

Курс: Функциональное программирование

Практика 9. Трaversы, парсеры, моноидальные функторы

Разминка

► Устно вычислите значения выражений и проверьте результат в GHCi. Полезно во всех заданиях сначала ответить на вопрос, какой аппликативный функтор здесь используется.

```
sequenceA [Right 3,Right 4,Right 5]

sequenceA [Right 3,Left 4,Right 5]

sequenceA [Left 3,Left 4,Right 5]

sequenceA [Right 3,Left 4,undefined]

sequenceA [undefined,Left 4,Right 5]

traverse (\x -> if odd x then Right x else Left x) [1,3,5,7]

traverse (\x -> if odd x then Right x else Left x) [1,2,6,7]

sequenceA [(+3),(+2),(+1)] 3

traverse (+) [1,2,3] 5

traverse (\x -> (show x,x)) [1,2,3]

sequenceA [[1,2,3],[4,5,6]]

sequenceA [[1,2],[3,4],[5,6]]

(getZipList . sequenceA . map ZipList) [[1,2,3],[4,5,6]]
```

Класс Traversable

- (Stepik, 1 балл) Сделайте тип данных `Result` представителем класса типов `Traversable`.

```
data Result a = Ok a | Error String  
deriving (Eq,Show)
```

```
GHCi> traverse (\x->[x+2,x-2]) (Ok 5)  
[Ok 7,Ok 3]  
GHCi> traverse (\x->[x+2,x-2]) (Error "!!!!")  
[Error "!!!!"]
```

- (Stepik, 1 балл) Сделайте тип данных непустого списка `NEList` представителем класса типов `Traversable`. Выполните реализацию представителя через функцию `sequenceA`.

```
data NEList a = Single a | Cons a (NEList a)  
deriving (Eq,Show)
```

```
GHCi> sequenceA $ Cons (Right 3) (Cons (Right 4) (Single (Right 5)))  
Right (Cons 3 (Cons 4 (Single 5)))  
GHCi> sequenceA $ Cons (Right 3) (Cons (Left 4) (Single (Right 5)))  
Left 4  
GHCi> traverse (\x->[x+2,x-2]) (Cons 20 (Single 30))  
[Cons 22 (Single 32),Cons 22 (Single 28),Cons 18 (Single 32),Cons 18 (Single 28)]
```

Функтор `Const` и реализации методов `Foldable` по умолчанию

- Рассмотрим фантомный тип (phantom type)

```
newtype Const c a = Const { getConst :: c }  
deriving ( Eq, Show)
```

Контейнер, который не содержит ни одного элемента (типа `a`).

```
instance Functor (Const c) where
    fmap :: (a -> b) -> Const c a -> Const c b
    fmap _ (Const v) = Const v
```

Этот функтор перепаковывает то же значение в другой тип:

```
GHCI> Const 'z'
Const 'z'
GHCI> :t Const 'z'
Const 'z' :: Const Char b
GHCI> fmap length (Const 'z')
Const 'z'
GHCI> :t fmap length (Const 'z')
fmap length (Const 'z') :: Const Char Int
```

Представители для остальных интерфейсов похожи на пары с отсутствующим вторым элементом

```
instance Foldable (Const c) where
    foldMap :: Monoid m => (a -> m) -> Const c a -> m
    foldMap _ _ = mempty
```

```
GHCI> foldMap Sum (Const 'z')
Sum {getSum = 0}
GHCI> foldMap Any (Const 'z')
Any {getAny = False}
GHCI> foldMap (\x->[x+1,x+2]) (Const 'z')
[]
```

```
instance Monoid c => Applicative (Const c) where
    pure :: a -> Const c a
    pure _ = Const mempty
    (<*>) :: Const c (a -> b) -> Const c a -> Const c b
    Const m1 <*> Const m2 = Const (m1 `mappend` m2)
```

```
GHCi> pure 'z' :: Const [a] Char
Const []
GHCi> Const "AB" <*> Const "CDE"
Const "ABCDE"
GHCi> :t Const "ABCDE"
Const "ABCDE" :: Const [Char] b
```

Представитель `Traversable` аналогично функтору делает простую перепаковку значения **и не нужен для дальнейшего**

```
instance Traversable (Const c) where
    traverse :: (a -> f b) -> Const c a -> f (Const c b)
    traverse _ (Const v) = pure $ Const v
```

Теперь можем написать реализацию представителя класса `Foldable` по умолчанию:

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Действительно:

```
f :: a -> m

Const :: m -> Const m b

Const . f :: a -> Const m b

traverse :: (a -> Const m b) -> t a -> Const m (t b)

traverse (Const . f) :: t a -> Const m (t b)

getConst :: Const m a -> m

getConst . traverse (Const . f) :: t a -> m
```

Аппликативный парсер

- (Stepik, 1 балл) Реализуйте на основе библиотеки парсеров, разработанной на лекции, парсер

```
nat :: Parser Char Int
nat = undefined
```

обеспечивающий следующее поведение

```
GHCi> multiplication2 = (*) <$> nat <*> char '*' <*> nat
GHCi> runParser multiplication2 "14*30"
Just ("",420)
```

Моноидальные функторы

Следующий класс типов изоморфен аппликативному функтору:

```
class Functor f => Monoidal f where
    unit :: f ()
    (*&*) :: f a -> f b -> f (a,b)
```

Метод `unit` обворачивает в контейнер что-то неинтересное, а `(*&*)` делает контейнер пар из пары контейнеров.

```
instance Monoidal [] where
    unit :: []
    unit = []
    (*&*) :: [a] -> [b] -> [(a,b)]
    xs *&* ys = [ (x,y) | x <- xs, y <- ys ]

instance Monoidal ZipList where
    unit :: ZipList ()
    unit = ZipList (repeat ())
    (*&*) :: ZipList a -> ZipList b -> ZipList (a,b)
    (ZipList xs) *&* (ZipList ys) = ZipList (zip xs ys)
```

```
GHCi> [1,2] *&*> [3,4,5]
[(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)]
GHCi> getZipList $ ZipList [1,2] *&*> ZipList [3,4,5]
[(1,3),(2,4)]
```

(Задачи этого раздела теперь добавлены в дз на [stepik.org](#))

► Напишите представителя моноидального функтора для [Maybe](#):

```
GHCi> Just 3 *&*> Just 5
Just (3,5)
GHCi> Just 3 *&*> Nothing
Nothing
```

► Напишите представителя моноидального функтора для пары:

```
GHCi> ("This is ",3) *&*> ("a pair!",5)
("This is a pair!",(3,5))
```

► Напишите представителя моноидального функтора для [\(\(→\) e\)](#):

```
GHCi> (^2) *&*> (*2) $ 5
(25,10)
```

► Покажите, что всякий аппликативный функтор моноидален, выразив методы второго интерфейса через первый.

```
unit' :: Applicative f => f ()
unit' = undefined

pair' :: Applicative f => f a -> f b -> f (a,b)
pair' = undefined
```

► Покажите, что всякий моноидальный функтор аппликативен, выразив методы второго интерфейса через первый.

```

pure' :: Monoidal f => a -> f a
pure' = undefined

ap' :: Monoidal f => f (a -> b) -> f a -> f b
ap' = undefined

```

Обобщение Alternative?

- Имеет ли смысл альтернативный моноидальный функтор, базирующийся не на произведении, а на сумме?

```

class Functor f => AltMonoidal f where
    zero   :: f Void
    (*|*)  :: f a -> f b -> f (Either a b)

instance AltMonoidal [] where
    zero = []
    xs *|* ys = fmap Left xs ++ fmap Right ys

```

```

GHCi> [1,2] *|* "ABC"
[Left 1,Left 2,Right 'A',Right 'B',Right 'C']

```

```

instance AltMonoidal Maybe where
    zero = Nothing
    Nothing *|* y = fmap Right y
    Just x *|* _ = Just (Left x)

```

```

GHCi> Just 3 *|* Just 'A'
Just (Left 3)
GHCi> Nothing *|* Just 'A'
Just (Right 'A')

```

* Законы для моноидальных функторов

```
-- (1) Left identity
snd <$> (unit *&*> v) ≡ v

-- (2) Right identity
fst <$> (u *&*> unit) ≡ u
```

Мораль первых двух законов в том, что `unit` безэффектен.

Для формулировки третьего закона нужны вспомогательные комбинаторы:

```
asl :: (a, (b, c)) -> ((a, b), c)
asl (x, (y, z)) = ((x, y), z)

asr :: ((a, b), c) -> (a, (b, c))
asr ((x, y), z) = (x, (y, z))
```

Пара таких функций задает изоморфизм между типами `(a, (b, c))` и `((a, b), c)`:

```
asl . asr ≡ id :: ((a, b), c) -> ((a, b), c)
asr . asl ≡ id :: (a, (b, c)) -> (a, (b, c))
```

Тогда

```
-- (3) Associativity
asl <$> (u *&*> (v *&*> w)) ≡ (u *&*> v) *&*> w
```

Здесь тип обеих частей `f ((a, b), c)` в предположении, что `u :: f a`, `v :: f b` и `w :: f c`.

```
-- (4) Naturality
(g `bimap` h) <$> (u *&*> v) ≡ (g <$> u) *&*> (h <$> v)
```

типа обеих частей `f (a', b')` в предположении, что `g :: a -> a'`, `h :: b -> b'`, `u :: f a`, `v :: f b`. Здесь использовалась функция `bimap` для пары как представителя класса типов `Bifunctor`:

```

class Bifunctor p where
    bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
    bimap f g = first f . second g
    first :: (a -> b) -> p a c -> p b c
    first f = bimap f id
    second :: (b -> c) -> p a b -> p a c
    second = bimap id

instance Bifunctor (,) where
    bimap f g p = (f (fst p), g (snd p))

```

Отметим, что 4 закон — свободная теорема для типа $*\&*$.

Докажем, что аппликативные законы следуют из моноидальных.

Дано: f — «законный» моноидальный функтор. **Доказать:** реализованная вами выше пара `pure'` и `ap'` удовлетворяет всем аппликативным законам.

► Нулевой закон

```

-- (0') functor/applicative
pure f <*> v ≡ f <$> v

```

Нам понадобится Лемма 1

```

uncurry ($) . (const f `bimap` id) ≡ f . snd

```

Докажите ее.

```

lemma1 :: (b -> c) -> (a, b) -> c
lemma1 f =
    uncurry ($) . (const f `bimap` id)
==== undefined
==== f . snd

```

Теперь докажем нулевой закон:

```

app0Law :: Monoidal f => (a -> b) -> f a -> f b
app0Law f v =
  pure' f `ap` v
  === ap' (pure' f) v
  === ap' (const f <$> unit) v
  === uncurry ($) <$> ((const f <$> unit) *&*> v)
  === uncurry ($) <$> ((const f <$> unit) *&*> (id <$> v)) -- 4 Monoidal
  === uncurry ($) <$> ((const f `bimap` id) <$> (unit *&*> v)) -- 2 Functor
  === (uncurry ($) . (const f `bimap` id)) <$> (unit *&*> v) -- lemma1
  === (f . snd) <$> (unit *&*> v)
  === f <$> (snd <$> (unit *&*> v))
  === f <$> v

```

► Первый закон

```

-- (1') identity
pure id <*> v ≡ v

```

тривиально следует из предыдущего и первого закона функторов.

► Самостоятельно.

```

-- (2') homomorphism
pure f <*> pure x ≡ pure (f x)

```

► Самостоятельно, дома, доп.задание.

```

-- (3') interchange
u <*> pure y ≡ pure ($ y) <*> u

```

► Самостоятельно, дома, доп.задание. (Довольно сложно, точнее нудно.)

```

-- (4') composition
pure (.) <*> u <*> v <*> w ≡ u <*> (v <*> w)

```