Курс: Функциональное программирование
Практика 8. Аппликативные функторы

Разминка

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
"ABCD" *> [1,2]

"ABCD" <* [1,2]

[(*0),(+100),(^2)] <*> [1,2,3]

(++) <$> ["ha","heh","hmm"] <*> ["?","!","..."]

[(+),(*)] <*> [1,2] <*> [30,40]

getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"

(,,) <$> "dog" <*> "cat" <*> "rat"
```

Аппликативные функторы

▶ (Stepik) Для типа данных, изоморфного `Either`,

```
data E l r = L l | R r  deriving (Eq, Show)
```

напишите представителя класса типов `Applicative` с семантикой, подобной `Maybe`:

```
instance Applicative (E l) where
  pure  = undefined
  (<*>) = undefined
```

Не забудьте для `E` реализовать представителя класса типов `Functor`.

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
(*) <$> R 2 <*> R 3

(*) <$> R 2 <*> L "Oh."

(*) <$> L "Ha!" <*> L "Oh."

(*) <$> L "Ha!" <*> undefined
```

Обсудим реализацию представителя аппликативного функтора для `((->) a)`

```
instance Applicative ((->) e) where
  pure = undefined
  (<*>) = undefined
```

Попробуем записать тип `(<*>)` для частично примененной стрелки

```
f (a -> b) -> f a -> f b
  ≡ (e -> (a -> b)) -> (e -> a) -> (e -> b)
  ≡ (e -> a -> b) -> (e -> a) -> e -> b
```

▶ Что это за комбинатор?

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
(pure 3) "blah"

(+) <*> (*3) $ 4

zip <*> tail $ [1..10]

(+) <$> (+2) <*> (*3) $ 10
```

```
(\a b c -> [a,b,c]) <$> (+5) <*> (*3) <*> (/2) $ 7
```

▶ Что делает следующая функция?

```
fun = 0 : 1 : (zipWith (+) <*> drop 1) fun
```

▶ Каков тип следующих конструкций для `((->) a)`, и как их можно было бы записать, не используя аппликативный стиль:

```
\f g h -> f <*> g <*> h ≡ \f g h -> ???
\f g h -> f <$> g <*> h ≡ \f g h -> ???
```

Аппликативные функторы: законы

```
-- (0)
fmap g xs  ≡  pure g <*> xs
-- (1) Identity
pure id <*> v  ≡  v
-- (2) Interchange
u <*> pure x  ≡  pure ($ x) <*> u
-- (3) Homomorphism
pure g <*> pure x  ≡  pure (g x)
-- (4) Composition
pure (.) <*> u <*> v <*> w  ≡  u <*> (v <*> w)
```

▶ Проверьте, что законы аппликативных функторов выполняются для типа `Maybe`.

▶ Проверьте, что законы аппликативных функторов выполняются для типа `((->) a)`. (дома, как дополнительное задание).

Композиция уровня типов

▶ Рассмотрим следующий трёхпараметрический конструктор типа, инкапсулирующий композицию двух однопараметрических конструкторов типа

```
newtype Cmps f g x = Cmps { getCmps :: f (g x) }
```

Каков кайнд этого конструктора типов? Приведите пример простого типа, сконструированного с помощью `Cmps`, и пример терма этого типа.
▶ Определите функцию

```
ffmap h = getCmps . fmap h . Cmps
```

и объясните её выведенный тип. Попробуйте осуществить вызов

```
GHCi> ffmap (+42) $ Just [1,2,3]
```

В чём причина ошибки?

▶ (Stepik) Чтобы обеспечить работоспособность подобного вызова, сделайте тип `Cmps` представителем класса типов `Functor`

```
instance (Functor f, Functor g) => Functor (Cmps f g) where
  fmap = undefined
```

Проверьте работоспособность на примерах

```
GHCi> ffmap (+42) $ Just [1,2,3]
Just [43,44,45]
GHCi> ffmap (+42) [Just 1,Just 2,Nothing]
[Just 43,Just 44,Nothing]
```

▶ Проверьте, что все законы функторов выполняются для этого представителя.

Таким образом **композиция функторов является функтором**. Подобное утверждение верно для `Applicative`, `Foldable`, `Traversable`, но неверно для `Monad`.