Класс типов Traversable

Минимальное полное определение: `traverse` или `sequenceA`.

```
class (Functor t, Foldable t) => Traversable t where
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g = sequenceA . fmap g
```

`sequenceA`: обеспечиваем правило коммутации нашего функтора `t` с произвольным аппликативным функтором `f`. Структура внешнего контейнера `t` сохраняется, а апликативные эффекты внутренних `f` объединяются в результирующем `f`. `traverse` — это `fmap` с эффектами: проезжаем по структуре `t a`, последовательно применяя функцию к элементам типа `a` и монтируем в точности ту же структуру из результатов типа `b`, параллельно «коллекционируя» эффекты.

Представители класса типов Traversable

```
instance Traversable Maybe where
  traverse :: Applicative f =>
                  (a -> f b) -> Maybe a -> f (Maybe b)
  traverse _ Nothing  = pure Nothing
  traverse g (Just x) = Just <$> g x
```

```
instance Traversable [] where
  traverse :: Applicative f =>
                    (a -> f b) -> [a] -> f [b]
  traverse _ []     = pure []
  traverse g (x:xs) = (:) <$> g x <*> traverse g xs
```

```
GHCi> traverse (\x -> [x+10,x+100]) (Just 7)
[Just 17,Just 107]
GHCi> traverse (\x -> [x+10,x+100]) [7,8]
[[17,18],[17,108],[107,18],[107,108]]
```

Сравнение реализаций Traversable и Functor

```
instance Traversable Maybe where
  traverse _ Nothing  = pure Nothing
  traverse g (Just x) = pure Just <*> g x

instance Functor     Maybe where
  fmap     _ Nothing  =      Nothing
  fmap     g (Just x) =      Just    (g x)
```

```
instance Traversable [] where
  traverse _ []     = pure []
  traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs

instance Functor     [] where
  fmap _ []         =      []
  fmap g (x:xs)     =      (:)    (g x)   (fmap     g xs)
```

Первый закон Traversable

```haskell
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap g (Identity x) = Identity (g x)

instance Applicative Identity where
  pure = Identity
  Identity g <*> v = fmap g v
```

(1) identity

```haskell
traverse Identity  ≡  Identity
```

```
GHCi> traverse Identity [1,2,3]
Identity [1,2,3]
```

Реализации fmap по умолчанию

Всякий `Traversable` — это `Functor`: имея `traverse` мы можем универсальным образом реализовать `fmap`, удовлетворяющий законам функтора.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault g = runIdentity . traverse (Identity . g)
```

- `Identity :: b -> Identity b`
- `(Identity . g) :: a -> Identity b`
- `traverse (Identity . g) :: Identity (t b)`
- `runIdentity . traverse (Identity . g) :: t b`

Законы Traversable (2) и (3)

(2) composition

```
traverse (Compose . fmap f2 . f1)  ≡
   Compose . fmap (traverse f2) . traverse f1
```

Здесь обе части имеют тип `t a -> Compose g2 g1 (t c)` в предположении, что `f1 :: a -> g1 b` и `f2 :: b -> g2 c`.

(3) naturality

```
h . traverse f  ≡  traverse (h . f)
```

где `h :: (Applicative f, Applicative g) => f b -> g b` произвольный аппликативный гомоморфизм, то есть функция удовлетворяющая требованиям:
(1) `h (pure x) = pure x`;
(2) `h (x <*> y) = h x <*> h y`.
В предположении, что `f :: a -> f b`, обе части имеют тип `t a -> g (t b)`.

Законы Traversable: практический смысл

Законы `Traversable` дают следующие гарантии:

- Траверсы не пропускают элементов.
- Траверсы посещают элементы не более одного раза.
- `traverse pure` дает `pure`.
- Траверсы не изменяют исходную структуру — она либо сохраняется, либо полностью исчезает.

```
GHCi> traverse Just [1,2,3]
Just [1,2,3]
GHCi> traverse (const Nothing) [1,2,3]
Nothing
```