Структуры данных в Haskell (дополнение)

Тип данных `Map` и его возможности

В различных приложениях — мобильных, веб, на десктопах одна из наиболее часто встречающихся ситуаций, когда надо передать или рассмотреть конфигурацию (собственно, файлы конфигураций), словари терминов, списки цен на товары, телефонные книги и т.п. Для их поддержки существует ряд очень мощных форматов, как текстовых, так и бинарных: ini-файлы, JSON, TOML, YAML, XML, BSON, CBOR, GOB, MessagePack, Google Protocol Buffers — хотя часто их избыточная мощность используется лишь для описания и передачи множества пар (величина: её значение).

А как эти множества представляются внутри языков программирования, в частности, Haskell?

Ассоциативные списки (или в других языках: ассоциативные массивы, отображения, словари, иногда хэши или хэш-таблицы). Такие структуры данных, которые позволяют хранить пары вида «(*ключ*, *значение*)» и поддерживают операции добавления пары, а также поиска и удаления пары по ключу. Позволяют, таким образом, создавать простые словари, справочники и т.п.

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

wikipedia: Ассоциативный массив

Различные реализации могут требовать упорядоченность по ключу или не использовать её. Но требуется возможность сравнивать по равенству — в терминах Haskell, для типа, реализующего ключ, требуется быть воплощением класса **`Eq`**.

В Haskell функциональность отображений вынесена в отдельный модуль Data.Map, составляющий часть базового пакета containers, стандартно входящего в `Haskell Platform`. Но так как имена некоторых функций из этого модуля совпадают с именами функций из **`Prelude`**, то рекомендуется импортировать модуль *квалифицированно*, а чтобы потом сократить текст, то дать ему «псевдоним»:

```
import qualified Data.Map as M
```

Отображение из ключей типа `k` в значения типа `a` (`k` и `a` — это переменные для типов) задаются следующей декларацией, особенности которой скрыты в модуле Data.Map:

```
data Map k a
```

Модуль скрывает особенности реализации как «настоящий» АДТ и не экспортирует конструкторы данных. Вместо этого предоставляются удобные функции (API) для создания и манипуляциям с ассоциативными списками. Таким образом, авторы оставляют для себя возможность свободно менять особенности внутренней реализации. В настоящий момент такая реализация выполнена на основе *сбалансированных бинарных деревьев*. К слову, в Haskell можно было легко сделать наивную реализацию ассоциативных списков на основе списков пар «(ключ, значение)»:

```
phoneBook =
  [("Оля",  "555-29-38")
  ,("Женя", "452-29-28")
```

```
  ,(”Катя”, ”493-29-28”)
  ,(”Маша”, ”205-29-28”)
  ,(”Надя”, ”939-82-82”)
  ,(”Юля”,  ”853-24-92”)
  ]
```

Липовача, С.136

Только в этом случае, программисту придётся самому реализовывать необходимый функционал. Ну и эффективность будет существенно ниже стандартной реализации.

Кстати, как отмечает (Липовача, С.138), значения ключей должны не только уметь сравниваться (иметь экземпляр класса типов **Eq**), но и должна быть

> возможность их упорядочить (класс типов **Ord**). Это существенное ограничение модуля `Data.Map`. Упорядочиваемые ключи нужны ему для того, чтобы размещать данные более эффективно.

Рассмотрим возможности, которые предоставляются стандартным модулем `Data.Map`.

Создание отображений

Создание пустого ассоциативного списка:

```
empty :: Map k a
```

Создание отображения с единственной парой ключа и значения:

```
singleton :: k -> a -> Map k a
```

Пример:

```
dict = Data.Map.singleton ”привет” ”hello”
```

Создание отображений из списка пар «(ключ, значение)»:

```
fromList :: Ord k => [(k, a)] -> Map k a
```

(отметим требование к поддержке ключами возможности сравнений по типу линейного порядка **Ord**).

Если список содержит более чем одно значение для одного и того же ключа, сохраняется только последнее значение из передаваемых:

```
Prelude Data.Map> Data.Map.fromList [(”peter”,234),(”peter”,8)]
fromList [(”peter”,8)]
```

Пример создания отображения из списка пар, определённого выше:

```
phBk = Data.Map.fromList phoneBook
```

Более быстрая конвертация из списка будет, если использовать список, заранее упорядоченный по ключам, и функцию `fromAscList`:

```
fromAscList :: Eq k => [(k, a)] -> Map k a
```

При этом, упорядоченность не проверятся (ответственность лежит на программисте)

```
fromAscList [(3,'b'), (5,'a')]          == fromList [(3,'b'), (5,'a')]
fromAscList [(3,'b'), (5,'a'), (5,'b')] == fromList [(3,'b'), (5,'b')]
```

Но особенность реализации приводит к монстрам вида:

```
fromAscList [(5,'a'), (3,'b'), (5,'b')] ==
   fromList [(5,'a'), (3,'b'), (5,'b')]
```

:) :) :o)

В таких случаях, можно делать проверку:

```
valid (fromAscList [(3,'b'), (5,'a'), (5,'b')]) == True
valid (fromAscList [(5,'a'), (3,'b'), (5,'b')]) == False
```

Обратно привести ассоциативный список к списку пар можно функциями:

```
toList :: Map k a -> [(k, a)]
toList (fromList [(5,'a'), (3,'b')]) == [(3,'b'), (5,'a')]
toList empty == []

toAscList :: Map k a -> [(k, a)]
toAscList (fromList [(5,'a'), (3,'b')]) == [(3,'b'), (5,'a')
```

при этом они синонимы: `toList = toAscList`.

Вставка-удаление

Напоминание: *отображения, как и все ранее изученные структуры — иммутабельны, поэтому, вставки-удаления не меняют текущую структуру (объект)!* Но, как утверждается, всё-таки вставки делаются по мере возможности максимально эффективно:

Is the whole Map copied when a new binding is inserted?

Haskell Immutable data structure - the Map data type

Вставка-удаление осуществляются для пар

```
insert :: Ord k => k -> a -> Map k a -> Map k a

insert 5 'x' (fromList [(5,'a'), (3,'b')])
  == fromList [(3,'b'), (5,'x')]
insert 7 'x' (fromList [(5,'a'), (3,'b')])
  == fromList [(3,'b'), (5,'a'), (7,'x')]
insert 5 'x' empty
  == singleton 5 'x'
```

Если пара для данного ключа уже была, то она заменяется на пару с новым значением.

Удаление осуществляется тоже для пар (хотя передаём только ключ), при этом, если удаляемой пары не было, то возвращается исходный ассоциативный список

```
delete :: Ord k => k -> Map k a -> Map k a

delete 5 (fromList [(5,'a'), (3,'b')])
  == singleton 3 'b'
```

```
delete 7 (fromList [(5,'a'), (3,'b')])
  == fromList [(3,'b'), (5,'a)]
delete 5 empty
  == empty
```

Проверка наличия ключа в ассоциативном списке

Требуется узнать, содержится ли в ассоциативном списке ключ, независимо от ассоциированного с ним значения.

```
member :: Ord k => k -> Map k a -> Bool

member 5 (fromList [(5,'a'), (3,'b')]) == True
member 1 (fromList [(5,'a'), (3,'b')]) == False
```

Поиск по ключу

Функция **lookup** принимает ключ и «ассоциативный список» и пытается найти соответствующее ключу значение. Если всё прошло удачно, возвращается обёрнутое в **Just** значение; в противном случае — **Nothing**:

```
lookup :: Ord k => k -> Map k a -> Maybe a
```

Так как в **Prelude** уже есть функция **lookup** со схожей функциональностью:

```
Prelude> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

которая работает со обычными списками «(ключ, значение)», то необходимо либо импортировать модуль `Data.Map` квалифицировано:

```
import qualified Data.Map
```

либо делать сокрытие функции **lookup** из **Prelude**:

```
import Prelude hiding (lookup)
import Data.Map
```

После этого можно попробовать поиск:

```
import Prelude hiding (lookup)
import Data.Map

listOfBalance =
  [(”Оля”,  234)
  ,(”Женя”, 652)
  ,(”Катя”, 4)
  ,(”Маша”, 25)
  ,(”Надя”, 41)
  ,(”Юля”,  302)
  ] :: [(String,Int)]

balance = fromList listOfBalance
testLookUp = lookup ”Юля” balance
```

```
*Main> testLookUp
Just 302

*Main> lookup ”Вася” balance
Nothing
```

Ну или можно использовать функцию (!?), которая делает то же самое, но с иным порядком аргументов:

```
*Main> balance !? ”Юля”
Just 302


*Main> balance !? ”Вася”
Nothing
```

или функцию (!), которая не делает оборачивания контейнером **Maybe** и возвращает ошибку в случае отсутствия ключа:

```
*Main> balance ! ”Юля”
302

*Main> balance ! ”Вася”
*** Exception: Map.!: given key is not an element in the map
CallStack (from HasCallStack):
  error, called at
```

Ряд полезных утилит

Проверка, является ли отображение пустым:

```
null :: Map k a -> Bool

Data.Map.null (empty)           == True
Data.Map.null (singleton 1 'a') == False
```

Размер, т.е. число пар:

```
size :: Map k a -> Int

size empty                                   == 0
size (singleton 1 'a')                       == 1
size (fromList([(1,'a'), (2,'c'), (3,'b')])) == 3
```

Слияние двух ассоциативных списков (одинакового типа):

```
union :: Ord k => Map k a -> Map k a -> Map k a

union (fromList [(5,”a”), (3,”b”)]) (fromList [(5,”A”), (7,”C”)]) ==
  fromList [(3,”b”), (5,”a”), (7,”C”)]
```

при дублях ключей предпочтение отдаётся встреченному в первому отображении.

Также есть утилиты для пересечения и разности.

Обходы или `map` для `Map`

Для значений в парах внесём какое-нибудь изменение, например, добавим суффикс «x»:

```
map :: (a -> b) -> Map k a -> Map k b

map (++ ”x”) (fromList [(5,”a”), (3,”b”)])
  == fromList [(3,”bx”), (5,”ax”)]
```

или мы захотим пополнить баланс для всех участников в примере выше:

```
tmap = map (*10) balance
```

Но функцию **`map`** тоже надо импортировать, скрывая аналогичную в **`Prelude`**, или делая импорт квалифицировано. Или использовать функтор `fmap`, который в данном случае делает то же самое и не требует дополнительных усилий по сокрытию.

Свёртки

Для отображений определены свёртки **`foldr`** и **`foldl`**:

```
foldr :: (a -> b -> b) -> b -> Map k a -> b
foldl :: (a -> b -> a) -> a -> Map k b -> a
```

для которых действуют такие же ограничения при импортировании, как и для **`map`** (к сожалению, свёртки не реализованы как прозрачные воплощения класса Foldable, что мы наблюдали в Data.Tree или в Data.Array).

Вот пример, если мы хотим посмотреть общий баланс для всех участников из примера выше:

```
import qualified Data.Map as M

listOfBalance = ...
balance = M.fromList listOfBalance

totalsum = M.foldr (+) 0 balance
```

Для свёрток также определены строгие аналоги **`foldl'`** и **`foldr'`**, полезные в некоторых случаях.

Фильтрация

Как и для списков, для ассоциативных списков есть функция фильтрации с тем же названием **`filter`**. Аналогично, в качестве аргумента она берёт предикат и ассоциативный список, а в качестве результат возвращает ассоциативный список, отфильтрованный на значениях по предикату:

```
filter :: (a -> Bool) -> Map k a -> Map k a

filter (> ”a”) (fromList [(5,”a”), (3,”b”)]) == singleton 3 ”b”
filter (> ”x”) (fromList [(5,”a”), (3,”b”)]) == empty
filter (< ”a”) (fromList [(5,”a”), (3,”b”)]) == empty
```

Аналогично, необходимо сокрытие одноимённой функции из **`Prelude`**.

функции …WithKey

Для функций **map**, **foldl**, **foldr**, **filter** есть функции-двойняшки, которые в качестве аргумента кроме значений принимают и ключи, и соответственно могут их обрабатывать:

```
mapWithKey :: (k -> a -> b) -> Map k a -> Map k b
foldrWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b
foldlWithKey :: (a -> k -> b -> a) -> a -> Map k b -> a
filterWithKey :: (k -> a -> Bool) -> Map k a -> Map k a
```

пара примеров из документации:

```
f key x = (show key) ++ ":" ++ x
mapWithKey f (fromList [(5,"a"), (3,"b")]) ==
  fromList [(3, "3:b"), (5, "5:a")]

filterWithKey (\k _ -> k > 4) (fromList [(5,"a"), (3,"b")])
  == singleton 5 "a"
```

Множество ключей и множество значений

Последние из значимых функций в пакете.

```
elems :: Map k a -> [a]

elems (fromList [(5,"a"), (3,"b")]) == ["b","a"]
elems empty == []

keys :: Map k a -> [k]

keys (fromList [(5,"a"), (3,"b")]) == [3,5]
keys empty == []
```

Сам модуль немного производит ощущение архаичной организации (см. функции `fold`) в сравнении с аналогичными модулями ниже, но включен в дефолтную поставку.

Ссылки по теме

Data.Map

An example of using Data.Map in Haskell

containers: Maps, Sets, and more

Работа с Haskell-типами (есть немного про работу с JSON)

containers - Introduction and Tutorial: Maps

HashMap

В случае, когда нам не нужна упорядоченность ключей (скорее всего, это большинство ситуаций по использованию ассоциированных списков), можно рассмотреть более эффективную реализацию, предлагаемую пакетом hashmap, основанную на хэш-функциях.

Модуль Data.HashMap предлагает функциональность, схожую с предоставляемой модулем Data.Map. Но, в отличие от пакета containers, пакет hashmap надо ставить самостоятельно:

```
cabal update
...
cabal install hashmap
```

Большинство функций имеют те же самые имена, как и их аналоги в модуле Data.Map, и импорт надо осуществлять примерно аналогично:

```
import qualified Data.HashMap as H

listOfBalance =
  [("Оля",  234)
  ,("Женя", 652)
  ,("Катя", 4)
  ,("Маша", 25)
  ,("Надя", 41)
  ,("Юля",  302)
  ] :: [(String,Int)]

balance = H.fromList listOfBalance

testLookUp = H.lookup "Юля" balance
tmap = H.map (*10) balance

totalsum = H.fold (+) 0 balance
```

Функции **foldl** и **foldr** реализованы как воплощения методов класса Foldable, и есть собственная функция fold (примерно как в Data.Tree, но здесь своя функция fold определена просто с помощью композиции **foldr** и **elems**). Отсутствуют функции fromAscList и valid. А те функции, которые возвращали упорядоченные списки (как toList) в Data.HashMap возвращают списки в произвольном порядке.

Benchmarks for dictionary data structures: hash tables, maps, tries, etc

Lookup Int (Randomized)

Name	10	100	1000	10000	100000	1000000
Data.Map.Lazy	113.9 ns	1.697 µs	67.91 µs	1225 µs	21.89 ms	556.1 ms
Data.Map.Strict	113.5 ns	1.782 µs	67.75 µs	1258 µs	21.68 ms	543.4 ms
Data.HashMap.Lazy	156.9 ns	2.151 µs	31.06 µs	531.4 µs	17.10 ms	339.4 ms
Data.HashMap.Strict	166.1 ns	2.157 µs	27.52 µs	517.6 µs	16.62 ms	335.9 ms
Data.IntMap.Lazy	138.7 ns	1.830 µs	71.38 µs	1203 µs	24.10 ms	649.5 ms
Data.IntMap.Strict	138.6 ns	1.835 µs	70.28 µs	1192 µs	25.18 ms	629.8 ms

Benchmarks for dictionary data structures: hash tables, maps, tries, etc

В настоящий момент вместо пакета hashmap рекомендовано использование более современного пакета unordered-containers

Ссылки по теме

Data.HashMap

unordered-containers

hashmap

Тип данных `Data.Sequence` и его возможности

В связи с массивами, мы уже обсуждали неэффективность коллекций **`List`** для различных практических задач программирования (и в следующих темах также будем обсуждать эти проблемы на примере работы с текстовыми данными).

Одним из промежуточных решений было представление конечных последовательностей с помощью особого типа хорошо сбалансированных деревьев («пальчиковых деревьев»):

Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure

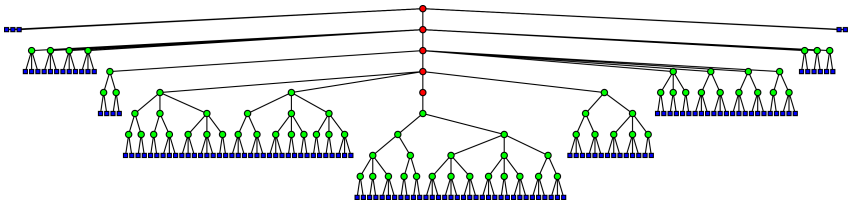

Рис. 1: Пальчиковые деревья

Это идея реализована в модуле Data.Sequence, входящем в базовый пакет containers (как и ранее рассмотренный модуль `Data.Map`). Последовательности, таким образом, за счёт внутренней организации, обеспечивают примерно логарифмическое время доступа к отдельному элементу (точнее, $O(\log(in(i, n-i))$ для i-го элемента в последовательности длины n). И подобно спискам, обеспечивается константное время добавление элемента (но в оба края) и довольно быстрое (логарифмическое) время слияние последовательностей, вставку элемента и разбиение на подпоследовательности.

Для внешнего использования модуля используется API, в целом напоминающий таковой функционал списков. Многие функции в этом модуле имеют те же имена, что и функции в **`Prelude`** или `Data.List`. Почти во всех случаях эти функции ведут себя аналогично. Например, **`filter`** фильтрует последовательность точно так же, как фильтрует список.

Но в то время как списки могут быть конечными или бесконечными, последовательности всегда конечны. В результате последовательность становится строгой по длине. Игнорируя эффективность, можно представить, что `Seq` определяется таким образом:

```
data Seq a = Empty | a :<| !(Seq a)
```

(восклицательный знак обозначает строгость конструктора, а конструктор `:<|` является аналогом конструктора `:` для списков)

Таким образом, если для списков такое поведение было вполне нормальным:

```
(1 : undefined) !! 0 = 1
```

то для последовательностей аналогичный фокус не проходит:

```
(1 :<| undefined) `index` 0 = undefined
```

Создание последовательностей

Создание пустой последовательности:

```
empty :: Seq a
```

Создание последовательности с единственным значением:

```
singleton :: a -> Seq a
```

Пример:

```
Prelude Data.Sequence> singleton 9.1
fromList [9.1]
```

Создание последовательности из списка значений:

```
fromList :: [a] -> Seq a
```

Пример:

```
Prelude Data.Sequence> fromList [True,True,False]
fromList [True,True,False]
Prelude Data.Sequence> :t fromList [True,True,False]
fromList [True,True,False] :: Seq Bool
```

Создание последовательности из значения размера последовательности (её длины) и данной функции (на целых числах **Int** от нуля):

```
fromFunction :: Int -> (Int -> a) -> Seq a
```

```
Prelude Data.Sequence> fromFunction 4 (\x -> x^2)
fromList [0,1,4,9]
Prelude Data.Sequence> :t (fromFunction 4 (\x -> x^2))
(fromFunction 4 (\x -> x^2)) :: Seq Int
```

Обработка последовательностей

Добавление элемента слева функцией:

```
(<|) :: a -> Seq a -> Seq a   (infixr 5)
```

конструктором (на самом деле «синонимом паттерна»)

```
(:<|) :: a -> Seq a -> Seq a   (infixr 5)
```

Pattern synonyms

Pattern Synonyms

Пример:

```
Prelude Data.Sequence> 3 <| 2 <| singleton 1
fromList [3,2,1]
Prelude Data.Sequence> :t (3 <| 2 <| singleton 1)
(3 <| 2 <| singleton 1) :: Num a => Seq a
```

Можно смешивать стили:

```
Prelude Data.Sequence> 3 :<| 2 :<| 1 :<| Empty
fromList [3,2,1]
Prelude Data.Sequence> 3 :<| 2 <| 1 :<| Empty
fromList [3,2,1]
Prelude Data.Sequence> 3 :<| 2 <| 1 :<| empty
fromList [3,2,1]
```

Добавление элемента справа:

```
(|>)  :: Seq a -> a -> Seq a   (infixl 5)
(:|>) :: Seq a -> a -> Seq a   (infixl 5)
```

Пример:

```
Prelude Data.Sequence> singleton 1 |> 2 |> 3
fromList [1,2,3]
```

Конкатенация двух последовательностей:

```
(><) :: Seq a -> Seq a -> Seq a    (infixr 5)
```

Требует время $O(\log(\min(n_1, n_2)))$.

Пример работы (необходима аккуратность в совместном использовании операторов `><`, `|>`, `<|` — лучше поставить скобки в сложных местах)

```
Prelude Data.Sequence> (singleton 1 |> 2 |> 3) >< 4 <| 5 <| singleton 6
fromList [1,2,3,4,5,6]
```

Индексация элементов

```
index :: Seq a -> Int -> a
```

(есть ещё функции **lookup** и `!?`)

Изменение элемента:

```
update :: Int -> a -> Seq a -> Seq a
```

Требует времени порядка $O(\log(\min(i, n-i)))$. Заменяет элемент на данной позиции. Если позиция за пределами индекса, то возвращается оригинальная последовательность.

Примеры:

```
Prelude Data.Sequence> update 0 10  $ fromList [1..5]
fromList [10,2,3,4,5]
Prelude Data.Sequence> update 20 10  $ fromList [1..5]
fromList [1,2,3,4,5]
```

Паттерн-матчинг:

```
getFirst3 :: Seq a -> Maybe (a,a,a)
getFirst3 (x1 :<| x2 :<| x3 :<| _xs) = Just (x1,x2,x3)
getFirst3 _ = Nothing
> getFirst3 (fromList [1,2,3,4]) = Just (1,2,3)
> getFirst3 (fromList [1,2]) = Nothing
```

Обходы и свёртки последовательностей

Так как **map** традиционно определен для списков, используется перегружаемая версия fmap:

```
Prelude Data.Sequence> fmap (\x -> x^2) (fromList [1..5])
fromList [1,4,9,16,25]
```

или вариант **map** с обработкой индекса:

```
mapWithIndex :: (Int -> a -> b) -> Seq a -> Seq b
```

пример:

```
Prelude Data.Sequence> mapWithIndex (\x y -> y:(show x)) (fromList ”hello”)
fromList [”h0”,”e1”,”l2”,”l3”,”o4”]
```

Работают свёртки (**foldl**, **foldr**, **foldl1**, **foldr1**, но не **foldl'**) в обычных версиях:

```
Prelude Data.Sequence> foldl1 (+) (fromList [1..5])
15
```

Фильтры (но необходимо либо вызывать квалифицированно, либо «прятать» версию из **Prelude**):

```
Prelude Data.Sequence> Data.Sequence.filter (>3) (fromList [1..5])
fromList [4,5]
```

Также доступны версии функций (**elem**, **null**, **sort**, **reverse**), схожих по обработке списков…

(to be continued… =)

Просмотр крайнего левого и правого элементов

Особенностью пакета является «специфичные» представления крайних элементов с помощью отдельных типов данных со своими конструкторами:

```
viewl :: Seq a -> ViewL a
viewr :: Seq a -> ViewR a
```

где `ViewL a` с конструкторами `EmptyL` и `a :< (Seq a)` используется для представления левой границы, а `ViewR a` с конструкторами `EmptyR` и `(Seq a) :> a` — для представления правой границы.

Пример использования:

```
Prelude Data.Sequence> viewl empty
EmptyL
Prelude Data.Sequence> viewl $ singleton 1
1 :< fromList []
Prelude Data.Sequence> viewl $ fromList [1..5]
1 :< fromList [2,3,4,5]
Prelude Data.Sequence> viewr $ fromList [1..5]
fromList [1,2,3,4] :> 5
```

Таким образом, если нам хочется написать аналог **head** и **last**, это будет примерно так:

```
Prelude Data.Sequence> headme' (x :< _ ) = x
Prelude Data.Sequence> headme = headme' . viewl
Prelude Data.Sequence> headme $ fromList [1..5]
1
```

Ссылки по теме

Data.Sequence

containers - Introduction and Tutorial: Sequences

Тип данных `Vector` (часть-1)

Мы уже подробно разбирали в предыдущей лекции возможности массивов и тип данных `Data.Array`. Кратко рассмотрим теперь более новый и устойчиво развивающийся пакет vector. Вполне возможно, что в будущих лекциях, при детальном изучении изменяемых состояний, мы вновь вернёмся к нему. Пакет и предоставляемые им типы данных (такие как `Data.Vector`, `Data.Vector.Unboxed`, `Data.Vector.Storable`, `Data.Vector.Mutable` и т.п.) в целом дают те же самые возможности, что и ранее рассмотренные массивы, а при «поверхностном взгляде», и списки. В большей части, эти возможности обеспечиваются функциями с тем же самыми именами.

Несмотря на все позитивные моменты (актуальность и постоянная поддержка, производительность, поддержка различных форм изменений и т.п.), пакет не включён в дефолтную поставку и вам нужно его устанавливать самостоятельно:

```
cabal update

cabal install --lib vector
```

Далее, чтобы не смешивать имена импортируемых функций с таковыми в **Prelude**, предлагается импортировать необходимые модули квалифицированно:

```
import qualified Data.Vector as V
```

Вот, по большей части, хорошо знакомый нам набор функций:

```
import qualified Data.Vector as V

e1 = V.empty
s1 = V.singleton True
list = [1..10] :: [Int]
v1 = V.fromList list
list2 = V.toList v1
v2 = V.filter odd v1
v3 = V.map (*2) v1
v3' = fmap (*2) v1
v3'' = V.imap (+) v3
v4 = V.reverse v1
v5 = V.head v1
v6 = V.tail v1
```

```
sm = V.foldr1 (+) v1
sm' = V.foldl' (+) 0 v1
r = V.length v1
v7 = V.slice 4 3 v1
```

Возможности индексации беднее, чем у `Data.Array`, индексы берутся от 0 с типом `Int`:

```
> v6 V.! 0
2
> v6 V.! (1::Integer)
>:3:9: error:
    * Couldn't match expected type `Int' with actual type `Integer'
> v6 V.!? 0
Just 2
> v6 V.!? 9
Nothing
> v6 V.! 9
*** Exception: index out of bounds (9,9)
```

Но зато, нам не нужны функции, подобные `range`, `index`, `inRange`, `bounds`, `elems` из `Data.Array`.

Есть возможность делать эффективные срезы функцией `slice` — эффективность достигается за счёт иммутабельности, и данные при этом не копируются!

Можно добавлять элементы в начало и в конец векторов функциями `cons` и `snoc`, склеивать два вектора:

```
> v1 V.++ v3
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20]
> v1 <> v3
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20]
```

И можно делать обновления векторов в стиле массивов с помощью функции `(//)`:

```
V.fromList [5,9,2,7] V.// [(2,1),(0,3),(2,8)] == [3,9,8,7]
```

Векторы, как было отмечено выше, бывают разных типов. Изменяемые мы рассмотрим позже, а сейчас отметим такие как Data.Vector.Unboxed — неупакованные, и Data.Vector.Storable — сохраняемые. Сохраняемые будут полезны при передаче и приёме данных с подключаемыми внешними Си-библиотеками, они содержатся в области памяти, сохраняемой от действий «сборщика мусора» (garbage collector) и таким образом, ситуация может быть подвержена фрагментации памяти. Нам пока нет нужды их использовать. Неупакованные векторы рекомендуется использовать для работы с примитивными типами, такими как `Bool`, `Char`, `Double`, `Float`, `Int`, `Int8` (16, 32, 64), `Word`, `Word8` (16, 32, 64). В этом случае, мы значительно сокращаем издержки на обработку

```
import qualified Data.Vector.Unboxed as U

list = [1..10] :: [Int]
v1 = U.fromList list
```

В заключение стоит добавить, что существует пакет vector-algorithms, эффективно реализующий для описанного типа данных ряд полезных алгоритмов, в основном сортировок.

vector: Efficient Packed-Memory Data Representations

Numeric Haskell: A Vector Tutorial

MONDAY MORNING HASKELL. Vectors

Haskell: Lists, Arrays, Vectors, Sequences