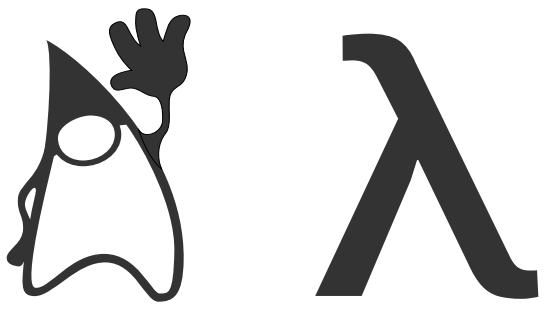
Введение в ФП

3 сентября в 11:08



Функциональные абстракции и элементы достаточно прочно вошли даже в языки, которые не рассматривались как функциональные. Все больше и больше разработчиков имеют опыт работы в функциональной парадигме. Но встречаются и те, кто хотел бы получить общее представление перед тем как пускаться в долгий путь детального изучения. Эта статья нацелена на формирование общей картины мира функционального программирования с примерами из широко распространенных языков. Статья ориентирована на разработчиков знакомых с языками базирующимися на jvm (желательно с базовым опытом Java), но должна быть полезной и остальным заинтересованным в теме.

Теоретический минимум

Функциональное программирование базируется на математическом аппарате из нескольких областей: теории множеств, лямбда-исчисления, теории групп, теории категорий и прочих. Давайте рассмотрим некоторые базовые концепции из этих теорий имеющих воплощения в программировании.

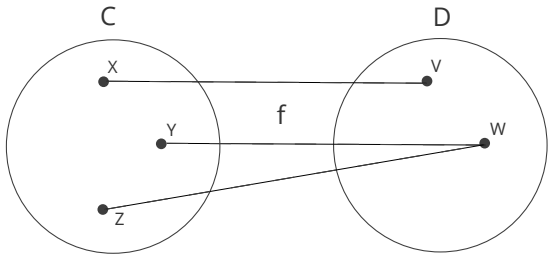
В силу своей краткости, данный теоретический минимум может иметь существенный порог входа. Если вы сталкиваетесь с упомянутыми разделами математики впервые, рекомендуется использовать этот раздел как конспект, дающий краткое описание концепций, но подразумевающий рассмотрение других более детальных источников, позволяющих двигаться мелкими шагами для лучшего усвоения материала.

Существенная часть примеров и описания - свободное изложение части лекций из курса теория категорий для программистов от Bartosz Milewski.

Функция

Под функцией будем понимать отношение двух множеств, где каждому элементу первого множества соответствует строго один элемент второго множества (функция f как отношение множества C к множеству D).

Это означает, что для определенного аргумента функции, всегда определен строго один результат (мономорфизм). Другими словами "чистая" функция не может иметь какое-либо скрытое состояние влияющее на результат. В тоже время допустимо чтобы несколько элементов первого множества соответствовали одному элементу второго (эпиморфизм, Y и Z аргументы для f дают W).



В математике, функция определяет результат, отсутствует понятие вычисления. В программировании функция имеет время вычисления и может не завершится. В большинстве концепций далее мы будем игнорировать незавершающиеся функции.

Композиция

Композиция это объединение двух и более объектов. Вероятно для программистов, наиболее очевидный пример это композиция функций, когда мы получаем новую функцию путем объединения двух и более функций через передачу результата первой в качестве аргумента второй и так далее. Но в общем случае согласно теории категорий мы говорим о композиции как об объединении объектов, где объектами могут выступать различные сущности, например морфизмы.

Мы сказали, что композиция это объединение двух и более объектов. Это достигается за счет объединения двух объектов и далее использования результата для объединения с третьим, при большем количестве объектов мы повторяем действие.

Для последующей беседы нам будет важно каким образом мы можем выбирать пары для композиции и как это влияет на результат. Поэтому стоит четко понимать разницу между ассоциативной и коммутативной операциями.

Ассоциативность подразумевает, что при композиции нескольких объектов результат не изменяется в зависимости от того в какой последовательности выбраны пары объектов, при это объекты должны иметь строгий порядок.

Коммутативность же присуща композиции если результат остается неизменным вне зависимости от порядка и очередности выбора пар.

Например операция сложения является как ассоциативной, так и коммутативной. Конкатенация же строк является только ассоциативной, потому как если мы переместим местами части строк, результат будет иным.

Далее мы будем говорить о композиции морфизмов (часто функций). Для обозначения композиции используется небольшой круг между двумя объектами

$$g \circ f$$

, что читается как g после f. Частным примером композиции объектов является композиция функций, которую можно записать в более привычной форме как g(f(x)).

Морфизм

Понятие морфизма встречается в нескольких областях математики. Морфизм это отображение одного объекта на другой с сохранением внутренней структуры. Слишком абстрактно? Давайте рассмотрим пример.

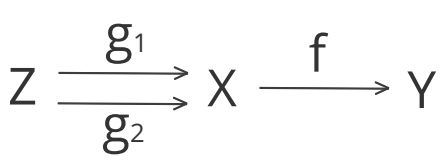
В категории множеств (что такое категория описано далее), объекты представлены множествами, а морфизмы функциями. Структура объектов в этом случае достаточно проста, все элементы множества имеют одно отношение - включение в множество. Далее рассматривая различные категорий мы увидим примеры сохранения других типов внутренних структур.

Стоит отметить, что морфизм в теории категорий это именно отношение двух объектов (направленная связь). И только лишь, в частном случае некоторых категорий, морфизмы рассматриваются как функции.

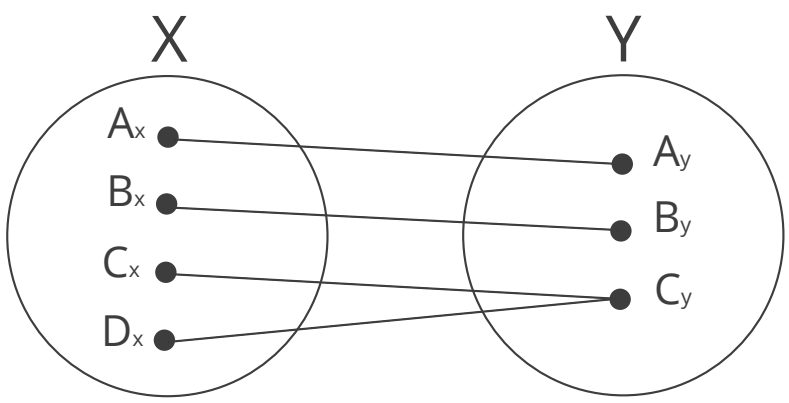
Описывая концепцию функции выше, были упомянуты мономорфизм и эпиморфизм. Рассмотрим оба типа морфизмов более детально.

Эпиморфизмом называется такой морфизм f между объектами X и Y, что для любого Z и всех морфизмов g1 и g2 верно следующее:

$$g_1 \circ f = g_2 \circ f \Longrightarrow g_1 = g_2$$



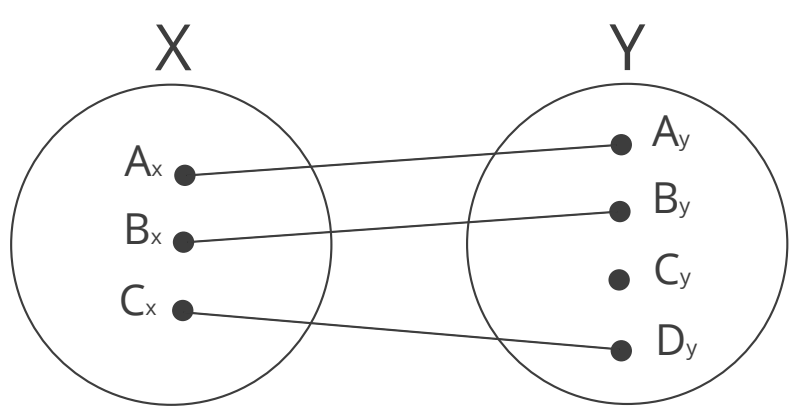
Эпиморфизм это обобщенный аналог понятия сюръективной функции (для каждого элемента множества X есть хотя бы один соответствующий элемент множества Y).



Мономорфизмом называется такой морфизм f между объектами X и Y, что для любого Z и всех морфизмов g1 и g2 верно следующее:

$$f \circ g_1 = f \circ g_2 \Longrightarrow g_1 = g_2$$

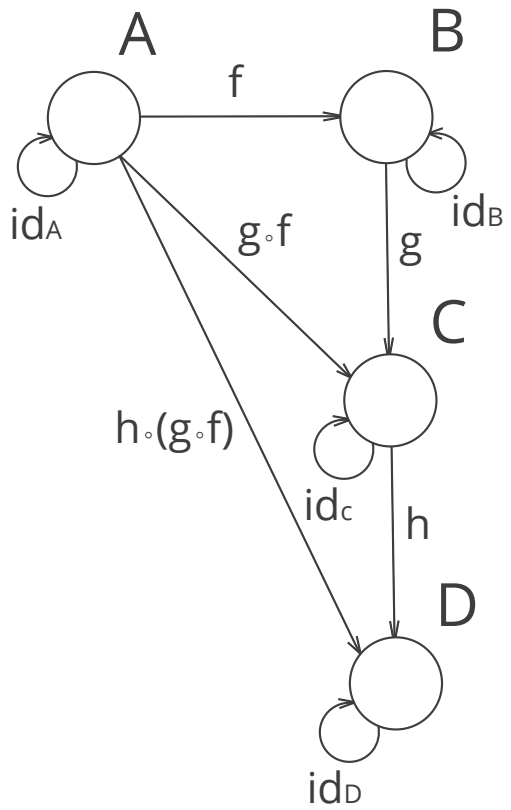
Мономорфизм это обобщенный аналог понятия инъективной функции (для каждого отдельного элемента множества X есть отдельный элемент множества Y).



Изоморфизм есть обратимый морфизм (возможно однозначно сопоставить объекты с обоих сторон). Любой изоморфизм одновременно является эпиморфизмом и мономорфизмом. Изоморфные функции позволяют транслировать аргумент и результат друг в друга, другими словами являются обратимыми.

Категория

Категория это множество объектов с определенными между ними морфизмами (стрелками или связями). Каждый морфизм определяет связь только между двумя объектами. При этом возможен морфизм с определенного объекта на него самого, тогда он называется тождественный (identity). Для пары морфизмов имеющих общий объект, определена композиция. Операция композиции должна быть ассоциативна.



Ассоциативность композиции

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Композиция с тождественным (identity) морфизмом

$$f \circ id_A = id_b \circ f$$

Еще раз следует повторить, что в теории категорий морфизм абстрактное понятие, означающее направленную связь двух объектов. И рассмотренный выше пример морфизма как функции между двух множеств, является частным случаем в категории множеств.

Важно понимать, что теория категории не изучает, что происходит внутри объектов, значения имеют только связи между ними и характер этих связей. Внимательный читатель может спросить: "Как же так, ведь мы говорили об объектах как о множествах?". Но это был лишь частный случай, который наиболее близок и понятен программисту.

Давайте рассмотрим пару примеров, что может быть описано как категория.

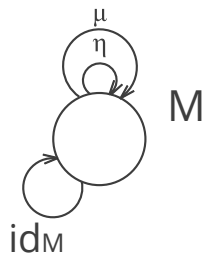
- Направленный граф, где все связи между узлами являются морфизмами, который необходимо расширить новыми связями через композицию морфизмов, то есть соединить все узлы из и в которые можно добраться по направленным связям. В добавок нужно не забыть, что в категории у нас есть identity морфизмы.
- Предпорядок (preorder) это рефлексивное и транзитивное бинарное отношение на множестве. Для тех кто не знаком с отношением порядка, предпорядок представляет собой просто отношение меньше или равно на элементах множества, то есть

$$a \leq b \leq c$$

Моноид

В теории множест, моноид - это множетво с определенной бинарной ассоциативной операцией и единичным элементом (unit). В теории групп - полугруппа с нейтральным элементом. А в теории категорий моноид - это категория с одним единственным элементом и определенными над ним морфизмами μ (умножение, multiplication, он же бинарная ассоциативная операция) и η(единица, unit). То есть, разделы математики рассматривают одну концепцию под разными "углами".

$$\mu : M \otimes M \to M$$
$$\eta : 1 \to M$$



С какой стороны мы бы не смотрели это всегда некий единый объект (единый тип с двумя определенными операциями). Снова звучит слишком абстрактно? Давайте рассмотрим несколько примеров моноидов в формате "сущность/тип, бинарная ассоциативная операция и единичный элемент".

- Целые числа, операция сложения и 0
- Целые числа, операция умножения и 1
- Строки (как список символов), операция конкатенации и пустая строка
- Более общий вариант строк, список с элементами любого типа, операция конкатенации списков и пустой список.

Хотя операции сложения и умножения коммутативны, данное свойство не является требуемым для моноида. Что мы и видим в случае со списками, где операция конкатенации списков ассоциативна, но не коммутативна.

Начальный и терминальный объекты

Начальный объект (initial object) - это объект который имеет один и только один морфизм к каждому объекту категории. Данное определение не гарантирует уникальность такого объекта, но можно говорить об уникальности согласно изоморфизму. Уникальность согласно изоморфизму подразумевает, что два объекта являются идентичными по наличию идентичных морфизмов от них (как в случае с начальным объектом) и/или по направлению к ним. Терминальный объект (terminal object) - это объект который имеет один и только один морфизм от каждого объекта категории. Аналогично предыдущему утверждению, это гарантирует уникальность согласно изоморфизму.

Для определения начального и конечного объектов используется так называемая универсальная конструкция. Она определяет "лучший" объект, который является единственным согласно изоморфизму. Ранжирование для определения "лучшего" объекта осуществляется путем утверждения наличия определенных морфизмов, как в случае с начальным объектом (речь идет о морфизмах от него ко всем остальным объектам категории), так и в случае с терминальным объектом (морфизмами от всех объектов категории к нему). В контексте универсальной конструкции (мы определяем объекты и отношение между ними с помощью морфизмов), как и в любой категории, присутствует дуализм. Любая категория может иметь обратную, такую в которой все морфизмы инвертированы. Начальный объект является конечным объектов в обратной категории. Дуализм свойственен всем сущностям теории категорий. Однако нужно понимать, что далеко не все концепции описываемые в теории категорий получили воплощение в виде инструментария в программировании. К дуализму мы еще вернемся.

Определения концепций в теории категорий, как обычно абстрактны и обобщают более частные понятия. Рассмотрим примеры как концепции начального и терминального объектов могут выглядеть в других разделах математики (не теории категорий) и программирование.

Начальный объект в теории категорий это обобщение пустого множества из теории множеств. Любое

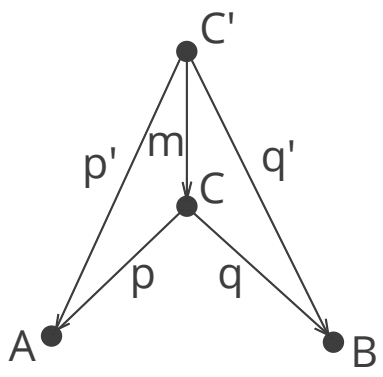
множество является производным от пустого множества путем добавления элементов, что в свою очередь является ни чем иным как морфизмами (добавление каждого элемента это морфизм в категории множеств). В теории типов, начальный объект это тип у которого нет значений. Тип `Void` в java можно рассматривать как начальный объект в контексте отсутствия значений (но будьте внимательны в случае рассмотрения `null` как значения типа `Void`, при таком взгляде на `Void` этот тип уже может рассматриваться как терминальный объект (смотри далее). При другом "угле" зрения, `null` в java можно воспринимать как начальный объект, но уже не в категории типов, а в категории инстансов (категория в которой объектами являются не типы, а инстансы). `Nothing` в Scala и `Void` в Haskell тоже являются воплощением концепции начального объекта в категории типов. Приведенные примеры еще раз подчеркивают абстрактность теории категорий. В случае с категориями типов и инстансов мы можем видеть концепцию начального объекта в разных воплощениях. Часто именно абстрактность и вызывает сложности в освоении теории категорий у программистов, особенно на начальном этапе. Потому, как это создает путаницу с кажущимися несвязанными примерами. Все дело в абстрактности концепций, которая не сразу очевидна из примеров имеющих на первый взгляд мало общего. В том числе поэтому, может быть полезно изучать базовые концепции из теории категорий, а не ограничиваться примерами из программирования.

Терминальный объект в теории категории это единичный тип (unit type) в теории типов. Другими словами это тип с единственным значением. В теории множеств аналогией выступает единичное множество (читатель же помнит, что можно рассматривать тип как множество значений?). Не все языки программирования имеют воплощение концепции терминального объекта в категории типов. Но некоторые поддерживают эту концепцию в явном виде, такие как `Unit` в Scala, `()` в Haskell или `NoneType` в Python. В java мы могли бы реализовать эту концепцию как `enum Unit { Nothing }` и использовать в случаях аналогичных тем, когда нам требуется тип `void` в методах. С некоторым допущением, что `null` не является объектом определенного типа. В java любой тип, который нельзя инстанциировать, можно рассматривать как терминальный объект при использовании `null` в качестве единственного значения такого типа. Попробуйте провести аналогию между следующими примерами:

```
public static Unit f(Object o) { return Nothing; }
public static Void f(Object o) { return null; }
```

Произведение

Произведение (product) двух объектов A и B есть объект C с двумя проекциями - морфизмами (projections) такими, что для любого другого C' с двумя проекциями можно определить такой морфизм m, который факторизует проекции.



Говоря о наличие морфизма m между C' и C в определении произведения, мы декларируем универсальное свойства. То есть фактически ранжируем все возможные варианты с помощью определения критериев "лучшего". Аналогично мы определяли начальный и терминальный объекты через указания критериев "лучшего". Например, если мы возьмем кортеж из трех элементов, очевидно

что существуют несколько морфизмов (m) позволяющих преобразовать его в кортеж из двух элементов (мы можем выбрать первый и второй элементы игнорирую третий, либо любую из оставшихся комбинации двух элементов). Тем самым, мы показываем, что кортеж из двух элементов "лучше" чем кортеж из трех согласно определению произведения (универсальному свойству). То есть, при наличие кортежа из трех элементов, у нас есть несколько способов реализовать две проекции из него, поэтому он всегда может быть преобразован (морфизм m) в кортеж из двух элементов.

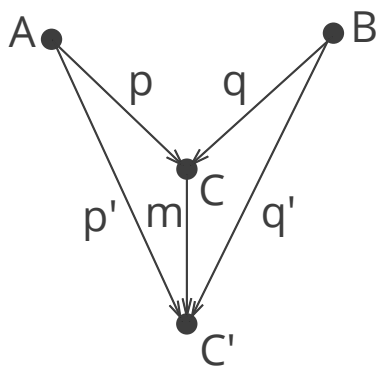
Произведение в теории категорий есть ничто иное как обобщение для декартова произведения двух множеств из теории множеств.

Примером произведения в программировании является кортеж из двух элементов (tuple, pair). Для кортежа из двух элементов можно выделить два морфизма возвращающие первый и второй элементы. Как уже упоминалось ранее, объекты в категориях (применительно к программированию) мы чаще всего будем рассматривать как типы данных. Изображенное на схеме произведение в терминах теории категории, можно трактовать как объект типа C - кортеж из двух элементов и его члены с типами A и B.

Копроизведение

Подобно присутствию дуализма для начального и терминального объектов. У произведения есть дуальная концепция - копроизведение (coproduct). Другими словами копроизведение есть произведение с инвертированными морфизмами.

Копроизведение двух объектов A и B есть объект C с двумя вложениями - морфизмами (injections) такими, что для любого другого C' с двумя вложениями можно определить морфиз m факторизующий вложения.



Примером концепции копроизведения в программировании является такой тип данных, который подразумевает возможность включение одного из двух значений разных типов. `Either` является достаточно устоявшимся названием для подобны типов. В haskell концепция копроизведения воплощена в механизме декларирования новых типов, фактически при декларировании вы можете перечислить все допустимые значения для определенного типа. Если вы не знакомы с вышеупомянутыми примерами, возможно вы знаете, что такое `union` в C/C++ - фактически это тоже копроизведение (с небольшими оговорками о том каким образом мы определяем, что в данный момент сохранено в объединении). Enum из java может служить еще одним примером копроизведения, то есть мы говорим о том, что инстанс данного типа может иметь только одно единственное значение в каждый определенный момент (два морфизма подразумевают наличия только двух возможных значений в enum, но это легко экстраполировать до любого количества значений, если представить вложенную структуру, где вместо второго возможного элемента будет новое копроизведение и так далее).

Алгебраические типы данных

Алгебраические типы данных (ADT - algebraic data types) - это общие составные тип данных

используемые в теории типов и программировании. Если мы возьмем произведение и копроизведение, то фактически сможем составить сколь угодно сложную конструкцию (тип данных), для этого мы можем составлять композицию из вложенных произведений и копроизведений.

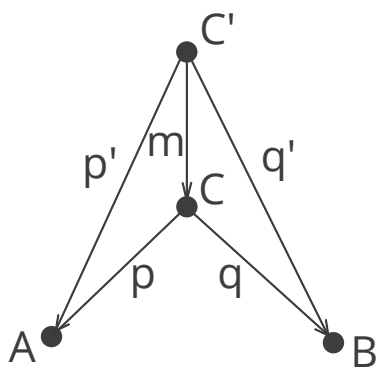
Для иллюстрации примеров с конкретными типами реализуем классы Pair и Either как два типа воплощающих концепции произведения и копроизведения из категории типов. `Pair` тип который включает в себя два значения любых заранее определенных типов, то есть существуют две функции (морфизма), которые возвращают первый или второй элемент из `Pair`. Тип `Either` описывает структуру, которая может содержать один элемент в один момент времени, при этом определено только два типа для таких элементов. Ниже приведены два примера реализации данных типов на java, в случае если вы не знакомы с аналогами из разных языков и библиотек, таких как Haskell, Scala, Scalaz, Cats и прочие.

```
class Pair<L, R> {
    private L left;
    private R right;

    public Pair(L left, R rigth) {
        this.left = left;
        this.right = right;
    }

    public L getLeft() {
        return left;
    }

    public R getRight() {
        return right;
    }
}

abstract class Either<L, R> {

    public abstract boolean isLeft();
    public abstract boolean isRight();
    public abstract L getLeft();
    public abstract R getRight();

    public static Either<L, R> left(L value) {
    return new Left(value);
}

public static Either<L, R> right(R value) {
    return new Right(value)
}

public static class Left<L, R> extends Either<L, R> {

    private L value;

    public Left(L value) {
        this.value = value;
    }

    public abstract boolean isLeft() {return true};
    public abstract boolean isRight() {return false};
    public abstract L getLeft() {return value;}
```

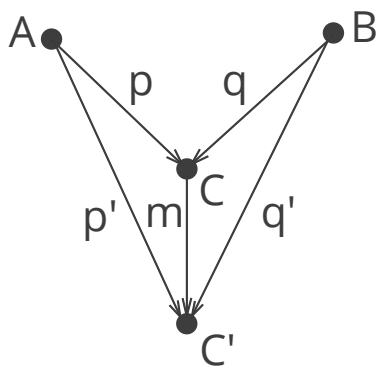
```
        public abstract R getRight() {throw new IllegalStateException()};
    }

    public static class Right<L, R> extends Either<L, R> {

        private R value;

        public Right(R value) {
            this.value = value;
        }

        public abstract boolean isLeft() {return flase};
        public abstract boolean isRight() {return true};
        public abstract L getLeft() {throw new IllegalStateException()}
        public abstract R getRight() {return value;};
    }
}
```

[Пример кода](#) выше намерено упрощен. В существующих библиотеках функционал подобных типов расширен, как минимум реализацией свойств функтора и монады (смотри описание этих концепций далее).

Рассматривая [Pair](#) и [Either](#) с точки зрения теории категории, мы можем выделить две целевые категории: категорию типов, в которой объекты являются классами java, и категорию инстансов, в которой объекты являются инстансами классов java.



`Pair<L, R>` в категории классов является объектом C, а типы L и R являются объектами A и B, а связи классов `Pair` с L И R на уровне объявления дженерик класса `Pair<L, R>` соответствуют морфизмам p и q. В категории инстансов, мы можем говорить об инстансе типа `Pair` как об объекте C, инстансы L и R будут являться объектами A и B, а морфизмами p и q являются соответственно методы `getLeft` и `getRight`, или ссылки на инстансы в атрибутах `left` и `right`.



В случае с реализацией `Either`, вложенные классы `Left` и `Right` могут привносить некоторую неясность. По аналогии с `Pair`, в категории типов, `Either` как копроизведение является объектом C из схемы, а L и R соответствуют объектам A и B. В таком случае классы `Left` и `Right` эквивалентны L и R

согласно изоморфизму, так как это всего лишь обертки и имея левое или правое значение всегда можно инстанцировать объект (в этом случае речь об объекте java) класса `Left` или `Right`, содержащий соответствующее значение, и наоборот имея такой инстанс можно извлечь значение из него. Морфизмы p и q в категории типов это связь Either с классами L и R образующаяся при объявлении дженерик класса, аналогично тому что было описано для `Pair`. То есть, на уровне типов `Either` может быть инстанцирован как `Left` или как `Right`. В категории инстансов, инстансы классов L и R являются объектами A и B, а инстанс класса `Either<L, R>`, является объектом C. Статические методы `left` и `right` позволяющие инстанцировать соответсвующий объект класса `Either<L, R>` являются морфизмами p и q.

В java обычно не говорят о конструировании типов, то что описано выше с точки зрения категории типов на уровне декларирования типов. То есть в системе типов java отсутствует подобный встроенный механизм, поэтому выше обозначается связь между классами, которая не имеет никакого формального названия (в примере с `Either` не стоит путать данную связь с наследованием `Left` и `Right`, хотя некая аналогия прослеживается, но все же мы говорим о связи классов `Either` с L и R на уровне декларирования дженерик `Either<L, R>`).

Для формирования интуиции, давайте рассмотрим еще один пример. Представьте, если бы нам необходимо было реализовать тип Boolean на java. Мы могли бы осуществить это как `enum Boolean { TRUE, FALSE }`. Переменная такого типа может иметь только одно из двух доступных значений в один момент времени. То есть присутствуют два морфизма `Boolean.TRUE` и `Boolean.FALSE` и единственное значение объекта типа Boolean. Это формулировка в рамках категории инстансов. Но если бы `TRUE` и `FALSE` сами были бы классами, по аналогии с `Left` и `Right` из `Either`, но с единственными значениями, то можно было бы говорить о конструировании типа `Boolean` из значений типов `TRUE` и `FALSE`.

В таких языках как haskell, концепция копроизведения воплощенная в механизмах декларирования типов является крайне полезной. Например вы можете конструировать типы явно поддерживающие индикатор отсутствия значения, как значение данного типа. Тоже самое как если бы `null` в java имел конкретный тип. Например значения `NullableInteger` включали в себя все `Integer` и `null`.

Если это ваше первое знакомство с теорией категорий, на этом моменте вы можете чувствовать, что теряете понимание сути концепций. Не стоит отчаиваться, так как данное введение содержит очень краткое описание и поэтому достаточно емкое. Как следствие, движение по материалу происходит в большом темпе который может ввести к недопониманию. Попробуйте рассмотреть другие источники по теории категорий, и возвращаться в данному теоретическому минимуму как к шпаргалке.

Как же быть если необходимо описать тип данных, который содержит более двух элементов? В примерах выше мы уже касались ответа на этот вопрос, но давайте повторим. Несколько элементов можно поддержать путем составления композиции из вложенных произведений, тем самым обеспечивается любое количество элементов как в следующем примере: `Pair(1, Pair("2", Pair(3, "4")))`. Подобное же применимо к копроизведениям или любым их комбинациям с произведением. Разумеется, это не всегда практично с технической точки зрения, но в данный момент мы смотрим на конструкции в концепциях теории категорий.

Вы спросите, а в чем заключается алгебраичность. Алгебра в общей своей форме есть раздел математики, где вводятся абстрактные элементы обозначающиеся буквами, что позволяет исследовать обобщенные операции и их свойства. Как пример это обобщение арифметики в виде уравнений. В более частном случае алгебарми называют математические структуры с определенными операциями сложения и умножения, также в некоторых случаях определяя требования ассоциативности и/или коммутативности. Типы произведение и копроизведение как составные элементы алгебраических типов данных являются аналогами логических операций "или" и "и"

согласно изоморфизму Карри-Хорварда-Ламбека. Ламбек показал связь между логикой, типизированным лямбда исчислением и декартово замкнутыми категориями. Далее аналогию с категориями множеств, по причине возможности оперировать с функциями в декартово замкнутых категориях, можно выйти на дизъюнктное объединение и декартово произведение, откуда мы можем говорить о типах суммы и произведения.

Как следствие мы можем говорить об алгебре типов.

	java/псевдокод	математическое представление	
начальный объект	void (тип, значение которого нельзя инстанцировать)	0	
терминальный объект	тип имеющий единичное значение, единичное значение	1	
копроизведение	Either(a, b)	a + b	
произведение	Pair(a, b)	a * b	
	Pair(a, Either(b, c)) = Either(Pair(a, b), Pair(a, c))	a * (b + c) = a * b + a * c	для java кода значения не в прямом смысле равные, а эквивалентные
	enum Boolean {TRUE, FALSE	2 = 1 + 1	также любое другое количество единиц для типов суммы с болшем количеством значеним
	Optional с empty() и of(a)	1 + a	единичное значение как терминальный объект, эквивалентно типу Either(a, "None")

Источник примеров в таблице: Simple Algebraic Data Types

Если вам требуется более глубокое понимание математических основ алгебры типов, то это остается за рамками данной статьи в качестве дополнительного задание. Здесь же мы ограничиваемся обзорным описанием концепций используемых в программировании. В дополнении вы можете рассмотреть логические выражения со следующими аналогиями начальный объект - ложное значение, терминальный объект - истинное значение, `Either` - логическое "или", `Pair` - логическое "и". Перед дополнительными изысканиями не забудьте заглянуть в раздел с описанием Лямбда исчисления.

Функтор

Функтор (functor) это отображение между категориями сохраняющее структуру морфизмов между объектами.

Рассмотрим две категории C и D, а также функтор F между ними. При этом

- каждый объект X категории C имеет отображаемый объект F(X) в категории D
- каждый морфизм

  $$f : X \to Y$$

  в категории C сопоставлен с морфизмом

  $$F(f) : F(X) \to F(Y)$$

  в категории D
-

  $$F(id_X) = id_{F(X)}$$

- $$F(g \circ f) = F(g) \circ F(f)$$

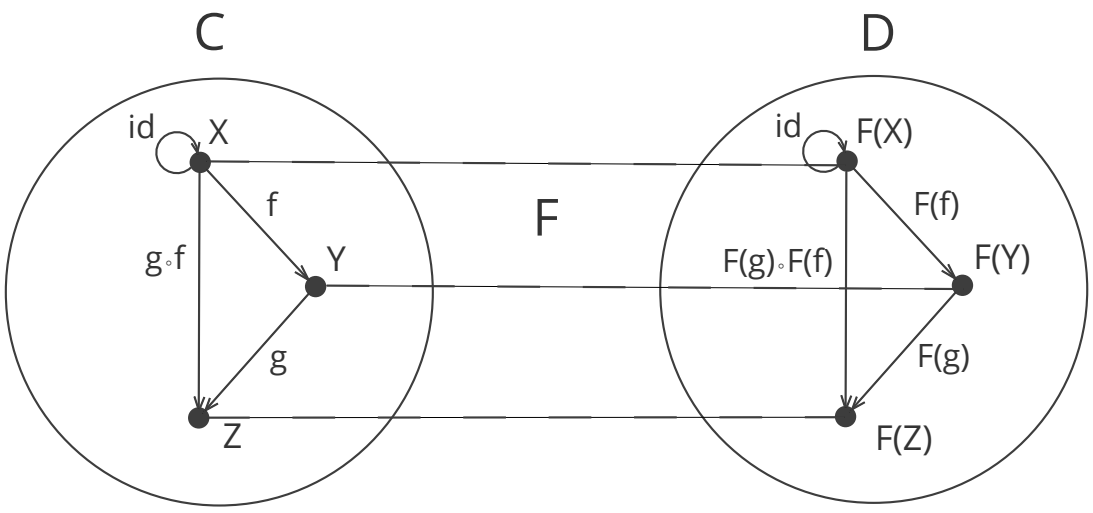
  для всех

  $$f : X \to Y$$

  и

  $$g : Y \to Z$$

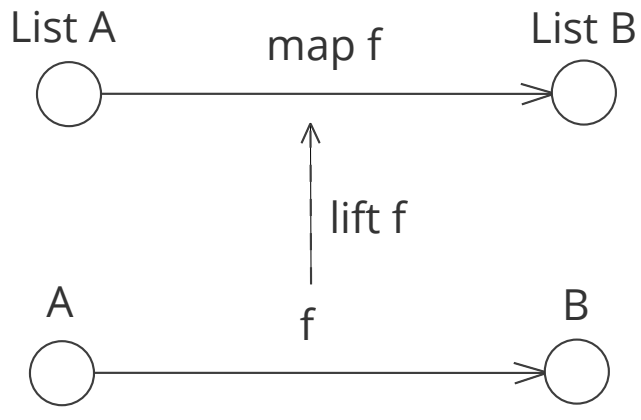
  из категории C



Выше дано определение, что функтор это отображение одной категории на другую, но также возможен функтор отображающий категорию саму на себя. Такой функтор называется эндофунктором. По причине повсеместного распространения концепции эндофукнтора в программировании, его часто называют просто функтором. Как правило в программировании используют концепцию эндофуктора на категории типов. Вероятно наиболее знакомым всем случаем реализации концепции эндофунктора является использование map (`fmap` более каноническое название для функциональных языков) над списком (коллекцией/массивом/вектором и т.д.). Список, как правило, имеет тип элементов. Как следствие map отображает список на список с изменением значений, но связь объектов в списке остается неизменной. Или в понятиях теории категорий - список это структура объектов и морфизмов в категории типов (в списке мы определяем порядок объектов, это и есть связь между ними - морфизмы), а `map` - эндофунктор.

Что делает map? map применяет функцию к каждому значению в контейнере. Стоит оговориться, что контейнер это одна из абстракций, для которых обычно применяют концепцию функтора. В более широком смысле, лучше говорить о применении функции к значению в неком контексте. В случае со списком, контекстом является структура списка. Хотя, вероятно, это не лучший пример для иллюстрации понятия контекста. Пример с `Maybe` (или `Optional` из Java, `Option` из Scala) нагляднее представляет контекст в виде наличия или отсутствия значения. Ну и еще один пример проявляющий наглядность понятия контекста, это `CompletableFuture.thenApply` из Java, `Future.map` из Scala, в которых контекст означает выполнение функции над значением, когда это значение вычислено в неопределенный момент времени по отношению к этапу вызова map.

И в завершении разговора о функторах, стоит упомянуть понятие лифтинга (lifting). Его достаточно просто разобрать через рассмотрение следующей диаграммы.



Фактически, применяя функцию через map, мы "поднимаем" эту функцию на уровень контекста.

Следующие методы являются примерами функторов из Java:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper) из Stream

<U> Optional<U> map(Function<? super T, ? extends U> mapper) из Optional

<U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn) из CompletionStage (CompletableFuture)
```

Аппликативный функтор

Концепция аппликативного функтора (applicative functor), в отличие от концепций описанных выше, изначально появилась именно в контексте программирования. Она впервые была описана в 2007 году Конором Мак Брайдом (Conor McBride) и Росс Патерсон (Ross Paterson) в их работе "Functional Pearl: applicative programming with effects".

В теории категорий аппликативный функтор это слабый моноидальный функтор. Моноидальный функтор есть не что иное как функтор между двумя моноидальными категориями сохраняющий моноидальную структуру категорий. Давайте сразу перейдем к примерам, будет намного проще разобраться с этой концепцией. А детальное рассмотрение аппликативного функтора с точки зрения теории категорий оставим за рамками этой статьи, в качестве домашнего задания для читателя.

Допустим мы имеем несколько `Optional` значений. Что если мы хотим вычислить функцию над ними при этом учитывая семантику `Optional`. Например `Optional[(a, b) -> a + b]` вычислить над `Optional[10]` и `Optional[20]`. С решеним подобной задачи в общей форме помогает концепция аппликативного функтора.

Перед тем как мы рассмотрим пример на Java, необходимо обсудить каррирование (currying). Собственно каррирование это вычисление функции с множеством аргументов как последовательности функций с одним аргументом. Каждая такая функция в последовательности возвращает очередную функцию. Частичное применение (parial application) это возможность зафиксировать значения некоторых аргументов функции и получить новую функцию принимающую оставшееся подмножество аргументов. В Java отсутствует полноценная поддержка каррирования и частичного исполнения любой функций, но это может быть реализовано через объявление типа подобного `Function<A, Function<B, Function<C, D>>>`. Примером конструкции такого типа может служить лямбда выражение a -> b -> c -> a + b + c;

Следующий пример не является идиоматической реализацией аппликатива в силу ограничений Java, но его должно быть достаточно для иллюстрации.

Перед рассмотрением примера рассмотрим его недостатки:

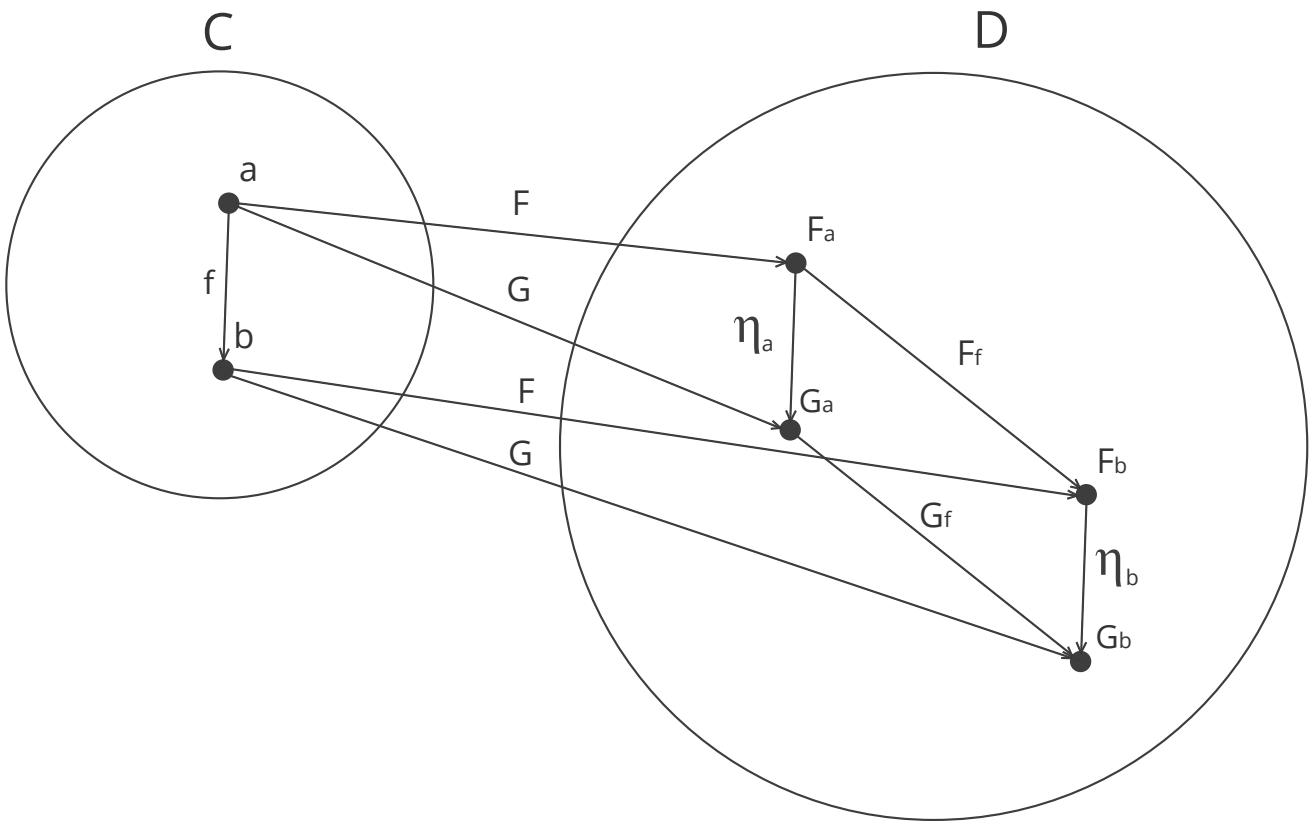
- `OptionalApplicative` не является `Optional` напрямую (`Optional` объявлен как final класс который нельзя расширять);
- `pure` принимает аргумент `Function` а не любой производный тип, что не является существенной проблемой для примера, так как `OptionalApplicative` все равно не является `Optional`;
- `OptionalApplicative` не имеет интерфейса функтора, аналогично `map` в `Optional`. Фактически мы вывернули интерфейс функтора наизнанку, функция содержится в самом `OptionalApplicative`, а `Optional` передается в качестве аргумента.

Избежать вышеописанных ограничений, позволила бы реализация полной иерархии `Functor`, `Optional` и OptionalApplicative, но это усложнило бы пример и отвязала бы его от существующих классов стандартной библиотеки.

```
public class OptionalApplicative<F, T> {
    private Optional<F> valueOpt;

    private OptionalApplicative(Optional<F> valueOpt) {
        this.valueOpt = valueOpt;
    }

    public static <F extends Function<T, ?>, T> OptionalApplicative<F, T> pure(F f) {
        return new OptionalApplicative(ofNullable(f));
    }

    public OptionalApplicative<Function<T, ?>, T> ap(Optional<T> maybeValue) {
        if (maybeValue.isPresent()) {
            return new OptionalApplicative(valueOpt.map(f -> ((Function) f).apply(maybeValue.get())));
        }

        return new OptionalApplicative<>(Optional.empty());
    }

    public Optional<F> toOptional() {
        if (valueOpt.filter(v -> v instanceof Function).isPresent()) {
            throw new IllegalStateException("Value is not calculated");
        }
        return valueOpt;
    }
}

Function<Integer, Function<Integer, Function<Integer, Function<Integer, Integer>>>> f =
    a -> b -> c -> d -> a * b + c - d;


var r = OptionalApplicative.pure(f)
        .ap(ofNullable(1))
        .ap(ofNullable(2))
        .ap(ofNullable(3))
        .ap(ofNullable(4));

assertEquals(ofNullable(1), r.toOptional());
```

Естественное преобразование

Естественное преобразование (natural transformation) это способ преобразования одного функтора в другой. В категории функторов (объекты категории функторы), естественное преобразование будет являться морфизмом.



Естественное преобразованием \eta между функторами F и G для соответствующих категорий C и D, есть множество морфизмов между объектами (Fa и Ga, Fb и Gb) в категории D, такое что для всех таких объектов выполняется

$$\eta_a \circ G_f = F_f \circ \eta_b$$

Если функтор есть не что иное, как изменение содержимого контейнера/контекста с сохранением структуры, то естественное преобразование наоборот не изменяет содержимое контейнера/контекста, а трансформирует структуру.

Рассмотрим примеры реализации концепции естественного преобразования на практике, в языках программирования. Фактически любая функция преобразующая один контейнер/контекст в другой сохраняя значения будет являться воплощением концепции естественного преобразования. Им же будет являться и трансформация любого константного значения в контейнер/контекст, потому как константное значение можно представить в виде тождественного функтора (identity functor). Следует сделать оговорку, что такая функция должна быть полиморфной (поддержка значений любых типов).

`Optional<T> findFirst()` или `Optional<T> findAny()` из `Stream` являются примерами естественных преобразований из стандартной библиотеки Java. Согласно определению естественного преобразования, очевидно, что следующее равенство верно:

```
List.of(1).stream().map(Object::toString).findFirst() = List.of(1).stream().findFirst().map(Object::toString)
```

Хотя на уровне результата обе части предыдущего равенства эквивалентны, они различаются по процессу вычисления (в правой части не требуется преобразование всех элементов). Но когда мы смотрим на данное равенство с точки зрения теории категорий, мы игнорируем разницу вычислительного процесса.

Стрелки Клейсли

Для того чтобы рассмотреть категории Клейсли математически, мы должны понимать концепцию монад. Но в то же время, интерфейс монад в языках программирования и библиотеках завязан на стрелки Клейсли (kleisli arrows). Как следствие мы рассмотрим каким образом концепция стрелок Клейсли появляется в программировании, а изучение категорий Клейсли в терминах теории категорий останется на читателя.

Рассмотрим следующий класс являющийся неким контекстом со значением произвольного типа и агрегированным списком текстовых сообщений. Цель данного класса обеспечить возможность работы с неким значением и логом текстовых сообщений без наличия сторонних эффектов. Во многих императивных реализация, логирование используется как операция со сторонним эффектом. То есть вывод происходит в некоторое состояние (буфер или напрямую устройство вывода) существующее за пределами функции. Другими словами, мы теряем ссылочную прозрачность для функции с логированием, потому как фактически результат работы функции не соответствует возвращаемому значению.

Основное внимание следует обратить на метод `flatMap`, который заботится об агрегации текстовых сообщений. Нам лишь необходимо вернуть значение из нашей функции обернутое в данный класс с помощью метода `LogAndValue<T> pure(T value, String log)`.

```
public class LogAndValue<T> implements Functor<T> {

    private final T value;
    private final List<String> logs;

    public LogAndValue(T value, List<String> logs) {
        this.value = value;
        this.logs = logs;
    }

    public T getValue() {
        return value;
    }

    public List<String> getLogs() {
        return logs;
    }

public <P> LogAndValue<P> map(Function<T, P> f) {
    return new LogAndValue<>(f.apply(value), logs);
}

public <P> LogAndValue<P> flatMap(Function<T, LogAndValue<P>> f) {
    return flat(map(f));
}

public static <T> LogAndValue<T> flat(LogAndValue<LogAndValue<T>> ll) {
    var resultLogs = new LinkedList<>(ll.getLogs());
    resultLogs.addAll(ll.getValue().getLogs());
    return new LogAndValue<>(ll.getValue().getValue(), resultLogs);
}

public static <T> LogAndValue<T> pure(T value) {
    return new LogAndValue<>(value, emptyList());
}
```

```
    public static <T> LogAndValue<T> pure(T value, String log) {
        return new LogAndValue<>(value, singletonList(log));
    }
}
```

Согласно упомянутой выше цели данного класса мы можем осуществить композицию чистых функций, действия в которых должны сопровождаться текстовыми сообщениями.

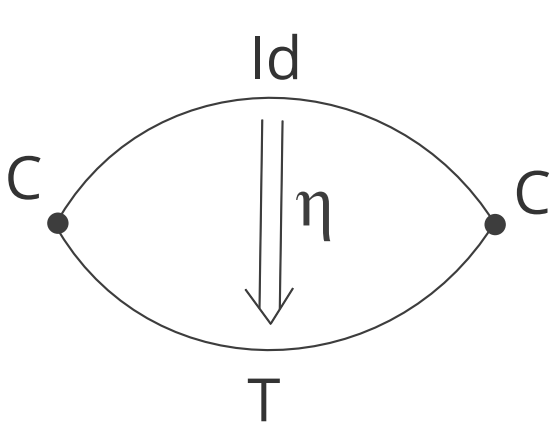
```
var r = LogAndValue.pure(10)
            .flatMap(v -> LogAndValue.pure(v + 5, "Added five"))
            .flatMap(v -> LogAndValue.pure(v - 3, "Subtracted three"))
            .flatMap(v -> LogAndValue.pure(v.toString(), "Converted to string"));

assertEquals("12", r.getValue());
assertEquals(List.of("Added five", "Subtracted three", "Converted to string"), r.getLogs());
```

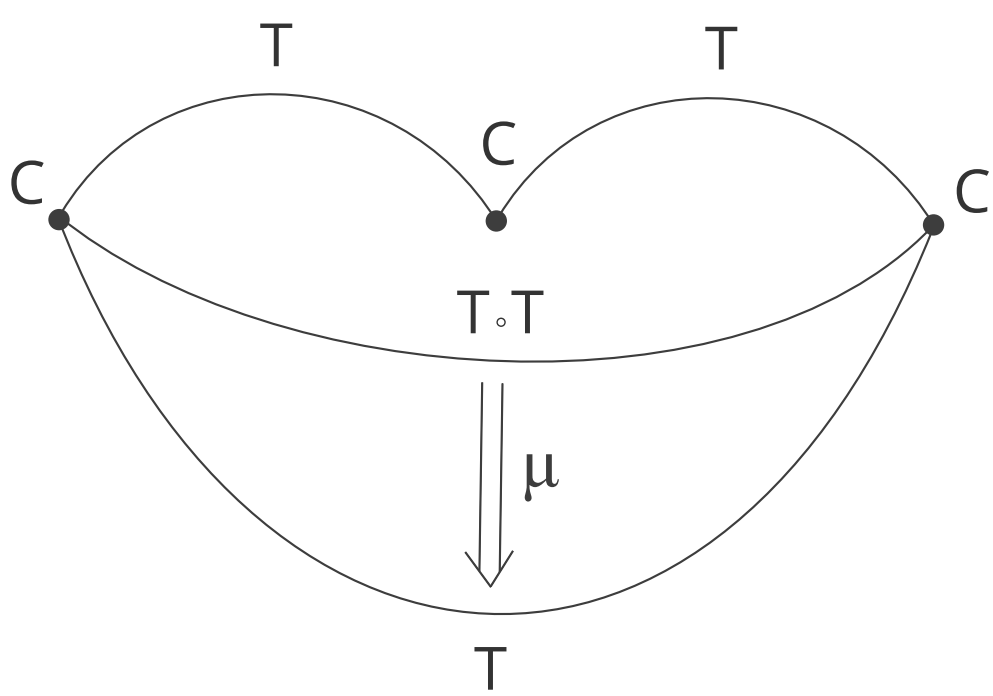
Цепочка вызовов flatMap есть не что иное, как пример стрелок Клейсли.

Монада

Монада (monad) - наверное самая популярная концепция при обсуждении функционального программирования. Мы уже рассмотрели функтор, как абстракцию позволяющую работать со значением в некотором контексте. То есть функтор позволяет изменять такое значение. Но с функтором, мы не можем влиять на сам контекст. Для решения подобной задачи и применяются монады.

В теории категорий, монада это моноид в категории эндофунторов. Звучит обнадеживающе, если необходимо отпугнуть тех кому нужно быстро понять, что это такое или тех кто не знаком с теорией категорий. Но для того чтобы быть последовательным нам все же необходимо рассмотреть математическое определение. Далее мы перейдем к примерам которые в полной мере должны пролить свет на эту концепцию.





T есть эндофунктор из категории C в C. И два естественных преобразования:

$$\eta = Id \Rightarrow T$$
$$\mu = T^2 \Rightarrow T; T^2 = T \circ T$$

Если вернуться назад и рассмотреть равенства описывающие моноид, то можно заметить явное

сходство. И даже более того, равенства описывают одну и ту же концепцию используя слегка отличные обозначения. Это и не удивительно, так как мы говорим о моноиде в категории эндофункторов. То есть в качестве объектов которыми оперирует моноид являются эндофункторы (или просто функторы как это принято в программирование).

Помимо наличия двух естественных преобразований, каждая монада должна подчиняться так называемым условиям согласованности (coherence condition).

$$\mu \circ T\mu = \mu \circ \mu T$$
$$\mu \circ T\eta = \mu \circ \eta T = 1_T$$

В стандартной библиотеке Java, как вы вероятно догадались, нет обобщенного интерфейса для монад, но есть частные реализации в виде различных классов. К тому же, в программировании распространен несколько иной интерфейс монад в отличие от описанного определения в теории категорий. Такой интерфейс базируется на категориях Клейсли.

Можно заметить, что название `flatMap` которое было использовано в примере из описания стрелок Клейсли не случайно. Согласно ему подразумевается, что метод должен делать нечто подобное map в функторе, но при этом результат должен избавляться от вложенности (`flat`, делаться плоским). Как раз об этой вложенности и идет речь в определении монады в естественном преобразовании μ.

Фактически, класс `LogAndValue` из секции про категории Клейсли, реализует интерфейс монад принятый в программировании: метод pure - как преобразование константного значения в обернутое (естественное преобразование η); метод `flatMap` - реализация которого основана на `map` и `flat` (естественное преобразование μ). Еще несколько примеров монад из стандартной библиотеки Java. Проведите аналогию между `pure` и `flatMap` из `LogAndValue`.

```
Optional
<T> Optional<T> of(T value)
<U> Optional<U> flatMap(Function<? super T, ? extends Optional<? extends U>> mapper)

CompletableFuture
<U> CompletableFuture<U> completedFuture(U value)
<U> CompletableFuture<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)
```

Условия согласованности из теории категорий преобразуются в иную форму контракта которому должна соответствовать любая реализация монады: равенства левой и правой идентичности, а также ассоциативности. Причина этому переход на контракт стрелок Клейсли с `flatMap`, вместо использования `flat` (μв математическом определении).

Давайте рассмотрим равенства на примере `Optional`.

```
Optional<Integer> m = Optional.of(1);
Function<Integer, Optional<Integer>> f = i -> Optional.of(i + 1);
Function<Integer, Optional<Integer>> g = i -> Optional.of(i + 2);
```

Левая идентичность

```
assertEquals(Optional.of(a).flatMap(f), f.apply(a));
```

Правая идентичность

```
assertEquals(m.flatMap(Optional::of), m);
```

Ассоциативность

```
assertEquals(m.flatMap(f).flatMap(g), m.flatMap(x -> f.apply(x).flatMap(g)));
```

Комонада и Стрелки Коклейсли

В описание начального и конечного объектов был подчеркнут дуализм, который проявляется для всех сущностей теории категории. Обычно дуальная сущность с инвертированными морфизмами называется аналогично основной сущности с префиксом "ко". Не трудно догадаться, что комонада (comonad) и есть дуальная сущность по отношению к монаде. Если о монаде мы говорили как о моноиде в категории эндофункторов, то комонада это не что иное как комоноид в категории эндофункторов. Инвертировав естественные преобразования описанные в предыдущей секции, получаем естественные преобразования для комонад.

$$\varepsilon = W \Rightarrow Id$$
$$\delta = W \Rightarrow W^2; W^2 = W \circ W$$

Подобно монадам, каждая комонада должна подчиняться следующим равенствам (согласно изоморфизму).

$$W\delta \circ \delta = \delta W \circ \delta$$
$$W\varepsilon \circ \delta = \varepsilon W \circ \delta = 1_W$$

Аналогично другим концепциям из теории категорий, непросто моментально построить ментальную модель концепции и как она может выглядеть в программирование. Давайте рассмотрим пример.

Неизменяемые структуры данных не позволяют легко модифицировать значения находящие внутри (представьте производную иерархию вложенных типов произведения - кортежей). Фактически, необходимо полностью пересобирать структуру данных подставляя свое собственное значение в определеную позицию (допустим изменяя значение на третьем уровне вложенности). Для решения подобной задачи часто используют концепцию так называемого зиппера (zipper, аналогия с замками молниями на одежде). Концепция зиппера заключается в том, что мы имеем возможность навигации по структуре. Текущая позиция обозначается как фокус, а сама структура разделена на две части этим фокусом. Возьмем для примера неизменяемый список. В таком случае фокус будет указывать на текущую позицию из списка, а левая и правая половина зиппера будут содержать элементы до и после текущей позицией.

Рассмотрим неизменяемый список и зиппер для него. Класс ImmutableList и его интерфейс будет несколько непривычен разработчикам привыкшим императивным реализациям. Данные класс реализует список через вложенность типа произведения (product) которым сам и является. Первый морфизм ссылается на "голову" списка, а второй на его "хвост".

```
public class ImmutableList<T> implements Functor<T> {

    public static final ImmutableList EMPTY_IMMUTABLE_LIST = new EmptyImmutableList();

    private T head;
```

```
    private ImmutableList<T> tail;

    public ImmutableList(T head, ImmutableList<T> tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public <U> ImmutableList<U> map(Function<T, U> f) {
        return new ImmutableList<U>(f.apply(head), tail.map(f));
    }

    public ImmutableList<T> add(T newHead) {
        if (newHead == null) {
            return this;
        }

        return new ImmutableList<>(newHead, this);
    }
    // ... код пропущен
}

public class ImmutableListZipper<T> implements Functor<T> {

    private final ImmutableList<T> a;
    private final T focus;
    private final ImmutableList<T> b;

    public ImmutableListZipper(ImmutableList<T> a, T focus, ImmutableList<T> b) {
        this.a = a;
        this.focus = focus;
        this.b = b;
    }

    public ImmutableListZipper(ImmutableList<T> a) {
        this(a.tail(), a.head(), EMPTY_IMMUTABLE_LIST);
    }

    @Override
    public <U> ImmutableListZipper<U> map(Function<T, U> f) {
        return new ImmutableListZipper<>(a.map(f), f.apply(focus), b.map(f));
    }

    public <P> ImmutableListZipper<P> coflatMap(Function<ImmutableListZipper<T>, P> f) {
        return coflat(this).map(f);
    }

    public ImmutableListZipper<T> forward() {
        return new ImmutableListZipper<>(a.add(focus), b.head(), b.tail());
    }

    public ImmutableListZipper<T> backward() {
        return new ImmutableListZipper<>(a.tail(), a.head(), b.add(focus));
    }

    public ImmutableListZipper<T> set(T newFocus) {
        return new ImmutableListZipper<>(a, newFocus, b);
    }
```

```
    public T focus(T defaultFocus) {
        if (focus == null) {
            return defaultFocus;
        }

        return focus;
    }

    public static <T> ImmutableListZipper<ImmutableListZipper<T>> coflat(ImmutableListZipper<T> l) {
        // ... Генерация Zipper который содержит список Zipper'ов каждый из которых имеет в качестве фокуса теку
    }

    // ... код пропущен
}
```

Метод фокус является естественным преобразованием εиз определения комонады в рамках теории категорий. coflat же является естественным преобразованием δ

И примеры каким образом можно работать со списком и зиппером.

```
var list = new ImmutableList<>(1)
                .add(2).add(3).add(4).add(5).add(6).add(7).add(8).add(9);

var zipper = new ImmutableListZipper<>(list)
        .backward()
        .backward();

assertEquals(
        "<1, 2, 3, 4, 5, 6, *7*, 8, 9>",
        zipper.toString());
assertEquals(
        "<2, 4, 6, 8, 10, 12, *14*, 16, 8>",
        zipper.coflatMap(z -> z.forward().focus(0) + z.backward().focus(0)).toString());
```

Как видно из кода, Коклайсли стрелки (Cokleisli arrows, метод `coflatMap`) позволяют получать некий контекст, извлечь из него необходимые данные (в зависимости от семантики, как в случае с ImmutableListZipper, извлекаются предыдущая и последующая позиции и вычисляется их сумма), производить манипуляции с полученными данными и возвращать результат который модифицирует контекст заранее определенным способом. Другими словами мы говорим о вычислении в контексте, с невозможностью его замены или модификации в противоположность монадам. То есть семантика контекста предопределена заранее в реализации комонады.

Линзы

Концепция линз (lens) достаточно проста, имея неизменяемую вложенную структуру данных мы должны каким либо образом извлекать из нее данные и получать модифицированные копии. Другими словами наличие неизменяемой вложенной структуры данных имеющие единственное значение выглядит малопрактичным. Используя понятие неизменяемая структура данных в данном контексте мы говорим о таких структурах, операции с которыми обладают ссылочной прозрачностью (referential transparency). Другими словами такие операции всегда имею предопределенный результат на заданный ввод. То есть отсутствует побочный эффект (side effect).

Определение линзы представляет собой две функции get и set. Первая функция get принимает в качестве аргумента - некое целое w (кортеж, контейнер и т.д.) и возвращает значение - некоторое частное (часть целого, например поле из кортежа - типа произведения) p. Вторая функция set принимает в качестве аргумента целое w и значение частного p и возвращает новое целое w содержащее уже новое значение частного p.

```
get(w) = p
set(w, p) = w
```

Но не любые такие функции будут являться линзой, для этого должны выполняться следующие правила:

- set w (get w) = id получение частного из целого и установка его же в целое идентично изначальному целому
- get (set w p) = p установка частного в целое и получение частного эквивалентно частному
- set (set w p) p' = set w p' установка первого частного и последующая установка второго частного эквивалентно установке только второго частного

Сама концепция линз кажется очень простой и не слишком полезной. Однако поддержка композиции линз может добавить возможность доступа к вложенным сущностям. Также в реальных библиотеках, существуют линзы над итерируемыми сущностями, как и механизмы для их создания могут выглядеть куда более удобными для практического использования в отличие от примера ниже. Вместо создания отдельного класса линзы для доступа к каждому полю неизменяемого объекта, мы могли бы генерировать такие классы динамически во время загрузки приложения. Синтаксис вызова get или set на линзе мог бы быть более декларативным, что предоставило бы куда большие удобства. Все это привносит практичность в конечный инструмент, который мы все еще называем линзой. Следующий пример предназначен для демонстрации изначальной концепции линзы с минимальным дополнением, поэтому он обладает ограниченной практичностью.

```
public interface Lens<W, P> {

    P get(W w);
    W set(W w, P p);

    default <P1> Lens<W, P1> compose(Lens<P, P1> l2) {
        var l1 = this;
        return new Lens<W, P1>() {
            @Override
            public P1 get(W w) {
                return l2.get(l1.get(w));
            }

            @Override
            public W set(W w, P1 p1) {
                return l1.set(w, l2.set(l1.get(w), p1));
            }
        };
    }
}
```

```
@Data
@RequiredArgsConstructor
class Address {
    private final String street;
    private final String premise;
}

@Data
@RequiredArgsConstructor
class Person {
    private final String name;
    private final Address address;
}

class StreetAddressLens implements Lens<Address, String> {
    @Override
    public String get(Address address) {
        return address.premise;
    }

    @Override
    public Address set(Address address, String p) {
        return new Address(address.street, p);
    }
}

class AddressPersonLens implements Lens<Person, Address> {
    @Override
    public Address get(Person person) {
        return person.address;
    }

    @Override
    public Person set(Person person, Address p) {
        return new Person(person.name, p);
    }
}

var person = new Person("George", new Address("W 23 St.", "10"));

var streetLens = new StreetAddressLens();
var addressLens = new AddressPersonLens();

assertEquals(
    new Person("George", new Address("W 23 St.", "26")),
    addressLens.compose(streetLens).set(person, "26")
);
```

В примере выше, с помощью линз мы абстрагировали работу с неизменяемыми вложенными сущностями. В противном случае модификация поля из вложенной сущности выглядела бы как пересоздание иерархии сущностей с измененным полем. В таком коде, это вероятно выглядело бы удовлетворительно. Но если говорить о большем количестве уровней вложенности или наличие коллекций, то подобный механизм существенно облегчает задачу.

Мы рассмотрели линзы. Они действуют над типом произведение (product), когда мы говорим о том, что выбираем из целого частное, мы фактически получаем одно из значений произведения (кортежа).

Существует аналогичная концепция над типом копроизведения (coproduct), которая называется призма (prism). Если в линзе мы всегда можем получить значение, то в призме мы можем получить одно из поддерживаемых значений. Частный пример призмы является получение внутренного поля как `Maybe` (`Optional` в Java). Так как `Maybe` есть ни что иное как копроизведение с двумя значениями: 1. непосредственно само значение 2. пустое значение.

Линзы, призмы и функциональность выросшая вокруг этих концепций называют оптикой (Optics). Дальнейшее углубление в оптику за рамками данной статьи и остается на усмотрение читателя.

Лямбда исчисление

Лямбда исчисление это теория разработанная Алонзо Черчем для описания универсальной модели вычисления и исследования вычислимости. Базовый аппарат теории составляют несколько простых концепций: переменная, абстракция, аппликация и выражение.

Переменная - символ обозначающий параметр или некоторое значений (аналогично тому, что принято в уравнениях).

Абстракция - сущность определяющая некую функцию `λx.fx`, что означает функцию одного аргумента x выполняющую некоторое действие `f` над `x`.

Аппликация - это аналог вызова функции с аргументом `f(a)`, но нотация записывается без скобок `f a`. Аппликация может трактоваться как вычисление алгоритма `f` с входным параметром `a`. Данная нотация подразумевает только унарную операцию, то есть, допускается единственный аргумент. При этом в качестве результата аппликации возможно возвращение абстракции, как следствие использование второго и последующих аргументов. Данный процесс называет каррирование (currying). То есть имея выражение `λa.λb.a+b` мы можем получить аналог функции двух переменных `f a b`. При этом допускается сокращать нотацию из нескольких лямбда выражений и записывать `λab.a+b`, но это лишь синтаксический сахар, логически выполнение должно происходить согласно исходной семантике.

Выражение - описывает последовательность действий с помощью произвольных математических операций, абстракций и аппликаций. Скобки могут использоваться для изменения приоритета операций. В случае когда выражение со скобками имеет тот же смысл, что и выражение без скобок, они могут быть опущены.

Весь синтаксис лямбда вычислений описывается как

```
expression = variable | expression expression | λ variable . expression | (expression)
```

Помимо синтаксиса определены следующие аксиомы

α-эквивалентность

`λx.x` эквивалентно `λy.y`, то есть в лямбда выражение допускается замена имени переменной.

β-редукция

`(λx.fx) a = fa`, то есть мы подставляем некую константу в качестве аргумента лямбда выражения и можем раскрыть его.

В случае когда для выражение невозможно применить β-редукцию, такое выражение называется β-нормальной формой.

η-редукция (преобразование)

`λx.fx` и `f` эквивалентны если и только если они дают одинаковый результат для любого возможного аргумента.

Несмотря на кажущуюся простоту, лямбда исчисление является тьюринг полным. То есть с помощью лямбда исчисления возможно представить любую вычислимую функцию/алгоритм. Следующая пара примеров показывает как с помощью лямбда исчисления можно описать арифметику и логические выражения, что иллюстрирует мощность нотации.

```
0 := λf.λx.x
1 := λf.λx.f x
2 := λf.λx.f (f x)
3 := λf.λx.f (f (f x))

PLUS := λm.λn.λf.λx.m f (n f x)
MULT := λm.λn.λf.m (n f)
```

Подставляя в переменные m и n натуральные числа представленные в виде лямбда выражений и раскрывая их согласно β-редукции, мы получаем конечный результат в виде лямбда выражения соответствующего определенному натуральному числу.

Аналогично применяя β-редукцию для логических операция представленных ниже, мы получаем операции булевой алгебры.

```
TRUE := λx.λy.x
FALSE := λx.λy.y

AND := λp.λq.p q p
OR := λp.λq.p p q
NOT := λp.p FALSE TRUE
```

Y-комбинатор

Комбинатором называется выражение использующее аппликации и абстракции все переменные которых связаны, то есть каждая используемая переменная используется как аргумент некого выражения (λx нотация).

Указывая на то, что лямбда исчисление является тьюринг полным, вы могли спросить каким же образом осуществлять повторяющуюся операцию - циклы или рекурсию. Ответ на этот вопрос кроется в Y комбинаторе.

```
Y = λf.(λx.f(xx))(λx.f(xx))
```

Раскрывая лямбда выражение по f мы получим бесконечную последовательность вложенных вызовов f. Разумеется используя абстракцию с двумя аргументами, мы можем реализовать счетчик итераций и завершать их по достижению определенного предела.

Это был краткий обзор основ безтипового лямбда исчисления, но этого достаточно для целей статьи. Разумеется, изучение теории остается на усмотрение читателя.

А что же на практике?

Функциональное программирование это парадигма базирующая на отсутствие глобального состояния и композиции функций. Это фундамент катализируют декларативный подход в формировании структуры программы. То есть, программа является выражением логики вычислений, нежели конкретных инструкций изменяющих глобальное состояние с последовательным порядком действий. Другими словами, мы компонуем логику вычислений и затем эта логика должна быть вычислена неким внешним вычислителем/исполнителем. Как простейший пример, можно рассмотреть композицию функций задающую логику вычисления, а вызов такой композиции с конкретным аргументом и будет являться внешним вычислителем/исполнителем. Если говорить о чисто функциональном языке, то он не должен позволять использовать/изменять глобальное состояние. Любая программа в таком случае должна выглядеть как некая большая композиция функций/некий пайплайн с входным аргументом и возвращаемым значением. Вы спросите каким образом это возможно? Ведь необходимо, как минимум, иметь способ взаимодействовать с внешним миром во время работы программы. Ответ на этот вопрос кроется в применении концепции монада (смотри выше). Но все по порядку.

Web-crawler

Давайте реализуем некий сервис - web-crawler, который принимает запросы с указанием какие веб страницы необходимо обойти и возвращает результат с их содержимым, включая обход ссылок на дочерние страницы. Рассмотрев реализацию на нескольких языках программирования, мы рассмотрим каким образом концепции описанные выше (смотри теоретический минимум) воплощаются как в различных языках программирования и фреймворках.

Web-crawler на Java

Для начала мы должны определить каким образом пользователи нашего сервиса смогу запрашивать обход веб-страниц. Объявим класс CrawlingRequest со следующей структурой:

```
public record CrawlingRequest(
        Map<String, String> urls,
        String childrenPattern,
        Integer childrenLevel) {}
```

где

- `urls` содержит url для обхода и их идентификаторы (произвольные строки)
- `childrenPattern` регулярное выражение ограничивающее url для дочерних ресурсов
- `childrenLevel` ограничивает максимальный уровень обхода вглубь по дочерним ресурсам

record в Java это не что иное как воплощение концепции произведения (сущность - целое, и морфизмы позволяющие получить части целого). К сожаление, эта реализация не слишком функциональна в силу ограничения языка. Например отсутствует возможность копирования с переопределением конкретного атрибута, как следствие это вынуждает вызывать конструктор и передавать в него значения всех атрибутов.

Очевидно, что последовательный обход иерархии веб страниц приведет к времени выполнения общего запроса равного сумме времени выполнения каждого подзапроса к конкретной веб странице. Поэтому вполне естественно будет сократить это время за счет запрашивания ресурсов параллельно, на

этапах, где это возможно (ссылки на дочерние веб страницы недоступны до получения родительской).

В стандартной библиотеке Java, присутствует класс для работы с асинхронными вычислениями реализующий контракты монады и функтора - это `CompletableFuture`. Монадический и функторный контракты позволяют определять цепочки вычислений. В контексте рассматриваемого примера, такой цепочкой действий будет являться получение тела веб страницы, извлечение из него ссылок на дочерние страницы и выполнения запросов к ним. Очевидно, что если http запросы выполняются асинхронно, мы должны вызывать следующее действие только тогда, когда предыдущее выполнено. Монадический и функторный контракт `CompletableFuture` как раз и способствует реализации подобной стратегии. В противном случае мы были бы вынуждены иметь дело с вложенными обратными вызовами (callback).

И так, рассмотрим реализацию метода обхода веб страниц из класса FutureBasedCrawler.

```
public CompletableFuture<CrawlingResponse> crawl(CrawlingRequest request) {
    return concat(
            request
                    .urls()
                    .entrySet()
                    .stream()
                    .map(entry ->
                            parseResponse(extractScheme(entry.getValue()), performRequest(entry.getValue()))
                            .thenCompose(singleResponse -> handleChildren(request.childrenPattern(), request.childre
                            .thenApply(singleResponse -> Map.entry(entry.getKey(), singleResponse)))
                    .collect(Collectors.toList()))
            .thenApply(l -> l.stream().collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)))
            .thenApply(CrawlingResponse::new);
}
```

Первое, что следует отметить, метод `crawl` реализован как композиция других методов. В контексте Java, любой метод принадлежит некоему объекту, хотя в текущей реализации методы не использует внешнее состояние, и мы можем говорить, что они являются чистыми функциями. У внимательного читателя возможно возникнут вопросы по поводу метода `performRequest`, который создает `WebClient`. Но в нашем контексте обсуждений этим можно пренебречь.

Рассмотрим реализацию обхода веб страниц. На входе мы имеем `CrawlingRequest` с Map содержащей идентификаторы ссылок и сами ссылки на веб ресурсы. Используя Stream API обрабатывается каждая ссылка из входящего запроса и в результате мы имеем новый список с сигнатурой типа `List<CompletableFuture<Map.Entry<String, SingleResponse>>>`. SingleResponse есть не что иное как результат обработки конкретной веб страницы со всеми подстраницами. Мы вернемся к деталям несколько позже. А сейчас давайте разберем следующий код в концепциях теории категорий описанных выше.

```
request
    .urls()
    .entrySet()
    .stream()
    .map(entry -> /* обработка каждой ссылки, содержимое пропущено */ )
    .collect(Collectors.toList())
```

В категории типов нашего приложения связь между `CrawlingRequest` и `Map<String, String>`

содержащую ссылки на веб страницы описывается морфизмом `CrawlingRequest.urls()`. Имея `Map<String, String>` мы преобразуем ее в `Set<Map.Entry<String, String>>`, что является реализацией концепции естественного преобразования. То есть эта операция изменяет контейнер, но не содержимое в нем. `stream()` еще один пример естественного преобразования из `Set` в `Stream`. Метод map есть ни что иное как реализация контракта функтора в классе `Stream`. И следом еще одно естественное преобразование из `Stream` в `List`. И да, этот пример показывает насколько Java не хватает лаконичности.

Рассмотрим реализацию функции, которую мы поднимаем (lift) в контекст функтора (Stream).

```
entry ->
    parseResponse(extractScheme(entry.getValue()), performRequest(entry.getValue()))
    .thenCompose(singleResponse -> handleChildren(request.childrenPattern(), request.childrenLevel(), singleResp
    .thenApply(singleResponse -> Map.entry(entry.getKey(), singleResponse))
```

Реализация данной функции построена в виде цепочки действий базирующихся на `CompletableFuture`. `performRequest` - выполняет запрос к целевому url из entry.getValue() и возвращает `CompletableFuture<ResponseEntity<String>>`. Возвращаемое значение `ResponseEntity<String>` обернуто в контекст `CompletableFuture`. На момент возвращения объекта `CompletableFuture`, значение `ResponseEntity<String>` с большой долей вероятности еще не вычислено, так как сам запрос к веб ресурсу происходит асинхронно возвращению значения из функции. Далее это значение передается в метод `parseResponse(String schema, CompletableFuture<ResponseEntity<String>> responseFuture)` в котором декларируется обработка результата запроса к веб ресурсу. То есть, метод только декларирует каким образом необходимо провести обработку результата запроса к веб ресурсу, но не выполняет такую обработку. Сама обработка осуществляется действиями объявленными в лямбда выражениях (смотри лямбда исчисление как источник концепции) которые вызываются в момент появления значения в `responseFuture`.

```
return responseFuture
        .thenApply(r -> new SingleResponse(
                Optional.of(r.getStatusCodeValue()),
                r.getStatusCode().getReasonPhrase(),
                r.getHeaders().toSingleValueMap(),
                Optional.of(r.getBody()),
                extractLinks(schema, r.getBody()),
                Optional.empty()
        ))
        .exceptionally(e -> new SingleResponse(
                Optional.empty(),
                e.getMessage(),
                Collections.emptyMap(),
                Optional.of(ExceptionUtils.getStackTrace(e)),
                Collections.emptyList(),
                Optional.empty()
        ));
```

`thenApply` является функторной семантикой `CompletableFuture`. То есть функция преобразующая `ResponseEntity<String>` в `SingleResponse` поднимается на уровень контекста `CompletableFuture` (снова lifting). Метод `exceptionally` служит для обработки ошибок, в случае их возникновение при выполнении запроса. Вместо того, чтобы передать ошибку (исключение) далее, мы хотим преобразовать его в нашу

стандартную форму ответа `SingleResponse`. Фактически exceptionally тоже является реализацией функтора. А тот факт, что контекст `CompletableFuture` может содержать либо значение, либо исключение должно наталкивать читателя на мысль о рассмотренной выше концепции копроизведения и частного примера ее реализации `Either`.

Следующим шагом идет

```
.thenCompose(singleResponse -> handleChildren(request.childrenPattern(), request.childrenLevel(), singleResponse
```

Метод `thenCompose` реализует монадическую семантику `CompletableFuture` (более каноническое имя для такого метода - `flatMap`). Согласно описанию интерфейсу монады принятому в программировании, `flatMap` позволяет влиять на контекст. Так как в нашем случае контекстом является асинхронность (`CompletableFuture`), то это означает, что мы можем декларировать дополнительные асинхронные действия, которые будут инициированы в момент вычисления значения singleResponse. Дополнительные асинхронные действия необходимы нам для обхода дерева веб ресурсов вглубь. Реализация метода `CompletableFuture<SingleResponse> handleChildren(String pattern, Integer level, SingleResponse response)` достаточно проста:

```
return level == 0 ?
    CompletableFuture.completedFuture(response) :
    crawl(toRequest(pattern, level, response))
            .thenApply(crawlingResponse -> new SingleResponse(
                response.code(),
                response.message(),
                response.headers(),
                response.body(),
                response.links(),
                Optional.of(crawlingResponse)
            ));
```

В случае достижения лимита глубины обхода дочерних веб ресурсов, возвращается текущий response. В обратной ситуации, генерируется запрос к дочерним ресурсам и рекурсивно вызывается метод `CompletableFuture<CrawlingResponse> crawl(CrawlingRequest request)`. При получение результата опроса дочерних ресурсов, конструируется новый объект `SingleResponse`, включающий в себя результаты обхода дочерних ресурсов - `crawlingResponse`. Пример бедности функционала record в Java, который упоминался выше (отсутствие механизма копирования содержимого неизменяемой сущности с переопределением только конкретных атрибутов).

Оставшиеся части реализации метода `crawl` должны быть очевидны в силу уже рассмотренной семантики метода `thenApply` из `CompletableFuture`, за одним исключением - метод concat. Он требуется для преобразования `List<CompletableFuture<T>> в CompletableFuture<List<T>>`. Обычно, подобные преобразования в функциональном программировании принято называть монад трансформером (monad transformer).

```
private static <T> CompletableFuture<List<T>> concat(List<CompletableFuture<T>> futures) {
    return CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()]))
            .thenApply(v -> futures.stream().map(CompletableFuture::join).collect(Collectors.toList()));
}
```

Семантически данный метод преобразует список `CompletableFuture` в `CompletableFuture` списка с соблюдением существующих контрактов. То есть, соблюдается контракт асинхронности и контракт элементов в списке. Реализация этого метода является оберткой над библиотечной `CompletableFuture.allOf`, предоставляющая работу с более подходящими типами в контексте примера.

Стоит подчеркнуть, что основная часть реализации является декларированием пайплайна - композиции функций, которые могут вызываться асинхронно относительно момента вызова `thenApply` и `thenCompose`. Почему "могут" используется в предыдущем предложении, потому что это зависит от наличия вычисленного значения в `CompletableFuture` в момент вызова этих методов.

Функторный и монадический контракты `CompletableFuture`, далеко не единственный пример. Цепочки асинхронных вычислений удобно представлять с подобной семантикой, чему может быть подтверждением еще одна реализация web-crawler базирующаяся на Reactor API.

Читатель может самостоятельно рассмотреть пример реализации в классе ReactorBasedCrawler. Использование `Mono` в этой реализации является прямым аналогом использования `CompletableFuture` из примера выше.

Web-crawler на Scala

Если о Java маловероятно услышать как о языке поддерживаемом функциональное программирования в полной мере, в Scala такая поддержка всегда декларировалась как полноценная. Но даже несмотря на дизайн языка Java мы увидели, что функциональные подходы весьма применимы и поддерживаемы. В Scala все должно быть несколько лучше в этом плане. Давайте рассмотрим пример реализации web-crawler на Scala.

Отметим: реализация всех примеров web-crawler в этой статье была выполнена в максимально схожей манере для облегчения сравнения. Несмотря на это осуществлена попытка продемонстрировать специфику конкретного языка, но не без ущерба идиоматичности.

Начнем рассмотрение Scala примера с главного пайплайна в методе `crawl`, аналогичного рассмотренному в Java реализации.

```
def crawl(request: CrawlingRequest)(implicit executor: ExecutionContext, system: ActorSystem[Nothing]): Future[C
    Future.sequence(
      request
        .urls
        .map(prepareRequestContext)
        .map {
          case (id, httpRequest) =>
            for {
              response <- Http().singleRequest(httpRequest).transformWith(parseResponse(httpRequest.getUri.getSc
              childrenResponse <- handleChildren(request.childrenPattern, request.childrenLevel, response)
            } yield (id, childrenResponse
        }
    ).map(responses => CrawlingResponse(responses = responses.toMap))
  }
```

Уже знакомым подходом, используя реализацию функтора мы преобразуем `Map[String, String]` в коллекция с элементами `Future[(String, SingleResponse))]` (если быть точнее, то не коллекцию, а `Iterable`). Метод `prepareRequestContext` делает простое преобразование из кортежа с идентификатором и `url`, в кортеж идентификатора и объекта `HttpRequest` используемого для

выполнения запроса. Наибольший интерес носит следущая функция поднимаемая (lift) в контекст коллекции `Map`.

```
case (id, httpRequest) =>
    for {
        response <- Http().singleRequest(httpRequest).transformWith(parseResponse(httpRequest.getUri.getScheme))
        childrenResponse <- handleChildren(request.childrenPattern, request.childrenLevel, response)
    } yield (id, childrenResponse)
```

Эта конструкция является частичной функцией (Partial Function), в данном случает это упрощенный вариант сопоставления с образцом (Pattern matching) для более читаемого доступа к содержимому кортежа `(id, httpRequest)`. Конструкция `for {} yield` в Scala это монадическая и функторная композиция аналогична той, что мы видели выше в примере на Java использующего `thenCompose` и `thenApply` из `CompletableFuture`. Выражение с `for` приведенное выше является полным аналогом следующего кода:

```
Http().singleRequest(httpRequest).transformWith(parseResponse(httpRequest.getUri.getScheme))
    .flatMap(response => handleChildren(request.childrenPattern, request.childrenLevel, response))
    .map(childrenResponse => (id, childrenResponse))
```

`Http().singleRequest(httpRequest)` возвращает `Future[HttpResponse]`. Далее вызывается метод `transformWith` который во много похож на `flatMap`, но его семантика расширена для обработки как основного результата, так и ошибки. Для этого используется тип `Try` (еще один пример реализации концепции копроизведения). Для любопытного читателя может быть интересным выяснить почему `Try` расширяет тип `Product`, хотя подчеркнуто, что это пример реализации концепции копроизведения.

```
def parseResponse(parentUrlSchema: String)(t: Try[HttpResponse]): Future[SingleResponse] = t match {
    case Success(httpResponse) => // конструирование SingleResponse в случае успешного ответа
    case Failure(exception) => // конструирование SingleResponse в случае ошибки
  }
```

Выше представлен сокращенный вариант реализации parseResponse. Обратите внимание на сопоставление с образцом (pattern matching) `t macth { case A => /// Case V => ///}`. Эта конструкция позволяет обрабатывать значение в зависимости от его содержимого.

Следующим шагом идет обработка дочерних веб страниц.

```
def handleChildren(pattern: String, level: Int, response: SingleResponse): Future[SingleResponse] =
    level match {
      case 0 => Future.apply(response)
      case _ => crawl(toRequest(pattern, level, response)).map(r => response.copy(children = Some(r)))
    }
```

В зависимости от текущего уровня (снова пример сопоставления с образцом) обработка завершается, либо выполняется запрос дочерних ресурсов. Реализация этого метода аналогична той, что мы видели в Java примере. Стоит отметить использование копирования неизменяемого значения `SingleResponse` с подстановкой нового значения в поле children в выражении `response.copy(children = Some(r))`. В Java

примере мы говорили о громоздкости копирования record с подстановкой одного нового значения по причине отсутствия возможности переопределить некоторые значения. В Scala эта проблема отсутствует, так как в языке есть поддержка указания конкретного поля по имени.

Future.sequence по аналогии с примером на Java (метод concat), если говорить упрощенно, преобразует коллекцию `Future` во `Future` коллекции. На самом деле семантика метода сложнее в плане используемых типов, но мы можем проигнорировать это в данном контексте.

Данный пример web-crawler'a демонстрирует, что реализация на Scala выглядит очень похоже тому, что мы видели на Java. Хотя безусловно сам язык и его библиотека позволяют писать функциональный код более лаконично (примеры с `for {} yeild`, сопоставление с образцом, работа с неизменямимыми объектами как случае с case-классом).

Web-crawler на Haskell

Следующим мы рассмотрим пример реализации web-crawler на чистом функциональном языке - Haskell. Поскольку язык является чистым функциональным, любые побочны эффекты (side effects) исключены. На самом деле это не полностью корректно, по причине наличия небезопасного (unsafe) API для системных нужд, но это исключение. Как же возможно избежать побочных эффектов ведь приложение должно каким то образом взаимодействовать с окружающим миром во время работы (выполнять ввод/вывод)? Ответом на этот вопрос является перенос работы с вводом/выводом за пределы языка в виде монадического API. Подобно тому как в примере на Java осуществляется работа с асинхронным вводом/выводом через использование `CompletableFuture`, в Haskell весь ввод/вывод базируется на использование `IO` монады. То есть, сама программа является декларативной композицией желаемых действий, которые выполняются внешним по отношению к ней механизмом.

Начнем рассмотрение реализации, как и в примерах выше, с функции `crawl`.

```
crawl :: CrawlingRequest -> IO CrawlingResponse
crawl req = do
  manager <- newManager tlsManagerSettings
  responses <- mapConcurrently (process manager) (toList (urls req))
  responsesWithChildren <- mapConcurrently processChildren responses
  return CrawlingResponse { responses = fromList responsesWithChildren }
  where
    processSafely manager url = try $ do
              request <- prepareRequest url
              response <- executeRequest request manager
              return response
  process manager (id, url) = fmap (\response -> (id, parseResponse url response)) (processSafely manager url)
  processChildren (id, response) = do
    responseWithChildren <- (crawlChildren (childrenPattern req) (childrenLevel req) response)
    return (id, responseWithChildren)
```

Функция `crawl` декларирована с аргументом `CrawlingRequest` и результатом `IO CrawlingResponse`. Аналогично примерам на других языках. Мы уже обсуждали каррирование (currying) и частичном применение (partial application) в секция об аппликативном функторе и лямбда исчислении. Но давайте еще раз рассмотрим эти понятия на примере Haskell. В это языке любая функция представлена не в виде сущности принимающей кортеж аргументов, а виде функции принимающей аргумент и возвращающей функцию принимающий следующий аргумент и так далее. Нотация декларирования функций отражает такую семантику, как можно заметить из примера:

```
function1 :: A -> B -> C
```

Функция `function1` принимает аргумент типа A и возвращает функцию принимающую аргумент типа B с результатом типа C. Каррирование есть не что иное, как подобная семантика работы с функциями. В языках, в которых функция является сущностью принимающей кортеж аргументов, каррирование требует дополнительных ухищрений. В Haskell же, каррирование встроено в язык. Частичное применение это смежное понятие к каррированию, но не идентичное ему. Оно подразумевает возможность передать только подмножество аргументов функции и получить в результате функцию принимающую оставшееся подмножество аргументов. Фактически мы получаем новую функцию с поведением аналогичным оригинальной, но с фиксированными значениями для подмножества аргументов. При поддержке каррирования, мы получаем возможность частичного исполнения функций фиксируя значения в аргументах по их порядку.

Следующей строкой начинается определение функции. req используется в качестве имени аргумента декларированного типом `CrawlingRequest`. Тело функции начинается с `do` выражения. В примере на Scala мы рассмотрели `for yield` нотацию для монадических типов. `do` в Haskell служит той же цели. Это синтаксический сахар для монадических цепочек.

```
do
    a <- action1 v1
    b <- action2 a
    c <- action3 v2
    return (a, b + c)

-- идентично

action1 >>= (\ a -> (action2 a) =>> (\ b -> (action3 v2) =>> (\c -> return (a, b + c))))
```

Оба выражения представленных выше идентичны. Давайте рассмотрим их в деталях:

- выражения вида `\a -> x + 1` являются анонимной функцией, лямбда выражением
- `\>>=` одна из функций связывания декларированных в классе `Monad` из Haskell. Это полный аналог `flatMap` рассмотренных выше. То есть эта функция принимает в качестве аргумента функцию с результатом предыдущего монадического действия и ожидает возвращения результата в виде новой монады
- `return` - есть не что иное, как естественно преобразование η рассмотренное нами в определении монады в терминах теории категорий. Эта функция принимает в качестве аргумента некое значение и возвращает его обернутым в монаду. Цель разработчиков Haskell была сделать `do` блок похожим на императивное построчное выполнение, отсюда и появилось название return по аналогии с возвращением значения из функции/метода

Строка `manager <- newManager tlsManagerSettings` создает менеджера соединений для http клиента, в контексте рассмотрения важно лишь то, что newManager возвращает значение монадического типа `IO Manager`.

Далее следует параллельная обработка запросов ко всем url текущего уровня `responses <- mapConcurrently (process manager) (toList (urls req))`. Каждый url из списка обрабатывается функцией process возвращающей `IO (Text, SingleResponse)` идентификатор запроса и ответ.

`mapConcurrently` осуществляет проход по списку с выполнением действия возвращающим `IO` монаду и преобразующим список монад в монаду списка, тем самым синхронизируя момент получения ответа по всем параллельно выполняющимся запросам.

Реализация функции process находится в секции where определения функции `crawl`. Реализация базируется на результате выполнения `processSafely` типе `IO (Either SomeException (Response L.ByteString))` который преобразуется через вызов функтора (`fmap`) и с помощью функции parseResponse в `IO (Text, SingleResponse)`.

Реализация функции `processSafely` базируется на цепочке монадических действий и блоке `do`, рассмотренных выше. Взглянем детальнее на начало реализации функции `try $ do` ... Это оборачивание всего блока `do` для перехвата всех возможных исключений, которые могли произойти ранее в цепочке вычислений. Результатом возвращаемом из `try`, в этом контексте, является значение типа `Either SomeException (Response L.ByteString)`. То есть либо исключение, либо результат http запроса.

После того как ответы со всех url текущего уровня получены, необходимо обработать запрос к дочерним страницам. Строка `responsesWithChildren <- mapConcurrently processChildren responses` вызывает метод обработки запросов к дочерним страницам. processChildren использует функцию `crawlChildren`, которую мы и рассмотрим ближе.

```
crawlChildren :: Text -> Int -> SingleResponse -> IO SingleResponse
crawlChildren _ 0 response = return response
crawlChildren pattern level response = do
  childrenResponse <- crawl $ toRequest pattern (level - 1) response
  return response { children = Just childrenResponse }
```

С помощью сопоставления с образцом, при нахождении на нижнем уровне происходит просто возвращение результата `crawlChildren _ 0 response = return response`. В противном случае выполняется запрос к дочерним страницам через уже знакомую нам функцию crawl. В результате текущий родительский response преобразуется в новый инстанс путем изменения значения поля children. Аналогично примерам на других языках - копирование неизменяемой сущности и переопределение поля.

Функции crawl завершается формированием ответа return `CrawlingResponse { responses = fromList responsesWithChildren }`.

Web-crawler на Kotlin

В завершении мы рассмотрим еще один пример реализации на языке Kotlin. В Kotlin, как и в большинстве современных языков, предоставляется множество средств для функционального программирования, хотя эта парадигма не является основной для него. Реализация представленная ниже, как и остальные примеры выполнена в схожей парадигме цепочки вызовов асинхронных действий. Хотя на этот раз, монадический контракт скорее полностью скрыт в языке. Давайте приступим к рассмотрению.

```
suspend fun crawl(request: CrawlingRequest): CrawlingResponse {
    val client = HttpClient(CIO)
    val responses = merge(
        request
        .urls
```

```
        .map { e ->
            suspend {
                Pair(
                    e.key,
                    handleChildren(request.childrenPattern, request.childrenLevel, parse {client.get(e.value)})
                )
            }
        }.toPersistentList()
    ).toMap().toPersistentMap()
    return CrawlingResponse(responses = responses)
}
```

Функция `crawl`, принимает на вход аргумент типа `CrawlingRequest` и возвращает значение типа `CrawlingResponse`. Эта сигнатура аналогична примерам на других языках представленных выше. Вы можете спросить: а где же асинхронная семантика? Ответ кроется в ключевом слове `suspend` и сопрограммах (coroutines). Асинхронная семантика в предыдущих примерах построена на монадах с использованием стрелок Клейсли (цепочки `flatMap`). В этом же примере, не видно подобного. Или нет?

Отмети: нижеследующее рассмотрение сопрограмм Kotlin осуществляется под "определенным углом" для демонстрации каким образом можно распознать концепции монады и стрелок Клейсли. Подобный взгляд на сопрограммы, может не разделяться читателем на его усмотрение.

Если кратко, то `suspend` функции, как заявляет документация Kotlin, это более безопасная и менее склонная к ошибкам абстракция по сравнению с `CompletableFuture` и `Future` из Java и Scala. Ключевое слово `suspend` означает, что реализация функции содержит асинхронную операцию по отношению к текущему контексту и на этом месте выполнение может прерваться в ожидании наступления события для продолжения (например появления данных в сокете). То есть, код который выглядит императивным, фактически является подобием рассмотренных выше механизмов `do` из Haskell и монадический `for` из Scala. Рассмотрим некий псевдокод с аналогией `suspend` и стрелок Клейсли из `flatMap`.

```
suspend fun f1(value: String): String

suspend fun f2(value: String): String

suspend fun f3(value: String): String

suspend fun f(value: String): String {
    val r1 = f1(value)
    val r2 = f2(r1)
    return f3(r2)
}

// концептуально аналогично

fun f1(value: String): Monad<String>

fun f2(value: String): Monad<String>

fun f3(value: String): Monad<String>

fun f(value: String): Monad<String> {
    return f1(value)
            .flatMap(r1 -> f2(r1))
```

```
            .flatMap(r2 -> f3(r2))
}
```

Как и в случае с монадами и стрелками клейсли, при их использование весь код выше по стеку вызова "заражается" ими. Тоже самое происходит в случае с ключевым словом suspend. Разумеется полноценного интерфейса монад мы не увидим в сопрограммах. Такие операций, например, как заворачивание скалярного значения в монаду (естественное преобразование η из определения монады в концепциях теории категорий), для сопрограмм в явном виде не имеют смысла, потому как их поддержка встроена в синтаксис языка и фактически управление `suspend` контекстом происходит за пределами синтаксиса. Заинтересованному читателю предлагается в качестве дополнительного упражнения попробовать доказать применимость условий согласованности для монады с использование `suspend` функции и скалярного значения.

Вернемся к обсуждению реализации. Общая структура функции `crawl` должна быть понятна по аналогии с уже рассмотренными примерами. Хотя несколько моментов требует пояснения для читателей не знакомых с Kotlin.

- `suspend {...}` возвращает функцию без аргумента и возвращающую значение с учетом suspend семантики
- `parse {client.get(e.value)}` функция parse принимает в качестве аргумента suspend функции для возможности обработки исключений внутри самой функции. И случай map на коллекции url и передача функции выполнения запроса к веб ресурсу объявляют `suspend () -> T` функции. Но в первом случае система типов не может вывести корректный тип из выражения, для этого и требуется явное указание suspend
- `suspend fun <T> merge(suspended: PersistentList<suspend () -> T>): PersistentList<T>` реализация функции для параллельной обработки списка suspend значений и возвращение suspend списка значений базируется на явном определении контекста сопрограмм coroutineScope и вызова параллельных вычислений с `async {}` блоком. То есть базовая семантика suspend подразумевает асинхронность в контексте последовательных действий позволяющая освобождать ресурсы текущего обрабатывающего потока в ожидании события, но не подразумевает параллельность suspend действий. Параллельность действий и обеспечивается конструкцией вида `async {}` возвращающая `Deferred` абстракцию схожую с `Future` рассмотренной выше, но с более ограниченным интерфейсом в области возможности композиции

```
suspend fun <T> merge(suspended: PersistentList<suspend () -> T>): PersistentList<T> =
    coroutineScope {
        suspended
            .map { f -> async { f.invoke() } }
            .fold(persistentListOf<T>()) { l: PersistentList<T>, d: Deferred<T> -> l.add(d.await()) }
    }
```

- `String?` и прочие типы значений с постфиксом `?` это типы позволяющие иметь `null` значения (по умолчанию типы не могут иметь `null` значения в отличие от java). `Nullable` типы совместно с безопасной операцией доступа к полям `?.` можно сравнить с реализацией контракта функтора в `Maybe` монаде (в частном случае использования для доступа к полям в поднимаемой функции). Попробуйте проследить аналогию в примере ниже

```
a?.b?.c
aMaybe.map(a -> a.b).map(b -> b.c)
```

- классы данных (data class) в Kotlin это аналог кейс классов из Scala, но с ограничениями относительно последнего. Например отсутствует поддержка использования классов данных в сопоставлении по образцу (pattern matching) с доступом к основным элементам. Инстансы классов данных являются неизменяемыми и имеют поддержку копирующих функций с возможностью переопределения конкретных полей (это момент мы уже обсуждали выше)

Функторный контракт коллекций и естественные преобразования в виде трансформации видов коллекций, вероятно уже распознаны читателем. Но для наглядности давайте посмотрим на несколько строк примеров.

```
.map { l -> Pair(l, l) }
.toMap()
.toPersistentMap()
```

Есть ли что-то еще?

Примеры, рассмотренные выше, дают детальную картину применения базовых концепций теории категорий в реальных условиях.

Исключая пример на haskell, где `io` монады неотъемлемая составляющая любой программы, наиболее явная мотивация появления монадического контракта это поддержка асинхронных вычислений. Разумеется это не единственная мотивация. Программы с подобной структурой способствуют формированию более изолированного и тестируемого кода. Принципы функционального программирования в общем и концепции теории категорий в частности - это в первую очередь о компонуемости и декларативности.

Хотя монадический контракт и проявляет себя во всей красе при использование для структурирования асинхронных действий, это не единственная область применения. В более общем контексте, подобные подходы применяются для изоляции и выноса побочных эффектов (например таких как ввод/вывод) за границы области, которую мы хотим сохранить функционально чистой (как это осуществлено в Haskell). Фактически любая область, где есть некий "контекст", обработку которого мы хотим изолировать от нашей логики, создает пространство для применения монад.

Нужно еще раз подчеркнуть, хотя большинство примеров выше иллюстрируют использования концепций монад и функторов, функциональное программирование не сводится только к элементам теории категорий. Персистентные структуры данных, ссылочная прозрачность, чистые функции, отсутствия сторонних эффектов, замыкания (лямбда выражения) и многое другое тоже являются частью того, что называют функциональным программированием.

В дополнение, рассмотрим еще несколько примеров. А именно, один из широко распространенных фреймворков для распределенной обработки данных и `Promise` из JavaScript.

Apache Spark

Аналогично тому что мы видели на примерах применения функторной и монадической семантики при работе с коллекциями, некоторые движки и фреймворки для обработки данных предоставляют подобные API. В добавлении к абстрагированию контекста в виде структуры связей элементов данных (как это происходит например в списке) часто можно отметить и контекст параллельных или/и

распределенных вычислений. Разумеется имеет место применение и других концепции из математических теорий рассмотренных выше: моноид, элементы лямбда вычислений и другие.

Apache Spark, как заявляет главная страница проекта, это унифицированный аналитический движок для обработки данных в большом масштабе. Нас же интересует предоставляемый им API. Apache Spark позволяет работать с большими массивами данных распределяя вычисления по множеству узлов. API поддерживает несколько подходов как для пакетной обработки (RDD, Spark SQL), так и для потоковой (Spark Streaming). В качестве примера давайте остановимся на RDD, остальные подходы выглядят более менее похожими в контексте применения функциональных конструкций.

`RDD` это устойчивый распределенный набор данных (resilient distributed dataset), то есть за этой абстракцией скрываются механизмы по работе с неизменяемым набором данных распределенным на множество узлов, с возможностью восстановления в случае фатальных ошибок или исчезновения некоторых узлов. Если бы нам как пользователям `RDD` было необходимо заботится о всех этих свойствах напрямую, механизм выглядел бы очень сложно. Только представьте себе обработку отказов узлов. Но благодаря абстрагированию всех этих свойств, мы фактически имеем интерфейс аналогичный тому, что реализован в локальных коллекциях. Следующий код считает количество с которым встречается каждая буква в потенциально очень большой последовательности слов. Рассмотрим этот пример построчно.

```
val actualRDD = spark.sparkContext
    .makeRDD(Seq("one", "two", "three"))
    .flatMap(_.toList)
    .map((_, 1))
    .reduceByKey(_ + _)

val expectedRDD = spark.sparkContext
    .makeRDD(
        Seq(('w',1), ('e', 3), ('t', 2), ('h', 1), ('o', 2), ('n', 1), ('r', 1)))

assertSmallRDDEquality(actualRDD.sortByKey(), expectedRDD.sortByKey())
```

Пример начинается с инициализации RDD из локальной коллекции: `spark.sparkContext.makeRDD(Seq("one", "two", "three"))`. В реальных же условиях, данные должны читаться из распределенного хранилища будь то файловая система (распределенная/кластерная) или NoSQL база данных. Далее мы видим строку `.flatMap(_.toList)` смысл которой должен быть понятен из рассмотренных ранее примеров монад. Стоит отметить, что `RDD.flatMap`, строго говоря, нарушает монадический контракт, поскольку список/коллекция не является `RDD`, фактический тип изменяется. Но возможно говорить, что с точки зрения пользователя эти типы изоморфны (если мы рассматриваем их как коллекции данных). Данное ограничение обусловлено реализаций RDD API. В следущей строке мы видим функторный контракт `RDD`, в которой преобразуем коллекцию букв в коллекцию кортежей где в качестве первого элемента буква, а второй всегда единица. Подобный тип данных нам нужен чтобы вычислить моноид в строке `.reduceByKey(_ + _)`. В данном случае моноид это операция сложения на целых числах (вторые элементы кортежа), в рамках разбиения по ключу (первые элементы кортежа). Затем следует несколько строк для формирования ожидаемого `RDD` и вызов метода сравнения ожидаемого и фактического `RDD`.

В рамках рассмотрения примера мы опустили еще одну важную концепцию часто встречаемую в функциональных языках/фреймворках - ленивые вычисления. На момент присваивания переменной `actualRDD`, не существует еще никакого финального результата вычислений. То есть сами вычисления еще не произведены. Другими словами `RDD` является ленивой коллекцией. Используя семантику `RDD`, до

присваивания `actualRDD`, мы декларировали какие действия необходимо выполнить чтобы произвести требуемые вычисления. Это является прямой аналогией отличий времени декларирования и исполнения пайплайнов из примеров web-crawler. Единственное, что в том контексте мы не использовали термин ленивость. Структура программ же выглядит идентично: сначала декларирование действий, и затем уже некий внешний механизм запускает их выполнения. Техническая разница в том что в случае, например, с haskell внешний механизм ввода/вывода запускает вычисления. В случае с Java и Scala, асинхронных механизм ввода/вывода. А в примере со Spark, это специальный метод инициирующий запуск распределенных вычислений и возвращение результата на драйвер (приложение координатор). В данном конкретном случае это происходит внутри реализации `assertSmallRDDEquality` при вызове `RDD.collect()`.

JavaScript Promise

Язык JavaScript известен, в том числе, своей асинхронной природой. В языке (его реализациях) отсутствует поддержка многопоточной работы. В то же время, его можно встретить как на клиентской, так и на серверной стороне. Очевидно, что ввод/вывод не может быть блокирующим в такой среде. Как следствие, все операции ввода/вывода в JavaScript асинхронные. Долгое время программы на JavaScript (по крайне мере не слишком искушенных разработчиков) тяготели к так называемому "аду" обратных вызовов (callback hell), потому как API ввода/вывода основано на передаче функций обратного вызова.

Promise API появилось, чтобы решить проблему композиции с обратными вызовами. И это делается, через уже знакомый нам монадический контракт. Promise является аналогом рассмотренных выше примеров с Future и CompletableFuture.

```
let request = req => {
    return new Promise((resolve, reject) => {
        let xhr = new XMLHttpRequest();
        xhr.open(req.method || "GET", req.url);
        if (req.headers) {
            Object.keys(req.headers)
                .forEach(key => {xhr.setRequestHeader(key, req.headers[key]);});
        }
        xhr.onload = () => {
            if (xhr.status >= 200 && xhr.status < 300) {
                resolve(xhr.response);
            } else {
                reject(xhr.statusText);
            }
        };
        xhr.onerror = () => reject(xhr.statusText);
        xhr.send(req.body);
    });
};

var resolveInitialAction = null;
new Promise(function(resolve, reject){
        resolveInitialAction = resolve;
    })
    .then(req => request(req))
    .then(resp => alert(resp));

setTimeout(() => resolveInitialAction({url: "https://api.ipify.org?format=json"}), 10000);
```

Пример выше демонстрирует использование Promise API. Из всего примера наибольший интерес, в нашем контексте, представляет инициализация объекта Promise и метода then. Но давайте рассмотрим его построчно.

Кода начинается с определения функции request реализующей `Promise` версию выполнения запосов из `XMLHttpRequest`. Инициализация объекта `Promise` принимает функцию с аргументами для изменения состояния `Promise`. В данном случае, при вызове request происходит инициализация запроса `XMLHttpRequest` и в зависимости от результата, `Promise` переводится в состояние fulfilled (заполнен значением) или rejected (отвергнут с ошибкой). Подобную инициализацию Promise мы видев в следующих двух строках. resolveInitialAction необходима для того обособленного запуска вычисления цепочки Promise.

Метод `then` является аналогом `flatMap` в одном из контекстов использования. Как и в многих других реализациях монад подобных API, названия методов и семантика могут отличаться от изначально распространенных. Но несмотря на это, на примере строки `.then(req => request(req))` можно четко опознать семантику `flatMap`. pure же можно сопоставить метод Promise.resolve, которые оборачивает значение в завершенный Promise.

Строка начинающаяся с `setTimeout` привносит асинхронность вызова исполнения цепочки `Promise` в отношении момента декларирования. Через 10 секунд после выполнения кода выше, таймер инициирует вызов функции `resolveInitialAction`, которая в свою очередь запускает исполнение всей цепочки: вызов публичного API возвращающего ваш IP и отображение результата в виде alert сообщения. Тем самым мы можем четко видеть разделение времени декларирования и времени исполнения.

Что изучать дальше?

В статье мы рассмотрели теорию и примеры некоторых подходов функционального программирования. По большей части фокус был сделан на демонстрацию компонуемости и декларативности основанных на монадическом и функторном контрактах. Были рассмотрены примеры воплощения концепций теории категорий, лямбда исчисления и других. Затронуты вопросы неизменяемых структур данных и самое начальное введение в оптику. Но это только базовые концепции. Функциональное программирование продолжает не стоит на месте и много тем не затронуто в данной статье. Давайте кратко рассмотрим некоторые из них, чтобы читатель заинтересованный в продолжении изучения, имел представление о направлениях дальнейшего движения.

Библиотеки и фреймворки

Первым на что вы, вероятно, захотите посмотреть будет один из фреймворков для вашего языка предоставляющий дополнительные механизмы функционального программирования. Их достаточно много, есть и лидеры и аутсайдеры, относительно новые и старые.

Java

Java не предоставляет серьезной поддержки функционального программирования. Но как и большинство распространенных языков получила некоторое развитие в этом направлении. Для того чтобы расширить возможности Java в поддержки функциональных подходов можно использовать одну из сторонних библиотек.

- vavr предоставляет достаточно большое количество инструментов для поддержки функционального программирования в Java: неизменяемые коллекции; абстракции для функций с поддержкой композиции, каррирования, частичного исполнения, мемоизации и других;

функциональные абстракции для значений; сопоставление с образцом (pettern matching).

- jOOL то функциональная библиотека в рамках проекта jOOQ. Предоставляет минимальное количество инструментов по работе с коллекциями и базовые абстракции для функций и кортежей.

- cyclops еще одна достаточно богатая библиотека для поддержки функционального программирования в Java. К сожалению, проект не развивается последние пару лет.

Scala

Несмотря на то что Scala поддерживает функциональное программирование как одну из своих парадигм и имеет множество встроенных абстракций эту поддержку можно расширить.

- Scalaz вероятно старейшая и все еще поддерживаемая Scala библиотека для функционального программирования. Библиотека является зрелой, но есть мнение, что ей не хватает согласованности. В добавок можно отметить недостаток актуальной документации.

- Cats библиотека предоставляющая абстракции для поддержки функционального программирования, нацелена на общее применение и построения других библиотек на ее основе. Целевая область покрытия сравнима со scalaz, но сама библиотек лучше документирована и более согласована в плане API.

- Cats effect библиотека для построения асинхронных сервисов базирующаяся на Cats и расширяющая ее. Дает фреймворк для построения функциональных приложений. Если кратко, то это развитие идей на которых реализованы примеры web-crawler.

- ZIO нацелен на решение той же задачи что и Cats Effect, но на своем стеке.

Kotlin

Kotlin, во многом как и Java, имеет некоторую поддержку для работы в функциональной парадигме, но эта поддержка неполна и может быть расширена.

- Arrow библиотека для поддержки функционального программирования в Kotlin, покрывает широкий спектр областей от базовых абстракций и оптики до асинхронного фреймворка.

Clojure/Lisp

В основном рассматривая статически типизированные языки к контексте функционального программирования (за исключением JavaScript), мы упустили из виду очень важную область - lisp. Lisp как один из самых старых языков программирования имеет ряд интересных особенностей. Если говорить о jvm мире, то Clojure как lisp-подобный язык общего назначения безусловно заслуживает внимания. Представьте себе язык, синтаксис которого является его же структурами данных. Это дает безграничные возможности к мета-программированию. Идеи базовых структур данных также крайне любопытны.

Персистентные структуры данных

Персистентные структуры данных это такие структуры, которые всегда поддерживают сохранение предыдущей версии при модификации, что делает такие структуры фактически неизменяемыми (immutable). Одно из техник реализации таких структур является копирование при записи, то есть каждое изменение порождает копирование данных. Такой подход является не слишком эффективным особенно при большом количестве небольших модификаций, поэтому существуют и другие подходы. Если представить любые вложенные структуры данных в виде дерева, то можно модифицировать

только изменяющиеся ветви и листья, тем самым сохраняя большую часть данных нетронутыми. Подобное реализацию можно найти в Clojure.

Неизменяемые структуры данных - один из базовых принципов функционального программирования, поэтому эффективная работа с такими структурами важна в контексте оптимальности потребления ресурсов.

И другое

И много другое заслуживает внимания в мире функционального программирования, который сильно шире базовых концепций функций высшего порядка и неизменяемых данных.

- полиморфизм по-запросу и type class
- проблемы монад трансформеров
- не упомянутые выше концепции и области математики, их отражение в программировании (типизированное лямбда исчисление, профунктор оптика, лемма Йонеды ...)
- free monad и tagless final

В качестве заключения

За свою историю работы в ИТ я видел как противников, так и сторонников функциональной парадигмы. Как у любой подхода у функционального программирования есть множество плюсов и минусов. Но трудно отрицать тот факт, что современная ИТ индустрия приняла данную парадигму. А если есть факт принятия, то эта область знаний заслуживает, как минимум, беглого изучения. Спасибо, что дочитали до конца. Надеюсь статья была полезна и помогла сформировать некую общую картину функционального программирования.

Дополнительные источники

- Исходный код всех примеров
- Лекции по теории категорий 1, 2, 3 и книга по ним от Bartosz Milewski (англ.), некоторые главы (рус.)
- Tofu club. Что такое tagless final?
- Persistent Data Structures and Managed References - Rich Hickey
- Lambda Calculus - Fundamentals of Lambda Calculus & Functional Programming in JavaScript
- From design patterns to category theory by Mark Seemann