

# Контейнеры и категории

---



Функциональное программирование  
*2023 г.*



## Pattern Matching в Haskell

- Сопоставление по образцам — сверху вниз (рассахаривание в case-оператор).
- Сопоставление с образцом — слева направо.
- Образцы линейные (без повторных переменных).
- Можно также внутри let и лямбда-объявлений.

```
ff y = let (x:4:y) = map (*2) [1,2,1]
      in x + 5
```



## Некоторые удобные конструкции

- As-образец (аналог where для образцов):

```
f x@(y:ys) = (:) y x
```

- Ленивые образцы (связывание с образцом не влечет выполнение аргумента):

```
f ~(x:xs) = x
```



## Примеры

```
-- bottom
f x = f x

let x = f 1 in 0
let (x,y) = f 1 in 0
(\(x,y) -> 0) (f 1)
(\~(x,y) -> 0) (f 1)
(\(x,y) -> 0) ((f 1),(f 1))
(\(x:xs) -> (x:x:xs)) (f 1)
```



## Теория категорий

### Определение

Категория  $\mathcal{C}$  — пара  $\{O, M\}$ , где  $O$  — набор объектов,  $M$  — набор стрелок (морфизмов), и выполнены следующие законы:

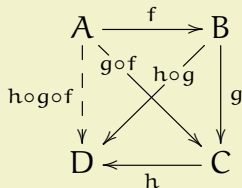
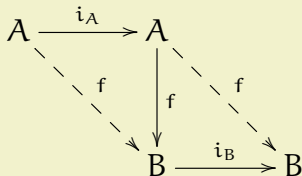
- $\forall A, B \in O \exists id_A, id_B \forall f : A \rightarrow B (id_B \circ f = f \circ id_A = f)$   
(тождественный морфизм)
- $\forall f : A \rightarrow B, g : B \rightarrow C \exists h : A \rightarrow C (f \circ g = h)$   
(существование композиции)
- $\forall f, g, h (f \circ (g \circ h) = (f \circ g) \circ h)$  (ассоциативность композиции)

Объекты — типы; морфизмы — функции над типами.

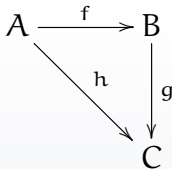


# Коммутативные диаграммы

## Диаграммы для законов категории



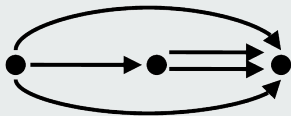
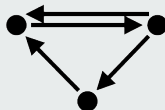
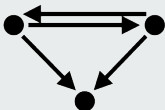
- Треугольник указанного ниже вида означает, что  $g \circ f = h$ .



- Пунктир указывает, что морфизм единственный.



## Задача



Диаграммы не обязательно коммутативные! Единичные морфизмы опускаются.



## Универсальная конструкция

Универсальная конструкция — конструкция, которая описывается только свойствами морфизмов, а не внутренними свойствами объектов, включённых в конструкцию.

Эпиморфизм — такой морфизм  $f$ , что

$$\forall g_1, g_2 (g_1 \circ f = g_2 \circ f \Rightarrow g_1 = g_2).$$

Мономорфизм — такой морфизм  $f$ , что

$$\forall h_1, h_2 (f \circ h_1 = f \circ h_2 \Rightarrow h_1 = h_2).$$

Универсальные конструкции определяются с точностью до изоморфизма.





## Универсальная конструкция

Универсальная конструкция — конструкция, которая описывается только свойствами морфизмов, а не внутренними свойствами объектов, включённых в конструкцию.

Эпиморфизм — такой морфизм  $f$ , что

$$\forall g_1, g_2 (g_1 \circ f = g_2 \circ f \Rightarrow g_1 = g_2).$$

Мономорфизм — такой морфизм  $f$ , что

$$\forall h_1, h_2 (f \circ h_1 = f \circ h_2 \Rightarrow h_1 = h_2).$$

Универсальные конструкции определяются с точностью до изоморфизма.

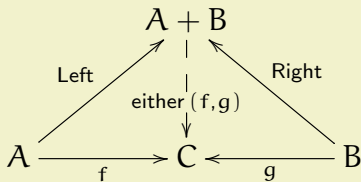
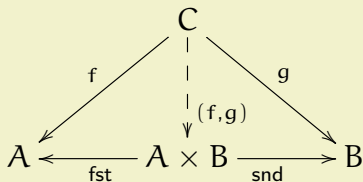
Инициальный объект — такой объект, что из него есть ровно одна стрелка в любой другой.

Терминальный объект — объект, в который есть ровно одна стрелка из любого другого.

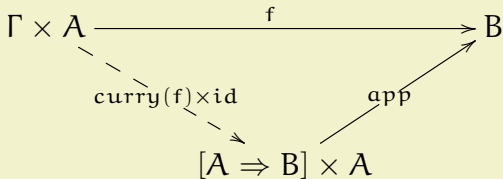


# Конструкторы типов

## Прямое произведение и прямая сумма



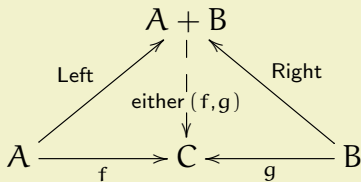
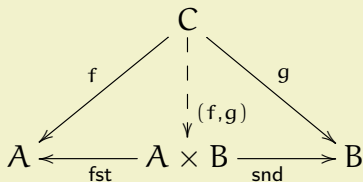
## Функциональный тип(экспонента) — пока без пояснений





# Конструкторы типов

## Прямое произведение и прямая сумма



Из каждого «кандидата» на произведение есть стрелка в «оптимальное» произведение.



## Алгебраические типы данных

- Структура — перечисление  $n$ -ок.
- Прямое произведение:  $(a, ()) \sim a$ ,  $(a, b) \sim (b, a)$ ,  
 $(a, (b, c)) \sim ((a, b), c)$ .
- Прямая сумма:  $\text{Either } a \text{ void} \sim a$ ,  
 $\text{Either } a \text{ (Either } b \text{ } c) \sim \text{Either (Either } a \text{ } b) \text{ } c$ .
- Дистрибутивность:  
 $(a, \text{Either } b \text{ } c) \sim \text{Either (} a, b \text{) (} a, c \text{)}$

Прямая сумма + прямое произведение + дистрибутивность =  
полукольцо.



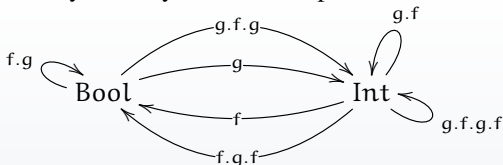
## Пример

Начинающий программист спроектировал систему с двумя типами: `int` и `bool` — и следующими преобразованиями `f`, `g` между ними.

```
f x
  | x >= 0 = True
  | x < 0 = False
g True = 1
g False = 0
```

Какая категория соответствует его системе? Какие изменения можно внести, чтобы категория упростилась?

Системе соответствует следующая категория.



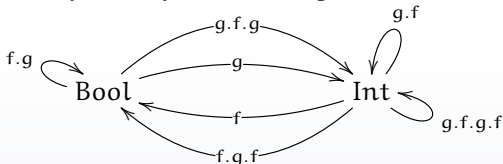


## Пример

```
f x
  | x >= 0 = True
  | x < 0 = False
g True = 1
g False = 0
```

Какая категория соответствует его системе? Какие изменения можно внести, чтобы категория упростилась?

Системе соответствует следующая категория.

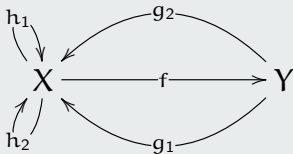


Чтобы ее улучшить, достаточно положить либо  $g \text{ False} = -1$ , либо  $f \text{ x} \mid x \leq 0 = \text{False}$ .



## Задача

Рассмотрим следующую категорию.

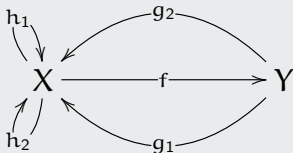


Построить систему типов  $X$ ,  $Y$  и преобразований между ними, которая ей соответствует.



## Задача

Рассмотрим следующую категорию.



Построить систему типов  $X$ ,  $Y$  и преобразований между ними, которая ей соответствует.

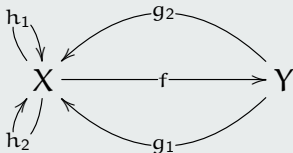
Композиции  $f.g_1$ ,  $f.g_2$  обе могут быть равны только  $\text{id}_Y$ . Поэтому  $f$  — сюръекция. Предположим теперь, что  $g_1.f = \text{id}_X$ . Тогда  $g_1.f.g_2 = g_1 = g_2$ , что невозможно. Поэтому ни  $g_1.f$ , ни  $g_2.f$  тождественными морфизмами быть не могут.

Композиции  $f.h_1$ ,  $f.h_2$  — морфизмы из  $X$  в  $Y$ . Такой морфизм на диаграмме один, это  $f$ . Поэтому  $f.h_1 = f.h_2 = f$ .





## Продолжение задачи



Положим, что тип  $Y$  состоит ровно из одного элемента  $a$  (то есть  $Y = \{a\}$ ). Тогда для всех  $x \in X$   $f(x) = a$ . Пусть  $g_1(a) = b_1$ ,  $g_2(a) = b_2$ . Тогда композиция  $g_1 \cdot f$  — это морфизм, отображающий все элементы  $X$  в  $b_1$ , а  $g_2 \cdot f$  отображает все в  $b_2$ . Назначим  $g_1 \cdot f = h_1$ ,  $g_2 \cdot f = h_2$ .

Проверим существование композиций.

$h_1 \cdot h_1 = h_1$	$h_1 \cdot g_1 = g_1$	$g_1 \cdot f = h_1$
$h_1 \cdot h_2 = h_1$	$h_1 \cdot g_2 = g_1$	$g_2 \cdot f = h_2$
$h_2 \cdot h_1 = h_2$	$h_2 \cdot g_1 = g_2$	$f \cdot g_i = \text{id}$
$h_2 \cdot h_2 = h_2$	$h_2 \cdot g_2 = g_2$	$f \cdot h_i = f$

Поскольку определенные нами преобразования являются функциями в алгебраическом смысле, ассоциативность композиции выполняется.

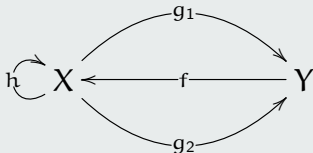


## Рекомендации

- 1 Ритуальная фраза про «функции в алгебраическом смысле» подходит для всех чистых функций (детерминированных). Главное при построении модели — убедиться, что существуют все композиции.
- 2 Если у объекта  $X$  есть петли (стрелки в себя), нужно убедиться, что существует композиция петли с собой, с другими петлями и со всеми входящими в  $X$  и исходящими из  $X$  стрелками.
- 3 Чем меньше элементов содержит тип, тем проще построить модель. Если к объекту  $X$  из каждого другого объекта ведет не больше, чем одна стрелка, и петель у  $X$  нет, можно попробовать объявить  $X$  одноэлементным типом. Это самое простое решение, но не обязательно оно подойдет.



## Это категория или нет?



Заведомо  $g_i \cdot f = \text{id}_Y$  (других морфизмов из  $Y$  в  $Y$  нет). Поэтому  $f$  — инъекция. Притом  $h \cdot f = f$ , что означает, что  $h$  совпадает с  $\text{id}_X$  на  $\text{codom}(f)$  (т.е. если  $\exists w(f(w) = v)$ , то  $h(v) = v$ ).

Известно, что  $h \neq \text{id}_X$ , то есть  $\exists a'(h(a') \neq a')$ . Поэтому  $a' \notin \text{codom}(f)$ , и  $f$  — не сюръекция. Значит, композиции  $f \cdot g_i$  не могут быть равны  $\text{id}_X$  (в частности, ни одна из них не может принимать значение  $a'$ ), следовательно, обе они равны  $h$ .

Поскольку  $g_1 \neq g_2$ , то  $\exists a(g_1(a) \neq g_2(a))$ . Поскольку  $f$  инъективна, то  $f(g_1(a)) \neq f(g_2(a))$ . С другой стороны, из рассуждений выше  $f(g_1(a)) = h(a) = f(g_2(a))$ . Противоречие.



## Другие преобразования типов

Что такое в терминах категорий список  $[a]$ ?

- Любой тип может порождать список.
- Структура преобразований над простыми типами переносится на структуру преобразований над списками.

Списки — это преобразование между категориями!

### Определение

Даны категории  $C_1, C_2$ . Функтор  $F$  — преобразование  $F : C_1 \rightarrow C_2$  такое, что

- если  $f : a \rightarrow b$  — морфизм  $C_1$ , то  $F(f) : F(a) \rightarrow F(b)$  — морфизм  $C_2$ ;
- $F(\text{id}_X) = \text{id}_{F(X)}$ ;
- $F(f \circ g) = F(f) \circ F(g)$ .



## Функторы

Функтор  $F$   $a$  — полиморфная структура, снабжённая функцией `fmap` (или `<$>`):

```
fmap :: (a -> b) -> F a -> F b
```

Логичное требование: если функция ничего не делает, её навешивание на структуру не должно ничего менять.

```
fmap id = id
```



# Функторы в Haskell

## Определение

```
class Functor f where  
fmap :: (a -> b) -> f a -> f b
```

## Примеры

```
-- СПИСКИ  
instance Functor [] where  
fmap = map  
  
-- Maybe  
instance Functor Maybe where  
fmap f (Just x) = Just (f x)  
fmap f Nothing = Nothing
```

fmap можно заменить инфиксным  $\langle \$ \rangle$ .



## ФЯ как исчисление

- 1 Строим тип требуемой полиморфной структуры в виде формулы логики.
- 2 Строим вывод этого типа, и заодно пытаемся извлечь из вывода построение.
- 3 Если построение неоднозначно, тогда тестируем варианты построения с помощью законов.
- 4 Если построение всё ещё неоднозначно, значит, их несколько. Только на этом этапе требуется согласовать конструкцию с библиотечной.
- 5 PROFIT! Почти всегда получаем корректно определённую структуру, 100% согласованную с дизайном языка.



## Функтор State s

```
-- F a раскрываем как s -> (a, s)
fmap :: (a -> b) -> (s -> (a, s)) -> (s -> (b, s))
```

В виде логической формулы:

$$(A \Rightarrow B) \Rightarrow ((S \Rightarrow A \ \& \ S) \Rightarrow (S \Rightarrow B \ \& \ S)).$$





## Функтор State s

```
-- F a раскрываем как s -> (a, s)
fmap :: (a -> b) -> (s -> (a, s)) -> (s -> (b, s))
```

В виде логической формулы:

$$(A \Rightarrow B) \Rightarrow ((S \Rightarrow A \ \& \ S) \Rightarrow (S \Rightarrow B \ \& \ S)).$$

Два возможных способа доказательства:

- $\backslash f \ x \ s \rightarrow \text{let } (x1, s1) = x \ s \text{ in } (f \ x1, s)$
- $\backslash f \ x \ s \rightarrow \text{let } (x1, s1) = x \ s \text{ in } (f \ x1, s1)$

Какой из них подойдёт для построения правильного функтора?

Здравого смысла у нас нет, есть только логика.



## Функтор State s

```
-- F a раскрываем как s -> (a, s)
fmap :: (a -> b) -> (s -> (a, s)) -> (s -> (b, s))
```

В виде логической формулы:

$$(A \Rightarrow B) \Rightarrow ((S \Rightarrow A \ \& \ S) \Rightarrow (S \Rightarrow B \ \& \ S)).$$

Два возможных способа доказательства:

- $\backslash f \ x \ s \rightarrow \text{let } (x1, s1) = x \ s \text{ in } (f \ x1, s)$
- $\backslash f \ x \ s \rightarrow \text{let } (x1, s1) = x \ s \text{ in } (f \ x1, s1)$

Какой из них подойдёт для построения правильного функтора?

Здравого смысла у нас нет, есть только логика.

(подсказка: законы)



## Функтор Either e I

```
-- F a раскрываем как Either e a
fmap :: (a -> b) -> (Either e a) -> (Either e b)
```

Требуемая формула типа:  $(A \Rightarrow B) \Rightarrow ((E \vee A) \Rightarrow (E \vee B))$ .

$\frac{\frac{\frac{*A \Rightarrow B}{*E \vee A} \quad \text{тип } f}{*E} \quad \text{тип } a \quad \frac{*A}{B} \quad \text{тип } f}{E \vee B \quad \text{Left } z1} \quad \frac{B}{E \vee B \quad \text{Right } (f \ z2)} \quad \frac{E \vee B \quad \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a}{(E \vee A) \Rightarrow (E \vee B)} \quad \lambda a \rightarrow \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a}{(A \Rightarrow B) \Rightarrow ((E \vee A) \Rightarrow (E \vee B))} \quad \lambda f \rightarrow (\lambda a \rightarrow \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a)$	$\frac{\frac{\frac{*A \Rightarrow B}{*E \vee A} \quad \text{тип } f}{*E} \quad \text{тип } a \quad \frac{*A}{B} \quad \text{тип } f}{E \vee B \quad \text{Left } z1} \quad \frac{B}{E \vee B \quad \text{Right } (f \ z2)} \quad \frac{E \vee B \quad \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a}{(E \vee A) \Rightarrow (E \vee B)} \quad \lambda a \rightarrow \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a}{(A \Rightarrow B) \Rightarrow ((E \vee A) \Rightarrow (E \vee B))} \quad \lambda f \rightarrow (\lambda a \rightarrow \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \text{Right } (f \ z2)) \ a)$
--	--

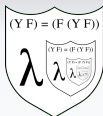


## Примеры функторов

### Задача

Может ли аппликация  $\langle \$ \rangle$  для списков быть определена как `fmap1`, `fmap2`, `fmap3`? Если нет, то какие законы функторов нарушаются?

```
fmap1 f [] = []  
fmap1 f (x:xs) = (f x : f x : fmap1 f xs)  
  
fmap2 f [] = []  
fmap2 f (x:xs) = mappend (fmap2 f xs) [f x]  
  
fmap3 f [] = []  
fmap3 f (x:xs)  
    | (f x) == x = (x:xs)  
    | otherwise = (f x : f x : fmap3 f xs)
```



## Бифункторы

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



## Функции > 1 аргумента

Проблема:

```
?fmap :: (a -> (b -> c)) -> F a -> ???
```

Получается алгебраическая структура «не до конца вычисленных» значений:  $F (b \rightarrow c)$ . Как вычислять её дальше?



## Функции > 1 аргумента

Проблема:

```
?fmap :: (a -> (b -> c)) -> F a -> ???
```

Получается алгебраическая структура «не до конца вычисленных» значений:  $F (b \rightarrow c)$ . Как вычислять её дальше? Понадобится функция типа:

```
??? :: F (b -> c) -> (F b -> F c)
```



## Функции > 1 аргумента

Понадобится функция типа:

```
<*> :: F (b -> c) -> (F b -> F c)
```

С помощью такой аппликации можно обрабатывать функции какого угодно числа аргументов, если добавить ещё «стандартный способ» погружения в алгебру:

```
pure :: a -> F a
```





# Аппликативные функторы

## Определение

```
class Functor f => Applicative f where
  pure :: a -> f a
  <*> :: f(a -> b) -> f a -> f b
```

## Примеры

```
-- Списки
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

-- Maybe
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> a = f <$> a
```



## Законы аппликативов

Пока смотрим:

`pure id <*> x = x`

`pure f <*> pure x = pure (f x)`

`u <*> pure x = pure (\f -> f x) <*> u`

`pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Алгебраически естественно выводятся из альтернативного представления аппликативных функторов через т.н. «тензорное умножение» на моноидальной структуре.



## Моноидальная структура

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

Законы моноидальных функторов:

```
u ** unit = u = unit ** u
fmap (f * g) (u ** v) = fmap f u ** fmap g v
u ** (v ** w) = (u ** v) ** w
```

Правила перевода:

```
pure x = fmap (const x) unit
fs <*> xs = fmap (uncurry ($)) $ fs ** xs
```



## Аппликатив для State s

Запишем подходящий тип:

$$(S \Rightarrow (A \Rightarrow B) \ \& \ S) \Rightarrow (S \Rightarrow A \ \& \ S) \Rightarrow (S \Rightarrow B \ \& \ S)$$

Начнём вывод:

$$*S \Rightarrow (A \Rightarrow B) \ \& \ S$$

тип  $f$

$$*S \Rightarrow A \ \& \ S$$

тип  $a$

$$*S$$

тип  $s$

*...и непонятно, что применять к  $s$ :  $f$  или  $a$ ?*

$$(S \Rightarrow (A \Rightarrow B) \ \& \ S) \Rightarrow (S \Rightarrow A \ \& \ S) \Rightarrow (S \Rightarrow B \ \& \ S)$$

Достроить вывод типа, извлечь терм и показать, что извлечённый терм не противоречит законам аппликативов номер 1 и 3 (законы 2 и 4 можно не проверять) — 3 балла. Решение здесь не единственное.



## Аппликатив для Either e

$$\begin{array}{c}
 *A \Rightarrow B \\
 *E \vee A \\
 \begin{array}{|l}
 *E \\
 \hline
 E \vee B \quad \text{Left } z1 \\
 E \vee B
 \end{array}
 \quad
 \begin{array}{|l}
 z1 *A \\
 \begin{array}{|l}
 *E \\
 \hline
 E \vee B \quad \text{Left } v1 \\
 E \vee B \quad \text{either } (\lambda v1 \rightarrow \text{Left } v1) (\lambda v2 \rightarrow \text{Right } (v2 \ z2)) f \\
 \text{either } (\lambda z1 \rightarrow \text{Left } z1) (\lambda z2 \rightarrow \\
 \quad (\text{either } (\lambda v1 \rightarrow \text{Left } v1) (\lambda v2 \rightarrow \text{Right } (v2 \ z2)) f)) \alpha
 \end{array}
 \end{array}
 \quad
 \begin{array}{|l}
 v1 *A \Rightarrow B \\
 \begin{array}{|l}
 B \\
 \hline
 E \vee B \quad \text{Right } (v2 \ z2)
 \end{array}
 \end{array}
 \quad
 \begin{array}{|l}
 v2 \\
 \hline
 v2 \ z2
 \end{array}
 \quad
 z2
 \end{array}
 \end{array}$$

$$\begin{array}{c}
 (E \vee A) \Rightarrow (E \vee B) \\
 (E \vee (A \Rightarrow B)) \Rightarrow ((E \vee A) \Rightarrow (E \vee B)) \\
 \text{either } (\lambda z1 \rightarrow \text{Left } z1) \\
 \quad (\lambda z2 \rightarrow (\text{either } (\lambda v1 \rightarrow \text{Left } v1) (\lambda v2 \rightarrow \text{Right } (v2 \ z2)) f)) \\
 \alpha
 \end{array}$$

тип  $f$   
тип  $\alpha$   
тип  $\lambda \alpha \rightarrow (\text{either } \dots)$   
тип  $\lambda f \alpha \rightarrow$

Не считая неудобной кодировки для разбора случаев, вывод простой и прямолинейный.



## Упражнения

```
-- K, S определяются как обычно  
f x y = K (*) 'S' (+) x 'S' (+) y  
> f 1 2 3
```

```
> [(*0), (+100), (-2)] <*> [1,2,3]  
> (,) <$> "dog" <*> "cat" <*> "rat"  
> (\a b c -> [a,b,c]) <$> (+5) <*> (*3) <*> (/2) $ 7
```



## Задача

Может ли аппликативный функтор на списках задаваться операциями `fapp1`, `fapp2`? Если нет, какие законы аппликативных функторов нарушаются?

```
fapp1 [] xs = []  
fapp1 (f:fs) (x:xs) = ((f x):(fapp1 fs xs))  
  
fapp2 [] xs = []  
fapp2 [f] xs = map f xs  
fapp2 (f:fs) xs = fapp2 fs (map f xs)
```



## Продолжение задачи

Ещё раз рассмотрим определение аппликации в контексте списков.

```
pure x = [x]
fapp1 [] xs = []
fapp1 (f:fs) (x:xs) = ((f x):(fapp1 fs xs))
```

Эта аппликация нарушает закон сохранения единичного элемента.

```
> [\x -> x] 'fapp1' [1,2,3]
[1]
```





## Продолжение задачи

Ещё раз рассмотрим определение аппликации в контексте списков.

```
pure x = [x]
fapp1 [] xs = []
fapp1 (f:fs) (x:xs) = ((f x):(fapp1 fs xs))
```

Эта аппликация нарушает закон сохранения единичного элемента.

```
> [\x -> x] 'fapp1' [1,2,3]
[1]
```

Можно переопределить pure так, чтобы fapp1 стало корректной аппликацией.

```
pure x = [x, x..]
```



## Еще о законах аппликативов

Если выполняются два первых закона, то четвертый всегда нужно проверять для списка функций, содержащего больше, чем один элемент. Например, рассмотрим определение:

```
fapp3 [] xs = []  
fapp3 (f:fs) xs = map f xs
```

Четвертый закон:

$\text{fapp3 fs [x]}$  должно быть равно  $\text{fapp3 } [\backslash h \rightarrow h x] \text{ fs}$

Положим  $\text{fs} = (\text{f}:\text{fs}')$ , тогда

$\text{fapp3 fs [x]} = [(f x)]$

$\text{fapp3 } [\backslash h \rightarrow h x] \text{ fs} = \text{map } (\backslash h \rightarrow h x) \text{ fs}$

Видно, что списки получаются разные. Чтобы убедиться в этом, достаточно рассмотреть  $\text{fs} = [(+1), (+2)]$ ,  $x = 0$ .



## Модальные логики

Модальность «необходимости»:  $\Box A$  ( $A$  выполнено во всех достижимых мирах).

Правила вывода, общие для большинства модальных логик:

$$\frac{A \text{ — теорема}}{\Box A} \quad \frac{\Box(A \Rightarrow B)}{\Box A \Rightarrow \Box B}$$

Ничего не напоминают?



## Модальные логики

Модальность «необходимости»:  $\Box A$  ( $A$  выполнено во всех достижимых мирах).

Правила вывода, общие для большинства модальных логик:

$A$  — теорема      $\Box(A \Rightarrow B)$

$\Box A$       $\Box A \Rightarrow \Box B$

Ничего не напоминают?

Аппликация! (и это *логичная* причина так называемого «applicative monad proposal» в Haskell)

Из интуиционистской логики можно перейти к модальной простым гомоморфизмом  $\mu$ :

$\mu(A) = \Box A$  ( $A$  — переменная)

$\mu(\Phi \Rightarrow \Psi) = \Box(\mu(\Phi) \Rightarrow \mu(\Psi))$

$\mu(\Phi \vee \Psi) = \mu(\Phi) \vee \mu(\Psi)$

$\mu(\Phi \& \Psi) = \mu(\Phi) \& \mu(\Psi)$

Помимо правил аппликативов, в полученной логике выполнены:

$\Box A \Rightarrow A$ ,  $\Box A \Rightarrow \Box \Box A$ . О смысле этих правил узнаем далее.



## Модальная логика S4

### Определение S4

- Модальность  $\Box$  — «необходимость». Правило вывода:  
$$\frac{A \text{ — теорема}}{\Box A}.$$

Аксиомы:

- $\Box A \Rightarrow A$ ;
- $\Box A \Rightarrow \Box \Box A$ ;
- $\Box(A \Rightarrow B) \Rightarrow \Box A \Rightarrow \Box B$
- Двойственная модальность  $\Diamond$  — «возможность».  
 $\Diamond A \Leftrightarrow \neg \Box \neg A$ . Выводимо  $A \Rightarrow \Diamond A$ .

Модели S4 — модели Крипке.



## Лабораторная работа 2 (любой функциональный язык)

- 1 По описанию графа в языке Graphviz автоматически проверить, описывает ли он категорию как множество преобразований над, самое большее, множествами из  $n$  элементов, где  $n$  — это максимальное число входящих стрелок в некоторый объект.
- 2 По записи терма в формате Haskell, содержащей аппликацию, абстракцию, пары и проекции, построить редуцированный терм, эквивалентный ему. Исходный терм не обязательно корректно типизируется.
- 3 По формуле в минимальной логике и входному числу  $n$  проверить, существует ли контрпример для этой формулы в моделях Крипке максимальной высоты  $n$ .
- 4 По  $\lambda$ -терму, записанному в формате Haskell, построить CPS-термы в стиле Call-by-Name и Call-by-Value.
- 5 По записи терма в формате Haskell, содержащей аппликацию, абстракцию, пары и проекции, построить его тип либо вывести сообщение о том, что терм не типизируется.
- 6 (на двоих) По  $\lambda$ -терму построить его представление в комбинаторах, с использованием ограниченного  $\eta$ -преобразования.
- 7 (на двоих) По описанию ADT в языке Haskell автоматически породить реализацию функтора для него.