Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Андрей Эдуардович Кислицын

Выпускная квалификационная работа

Создание Kotlin API для Grammar of graphics

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
д. ф.-м. н., профессор СПбГУ
А. С. Куликов

Консультант:
руководитель команды разработки JetBrains
Р. В. Белов

Рецензент:
к. ф.-м. н., с. н. с. МФТИ
А. А. Нозик

Санкт-Петербург
2022 г.

Содержание

Введение

Grammar of graphics — это грамматика для задания графиков, описанная Леландом Уилкинсоном в одноименной работе [1]. На основе этой работы была создана библиотека ggplot2 [2] [3] для построения графиков на языке программирования R [4]. В ней для описания графиков используется синтаксис, вдохновленный грамматикой Уилкинсона. На основе библиотеки ggplot2 в компании JetBrains была разработана библиотека Lets-Plot [5] для языка программирования Kotlin [6]. Однако API для построения графика был взят почти напрямую из ggplot2, и унаследовал его существенные недостатки — отсутствие типизации и типобезопасности, проблемы со структурой (такие как отсутствие иерархии элементов графика, разделение взаимосвязанных элементов графика). Он не задействует средства Kotlin для создания идиоматичного синтаксиса, — например, внешних контекстов с получателями.

Постановка задачи

Первоначально, данная работы была направлена на разработку удобного предметно-ориентированного языка на Kotlin для построения графиков в библиотеке Lets-Plot, использую идеи Grammar of graphics.

Однако, в дальнейшем, задача была в значительной мере изменена. Было принято решение сделать API более универсальным, а именно:

1. Разработать универсальное промежуточное представление графика, которое можно перевести в представления графиков в различные движки для визуализации графиков.

2. Разработать предметно-ориентированный язык на Kotlin для построения графиков, генерирующая данное представление, с возможностью (опционально) добавлять в него элементы, основанные на особенностях движка, для которого строится график

Помимо этого, в задачу входило написании трансляторов, преобразующих получаемое при создании на разработанном предметно-ориентированном языке промежуточное представление графика во внутренние представления графиков конкретных движков. Такими движками были выбраны, разумеется, Lets-Plot, а также библиотека ECharts [7] на языке JavaScript [8]. ECharts был выбран по ряду причин. Во-первых, формат, в котором в ECharts представлен график, - это JSON, который несложно сгенерировать средствами Kotlin. Во-вторых, он использует подход описания графиков, отличный от Lets-Plot и ggplot2, а также написан на другом языке. Таким образом, реализация трансляции ECharts демонстрирует универсальность разработанного представления.

Другой немаловажной задачей было интегрировать разработанную библиотеку с существующими продуктами в экосистеме Kotlin for Data Science. А именно:

1. Kotlin Jupyter Kernel [9] - это ядро для Jupyter Notebook [10], позволяющее запускать на нем код на Kotlin. Необходимо было реализовать визуализацию создаваемых графиков на движках внутри ноутбука.

2. Kotlin Dataframe [11] - это библиотека для работы с данными. В рамках работы нужно реализовать визуализации данных, используя Dataframe как источник данных.

Обзор литературы

Основными принципами, которыми руководствовались при создании нашего предметно-ориентированного языка для построения графиков, были принципы "Грамматики графики"(Grammar of grpahics), описанные в [1]. В ней подробно рассмотрен процесс перехода от данных к их визуализации. В работе подробно описан каждый шаг этого процесса. Для это используется специальная нотация. Элементы построения графика, описанные Уилкинсоном, использовались как основа разрабатываемого в рамках данной работы предметно-ориентированного языка и промежуточного представления графика.

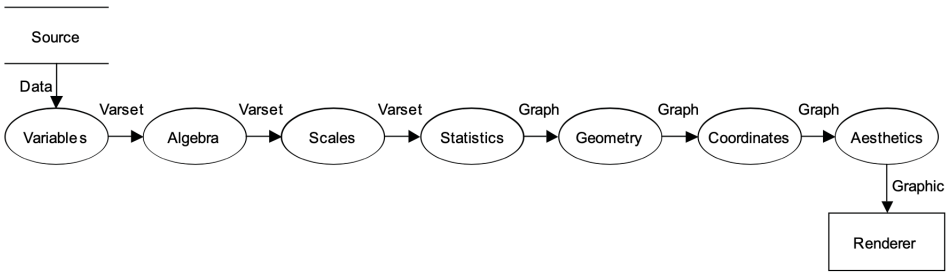

Рис. 1: Процесс визуализации в Grammar of graphics

Характерная черта подхода Grammar of graphics, отличающая его от большинства других, — императивность. Грамматика описывает **как** строить график, а не то, каким график должен получиться.

Помимо непосредственно работы Уилкинсона, для разработки API было важно ознакомиться с реализациями Grammar of graphics. Одна из самых известных — библиотека ggplot2 [2] для языка программирования R, широко использующаяся в области Data Science. В [3] её автор подробно описывает то, как теоретическая грамматика ложится на практический синтаксис. Одна из основных тенденций развития ggplot2 по сравнению с Grammar of graphics — структуризация синтаксиса, основанная на теоретической структуре графика. Внутри синтаксиса библиотеки значительное место занимают важнейшие элементы графика (слои, эстетические атрибуты, скейлы и другие) и их взаиморасположение. Поэтому при его разработке также был произведен шаг в

сторону декларативности относительно нотации Уилкинсона.

1. Визуализация данных. Структура графика

1.1. Модель данных

При визуализации данных важно, в первую очередь, определить модель данных. В ходе исследования, было решено остановиться на *табличной* модели данных, которая обладает следующими ключевыми особенностями:

1. В такой модели *объект данных* представляет собой набор *полей* — пар "ключ-значение"с уникальным набором ключей.

2. Набор ключей для всех объектов одинаковы, а значения с одинаковыми ключами имеют одинаковый тип.

3. Все объекты упорядочены (то есть пронумерованы натуральными числами).

4. Количество объектов конечно.

Также сделано допущение, что все значения в таблице определены (то есть не являются NULL).

Почему была выбрана именно такая модель? Прежде всего, подобная модель используется непосредственно в Grammar of graphics. В ggplot2 используется `data.frame`, а в Lets-plot — `Map<String, List<*>>`. Обе эти модели схожи с вышеописанной.

Безусловно, есть более сложные модели данных, в частности — иерархические и древовидные. Однако их визуализации с ними в данной работе не рассматривается в виду определенных сложностей.

Типы в языке Kotlin для работы с данной моделью в библиотеке будут описаны позже, а пока что введем следующие определения. *Источник данных* — это упорядоченный набор значений, соответствующих определенному ключу в данной таблице данных (то есть столбец таблицы). Он характеризуется названием ключа и типом значений. *Объект данных* — строка в данной таблице.

1.2. Визуальная структура графика

В этой секция описывается структура визуальных компонентов графика и их связь с данными.

1.2.1 Компоненты графика

Полотно графика — двумерное визуальное пространство, на котором располагаются все его остальные компоненты. Компоненты делятся на основные и вспомогательные.

Основные компоненты — *визуальные объекты*. Они характеризуются:

- *геометрическим объектом* или *геометрической сущностью*
- *эстетическими* (или *визуальными*) *атрибутами*:
  - *позиционными*, то есть своим расположением на полотне
  - *непозиционными* — цветом, формой, размером и другими
- некоторыми другими свойствами (например, позицией относительно друг друга)

Существуют различные вспомогательные компоненты. Но самые главные — это *гиды*, которые делятся на два типа — *легенды* и *оси*. Про их связь с компонентами и их значение для построения графика будет идти речь в следующем подразделе.

1.2.2 Визуализация и чтение графика

Постановка задачи визуализации — отобразить данные в компоненты графика. Чтобы понять, как происходит это процесс, проведем исследование процесса, обратного ему: восстановление данных из графика. Рассмотрим график 2[1].

Рассмотрим произвольный визуальный объект на данном графике. В первую очередь он характеризуется своим геометрическим объектом или

[1]Данный график построен с помощью разработанной библиотеки на движке Lets-plot. Данные взяты из открытого источника[12]

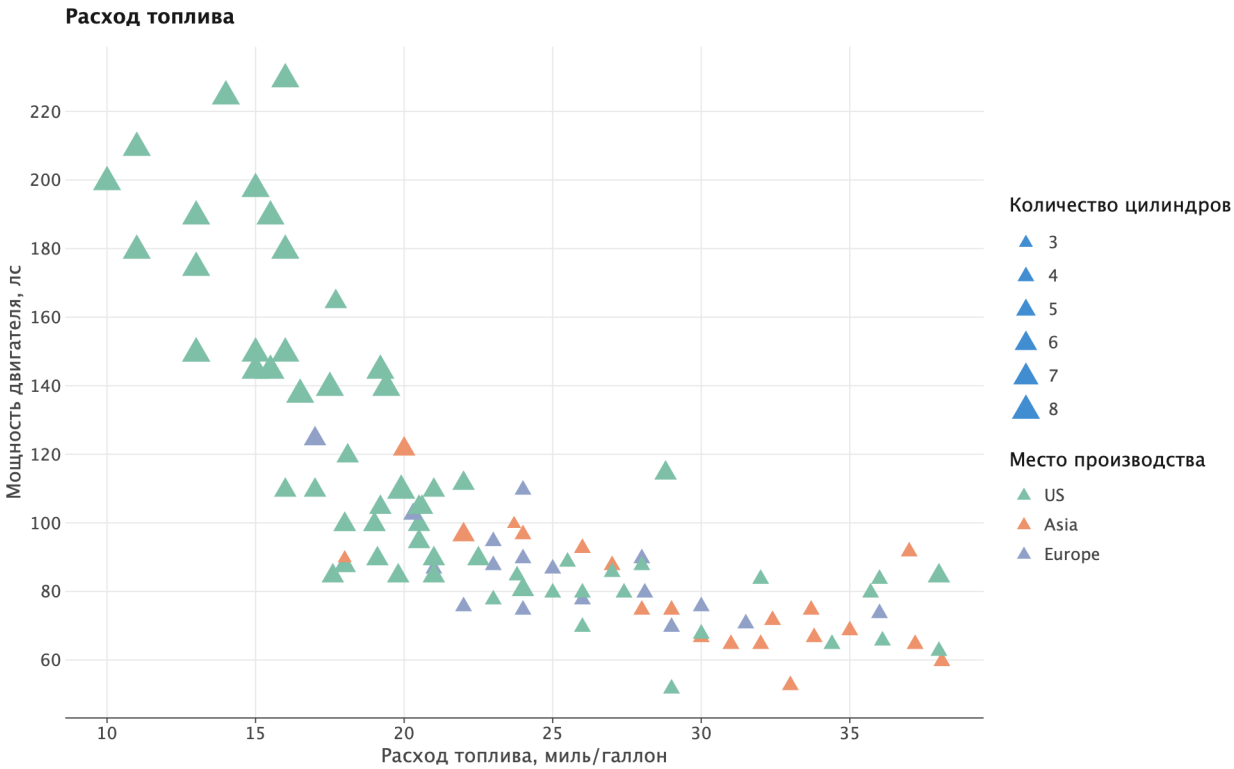

Рис. 2: График расхода топлива.

геометрической сущностью. В данном случае это *точка* (*point*). Объектов с другой сущностью на графике нет. Каждая точка характеризуется набором эстетических атрибутов:

- Позиционные — координаты вдоль осей X и Y. Чтобы восстановить данные из них, мы сначала считаем относительную позиция вдоль виртуальной геометрической оси (например, ортогональной проекцией на ось) и сопоставляем с осью-гидом, чтобы вычислить значение из изначальных данных (в данном случае с помощью линейной интерполяции). Таким образом, из координаты по каждой из осей мы получили поле объекта данных (а именно: "Мощность двигателя"и "Расход топлива");
- Непозиционные:
  - Размер. Сопоставляя размер объекта с легендой для размера, мы получаем поле "Количество цилиндров";
  - Цвет. По аналогии, вычислили поле "Место производство";

– Форма. Все точки имеют форму «треугольник». Этот атрибут не несет никакой информации, тем не менее имеет некоторое постоянное значение.

– Аналогично, некоторые константные в данном случае атрибуты, такие как прозрачность, размер и цвет границы точки не несут информации.

Таким образом, мы поняли, как читается график, и можем описать и формализовать процесс его построения.

1.3. Процесс визуализации

Основной момент в процессе визуализации — преобразование объекта данных в визуальный объект. Он определяется геометрической сущностью. Среди основных из них — это *точка* (*point*), *линия* (*line*) и *столбец* (*bar*); безусловно и есть более сложные — «ящик с усами» (*boxplot*), «японская свеча» (*candlestick*) и другие. Именно геометрическая сущность определяет набор эстетических атрибутов визуального объекта. Например для точки определен атрибут формы (shape), для линии — ширина (width). Стоит упомянуть, что некоторые сущности зависят (например, линия) от порядка (напомню, что объекты данных в модели упорядочены), и получаемые визуальные объекты зависят от нескольких объектов данных. Такие сущности называются *коллективными*.

Главной частью данного преобразования являются *скейлы* (*scale*). *Скейл* — обратимая функция отображения из источника данных в геометрический тип данных (то есть тип данных, характеризующих эстетический атрибут, к которому привязан данный скейл).

При чтении графика в предыдущем разделе, чтобы восстановить данные мы вычисляли функцию обратную скейлу. Делали это мы при помощи осей и легенд (гидов). Здесь становится понятна их настоящая природа: гиды — это график скейла. Оси и легенды не имеют какого-либо стандарта и сильно зависят от движка визуализации, однако должны тем или иным способом визуализировать скейл.

Позиционные атрибуты немного отличаются от непозиционных. Во-первых, они определяются не геометрической сущностью, а системой координат. Здесь стоит отметить, что в данной работе и в разработанной библиотеке мы работает с двухмерными графиками с декартовой системой координат (имеющая оси X и Y), однако тривиально обобщается на другие системы координат с произвольным числом измерений. Во-вторых, им соответствует особый геометрический тип — относительная позиция. На полотне эти значения определяются осями координат, но в большинстве библиотек для построения графиков положение этих осей на полотне определяются автоматически, а значит и область значений скейла для позиционного атрибута определяется автоматически.

2. Промежуточное представление графика

В этой главе рассматривается процесс разработки промежуточного представления графиков.

2.1. Данные

Для реализации вышеописанной модели данных,используется следующий тип данных:

```
typealias NamedData = Map<String, List<Any>>
```

Ключи данной хэш-таблицы — название столбцов, значения — списки с non-null значениями соответствующих столбцов. Эти списки должны иметь одинаковый столбец.

Для источника данных используется тип

```
data class DataSource<out T: Any>(val id: String, val type: KType).
```

Как видно, он не содержит данных, а лишь ссылается на столбец в таблице. С помощью рефлексии мы сохраняем тип данных.

2.2. Скейлы

Базовый интерфейс инкапсуляуции скейла — `Scale`.

В теории, мы определили скейл как функцию, которая преобразует значение из источника данных в некоторое геометрическое значение. Однако при реализации мы сталкиваемся с рядом трудностей.

Функцию в привычном в виде сложно сериализовывать, и на практике пользователю редко нужно определять скейл как сложную нетривиальную функцию. Большинство представление графиков и языков описания графиков поступает следующим образом — делит скейлы на 2 вида:

1. *Категориальный* (также *дискректный / кусочный / номинальный*) скейл. Чтобы его задать, достаточно определить его конечное множество определения — категории, и соответствующие каждой из категорий значение функции.

   Категориальные скейлы наследуют интерфейс `CategoricalScale`

2. *Континуальный* (или *непрерывный*) скейл. Его области определения и значений - это отрезки линейно упорядоченных континуальных множеств (область определения в большинстве случаев - числовой отрезок; областью значений может являться диапазон размера или отрезок линейного цветного градиента), между которыми некоторое строится функция соответствия f, удовлетворяющая равенствам

$$f(d1) = r1,$$
$$f(d2) = r2,$$

где $d1$ и $d2$ — пределы множества определения, а $r1$ и $r2$ — пределы множества значений. Чаще всего соответствие линейное, однако в теории может быть произвольной обратимой функцией (возможные функции зависят исключительно от движка).

Континуальные скейлы наследуют интерфейс `ContinuousScale`

Другая немаловажная деталь реализации — позиционные скейлы, то есть скейлы, отображающие в позиционные эстетические атрибуты (координаты). Как указывалось выше, в большинстве движков область значений позиционных скейлов не указывается явно. Для простоты, его можно представить как отрезок $[0, 1]$ (где 0 соответствует началу оси, а 1 — концу.). В случае категориального скейла, обычно(но не всегда) область определения - конечное множество равномерно расположенных точек этого отрезка. В случае континуального точно также строится сопоставление между отрезками, линейное или некоторое другое.

Позиционные скейлы наследуют интерфейс `PositionalScale`, непозициионные наследуют интерфейс `NonPositionalScale`

Помимо этого, пользователь может не указывать скейл явно (и в этом случае он определяется либо транслятором, либо движком). Такие скейлы наследуют интерфейс `DefaultScale`. Пользователь может не задавать скейл ни в каком виде, может задать только тип (категориальный или континуальный).

Также при реализации скейлов необходимо помнить про типы. Во первых, у скейлов должны быть типовые параметры — тип области определения

и тип области значений. Однако, у позиционных скейлов тип области значений неявный. Также не стоит забывать, что в Kotlin типовые параметры дженерик–классов не сохраняются. Поэтому, информацию о типах необходимо сохранять с помощью рефлексии Котлина и хранить в полях, имеющих тип `KType`. Стоит отметить, что в реализации они хранятся не непосредственно в классах, инкапсулирующих скейлы, а в классах, инкапсулирующих маппинг, так как при использовании неуточненных скейлов мы не знаем тип области значений, в то время как в маппинге всегда его знаем.

Вдобавок, важно иметь в виду, что скейл может быть задан образом, отличным от вышеописанного. Создание и интерпретация такого скейла зависит от движка и его транслятора. Для таких целей существует интерфейс `CustomScale`.

Безусловно, данная модель скейла не всеобъемлющая. В частности, скейл может быть функцией нескольких переменных. Однако, в большинстве движков такая интерпретация невозможна, и в данной работе скейл подразумевается как функция 1 аргумента.

Полная иерархия скейлов изображена на схемах 3, 4 и 5.

2.2.1 Гиды

Как упомянались выше, оси и легенды не имеют какого-либо стандарта и сильно зависят от движка визуализации. Для их реализации, необходимо наследовать интерфейсы `Axis` и `Legend`.

2.3. Эстетические атрибуты

Базовый интерфейс для всех эстетических аттрибутов — `Aes`.

В представлении движка визуализации эстетические атрибуты — всего лишь текстовая отметка. На практике при реализации элементов представления графика и предметно-ориентированного языка нам необходимо ввести небольшую иерархию эстетические атрибутов, наложив некоторые ограничения, а именно:

1. Позиционные и непозиционные эстетики (в которые действуют позиционные и непозиционные скейлы соответственно);

Они наследуют `PositionalAes` и `NonPositionalAes` соответственно.

2. Эстетики, которые можно связать с отображением из данных (а не только присвоить им конкретное значение) — `MappableAes`.

3. Эстетические атрибуты, в которые может действовать непосредственный скейл — `ScalableAes`.

   Что под этим подразумевается? Существуют так называемые «подпозиционные» атрибуты, которые отвечают за позиционирование вдоль некоторой оси, но не являются атрибутами, соответствующими этой оси непосредственно (например - параметры «коробки с усами»). Их скейл определяется скейлом для "старшего"атрибута.

4. Непозиционные эстетические атрибуты типизируются некоторым геометрическим значением. Например, цвет — специальным типом `Color`, размер, ширина, прозрачность - типом `Double`.

Система наследования эстетических атрибутов отображена на схеме 6.

В пакете `ir.aes` представлены примеры базовых эстетических аттрибутов:

```
val X = ScalablePositionalAes("x")
val Y = ScalablePositionalAes("y")

val SIZE = MappableNonPositionalAes<Double>("size")
val COLOR = MappableNonPositionalAes<Color>("color")
val BORDER_COLOR = NonPositionalAes<Color>("border_color")
```

2.4. Связывание

Процесс определения значения эстетического атрибута называется *связыванием*. Мы можем задать это значение некоторым постоянным значением, и тогда это называется *сеттинг*, либо задать скейл в данный атрибут, и тогда это называется *маппинг*.

Интерфейс `Setting` инкапсулирует сеттинг. На данный момент сеттинг определен только для непозиционных эстетических атрибутов (в виду

сложности поддержки в движках). NonPositionalSetting<T: Any> содержит эстетический атрибут и присвоенного значения соответствующего типа.

Интерфейс Mapping инкапсулирует маппинг. Он содержит информацию об эстетическом атрибуте, а также о DataSource (к которому, возможно, применили скейл), из которого происходит отображение на данный атрибут.

SourceScaled — это инкапсуляция для результата применения скейла на данные, то есть обертка, содержащая DataSource и применяемый к нему скейл. Наследники SourceScaled отличаются типом содержимого ими скейла и имеют простую иерархию, основанную на иерархию Scale. Данные классы параметризуются типами, параметризующих соответствующий скейл. Тип DataSource и области определения скейла должны совпадать.

Аналогичным образом, маппинги отличаются друг от друга типом внутреннего SourceScaled, и имеют схожую иерархию. Отдельно стоит отметить NonScalablePositionalMapping - он предназначен для обертки маппинга на NonScalablePositionalAes, и содержит "сырой"DataSource, а не преобразованный скейлом. Остальные же наследуются от интерфейса ScaledMapping и имеют внутри SourceScaled. Как писалось выше, мы сохраняем типы области определения и значений с помощью рефлексии; они имеют тип KType.

Маппинги имеют следующие ограничения:

1. Маппинг на позиционный эстетический атрибут может иметь только позиционный скейл. Аналогично для непозиционных.

2. В случае непозиционных маппингов не по-умолчанию также должно выполняться ограничение — тип, параметризующий непозиционный эстетический атрибут должен совпадать с типом области значений скейла.

2.5. Слой

Понятие *"слой"* взято из [3]. *Слой* — это набор данных, геометрическая сущность и набор маппингов и сеттингов. График может состоять как из одного, так и и из нескольких слоев.

В промежуточном представлении слой представлен следующим образом:

```
data class Layer(
    val data: NamedData,
    val geom: Geom,
    val mappings: Map<Aes, Mapping>,
    val settings: Map<Aes, Setting>,
    val features: Map<FeatureName, LayerFeature>
)
```

`Geom` — инкапсуляция геометрической сущности.

2.6. Особенности

Особенности — основной механизм для добавления вещей, специфичных для движков. В промежуточном представлении они представлены двумя интерфейсами — `PlotFeature` и `LayerFeature`, определяющие особенности графика в целом и конкретного слоя соответственно. Эти интерфейсы реализуются в модулях, отвечающих за конкретные движки. Для работы с ними используется специальный класс `FeatureName` — обертка над строковым литералом для идентификации особенности.

2.7. График

Сам график представлен крайне просто — это набор слоев, настройка верстки и набор особенностей:

```
data class Plot(
    val layers: List<Layer>,
    val layout: Layout,
    val features: Map<FeatureName, PlotFeature>
)
```

2.8. Схема промежуточного представления

Общая схема промежуточного представления показа на рисунке 7.

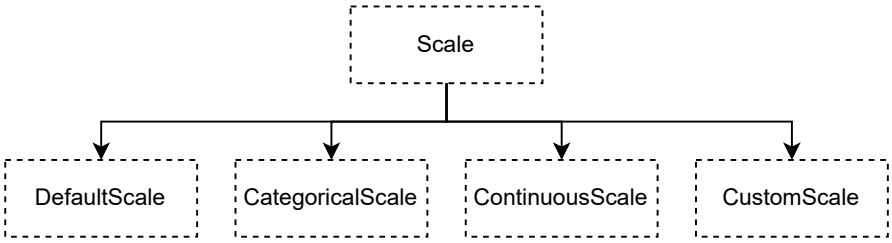

Рис. 3: Интерфейсы скейлов.

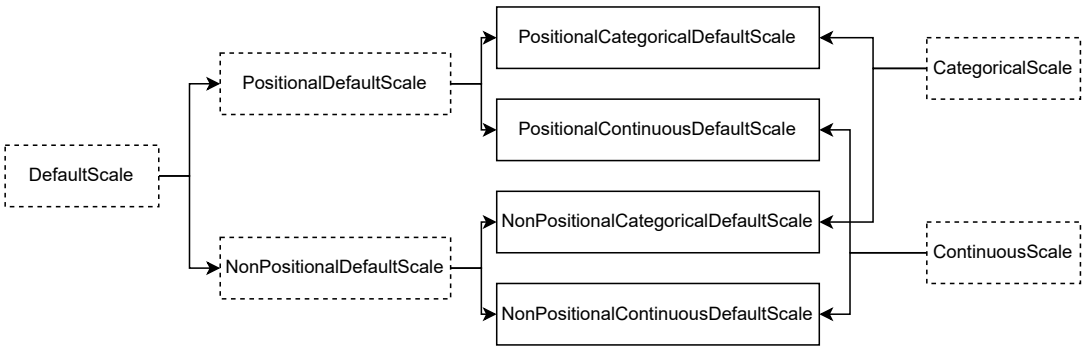

Рис. 4: Скейлы по умолчанию.

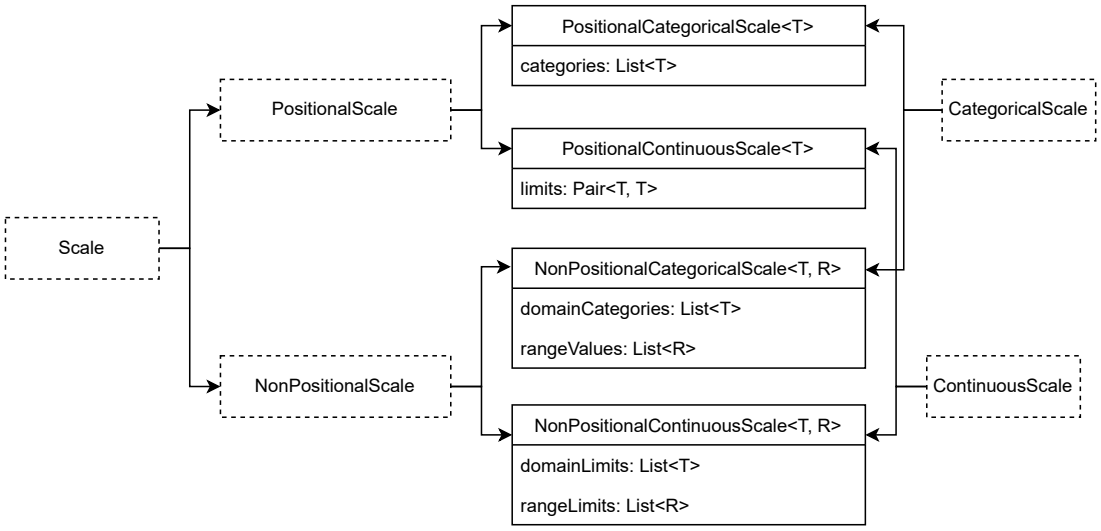

Рис. 5: Основные скейлы.

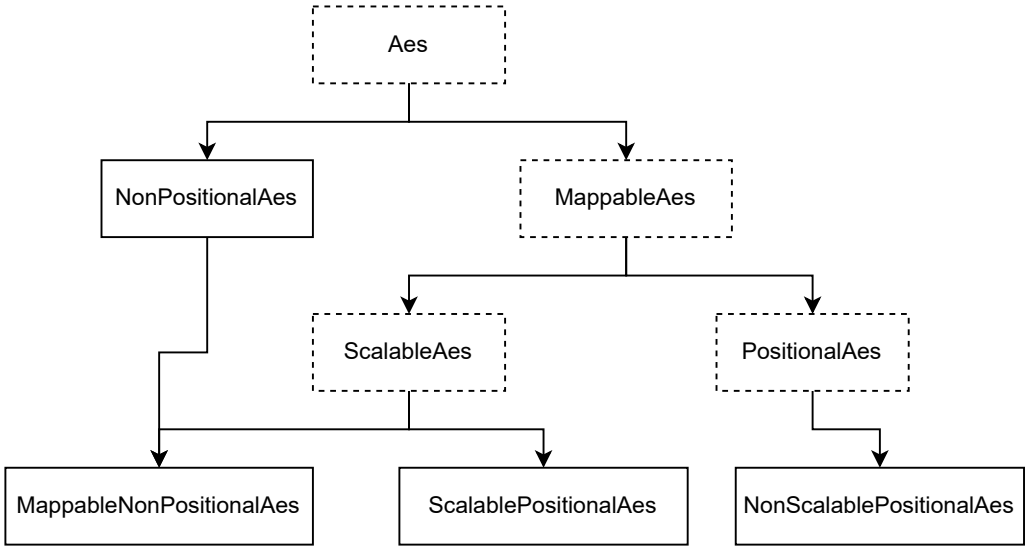

Рис. 6: Иерархия Aes.

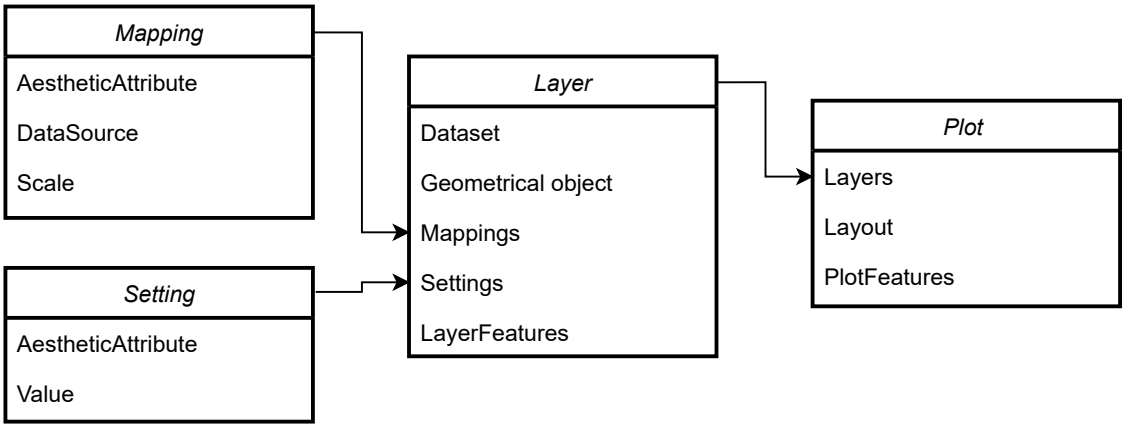

Рис. 7: Промежуточное представление.

3. Предметно-ориентированный язык

В данной главе подробно разбирается разработанный в ходе работы предметно-ориентированный язык для построения графиков.

3.1. Контексты

Для построения предметно-ориентированного языка используется парадигма контекстно-ориентированного программирования [13]. В синтаксисе языка Kotlin функции с определенным типом аргумента могут создавать контекст, то есть блока кода с получателем (указателем на экземпляр класса). Это позволяет определить для класса-получателя функции расширения и использовать их в данном контексте. В нашем языке определены следующие контексты.

3.1.1 Базовый контекст

`BaseContext` — абстрактный класс, от которого наследуются все остальные основные контексты.

Его ключевой особенностью является то, что внутри него можно создавать связывание (подробнее о них — в соответствующем разделе 3.4). Для их хранения `BaseContext` использует внутреннюю структуру `BindingCollector`. Изменяемое поле `data` — ссылка на изменяемый набор данных. По умолчанию, мы должны его наследовать из родительского контекста.

Для реализации связывания контексты имеют поля, соответсвующие эстетическим атрибутам. В базовом контексте определены поля `x` и `y` с соответствующими эстетическими атрибутами.

3.1.2 Контекст графика

Класс `PlotContext` наследуется от `BaseContext`.

Это верхний контекст, создающийся при вызове функции `plot`. Он содержит список слоев, хэш-таблицу особенностей (которая определена как `val features: MutableMap<FeatureName, PlotFeature>`), а также настройки верстки.

Для создания слоев используются расширения данного контекста. Например, в стандартной API есть 3 функции-расширения для контекста графика, которые создают слой и добавляют их в список.

```
fun PlotContext.points(block: PointsContext.() -> Unit): Unit

fun PlotContext.bars(block: BarsContext.() -> Unit): Unit

fun PlotContext.line(block: LineContext.() -> Unit): Unit
```

3.1.3 Контекст слоя

Интерфейс `LayerContext` позволяет создавать слои. В нем определена хэш-таблица особенностей (`LayerFeature`).

Наследники данного интерфейса — контексты, определяемые некоторой геометрической сущностью. В них есть поля, соответствующие эстетическим атрибутам, характерным для данной сущности.

В базовом API есть 3 таких контекста, соответствующие «точке», «линии» и «столбцу» — `PointsContext`, `LineContext`, `BarsContext` соответственно. Для их создания используются функции, упомянутые в 3.1.2.

3.2. Данные

Для визуализации данных с помощью библиотеки необходимо привести их в вид `NamedData`.

Для создания источника данных необходимо использовать функцию

```
inline fun <reified T: Any> source(id: String) : DataSource<T>.
```

Она определена вне какого-либо контекста.

3.3. Скейлы

Создать определенный скейл можно с помощью соответствующей функции.

Функции без аргументов `continuous()` и `continuousPos()` возвращают соответственно непозиционные и позиционный континуальные скейлы по умолчанию.

Функция

```
fun <DomainType : Any, RangeType : Any> continuous(
   domainLimits: Pair<DomainType, DomainType>? = null,
   rangeLimits: Pair<RangeType, RangeType>? = null,
): NonPositionalContinuousScale
```

создает непозиционный континуальный скейл с заданными пределами области определения и области значений. В аналогичной функции для позционного скейла только один аргумент — пределы области определения.

```
fun <DomainType : Any> continuousPos(
   limits: Pair<DomainType, DomainType>? = null
): PositionalContinuousScale
```

Аналогичным образом работают функции для создания категориальный скейлов `categorical()` и `categoricalPos()`, а также

```
fun <DomainType : Any, RangeType : Any> categorical(
   domainCategories: List<DomainType>? = null,
   rangeValues: List<RangeType>? = null,
): NonPositionalCategoricalScale
```

создающая категориальный скейл, область определения которого задается списком, как и соответствующие значения функции — область значений. В позиционном варианте единственный аргумент — список категорий.

```
fun <DomainType : Any> categoricalPos(
   categories: List<DomainType>? = null
): PositionalCategoricalScale
```

Далее, нам необходимо применить скейл на источник данных. Для этого у `DataSource` определено семейство функций-расширений `scaled`, создающие `SourceScaled`. Эти функции отличаются типом применяемого скейла и типом возврщаемого `SourceScaled`:

```
fun <DomainType : Any> DataSource<DomainType>.scaled():
   SourceScaledUnspecifiedDefault

fun <DomainType : Any> DataSource<DomainType>.scaled(
   scale: PositionalDefaultScale
): SourceScaledPositionalDefault

fun <DomainType : Any> DataSource<DomainType>.scaled(
   scale: NonPositionalDefaultScale
): SourceScaledNonPositionalDefault

fun <DomainType : Any> DataSource<DomainType>.scaled(
    scale: PositionalScale<DomainType>
): SourceScaledPositional

fun <DomainType : Any, RangeType : Any> DataSource<DomainType>
        .scaled(scale: NonPositionalScale<DomainType, RangeType>)
: SourceScaledNonPositional
```

Также здесь сохраняется ограничение на то, что источник данных и область определения скейла имеют имеют одинаковый тип (в случае, если второй явно задан).

Все вышеперечисленные функции объявлены вне какого-либо контекста. Они могут быть использованы для создания скейлов и источников с примененным скейлом для нескольких слоев и даже графиков.

3.4. Связывание

Связывания определены в `BaseContext` 3.1.1.

Реализованы они с помощью «вызова» (то есть применения оператора «круглые скобки») полей, соответствующих эстетическим атрибутам.

Функция

```
operator fun <T : Any> NonPositionalAes<T>.invoke(value: T)
```

создает сеттинг для данного непозиционного атрибута и данного значения соотвествующего типа.

Отдельно стоит выделить функцию

```
inline operator fun <reified DomainType : Any>
NonScalablePositionalAes.invoke(source: DataSource<DomainType>),
```

который создает `NonScalablePositionalMapping` (см. 2.4).

Остальные же функции создают маппинги в соответствии с типом атрибута и скейла. При этом выполняются все ограничения, описанные в 2.4.

3.5. Добавление особенностей

Несмотря на то, что в стандартном API нет примеров работы с особенностями, механизм работы с ними предполагается следующий — создание функций-расширений контекста, создающих и добавляющих в соответствующую (в контексте слоя или графика) хэш-таблицу особенности, либо полей-расширений, изменение которых влечет за собой те же действия.

Примеры подобных механизмов приведены в главе 4, посвященной движкам.

3.6. Схема предметно-ориентированного языка

На рисунке 8 показана общая схема предметно-ориентированного языка

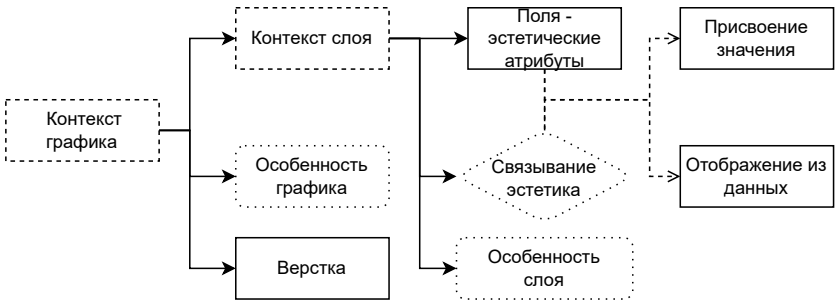

Рис. 8: Предметно-ориентированный язык.

4. Трансляторы и особенности движков

4.1. Lets-Plot

В данном разделе описывается то, что входит не в стандартный API, а в специальный API для движка Lets-Plot.

4.1.1 Особенности

В рамках работы было добавлено несколько особенностей движка Lets-Plot.

Фасетирование — это разделение графика на несколько графиков с одинаковым набором маппингов и скейлов, полученных путем разделения набора данных.

Фасетирование — это особенность графика:

```
class FacetGridFeature : PlotFeature
```

Внутри языка реализация фасетирования похожа на добавление слоя. Функция-расширение контекста графика

```
fun PlotContext.facetGrid(block: FacetGridFeature.() -> Unit): Unit
```

создает контекст фасетирования и добавляет полученную особенность в слой. В нем можно указать, по какому источнику данных происходит по осям X и Y (ситнаксис этого похож на маппинг, но без скейла), а также порядок (по-возрастанию или по-убыванию свойства из источника данных) расположения подграфиков.

Выглядит синтаксис фасетирования следующим образом:

```
plot {
   ... // layers
   facetGrid {
      x(originSource)
      y(nocSource)

      yOrder = OrderDirection.DESCENDING
```

```
    }
}
```

Также была реализована особенность под названием «позиционирование». Это особенность, которая позволяет настраивать взаиморасположение объектов внутри слоя. Сама особенность представлена `sealed` классом с приватным конструктором:

```
sealed class Position private constructor(val name: String) :
LayerFeature
```

Внутри него реализованы все его наследники, определяемые возможными способами позиционирования:

```
object Identity : Position("identity")
object Stack : Position("stack")
class Dodge(val width: Number? = null) : Position("dodge")
class Jitter(val width: Number? = null, val height: Number? = null) :
    Position("jitter")
class Nudge(val x: Number? = null, val y: Number? = null) :
    Position("nudge")
class JitterDodge(
    val dodgeWidth: Number? = null,
    val jitterWidth: Number? = null,
    val jitterHeight: Number? = null,
) : Position("jitter_dodge")
```

Внутри языка особенность реализована как поле-расширение контекста слоя:

```
var LayerContext.position: Position
```

При присваивании в него некоторого значения, соответствующее значение добавится в слой.

Помимо вышеперечисленных особенностей, в API для Lets-Plot реализованы новые слои для новых геометрических форм и с новыми эстетическими атрибутами, а также новые значения для базовых эстетических атрибутов (конкретно — новые значения `Symbol` и `LineType`)

4.1.2 Транслятор

Специфика Lets-Plot, как и его идейного вдохновителя ggplot2, заключается в использовании Grammar of graphics. Благодаря следованию идеям этой грамматике и в данной работе, реализация разработанного промежуточного представления в представление в Lets-Plot реализуется довольно несложно. Идея слоев была также взята из ggplot2, поэтому транслировать слои из нашего представления в слои Lets-Plot также не сильно затруднительно.

В представлении Lets-Plot есть одна особенность — в нем может быть только один скейл для каждого эстетика. Более того, он не привязан к конкретному маппингу или слою.

Чтобы обернуть слой, мы создаем класс — наследник базового слоя в Lets-Plot, и преобразуем наши геометрические сущности, маппинги, сеттинги и скейлы в таковые в API Lets-Plot. Дополнительная обработка скейлов по-умолчанию не требуется.

Полный код транслятора представлен в пакете `letsplot.translator`

4.2. ECharts

В данном разделе описываются детали API для движка ECharts.

4.2.1 Особенности

Особенность «стек» чем-то похожа на `Position.Stack` из Lets-Plot. Но в отличии от нее, она работает не внутри одного слоя, а настраивает стек, то есть укладку объектов типа «столбец» из разных слоев, имеющие одинаковое название стека.

Реализована особенность как поле-расширение контекста слоя столбцов.

```
var BarsContext.stack: Stack
```

Класс `Stack` оборачивает название стека, и создает функцией

```
fun stack(name: String): Stack
```

Также реализовано несколько видов анимации.

Анимация появления настраивается в соответствующем контексте, создаваемой функцией-расширением для контекста графика:

```
fun PlotContext.animation(block: AnimationFeature.() -> Unit): Unit


class AnimationFeature(
    var enable: Boolean = true,
    var threshold: Int = 2000,
    var duration: Int = 1000,
    var easing: AnimationEasing = AnimationEasing.CUBIC_OUT,
    var delay: Int = 0
): PlotFeature
```

Анимация с изменением данных — это анимация, заданная некоторым графиком и функцией изменения данных:

```
fun Plot.withDataChangeAnimation(
    interval: Int,
    dataChange: NamedData.() -> Unit
): DataChangeAnimation
```

Более общий случай анимации — анимация, состоящая из ограниченного набора графиков:

```
data class PlotChangeAnimation(
    val plots: List<Plot>,
    val interval: Int,
)
```

Подробнее о реализации этих 2 типов анимации — в разделе 5.1.1.

Также реализованы новые слои для новых геометрических форм. Добавлены новые значения эстетических атрибутов. Например, новые виды цветов — `LinearGradientColor` и `RadialGradientColor`.

4.2.2 Транслятор

Формат представления графика в ECharts — JSON. Для работы с ним в Koltin, были созданы соответствующие дата-классы.

В представлении графика в ECharts можно выделить элементы, соответствующими элементами промежуточного:

- `Series` — это слой;
- `Axis` — это позиционный скейл;
- `VisualMap` — это объединение маппинга и скейла (для непозиционных эстетических атрибутов).

Основная трудность — отсутствие скейлов по-умолчанию. Для их обработки необходимо написать генерацию ожидаемых скейлов.

5. Технические детали и аналоги

В данной главе описываются технологии разработки и тестирования, приводится сравнение с аналогами. Также в главе говорится об интеграции с другими проектами Kotlin for Data Science.

5.1. Интеграции

5.1.1 Kotlin Jupyter Kernel

Kotlin Jupyter Kernel — это ядро для Jupyter Notebook, позволяющее запускать в нем код на Kotlin.

Интеграция с данным ядром является частью API. В рамках нее, помимо импорта всех публичных пакетов, осуществлена реализация рендеринга графиков на движках.

Для визуализации графиков Lets-Plot используется интеграция, являющаяся частью API Lets-Plot.

Для рендера графиков ECharts, в том числе анимированных (см. 4.2.1) была написана интеграция, использующая API Kotlin Jupyter, язык JavaScript и API Echarts. Она является частью API соответствующего модуля.

5.1.2 Kotlin Dataframe

Kotlin Dataframe — библиотека для работы с табличными типами данных. Интеграция с данной бибилиотекой представлена в модуле `dataframe`. В ее рамках реализовано несколько нововведений.

`DataFrame` можно использовать в качестве источника данных, для этого используется функция-расширение

```
fun <T> DataFrame<T>.plot(
   block: context(DataFrame<T>, PlotContext).() -> Unit
): Plot
```

Эта функция использует новую особенность Kotlin — контексты с несколькими получателями.

Также, вместо `DataSource` для создания маппинга можно использовать ссылки на колонки `DataFrame` — `ColumnReference`. Примечательно то, что в плагине для Jupyter Notebook (а в будущем, в плагине для компилятора) они генерируются автоматически (как поля-расширения), и пользователь не должен создать их вручную, в отличие от `DataSource`. В сочетании с предыдущей функцией это значительно упрощает синтаксис.

Однако, неоднократно используемая при реализации вышеупомянутая особенность (контекст с несколькими получателями) в данный момент не поддерживается компилятором полноценно, поэтому данный модуль нельзя публиковать как артефакт системы maven и использовать в Kotlin Script (а следовательно, в Kotlin Jupyter). Тем не менее, данная интеграция полностью работает.

5.2. Тестирование

В библиотеке используется автотестирование с использованием библиотеки Kotlin Test.

Детали тестирования:

- Юнит-тесты и интеграционные тесты основного API (предметно-ориентированн
  языка и промежуточного представления);
- Юнит-тесты инъекций в предметно-ориентированный язык;
- Юнит-тесты и интеграционные тесты трансляторов;

5.3. Используемые технологии

В рамках работы использовались следующие технологии:

1. Для написания большей части кода использовался язык программирования Kotlin и его стандартная библиотека [6];
2. Система сборки Gradle [14];
3. Для исследования использовался язык программирования R [4] и библиотека ggplot2 [2];

4. Для исследования и реализации библиотеки была использована библиотека Kotlin Lets-Plot [5];

5. Язык JavaScript [8] и фреймворк Apache Echarts [7];

6. Kotlin Jupyter Kernel API [9];

7. Библиотека Kotlin Dataframe [11];

8. Библиотека Kotlin Test [15];

5.4. Сравнение с аналогами

Если говорить про представление графика, то, безусловно, в каждой библиотеке для визуализации графиков есть свое представление графика. Однако, каждое из них очень сильно привязано к движку. Архитектура промежуточного представления, разрабатываемая в рамках данной работы, позволяет хранить график в обобщенном виде с возможностью добавления особенностей, зависящих от движка, не теряя универсальности. Это позволяет использовать код графика (как в предметно-ориентированном языке для построения, так и в промежуточном представлении) для нескольких движков с минимальными изменениями.

Среди прочих выделяется библиотека Vega Lite. Она также основана на Grammar of graphics, довольно простая и удобная. Однако, как и в других библиотеках в ее представлении есть вещи, завязанные на движке. А также транслировать из ее представления в другие не очень удобно, как классы Kotlin (в том числе, из-за наличия типизации и полиморфизма).

Предметно-ориентированный язык на Kotlin. Для языка Kotlin существует не так много библиотек для построения графиков. Библиотека Lets-Plot, для которой изначально разрабатывалось новое API, имеет в своем синтаксисе ряд существенных недостатков, наследуемых от ggplot2:

1. Далеко не идеальная структура. Во-первых, в синтаксисе отсутствует какая-либо иерархия. Объявление графика, слоев, настройка верстки, скейлов и осей находится на одном уровне. Во-вторых, скейл никак не связан с маппингами.

2. Можно задать только один скейл для каждого эстетического атрибута.

3. Отсутствие типизации. Большинство полей имеют тип `Any`.

4. Отсутствие типобезопасности. В частности, типы скейлов никак не связаны с типами источников данных, невозможна проверка типов.

Библиотека kravis [16] схожа с Lets-Plot: ее API основан на ggplot2 и имеет те же недостатки.

Библиотека plotly.kt [17] предоставляет Kotlin API для библиотеки plotly [18]. В отличие от всех остальных аналогов, в ней используется контексты для построения предметно-ориентированного языка, также она содержит интересные идеи в отношении задания маппингов и сеттингов. Однако API сильно ограничен возможностями движка plotly и не подходит для описания графика в общем виде.

Также существует библиотека charts.kt. Она имеет закрытый исходный код.

Заключение

В ходе работы была разработаны предметно-ориентированный язык для построения графиков, который эффективно использует средства языка для достижения выразительности, читаемости и удобства. Он разработан таким образом, что в него можно вставлять особенности, зависящие от движка визуализации.

Разработанное промежуточное представление графика позволяет эффективно использовать один и тот же код для построения графика для визуализации в различных движках рендеринга. Благодаря простоте и универсальности, трансляции осуществляться во многие движки и несложно реализуется.

Реализованные трансляторы позволяют использовать разработанный язык для построения графиков и визуализировать их с помощью движков Lets-Plot и Apache Echarts.

Реализованные интеграции позволяют работать с важнейшими элементами экосистемы Kotlin for Data Science.

В результате работы все поставленные задачи были выполнены.

Разработанная библиотека в ближайшем будущем будет опубликована как часть официальных библиотек для языка Kotlin — kotlinx.

Весь код разработанного программного обеспечения доступен в публичных репозиториях на платформе GitHub:

- `https://github.com/AndreiKingsley/ggdsl`
- `https://github.com/AndreiKingsley/ggdsl-lets-plot`
- `https://github.com/AndreiKingsley/ggdsl-echarts`
- `https://github.com/AndreiKingsley/ggdsl-dataframe`

Список литературы

[1] L. Wilkinson: The Grammar of Graphics, Springer, 1999. ISBN 0-387-98774-6.

[2] Документация ggplot2. URL: https://ggplot2.tidyverse.org/index.html

[3] H. Wickham, D. Navarro, T. L. Pedersen: ggplot2: elegant graphics for data analysis. URL: https://ggplot2-book.org

[4] Язык программирования R. URL: https://www.r-project.org

[5] Библиотека Lets-Plot. URL: https://github.com/JetBrains/lets-plot-kotlin

[6] Язык программирования Kotlin. URL: https://kotlinlang.org

[7] Библиотека Apache ECharts. URL: https://echarts.apache.org/en/index.html

[8] Язык программирования JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[9] Kotlin Jupyter Kernel. URL: https://github.com/Kotlin/kotlin-jupyter

[10] Jupyter Notebook. URL: https://jupyter.org

[11] Библиотека Kotlin Dataframe. URL: https://github.com/Kotlin/dataframe

[12] The official U.S. government source for fuel economy information. URL: https://fueleconomy.gov

[13] A. Nozik: An introduction to context-oriented programming in Kotlin. URL: https://proandroiddev.com/an-introduction-context-oriented-programming-in-kotlin-2e79d316b0a2

[14] Система сборки Gradle. URL: https://gradle.org

[15] Библиотека Kotlin Test. URL: https://kotlinlang.org/api/latest/kotlin.test/

[16] Библиотека kravis. URL: https://github.com/holgerbrandl/kravis

[17] Библиотека plotly.kt. URL: https://github.com/mipt-npm/plotly.kt

[18] Библиотека plotly. URL: https://plotly.com