Теория категорий

Теория Категорий

Теория категорий (ТК) - это раздел математики изучающий объекты и любые их взаимоотношения, вне зависимости от свойств самих объектов.
Основным объектом изучения **ТК** является **категория,** состоящая из объектов и морфизмов между ними.

Теория Категорий

Мотивация. ТК широко применяется в разных разделах физики, математики и программирования. **ТК** предоставляет возможность подойти к решению проблем на крайне высоком уровне абстракции. Она, позволяет, не вдаваясь в детали предметной области решать многие задачи, основываясь только на анализе взаимоотношений объектов предметной области. По ссылке можно найти пример использования **ТК** для решения математических проблем. Для программиста бывает удобно представлять программу в терминах **ТК**. Где предметные области, участвующие в приложении, могут рассматриваться как категории, а функции и методы в них, как морфизмы, функторы и натуральные преобразования. Существует несколько библиотек для scala, такие как **scalaz**, **cats**, **monocle**, которые помогают выделить и применить категорные концепции на практике.

Теория Категорий

Объект - это элемент, член какой-либо категории, не обладающий какими либо характерными свойствами. Объекты в ТК можно сравнить с точками в геометрии.

Морфизм - это направленное отношение между членами категории. Иногда морфизм еще называют “стрелка”.
Морфизм обладает следующими свойствами

- Имеет начальный и конечный объект. Объектом начала и конца морфизма, может быть один и тот же объект
- Направлен. Т.е. морфизм из точки **A ->B** не тоже самое, что морфизм из **B -> A**

Композиция морфизмов Для двух морфизмов **M(A ->B)** и **N(B ->C)** определена операция композиции **N ○ M**(читается N после M или N compose M), которая порождает новый морфизм **C (A -> C).** Морфизмы ассоциативны, т.е. если найдется еще один морфизм **P(C -> D)**, то **(P ○ N) ○ M = P ○ (N ○ M)**

Теория Категорий

Категория - это любая общность объектов и морфизмов, удовлетворяющая следующим свойствам

- Содержит 0 или более объектов
- Для каждого объекта существует уникальный морфизм в этот же объект $\mathbf{Id_A}$ **,** который обладает следующим свойством:
  $\mathbf{Id_B} \circ \mathbf{M(A \to B) = M(A \to B) \circ Id_A = M(A \to B)}$
- Между двумя произвольными объектами может существовать произвольное количество морфизмов
- Для 2 -х подходящих морфизмов обязана существовать их композиция, также являющаяся морфизмом в этой категории

Из свойств видно, что не бывает категорий, состоящих из одних только морфизмов. Морфизмы задают структуру категории через отношения между объектами, не описывая характер объектов и суть этих взаимоотношений.

Теория Категорий

Примеры категорий

Set - категория, в которой объектами являются множества, включая пустое множество. Морфизмами являются функции, преобразующие одни множеств в другие, удовлетворяющие свойствам морфизмов.
Hom(a, b) set - это множество всех морфизмов из объекта **a** в объект **b. Hom** также являются объектами в категории **Set.**

Cat - категория (или 2-категория) всех категорий**.**

Малая категория - это любая категория в которой стрелки (и соответственно объекты) формируют множество

Локально малая категория - это категория, в которой каждый **Hom** представляет собой множество

Категория типов и функций. Применяя теорию категорий в программировании, в качестве базовой берется категория, объектами которой являются типы, а морфизмами функции над типами.

Теория Категорий

Примеры категорий

Кокатегория (двойственная категория)**.** Категория является кокатегорией к какой-либо категории, когда все объекты в исходной категории сохранены, а все морфизмы изменили свое направление на противоположное

2-категория, **n-категория**, **∞-категория** - категории высших порядков. 2-категория - это категория, в которой морфизмы существуют не только между объектами но и между морфизмами объектов.

Теория Категорий

Универсальное свойство (УС).

Как правило, для проведения каких-либо построений в ТК принято рассматривать всю категорию целиком. Чтобы дать определение какой-либо сущности или факту, выбирают некое универсальное свойство. Далее проводят построение, показывающее наличие или отсутствие объектов в категории, удовлетворяющих заданному свойству. Построения проводят для каждой категории в отдельности. Дадим определение начального и конечного объектов категории через УС

Терминальный объект(ТО) в категории **C** - это объект **1** в категории, обладающий следующим универсальным свойством:
для каждого объекта **x** в категории **C** существует уникальный морфизм **X -> 1**. Терминальный объект категории, если существует, он уникален вплоть до изоморфизма.
Начальный объект(НО) в категории C — это её объект **I**, такой что для любого объекта **X** в **C** существует единственный морфизм **I -> X**
Нулевым объектом называется любой объект, являющийся одновременно начальным и конечным

Теория Категорий

Примеры

В категории **Set**, одноэлементные множества в той же категории являются терминальными объектами. Терминальный объект уникален вплоть до изоморфизма.

Покажем что утверждение верно для терминального объекта.
Выберем в категории **Set** пустое множество и обозначим его **Unit.** Как мы знаем, стрелки в категории **Set -** это функции. Объявим тогда такую полиморфную функцию
def unit[T](prm: T): Unit = (). Ясно, что эта функция переводит любое множество **T** в множество **Unit.** Следовательно **Unit** является терминальным объектом.

В категории **Set**, пустое множество является единственным начальным объектом

Вопрос:
Как вы думаете, как будут выглядеть стрелки (функции) из пустого множества к каждому множеству категории **Set ?**

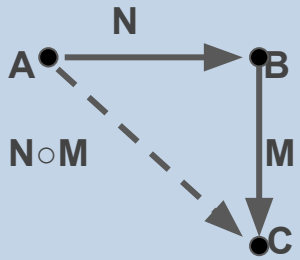
Теория Категорий

Коммутативные диаграммы (КД)

В теории категорий, для иллюстрации взаимоотношений объектов и морфизмов используют коммутативные диаграммы. Буквами в них, обозначают объекты категории, а стрелочками морфизмы. Коммутативность означает, что для любых выбранных начального и конечного объекта для соединяющих их ориентированных путей композиция соответствующих пути морфизмов не будет зависеть от выбора пути
Часто, при доказательстве существования некоего морфизма, искомый морфизм обозначают пунктирной стрелкой, а данные морфизмы - сплошной

Пример - композиция морфизмов **N** и **M**

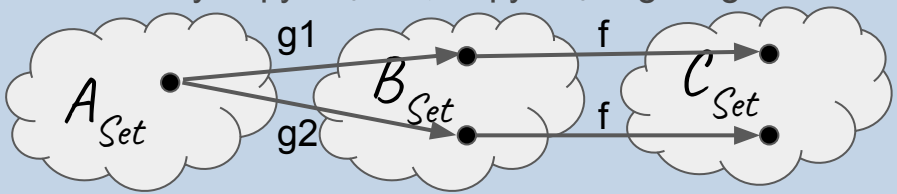
N(A, B) ○ M(B, C)



Теория Категорий

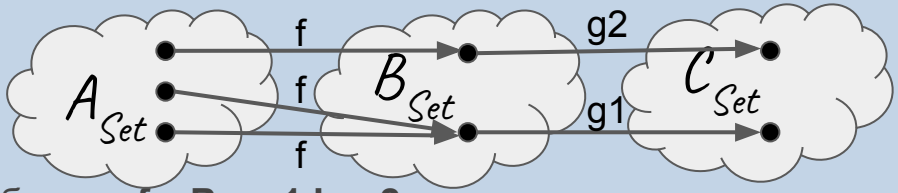
Виды морфизмов

Гомоморфизм (homomorphizm) - морфизм, сохраняющий ‘структуру’ объектов для которых определен. В случае конкретных категорий (категории групп, колец и т.д.) все морфизмы - это гомоморфизмы. Не все категории конкретны и потому не для любой категории можно говорить о гомоморфизме.
Мономорфизм (monomorphizm). f(т.е. инъективная функция**) -** это мономорфизм тогда, когда для 2-х морфизмов **g1** и **g2**
равенство **f** ○ **g1== f** ○ **g2** возможно только тогда, когда **g1 == g2**. Понять смысл определения на примере абстрактных категорий довольно сложно. Нас, как программистов, в первую очередь интересует категория **Set** (а точнее категория всех типов и функций) и на ее примере понять, что такое мономорфизм, значительно проще
Представим три объекта (A, B, C) в категории Set. Инъективную функцию f, и функции g1 и g2. Если функция инъективна, она всегда переводит разные параметры в разные выходные.
На картинке видно, что если **g1** и **g2** не равны, то и их композиции с **f** не будут никогда равны.



Теория Категорий

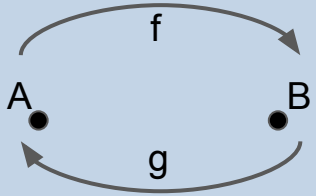
Эпипорфизм (эпичный!!)
Двойственное мономорфизму опредение.
Для того, чтобы морфизм **f(A -> B)** был эпиморфизмом необходимо чтобы выполнялось следующее: **∃ (g1, g2) : g1 == g2** для всех **f(a),**



т.е. для всех элементов **B**, которые являются образом **f** в **B** и **g1 != g2** для остальных элементов множества **B. f** эпиморфизм ⇔ **g2 ○ f == g1 ○ f** для **∀ (a ∋ A, b ∋ B , c ∋ C)**

Изоморфизм
В ТК не рассматривается понятие равенства объектов, т.к. трудно дать определение тому, что является равенством объектов. Вместо этого вводиться понятие изоморфизма. Он описывает тот факт, что из объекта **A** может быть получен объект **B** и из **B** обратно в объект **A**. Т.е. объекты **A** и **B** тождественны вплоть до некоего преобразования.

f ○ g == g ○ f
g ○ f == id_a
f ○ g == id_b
} **-** изоморфизм



Теория Категорий

Задания

- Рассмотрим категорию **Set** и в ней 2 объекта, множество всех четных чисел и множество всех нечетных. Покажите изоморфны ли эти множества. Что будут представлять собой морфизмы **h** и **g**
- Даны 2 множества - множество всех целых чисел **Z** и множество всех квадратов $\mathbf{N}^2$. Изоморфны ли они? Как могут выглядеть морфизмы?

Теория Категорий

Порядки. Морфизмы в ТК являются более общей сущностью, чем функция над объектами множества. Например существуют категории в которых морфизмом являеться факт наличия отношения между 2-я объектами. Такие категории чаще всего относятся к той или иной разновидности порядков (order)

Предпорядок(preset, preorder). Это категория в которой между 2-я произвольными объектами существует максимум один морфизм **≤**, обладающий следующими свойствами

- **Рефлексивность**: **x ≤ y => y ≤ x**
- **Транзитивность**: **x ≤ y ≤ z**

Частичный порядок(poset, partial order, thin category). Это категория, на объекты которой наложено дополнительное условие

- **Антисимметричность**: **if x ≤ y ⋀ y ≤ x => x == y**

Полный порядок (total order). Это категория в которой между каждыми 2-я объектами существует морфизм **≤,** обладающий свойствами рефлексивности, транзитивности и антисимметричности.

Теория Категорий

Задания

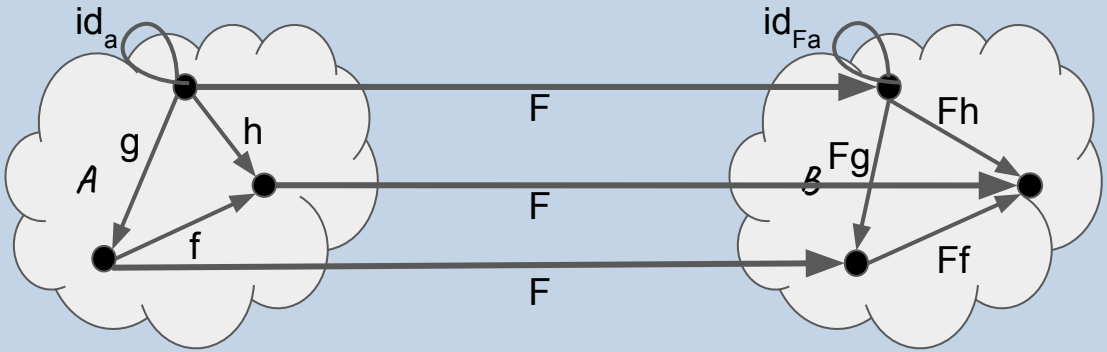
- Привести пример предпорядка, не являющегося частичным порядком
- Привести пример частичного порядка
- Привести пример полного порядка

Теория Категорий

Функтор

Функтор - это оператор, сопоставляющий объекты и морфизмы исходной и целевой категории. Функтор сохраняет структуру категории. Т.е., если морфизмы **f, g, h** удовлетворяли условию **f ○ g == h,** то функтор **F** переводит их в морфизмы, которые удовлетворяют:

- **F(f) ○ F(g) == F(h)**
- **F(Id_a) = Id_{Fa}**



Эндофунктор - это функтор, который переводит объекты и морфизмы категории в объекты и морфизмы той же категории.

Теория Категорий

Пусть существуют объекты **a** и **b** в категории C. Морфизмы a->b образуют set **$Hom_c(a,b)$**
Функтор F переводит этот set в **$Hom_c(F(A), F(b))$**

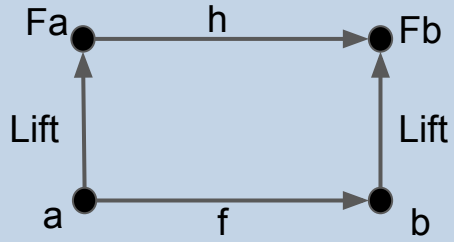
Faithful functor - это функтор для которого **$Hom_c(a,b)$** *инъективен* по отношению к **$Hom_c(F(A), F(b))$**

Full functor - это функтор для которого **$Hom_c(a,b)$** *сюръективен* по отношению к **$Hom_c(F(A), F(b))$**

Fully Faithful functor - это функтор для которого **$Hom_c(a,b)$** *биективен* по отношению к **$Hom_c(F(A), F(b))$**

Теория Категорий

Функтор(ковариантный) в программировании.
По сути функтор - это полиморфная функция, переводящая объекты в объекты, а функции в функции. Для того, чтобы быть функтором, тип должен иметь, как минимум 2 функции:
Первая функция обычно называется **lift**, она помещает объект в контекст функтора. На диаграмме ниже понятно, почему ей дали такое название.




```
def lift[A, B](fa: A): F[A]
```

Вторая обязательная функция для функтора обычно называется **map**

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

Теория Категорий

В scala функтор имеет несколько канонических форм

Одна из форм - контейнер значений, реализующий методы свойственные функтору. Например **List**, **Map**, **Option** - функторы. Сигнатуры метода **map**, тогда выглядит немного по-другому, а вместо lift применяют apply.

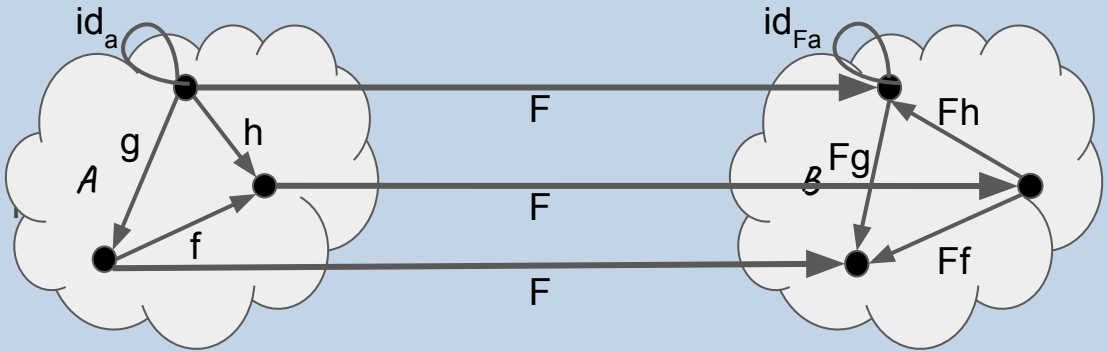
```
def map[A, B](f: A => B): F[B]

def apply(fa: A): F[A]
```

Альтернативная реализация - это type class, например, так, как это реализовано в библиотеке Cats

Поскольку в программировании мы имеем дело с категорией **Set**, все функторы - это эндофункторы.

Теория Категорий

Контравариантный функтор(контрафунктор, кофунктор) - это функтор, переводит объекты в объекты,а стрелки в обратные им стрелки



- **F(f) ○ F(g) == F(g ○ f)**
- **F(Id$_a$) = Id$_{Fa}$**
- Морфизму **f : a -> b**, кофунктор сопоставляет **F(f) = F(b) -> F(a)**

Примером контрафунктора может служить функция возведения в степень -1 в категории полного порядка, где объекты - это целые положительные числа > 0 и стрелки - это отношение ≤

Теория Категорий

Задания

- Реализуйте **lectures.cat.OptionFunctorLaw**. Докажите тем самым, что Option - это функтор
- Выполните задание в **lectures.cat.Functor.scala**
- Приведите еще пример контрафунктора

Теория Категорий

Аппликативный функтор(АФ).
АФ - это не категориальное понятие. Тем не менее эта абстракция тесно связана с категориальным функтором, но в каком-то смысле является более “мощной”

АФ в scala - это чаще всего type class, характеризующийся наличием 2-х следующих функций

```
def pure[A](a: A): F[A]

def apply[A,B](f: F[A])(ff: F[A => B]): F[B]
```

Иногда **АФ** представляют в альтернативной форме

```
def pure[A](a: A): F[A]

def map2[A, B, C](a: F[A], b: F[B])(ff: (A, B) => C): F[C]
```

Легко заметить, что функтор и Аппликативный функтор не имеют публичных методов для извлечения значений из контейнера F[_]. Они могут только добавляют еще один уровень. Именно поэтому Applicative оказыватся более мощным. Ведь превратить **f: A => B** в **f : F[A =>B]** возможно, а произвести обратное преобразование средствами одного функтора - невозможно.

Теория Категорий

Задание: lectures.cat.Applicative

Теория Категорий

Полугруппа - это **Set** вооруженный ассоциативной бинарной операцией.
Вопрос - может ли набор обладающий только ассоциативной операцией быть категорией?

Моноид - это полугруппа с единицей (нулем), в зависимости от операции.
Покажите, что моноид обладает всем необходимым, чтобы считаться категорией

Группа - это моноид, в котором для каждого элемента существует обратный

Группоид - это категория, в которой все морфизмы, являются изоморфизмами, т.е. все морфизмы обратимы

Задание: lectures.cat.MonoidLawTest

Теория Категорий

Принцип менее мощной абстракции

Мы познакомились с большим количеством функциональных паттернов, в том числе, пришедших к нам из теории категорий. Прежде чем мы двинемся дальше к изучению более “мощных” абстракций, таких как монады, например, хочется познакомить вас с **принципом менее мощной абстракции**.

Чуть позже, мы познакомимся с понятием, монада, и узнаем, что она, в том числе, является функтором. Это вызывает соблазн везде, где можно обойтись функтором, применить монаду, “на всякий случай”. Вот где вступает в игру принцип.

Дело в том, что “мощность” большинства абстракций не дается даром. Чтобы тип данных был монадой необходимо соблюдение более строгих законов, нежели при применении функтора или моноида.

Чуть позже будет приведен пример типа данных, которые являются функторами, но не являются монадами.

Теория Категорий

Стрелка Клейсли

Часто, на практике, мы сталкиваемся с функциями, результат которых помещен в какой-то контейнер, который наделяет их дополнительным смыслом.
Например, представим, что у нас есть источник данных, пусть это будет функция **data1.**
И нам необходимо, результат этой функции обработать, но функции из **F[Result] -> F[Response],** нет, а есть только **process: Result -> F[Response]**. Такая ситуация встречается сплошь и рядом.
Достаточно, например, заменить **F** на **Try** и станет понятно, что функция обработки результатов не обязана уметь обрабатывать еще и ошибки получения этих данных.

```
def data1(): F[Result] = ???  // Unit -> F[Res]

def process(r: Result): F[Response] = ???
```

Тем не менее нам хочется получить функцию, **Unit -> F[Response].** Здесь нам на выручку и приходит **Kleisli** и его “рыбий” оператор композиции таких функции.
Вообще **Kleisli** мощный функциональный паттерн, который применяется в реализации, аппликативов, моноидов и монад.

Теория Категорий

Стрелка Клейсли. “Fish” оператор

Ниже приведен пример минимального Kleisli оператора. Fish оператором называют операцию композиции, которую в Haskell принято обозначать набором символов, похожим на рыбку

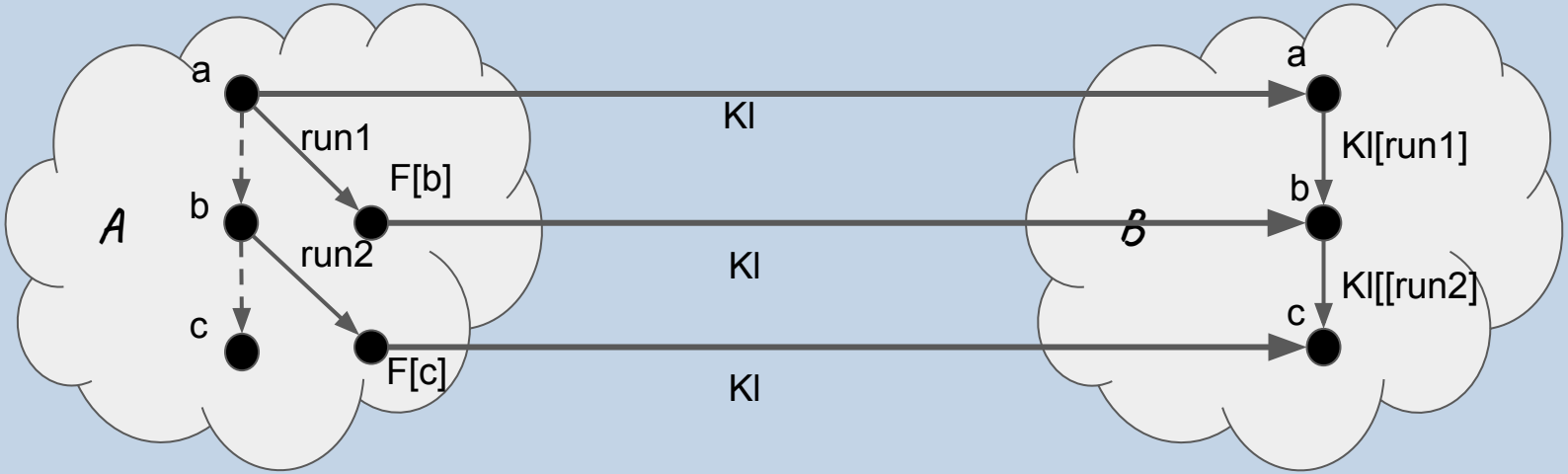
```
trait Kleisli[F[_], A, B](run: A => F[B]) { self =>

  def >=>[C](f: B => F[C]): Kleisli[F, A, C] = andThen(f)

 def andThen[C](f: B => F[C]): Kleisli[F, A, C] = ???

 def andThen[C](k: Kleisli[F, B, C]): Kleisli[F, A, C] =
   this andThen k.run

 def compose[Z](f: Z => F[A]): Kleisli[F, Z, B] = ???

 def compose[Z](k: Kleisli[F, Z, A]): Kleisli[F, Z, B] =
   this compose k.run

 def apply(a: A): F[B] = run(a)
}
```

Теория Категорий

Клейсли категория

Благодаря **Kleisli[F]** мы получили композицию функций вида **A => F[B]**. Более того Kleisli формирует структуру очень близкую к категории

- морфизм - это функция **A => F[B]**
- а композиция морфизмов - методы **compose** и **andThen**



Теория Категорий

Задание: lectures.cat.Kleisli

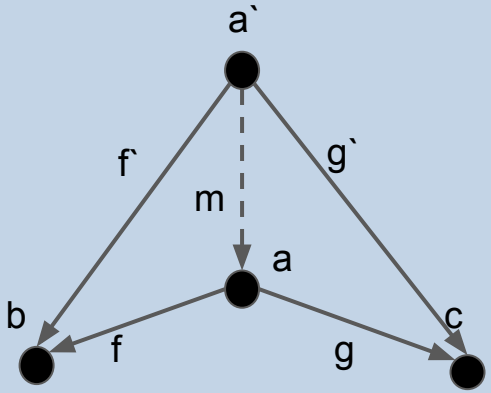
Products and Coproducts

ТК обобщает понятие декартова произведения, известного по теории множеств. Определение в ТМ декартова произведения звучит так - это множество, элементами которого являются все возможные упорядоченные пары элементов исходных множеств. Для того, чтобы понять, чем может являться декартово произведение в ТК, построим коммутативную диаграмму

Теория Категорий

Product (произведение) - это объект категории **a**, имеющий 2 морфизма **f(a -> b)** и **g(a ->c),** часто его обозначают как (**b × c**). Очевидно, что в большинстве категорий найдется много объектов, имеющих 2 морфизма. Но не все они будут произведениями. Для того, что бы стать правильным произведением, объект должен обладать следующим универсальным свойством:
Для любого другого объекта **a`**, обладающего 2-я морфизмами **f`(a`->b)** и **g`(a` -> c)** должен существовать уникальный морфизм **m(a` -> a).** Причем такой, что:

- **f ○ m = f`**
- **g ○ m = g`**



Из уравнений выше, видно, что **m** является общим множителем, который характеризует какое-то дополнительное преобразование. Универсальное свойство произведения - это категориальный способ сказать, что для того, чтобы быть произведением, морфизмы **f** и **g** не должны делать ничего, кроме как указывать на объекты **b** и **c**

Теория Категорий

Product

В scala есть тип **Tuple2[A, B](),** который является воплощением категориального произведения. Произведения могут быть и большей размерности (3, 4 и т.д.). Они все должны удовлетворять универсальному свойству для всех своих морфизмов.
Категории “оборудованные” произведениями, называют Cartesian Monoidal Category

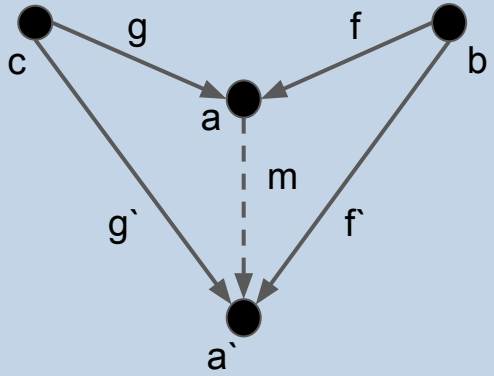
Coproduct

Благодаря свойству дуальности, у любой категориальной конструкции есть свое “альтер эго” полученное путем инвертирования морфизмов. У произведения это копроизведение или категориальная сумма.

Объект **a** - копроизведение, если существуют 2 морфизма **g(c ->a)** и **f(b -> a),** удовлетворяющие универсальному свойству.
Для любого объекта **a`** и морфизмов **g`** и **f`** существует уникальный морфизм **m,** такой что **g**

- **m ○ g = g`**
- **m ○ f = f`**

Копроизведение часто обозначают как **c ⊕ b**



Теория Категорий

Coproduct

По сути универсальное свойство копроизведения говорит, что для объекта **a** должны существовать морфизмы из объектов **b** и **c.** И эти морфизмы не должны делать ничего, кроме как помещать объекты в копроизведение. Их, поэтому, част называют injectors или canonical injectors

В scala примером копроизведения является тип Either[+A, +B]

Если проводить аналогию с теорией множеств, и считать объекты **b** и **c** множествами, то объект **a** будет размеченным объединением этих множеств

Функциональный объект. Экспонента

В функциональных языках, функция может быть передана в другую функцию как значение.
В ТК это значит, что должны существовать категории, в которых функцию являются объектами. Это так называемые “функциональные объекты”
ФО, должен обладать универсальным свойством, описанным коммутативной диаграммой ниже.

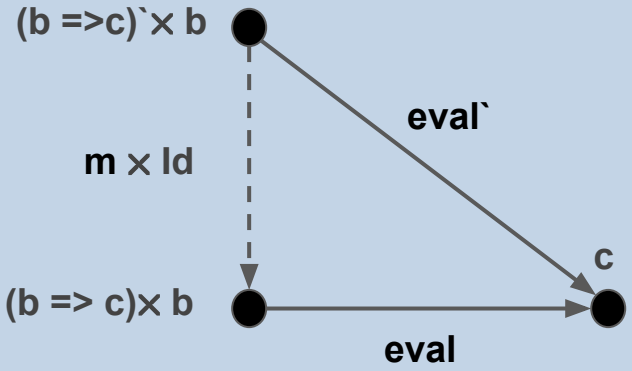
Теория Категорий

ФО. Экспонента

Объект **b => c** является **ФО**, тогда и только тогда, когда

- существует морфизм **eval ((b => c)× b) -> c**
- для любого другого морфизма **(b =>c)` × b** существует уникальный морфизма в ФО **(b => c)× b**

Смысл этого свойства в том, что в категории должен существовать морфизм, который не делает ничего кроме как применяет объект-параметр к функциональному объекту



Функциональный тип еще часто называют экспонентой и записываю как b^a . Это легко понять, если вернутся к множествам и подсчитать сколько значений может быть у этого типа, если **a** - будет, скажем, перечисление из 10 значений, а выходная строка - это тип boolean. На каждое входное значение перечисления функция такого типа может вернуть либо true либо false => всего значений 2^{10}, т.е. b^a

Теория Категорий

Терминальный и начальный объект в категории Set

Перед тем как перейти к полноценной алгебре типов нам осталось понять, что такое **1** и **0** с точки зрения теории категорий. Как мы помним, типы и функции в scala формируют категорию **Set,** т.е. тип - это множество (возможно бесконечное) всех своих значений, а морфизмы - это функции между типами.

В категории **Set** множество с одним объектом (**Unit**), является терминальным объектом т.к. удовлетворяет свойствам терминального объекта

- существует морфизм из каждого объекта категории в объект Unit
- этот морфизм **уникальный**

Unit, также, является единичным типом **(1),** мы убедимся в этом чуть позже

Начальным объектом в категории Set принято считать пустое множество **Void**. Лишь оно обладает обладает тем свойством, что из него существует уникальный морфизм к любому другому объекту. Морфизм этот - это искусственно введенная полиморфная функция **absurd,** которая принимает объект типа Void и возвращает объект нужного типа.

Теория Категорий

Полугруппа, свойства операций

Имея, определения для единицы, нуля, сложения и умножения, можно определить над типами алгебраическую структуру под названием rig (это ring без n - negative). Rig еще иногда называют полугруппой. Rig - это множество с 2-я операциями 0 и 1. В отличии от теории множеств, в ТК свойства операций в терминах изоморфизмов. Запись **a ≅ b** читается как ‘a изоморфна b’ или ‘между a и b существует изоморфизм’

- **$c \oplus b \cong b \oplus c$ (коммутативность сложения)**
- **$b \times c \cong c \times b$ (коммутативность умножения)**
- **$(a \times b) \times c \cong a \times (b \times c)$ (ассоциативность умножения)**
- **$(a \times b) \times c \cong a \times (b \times c)$ (ассоциативность слжения)**
- **$a \times (c \oplus b) \cong (a \times c) \oplus (a \times b)$ (дистрибутивный закон)**
- **$x \oplus 0 \cong x \cong 0 \oplus x$ (сложение с 0)**
- **$1 \times x \cong x \cong x \times 1$ (умножение на с 1)**

Некоторые из этих свойств доказаны в lectures.cat.RigOperationProperties

Теория Категорий

Алгебраические типы данных (ADT)

Алгебраическими типами данных называют типы

- имеющие начальные субтипы, входящие в домен алгебраических операций над типами
- образованные из субтипов, путем применения операций сложения, умножения и возведения в степень

Мы уже ни раз встречались с алгебраическими типами данных
Например, покажем, что **Option[T],** это копроизведение, т.к. можно реализовать изоморфизм с **Either[T, Unit]. lectures.cat.OptionToEitherIso**

Теория Категорий

Здания

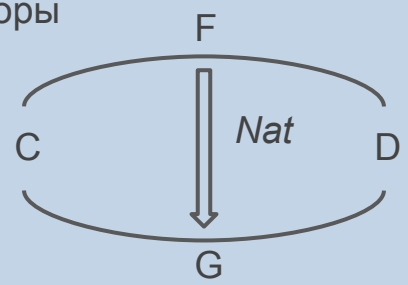
- Завершите доказательство свойств операций в **lectures.cat.RigOperationProperties,** напишите тесты на все неприватные методы объекта RigOperationProperties
- покажите, что непустой список - это произведение по аналогии с **lectures.cat.OptionToEitherIso[T]**

Теория Категорий

Естественные преобразования (ЕП, natural transformations)

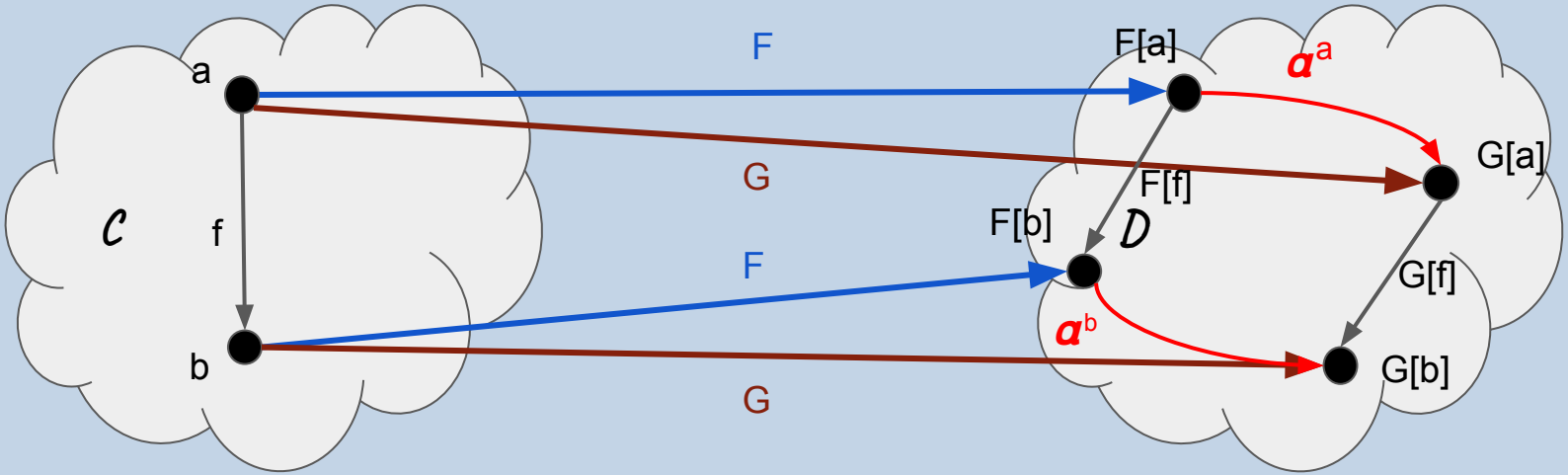
Как нам известно, функтор - это отображение объектов и морфизмов одной категории в объекты и морфизмы другой (или той же, если это эндофунктор). Такое преобразование должно обязательно сохранять “структуру” категории. Для функторов, в свою очередь, есть аналогичное отражение, одного функтора в другой, с сохранением “структуры”. Это отражение называется **естественным преобразованием**, **а** условие, обеспечивающее сохранение “структуры” - условие естественности (**naturality condition**)

Здесь **C** и **D** - категории, **F** и **G -** функторы и *Nat* - естественное преобразование



Теория Категорий

Для того, чтобы понять условие естественности, распишем коммутативную диаграмму чуть подробнее



Между функторами **F[C ->D]** и **G[C ->D]** существует ЕП, если для каждого объекта F[-] в категории D можно найти морфизм α = F[-] -> G[-], такой, что удовлетворяется следующее условие:

$\alpha^b \circ$ **F[f] = G[f]** $\circ \alpha^a$

Теория Категорий

Из диаграммы на предыдущем слайде, видно что естественное преобразование можно воспринимать, как семейство морфизмов ***α*,** которые еще называют компонентами натурального преобразования.

ЕП с точки зрения программирования
Программируя на скала мы находимся в категории **Set,** где объектами являются типы**.** В этой категории, как известно, существуют морфизмы между практически любыми объектами. Следовательно, мы всегда можем найти морфизмы **F[f]** и **G[f]**. Поэтому для того, чтобы **ЕП** существовало, нам достаточно найти все его компоненты ***α*.**

Иногда ЕП проще представить в виде функции, а не наборов морфизмов. Если вспомнить, что фунткор в скала, это полиморфная функция, чаще всего представляемая в виде трейта **trait Functor[F[_]],** то натуральное преобразование - это функция преобразования функторов, т.е **trait Nat[F <: Functor[_], G[_]<: Functor[_]]**

Теория Категорий

В библиотеке **cats**, есть трейт **cats.arrow.FunctionK[F[_], G[_]],** который по сути является “слабой” реализацие ЕП. **FunctionK** представляет собой часть ЕП, отвечающую за выбор компонентов ***α***

Теперь мы имеем в своем распоряжении:

- объекты **a,b,c,d - содержимое, данные и т.д.**
- морфизмы **a => b - способы модификации данных**
- функторы **F[A] => F[B] - способы композиции модификаций, контейнер для данных**
- и естественные преобразования **F[_] => G[_] - модификация контейнеров данных, модификация способов композиции**

Это базовые элементы из которых можно строить более сложные абстракции и целые приложения

Задание: lectures.cat.NatTransform

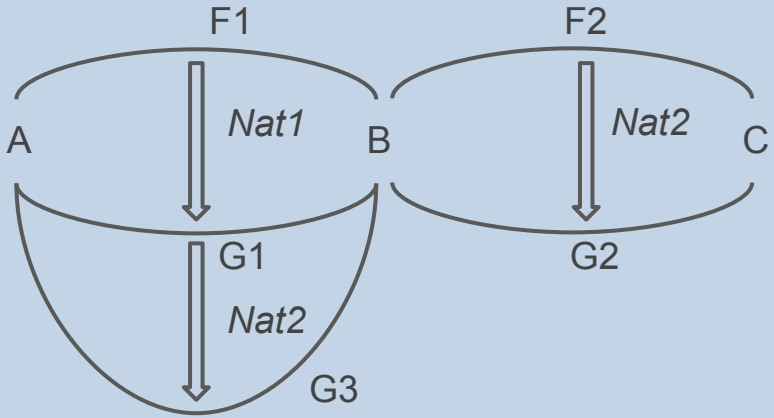
Теория Категорий

Теперь, познакомившись с естественными преобразованиями поближе, нам предстоит ответить на следующие вопросы

- что представляет из себя композиция функторов, какими свойствами обладает и как можно ее выразить
- мы знаем как поместить объект в функтор, как поместить функтор в функтор и т.д. Чего мы пока не знаем, это как сократить уровень вложенности данных. Т.е., проще говоря, как нам достать результат.

В ТК разделяют вертикальную и горизонтальную композицию функторов

- горизонтальная композиция - это композиция функторов “вдоль категорий” **F2** ○ **F1**
- вертикальная - вдоль натуральных трансформаций ***Nat2*** ○ ***Nat1***



Теория Категорий

Монада - это моноид в категории эндофункторов :))

На практике часто оказывается полезно уметь композировать применение одного и того же функтора (эндофунтора). Например, у нас есть функции **A -> Option[C]** и **B -> Option[D]**, нам хотелось бы применить какую-нибудь трансформацию к содержимому их результатов. В этом случае функция **map** функтора **Option** нам не подходит т.к. имеет сигнатуру **map(F[a])(f: A => B)** , а нам хотелось бы иметь что-то вроде **flatMap(F[a])(f: A => F[B])** или **flatMap(F[a])(f: F[A] => F[B])**

Если немного перефразировать, то нам хотелось бы иметь функторы для которых определены

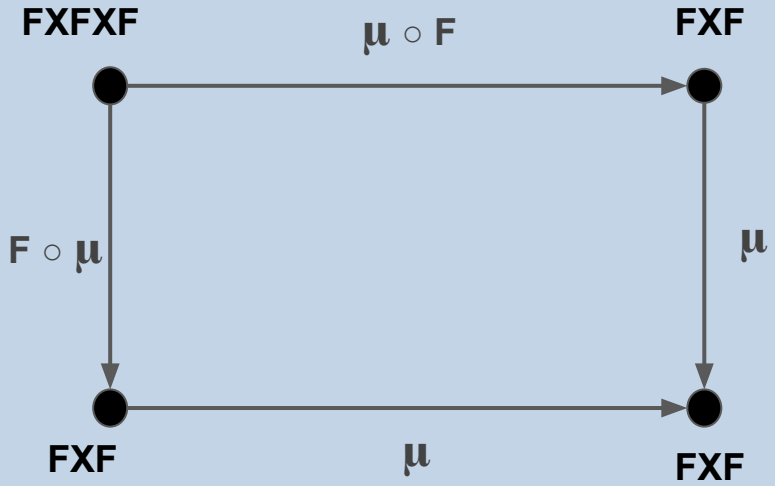
- естественное преобразование μ **= F X F -> F;** бинарная ассоциативная операция над функторами, которая представляет их композицию
- естественное преобразование η **= Id_x-> F;**
- ЕП μ и η должны обладать свойствами, выраженными на коммутативных диаграммах ниже

Перейдем в категорию **End(C),** объектами которой являются эндофункторы **F** из категории **C**, а морфизмами - натуральные преобразования (в том числе μ и η)

Теория Категорий

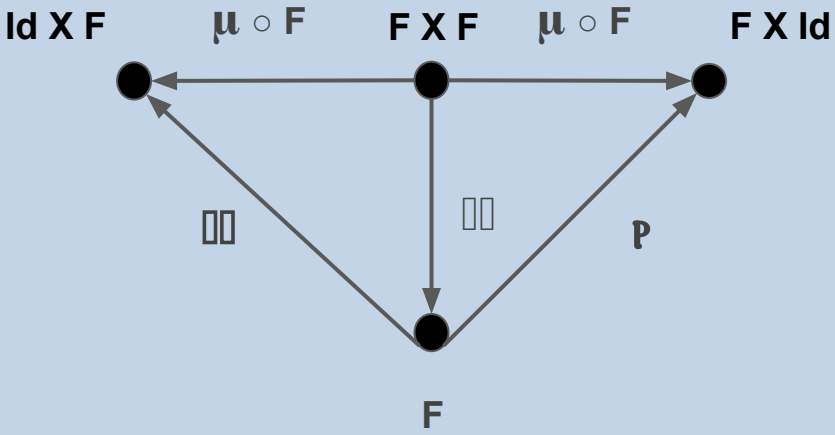
Свойства **End(C)**

- существование ассоциатора. Это естесвенное преобразование, такое, что
  α: **(F X F) X F ≅ F X (F X F)**
- левая единица λ: **(Id X F) -> F**
- правая единица ρ: **(F X Id) -> F**
- associativity square



Теория Категорий

- identity triangle



Теория Категорий

Выразить монаду на scala можно многими разными способами

Во-первых монада, это контейнер, как и функтор. В одном подходе значение этого контейнера может храниться в монаде, как это сделано для **Option** или **List.** В другом, монада может быть представлена отдельным классом, в методы которого передаются значения

Во-вторых монаду можно строить, реализуя разные базовые методы

- клейсли стрелку
- **flatMap[F[_], A, B]** и **unit[T]** : **F[T]**
- **flatten**[**F[_]**] и **unit[T]** : **F[T]**, потребовав, чтобы **F**, являлся функтором

Какой бы подход не был выбран, реализация монады должна следовать категорным законам, определенным для монад.

Рассмотрим несколько канонических представлений монад в **lectures.cat.Monad.scala**

Теория Категорий

задание: комментарий к lectures.cat.Monad.scala

Теория Категорий

Как только монады начали попадать в реальные проекты, стало понятно, что просто композиции функций, отвечающих за логику приложения, не достаточно. Почти всегда, на ряду с композицию, нужно “что-то еще”. То контекст протащить, через приложение, то логирование везде добавить, то результат побочный вернуть. Кроме этого, захотелось и с исключениями и с вводом-выводом в функциональном стиле работать. Одним их первых результатов работы, по решению вышепоставленных задач стала россыпь разнообразных монад. “Что-то еще” вшито в каждую из монад, так, чтобы программист мог сосредоточится на решении своей задачи.
Далее мы познакомимся с наиболее востребованными представителями этого функционального семейства. Можно считать их функциональными “паттернами”, ответом своим ООП-шным сородичам

Теория Категорий

Reader монада

С этим зверем мы уже познакомились в **lectures.di.reader.ReaderMonadProgram.scala.** Основная задача Reader - нести композицию функций туда, где ее не было. Вспомним, что функции вида **A => B** и **B => C,** прекрасно композируются, стандартными методами compose и andThan из Function1. А вот с композицией вида **A => F[B]** и **B => F[C]** уже сложнее. Здесь нам и приходит на помощь Reader. Достаточно, реализовать flatMap, map bind и пару других методов, для Reader[F, A, B]

```
type Id[A] = A
type ReaderT[F[_], A, B] = Kleisli[F, A, B]

object Reader {
 def apply[A, B](f: A => B): Reader[A, B] = ReaderT[Id, A, B](f)
}

final case class Kleisli[F[_], A, B](run: A => F[B]) {
 def map[C](f: B => C)(implicit F: Functor[F]): Kleisli[F, A, C] = ???
 def mapF[N[_], C](f: F[B] => N[C]): Kleisli[N, A, C] = ???
 def flatMap[C](f: B => Kleisli[F, A, C])(implicit F: FlatMap[F]): Kleisli[F, A, C] = ???
  ...
```

Теория Категорий

Writer монада

Эта монада родилась из необходимости накапливать какой-то контекст по ходу вычисления основной части программы. Часто ее применяют для накопления, например, данных для логирования.
Она оперирует с типом, вида **F(L, V)**, где **L**, это контекст вычисления, а **V** текущий результат.
Ключевым методом этой монады, является

```
def flatMap[U](f: V => WriterT[F, L, U])(implicit flatMapF: FlatMap[F], semigroupL: Semigroup[L]): WriterT[F, L, U]
```

Этот метод принимает **V => WriterT[F, L, U]**, как и осталные монады. Отличительной чертой его является то, что он берет на себя ответственность за композицию контекстов. Для того, чтобы это стало возможным, как видно из сигнатуры функции, контекст должен быть полугруппой. Проще говоря должен обладать бинарной ассоциативной операцией.

Упрощенный вариант этой монады из библиотеки cats, приведен ниже

Теория Категорий

```
final case class WriterT[F[_], L, V](run: F[(L, V)]) {

def written(implicit functorF: Functor[F]): F[L] =
  functorF.map(run)(_._1)

 def value(implicit functorF: Functor[F]): F[V] =
  functorF.map(run)(_._2)

 def ap[Z](f: WriterT[F, L, V => Z])(implicit F: Apply[F], L: Semigroup[L]): WriterT[F, L, Z] =
  WriterT(
    F.map2(f.run, run){
      case ((l1, fvz), (l2, v)) => (L.combine(l1, l2), fvz(v))
    })

 def flatMap[U](f: V => WriterT[F, L, U])(implicit flatMapF: FlatMap[F], semigroupL: Semigroup[L]): WriterT[F, L, U] =
  WriterT {
    flatMapF.flatMap(run) { lv =>
      flatMapF.map(f(lv._2).run) { lv2 =>
        (semigroupL.combine(lv._1, lv2._1), lv2._2)
      }
    }
  }

 def mapBoth[M, U](f: (L, V) => (M, U))(implicit functorF: Functor[F]): WriterT[F, M, U] =
  WriterT { functorF.map(run)(f.tupled) }
}
```

Теория Категорий

Writer монада

Рассмотрим пару примеров использования **writer**

lectures.cat.WriterTOps.scala

- **logActions - логирование с применением WriterT**
- **filterW - чуть более изощренный пример, фильтрация с WriterT**

На первый взгляд кажется, что с помощью помощью WriterT, может быть удобно реализовывать методы типа fold или aggregate. Но это не так, из-за того, что бинарная ассоциативная операция “вшита в тип” **L** и ее очень трудно переопределить.

задание lectures.cat.SeqWriterTOpsTest

Теория Категорий

State монада

Последняя из “большой тройки” монад. Она обеспечивает композицию функций вида **(S) => (S, A).** Где, **S -** состояние, а **A** текущий результат. **State,** в отличии, от **writer,** композирует функции**,** которые на вход принимают состояние. Т.е. если вычисления в приложении полностью задаются своим текущим состоянием, то это сигнал к тому, что в этом месте можно применить **state**. Иногда state применяют в качестве builder паттерна или в качестве “генераторов” каких-либо последовательностей. Например генератор случайных чисел легко можно реализовать с помощью **State Monad**.

Вариант реализации **cats.data.StateT**

Примеры применения **в lectures.cat.State.scala**