались с простыми примерами на булевой алгебре, рассмотрим более интересные ситуации.

Списки

```
instance Semigroup [a] where
    (<>) = (++)

instance Monoid [a] where
    mempty = []
    mconcat xss = [x | xs <- xss, x <- xs]
```

И в ghci:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]

ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```

Вообще, могли бы для списка сделать `mconcat = concat`, который, кстати, как раз `concat = foldr (++) []`

или не определять никак, а использовать реализацию класса. Но сделано именно так.

Законы моноидов для списков, очевидно, выполнены, проверять не будем.

Числа

Для чисел, как и для булевых алгебр, мы можем ввести моноид как для операции `+` с нулем в роли нейтрального элемента (единицы), т.е. `mempty`, так и для операции `*` с `1` в роли собственно единицы. Поэтому, поступили хитрее, в стиле Haskell, создали два новых полиморфных типа `Product` и `Sum` в контексте класса `Num`:

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Ord, Num, ...)

instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x * y)

instance Num a => Monoid (Product a) where
    mempty = Product 1
```

в реальности уже немного не так, см. [src](#)

```
(<>) = coerce ((* :: a -> a -> a)
```

где `coerce` взято из [Data.Coerce](#), и см. [Roles](#)

Работает это примерно так:

```
import Data.Monoid
```

```
test = Product 2 * Product 5  
test2 = Product 2 <> Product 5
```

Тестируем:

```
*Main> test  
Product {getProduct = 10}  
*Main> getProduct test  
10  
*Main> test2  
Product {getProduct = 10}  
*Main> getProduct test2  
10  
*Main> getProduct $ mempty <> Product 3  
3
```

Совершенно аналогичные определения теперь можно сделать и для сложения:

```
newtype Sum a = Sum { getSum :: a }  
    deriving (Eq,Ord,Num, ...)  
  
instance Num a => Semigroup (Sum a) where  
    Sum x <> Sum y = Sum (x + y)  
  
instance Num a => Monoid (Sum a) where  
    mempty = Sum 0
```

Вновь Булевы Алгебры

Для булевых алгебр у нас теперь есть решение в этом же стиле, например,

```
newtype All = All { getAll :: Bool }  
    deriving (Eq, ...)  
  
instance Semigroup All where  
    All x <> All y = All (x && y)  
    mempty = All True
```

для того, что мы делали ранее. И, аналогично,

```
newtype Any = Any { getAny :: Bool }  
  
    deriving (Eq, ...)  
  
instance Semigroup Any where  
    Any x <> Any y = Any (x || y)  
    mempty = Any False
```

Maybe

```
instance Semigroup a => Semigroup (Maybe a) where  
    Nothing <> b      = b  
    a      <> Nothing = a
```

```
Just a  <> Just b  = Just (a <> b)
```

```
instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

Endo

Эндоморфизм

Моноид эндоморфизмов. С точки зрения математики, эндоморфизм — это гомоморфизм какой-либо алгебраической системы в саму себя. В Haskell роль такой алгебраической системы несет параметр типа `a`, который может являться любой структурой (числами, булевой алгеброй, списками и т.п.), эндоморфизмами будут любые функции типа `a -> a`. Тожественная функция `id` будет выполнять роль единиц `mempty`, а композиция `.` будет выполнять роль моноидной операции `<>`.

```
newtype Endo a = Endo { appEndo :: a -> a }
    deriving (...)
```

```
instance Semigroup (Endo a) where
    Endo f <> Endo g = Endo (f . g)
```

```
instance Monoid (Endo a) where
    mempty = Endo id
```

В качестве примера можно рассмотреть применение списка функций для построения их общей композиции:

```
import Data.Monoid

f :: Endo Int
f = mconcat $ map Endo [(+1), (*2), negate]

main = print $ f `appEndo` 5
```

что при выполнении в `ghci` даёт:

```
*Main> main
-9
```

Сам пример очень простой и может быть понят следующим образом. Применение `map` `map Endo [(+1), (*2), negate]`

даёт список «тэгизированных» одноместных функций:

```
[Endo (+1), Endo (*2), Endo negate]
```

Применение к ним `mconcat` даёт «тэгизированную» композицию:

```
f === Endo ( (+1) . (*2) . negate )
```

А в основной программе деструктор `appEndo` фактически сначала освобождает функцию `f` от тэга и применяет к аргументу `5`. Более понятно это выглядит так:

```
(appEndo f) 5
```


ну или для другой ситуации

```
(appEndo $ Endo (+8)) 1
```

[Monoids Tour](#)

[Haskell Monoids and their Uses](#)

[Monoids in Haskell](#)

[От моноидов к алгебрам де Моргана. Строим абстракции на Haskell](#)

[Моноиды, полугруппы и все-все-все](#)

[Учим поросёнка на моноидах верить в себя и летать](#)

[Великая сила newtypes](#)

Data.Group

Отметим, что у группы тоже есть пакет [groups](#) с модулем [Data.Group](#), который описывает функциональность групп в математическом смысле. Операции (`<>`) и `mempty` наследуются, таким образом, необходимой минимальной операцией будет `invert`

```
class Monoid m => Group m where
    invert :: m -> m
    (~~) :: m -> m -> m -- infixl 7
    pow :: Integral x => m -> x -> m
```

где (`~~`) означает вычитание (или деление)

```
x ~~ y == x <> invert y
```

а `pow` означает многократное возведение в степень (при отрицательной степени применяем `invert`, т.е. обращаем результат).

Опять же, при создании воплощения данного класса, проверка законов группы, а в случае Haskell, функциональности и законов, определяемых модулями `Data.Semigroup` и `Data.Monoid`, и новыми соотношениями

```
a <> invert a == mempty
invert a <> a == mempty
```

лежит на программисте!

Foldable

Уже несколько лет назад свёртки на списках реализованы немного не так, как мы изучали на лекциях и делали в упражнениях. Вместо «узких» специализированных на списках функций `foldr`, `foldl` и т.п. эти функции работают на структурах (типах), реализующих полиморфный класс `Foldable a` — т.е. таких типах данных, которые можно свернуть. Например, помимо списков — это различные деревья и другие коллекции и типы данных (напр., `Text`).

[Data.Foldable](#)

Класс предоставляет несколько функций, две из которых являются минимально необходимыми. Одна из них — знакомая нам правая свёртка **foldr**. Иными словами, если мы на нужной нам структуре определяем функцию **foldr**, то все другие функции — будут. Но есть и другая интересная функция, имеющая отношение как раз к теме моноидов, что мы только что прошли:

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

Эта функция принимает два аргумента: второй **t a** — это значение какой-то нужной нам структуры: список **[a]**, дерево **Tree a**, значение типа **Maybe a** и т.п.

Первый аргумент **a -> m** — это функция, которая отображает базовые значения нашей структуры (например, элементы списка или значения в вершинах дерева) в какой-то моноидный тип: числа, булеву алгебру, списки и т.п.

Сама функция **foldMap** сначала работает как **map**, применяя первый аргумент-функцию, а потом как **fold**, сворачивая результат с помощью (**<>**) (**mapend**) в том моноидном типе, куда сделано отображение с помощью функции из первого аргумента.

Рассмотрим как это работает на примере. В лекции 8 мы познакомились с типом двоичных деревьев, например таким:

```
data ATree a = Leaf a | Branch (Tree a) a (Tree a)
    deriving (Eq, Ord, Show, Read)
```

и создали некоторый пример:

```
y = ABranch (ALeaf 1) 2 (ALeaf 3)
```

Давайте создадим пример чуть сложнее:

```
tree = ABranch
    (ABranch (ALeaf 1) 2 (ALeaf 3))
    4
    (
        ABranch (ALeaf 5)
        6
        (ABranch (ALeaf 7) 8 (ALeaf 9))
    )
```

Но для «красоты» (точнее, чтобы проиллюстрировать использование отображения в **empty**) вместо листьев **ALeaf** будем использовать пустые поддеревья **Empty**:

```
import Data.Monoid
```

```
data ATree a = Empty |
    ABranch (ATree a) a (ATree a)
    deriving (Eq, Ord, Show, Read)
```

```
tree = ABranch
    (
        ABranch
            (ABranch Empty 1 Empty)
            2
            (ABranch Empty 3 Empty)
    )
```

```

    )
      4
    (
      ABranch (ABranch Empty 5 Empty)
        6
      (ABranch
        (ABranch Empty 7 Empty)
          8
        (ABranch Empty 9 Empty)
      )
    )
  )

```

```
instance Foldable ATree where
```

```

  foldMap f Empty = mempty
  -- foldMap f (ALeaf x) = f x
  foldMap f (ABranch l x r) =
    foldMap f l <> f x <> foldMap f r

```

```

test1 = getAny $ foldMap (\x -> Any $ x == 3) tree
test2 = getAll $ foldMap (All . (> 3)) tree
test3 = getSum $ foldMap (Sum) tree
test4 = getProduct $ foldMap (Product) tree
test5 = foldMap (\x -> [x]) tree
test6 = foldMap (\x -> Just [x]) tree

```

```
*Main> test1
```

```
True
```

```
*Main> test2
```

```
False
```

```
*Main> test3
```

```
45
```

```
*Main> test4
```

```
362880
```

```
*Main> test5
```

```
[1,2,3,4,5,6,7,8,9]
```

```
*Main> test6
```

```
Just [1,2,3,4,5,6,7,8,9]
```

(Липовача, с.355–359)