Сложности дизайна декларативного type-safe API для библиотеки визуализации данных Scalaz-Viz.

Evgeniy T

2018

Опрос

- Functor
- Apply, Applicative
- Monad
- Free, FreeAp, Fix
- Coproduct
- Lens, Prism, Optics

Прежде чем начать / Disclaimer

- В данном докладе разбирается ”игрушечный” пример, а не финальная реализация библиотеки Scalaz-Viz
- Подход к реализации этой библиотеки может измениться в любой момент, т.к над проектом работает коллектив авторов и каждый преследует разные цели.
- Возможно выбор FreeAp или других подходов для реализации подобного функционала является не оптимальным, но я надеюсь, что это будет интересным примером для дальнейшего изучения Free или Tagless Finall под ваши нужды.

- *Source Code:* `https://github.com/strobe/scalaz-viz/` (branch: *ekb-meetup-29-11-2018*)
- *Слайды:* `https://github.com/strobe/ekb-meetup-29-11-2018-slides/`

Почему возникла необходимость в еще одной библиотеки?

В Scala экосистеме ситуация с библиотеками визуализации далека от идеальной:

- Множество оберток над java библиотеками, которые предлагают немногим больше, чем простые OO/императивные API
- Множество заброшенных проектов
- Обертки над JavaScript библиотеками, такими как D3 (которые покрывают узкий набор требований, особенно если учитывать потребности data science)

Что сейчас есть для визуализации в Scala? Множество библиотек которые частично решают задачу.

Library	Wrapper	Backends	Typesafe/FP	Clear API	Streams	S. overflows	Interactivity	Docs
Doodle	no	yes	yes	no (Free monads)	no	no	no	partial (at book)
Nspl	no	yes	partial?	no (Factory methods)	yes	?	partial	no
EvilPlot	no	partial	partial	yes	no?	?	no	yes
Vegas	yes	no	partial	no	no	?	yes	partial
Plot.ly	yes	no	partial	yes (Limited)	yes	?	yes	yes
Wisp	yes	partial	partial	no	no	?	no	no
BreezeViz	yes	no	partial	yes (Limited)	no	?	no	no
Smile	no	no	partial	yes (Limited)	no	?	no	partial

А что в других экосистемах?

Language	Library
R	ggplot2 Lattice
JavaScript	D3 Vega Vega-Lite Plotly
Java	JChart JFreeChart charts4j GRAL
Python	matplotlib pandas ggplot
MATLAB	MATLABs plotting library

Что нам нужно для идеальной библиотеки визуализации?

- гибкость композиции
- поддержка стриминга
- производительность
- различные бэкэнды (Scala.js, JVM, OpenGL, файлы ...)
- pure FP и строгая типизация
- интерактивность

А также:

- Понятное API
- Встроенная интеграция с множеством популярных проектов и библиотек:
  - Big Data Platforms (Spark, Flink, Kafka ...)
  - Notebooks (Jupyter, Zeppelin ...)
  - Scalaz (Analitycs, ML, Metrics, etc.)
- поддержка различных версий Scala

Архитектура

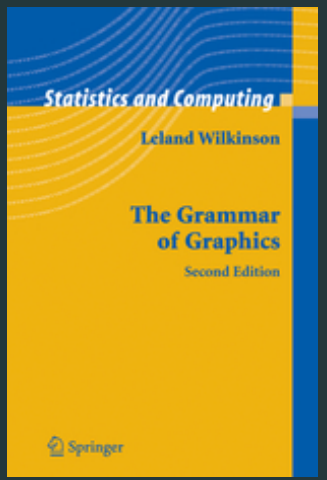
Для начала попробуем решить проблему API

Как должно выглядеть API библиотеки для пользователя?

Теоретические основы

Книга раскрывающая проблему дизайна API визуализации данных с теоретической стороны (в том числе со стороны теории категорий):

- **Leland Wilkinson, The Grammar of Graphic (2005)**



GGPLOT2 (R LANG) - ЭТО РЕАЛИЗАЦИЯ ИДЕЙ ИЗ КНИГИ.

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and geoms—visual marks that represent data points.

data **geom** x = F · y = A **coordinate system** **plot**

To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.

data **geom** x = F · y = A color = F size = A **coordinate system** **plot**

GGPLOT2 API

Complete the template below to build a graph.

```
ggplot (data = <DATA>) +
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
 stat = <STAT> , position = <POSITION> ) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>
```

required

Not required, sensible defaults supplied

Так как мы начинаем с API, нам нужен рендеринг бэкэнд - что выбрать?

Vega-Lite - Высокоуровневая спецификация. Позволяет быстро определять визуализации с помощью JSON синтаксиса и поддерживает интерактивные и много оконные визуализации.

Vega и Vega-Lite

Хорошее

- базируется на Grammar of Graphics
- декларативное API
- поддержка интерактивности

Плохое

- завязанно на семантику JS
- позволяет создавать конфликтующие декларации
- не достаточно типизировано
- Vega слишком низкоуровневая спецификация
- Vega Lite позволяет делать одну вещь множеством способов

Но не смотря на недостатки Vega-Lite имеет достаточно преимуществ чтобы можно было использовать в качестве рендеринг бэкенда на начальном этапе.

Пример Vega-Lite визуализации

- Набор точек в 2D координатах, классифицированных по одному признаку цветом.
- Среднее по Y координате.

Пример Vega-Lite спецификации

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v3.json",
  "description": "A scatterplot showing horsepower and miles per gallons for various cars.",
  "data": {
    "url": "data/cars.json"
  },
  "layer": [
    {
      "mark": "point",
      "encoding": {
        "x": { "field": "Horsepower", "type": "quantitative" },
        "y": { "field": "Miles_per_Gallon", "type": "quantitative" },
        "color": { "field": "Origin", "type": "nominal" }
      }
    },
    {
      "mark": "rule",
      "encoding": {
        "y": { "field": "Miles_per_Gallon", "type": "quantitative", "aggregate": "mean" }
      }
    }
  ],
  "config": {
    "style": {
      "rule": { "strokeWidth": 3, "stroke": "blue" }
    }
  }
}
```

Источник данных

Слой 1

Тип маркера

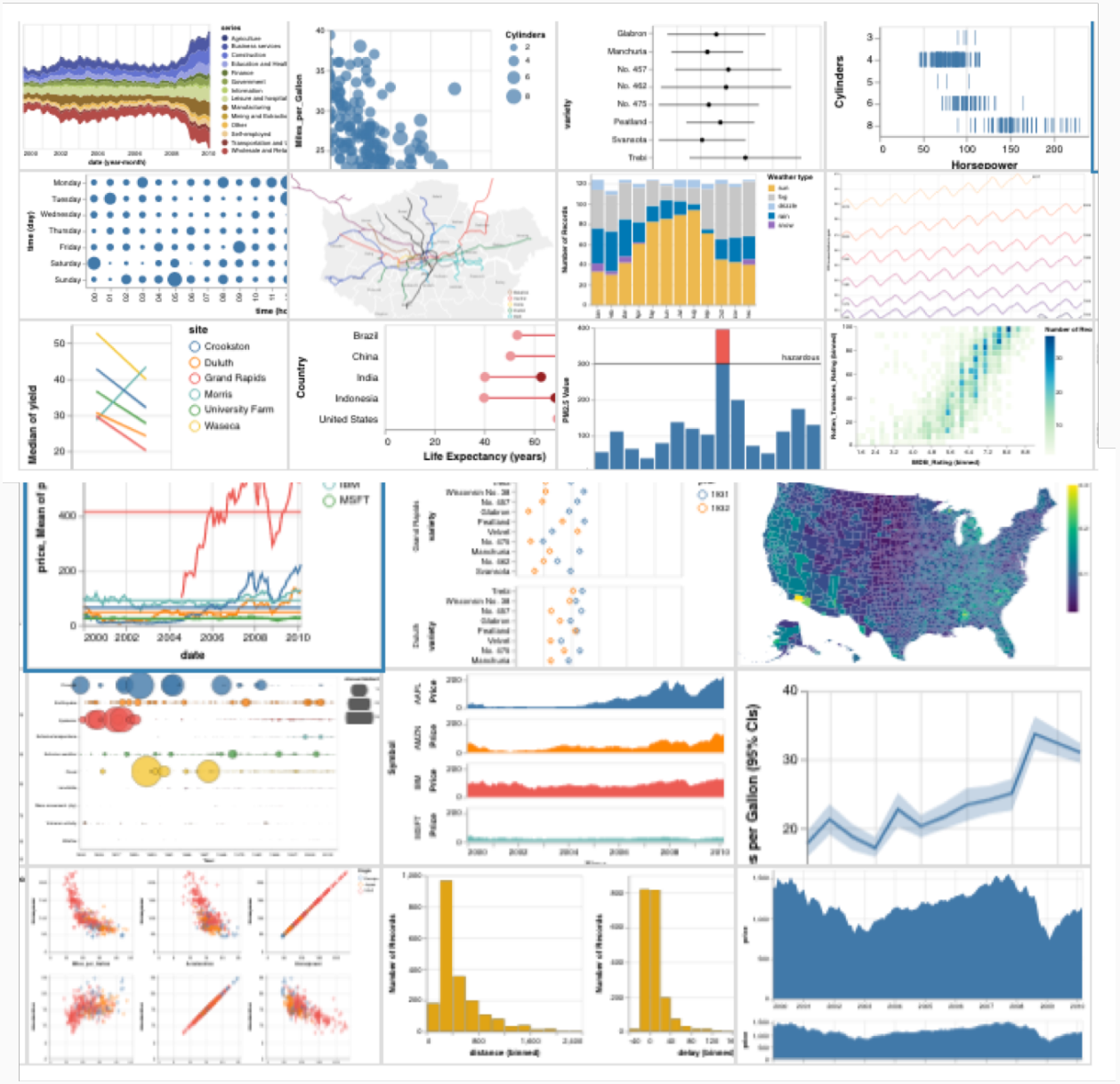
Слой 2

Функция трансформации

Тип маркера

Набор значений по умолчанию

Визуализации бывают разные (что может Vega-Lite?)



Практика.

Требования

Повторим основные требования к API:

- Декларативность
- Гибкость Композиции
- Строгая типизация
- Возможность трансляции в более низкоуровневое представление

Итак разберемся в доменной области (на самом деле нет)

Что нужно сделать чтобы получить визуализацию?

Figure 2.1 How to make a pie

Make a Pie - может быть даже одной функцией, которая делает всю работу, но для наших целей нужно намного больше гибкости.

Как то так?

```
trait Source[F[_], A] {
  def iterator(s: F[A]): Iterator[A]
}

implicit def listSource[A]: Source[List, A] = (s: List[A]) => s.toIterator

def plot[F[_], A](input: F[A])
  (implicit ev: Source[F, A]): IO[Unit] = IO[Unit] {
    ev.iterator(input).foreach(println)
}

plot(List(1,2,3)).unsafePerformIO()
```

- мы можем взять почти любую коллекцию или Stream и сделать интерпретатор, который будет универсальным и это хорошо, но что насчет гибкости композиции графических примитивов?

На самом деле все сложнее!

Grammar of Graphics определяет визуализацию как:

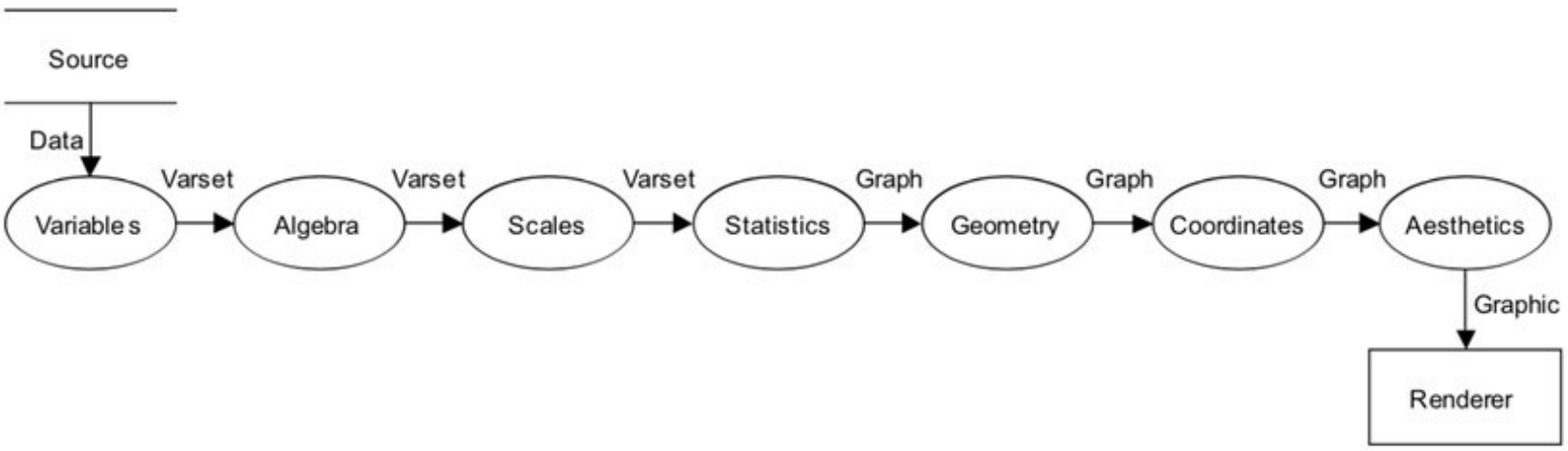
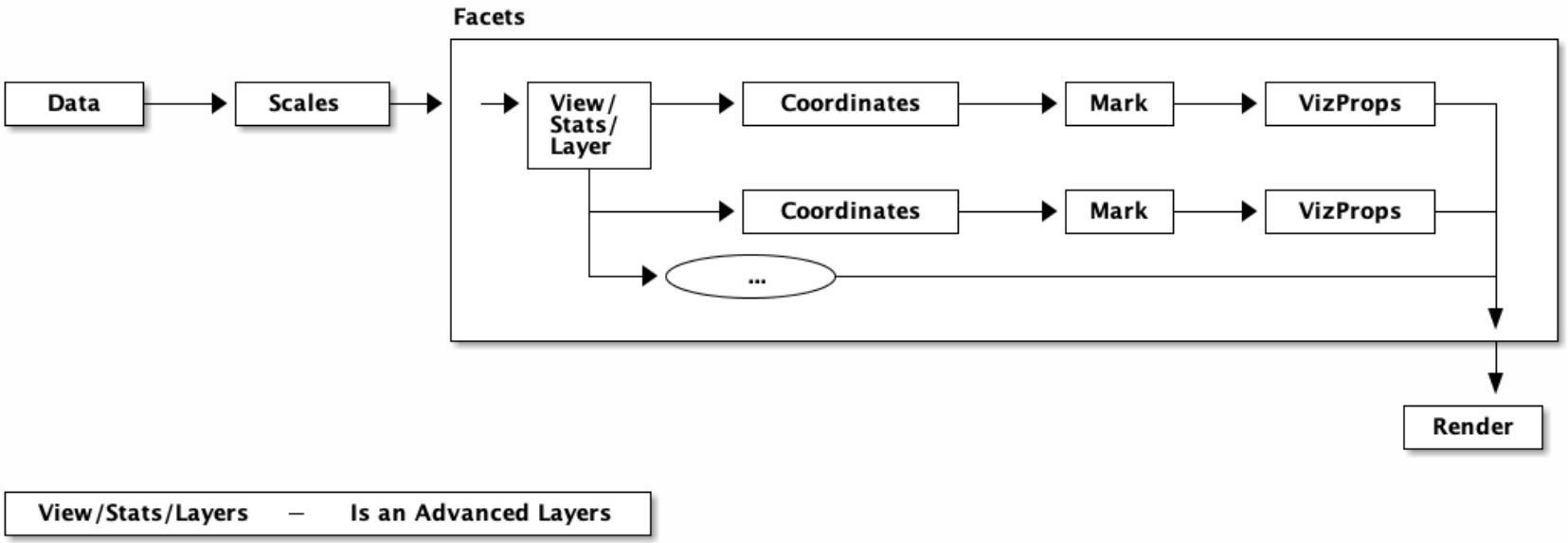


Figure 2.2 *From data to graphic*

Элементы Marks, Coordinates, Layers, Facets и т.д.



Как может выглядеть API для пользователя? (1)

Отсюда требования:

- структура DSL выражения должна позволять анализировать ее до выполнения (для трансляции оптимизации)
- у пользователя должна быть возможность использовать набор default значений для визуальных свойств и легко их модифицировать при композиции

Как может выглядеть API для пользователя? (2)

Попробуем изобразить это в виде псевдо кода (смеси Scala и Haskell):

```
val scaledSource = sourceFn <$> log <*> reverse

val stats1 = scaledSource.toStats

val stats2 = scaledSource.toStats
               <&> range(from 10, to end, _._1) <*> mean(_._2)

val coordinates = cartesian <$> flip

val mark1 = points.values .~ stats1.values
          $ points.style.color .~ rgb(255, 0,0, 0.5)
          $ points.style.border .~ 2
          $ points.title .~ "somePoints"
          $ defaults

val mark2 = mark1.points.style.color .~ Blue
...
```

Как может выглядеть API для пользователя? (3)

```
...

val aes = axes.color .~ blue
          $ axes.fontWeight .~ bold
          $ axes labelPos .~ center
          $ defaults
          |@|
          grid.span .~ 10
          $ grid.subdivide .~ true
          $ defaults

val aes1 = aes |@| label.name .~ stats1.columnsNames
val aes2 = aes |@| label.name .~ "mean"

val chart =  in |@| stats <*> coordinates <*> mark1 <*> aes1
                |@| stats1 <*> coordinates <*> mark2 <*> aes2

chart.renderToFile("points.png")
```

Чтобы все это описать нужно? Free/FreeAp, Lenses.. (1)

- FreeAp[F[_], A]
- Applicative, ap: (F[A], F[A => B]) => F[B]

Чтобы все это описать нужно? Free/FreeAp, Lenses.. (2)

В отличии от *Free*, FreeAp структуры полностью определяются статично, поэтому позволяют очень легкую композицию различных DSL выражений. Но из-за этого же свойства не позволяет на прямую определять последовательные вычисления, к счастью для простых случаев это ограничение легко обходится.

Чтобы все это описать нужно? Free/FreeAp, Lenses.. (3)

Для того чтобы легко модифицировать различные значения, определенные в виде case class и ADT, очень хорошо подходят такие примитивы как Lens и Prism (и другие Optics).

Lens - позволяет работать с Product types (*case class*, *Tuple*, *Hlist*, *Map*)

Prism - позволяет работать с Sum types (*Coproducts*), такими как *Option*, *sealed trait*

еще может быть полезна Iso, которая делает двунаправленные преобразования значений без потерь.

Algebra (1)

```
case class RawData(t: Int, v: Double)
type Source = Vector[RawData]
type TFunction = Source => Source

// data loading operations
sealed trait DataF[A]
case class LiftSource(s: Source) extends DataF[Source]

// scales data transformations operations
sealed trait ScalesF[A]
case class ReverseFn() extends ScalesF[TFunction]
case class LogFn() extends ScalesF[TFunction]

...
```

Algebra (2)

```
...

// stats/views data transformations operations
sealed trait StatsF[A]
case class MeanFn() extends StatsF[TFunction]
case class RangeFn(from: Int, to: Int) extends StatsF[TFunction]

// point mark mappings
sealed trait MarksF[A]
case class SetMapping(m: Mark,
                      property: MarkProperties,
                      columnId: ColumnId) extends MarksF[MarkDataRel]
case class SetAes(aes: Aes) extends MarksF[Aes2D]

// coordinates
abstract class CoordinatesF[A]
case class SetCoordinates(cs: Coordinates)
   extends CoordinatesF[Coordinates]
```

Для примера, какие бывают трансформации

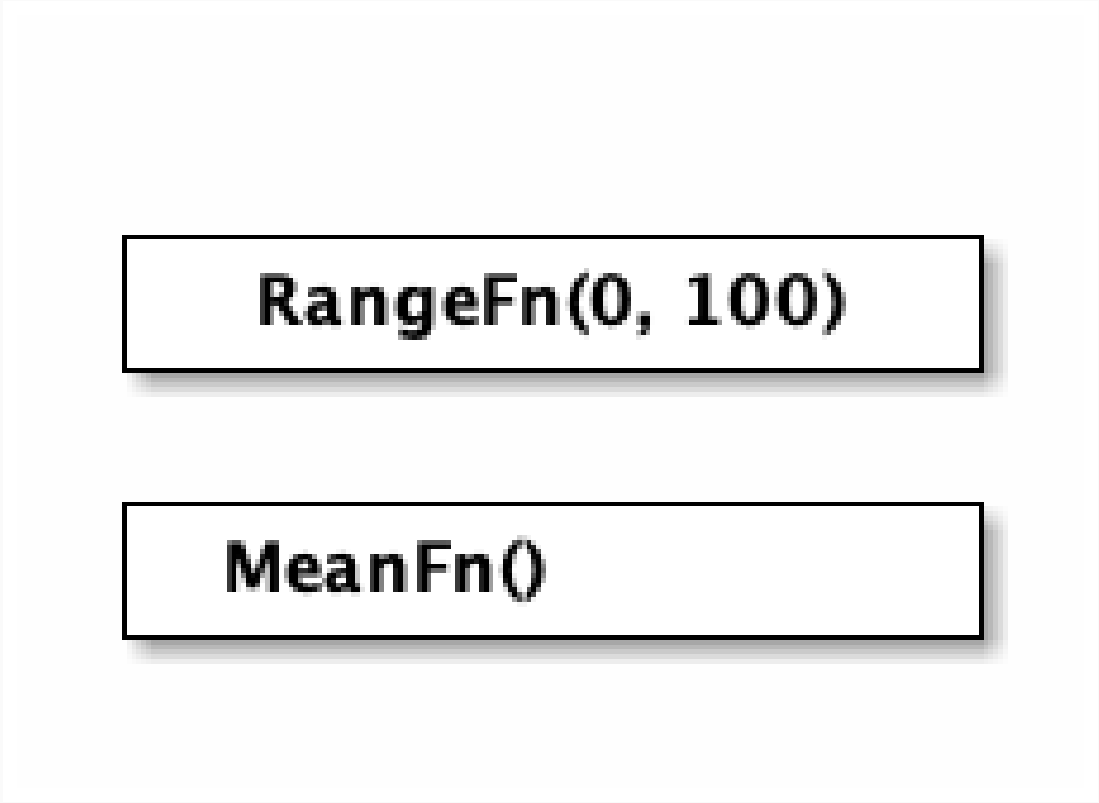
Table 6.4 Scales

Categorical	Interval	Time	One-bend	Two-bend	Probability
cat()	*linear()*	*time()*	*log()* *pow()*	*asn()* *logit()* *probit()* *atanh()*	*prob()*

Table 7.1 Statistical Methods

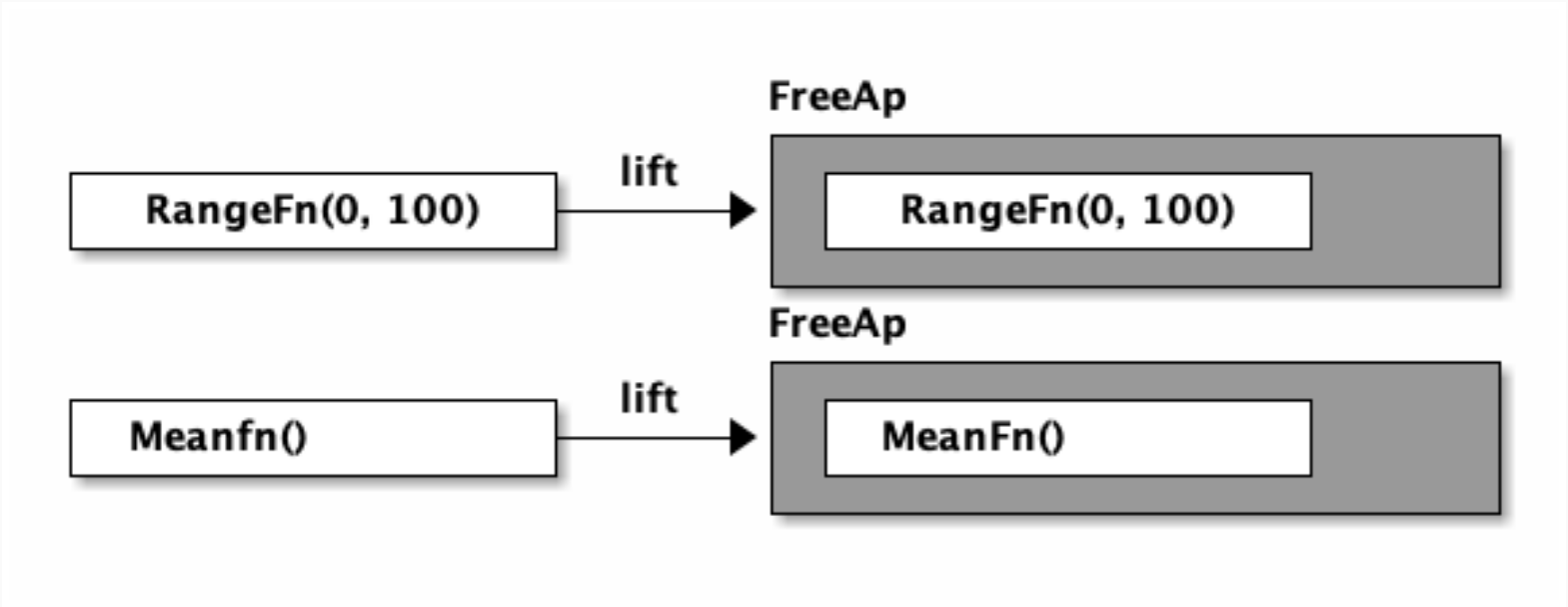
Bin	Summary	Region	Smooth	Link
rect *tri* *hex* *quantile* *boundary* *voronoi* *dot* *stem*	*count* *proportion* *sum* *mean* *median* *mode* *sd* *se* *range* *leaf*	*spread* *sd* *se* *range* *confi* *mean* *sd* *smooth*	*linear* *quadratic* *cubic* *log* *mean* *median* *mode* *spline* *density* *normal* *kernel*	*join* *sequence* *mst* *delaunay* *hull* *tsp* *complete* *neighbor*

Представление выражений DSL в виде данных



Lifting into FreeAp

Теперь нам требуется поместить их в FreeAp контекст



Smart Constructors

```
// lift helper
def lift[Op[_], F[_], A](op: Op[A])(implicit inject: Inject[Op, F]) =
  FreeAp.lift(inject.inj(op))


class StatsLifted[F[_]](implicit I: Inject[StatsF, F]) {
  object stats {

    def meanFn(): FreeAp[F, TFunction]                    =
      lift(MeanFn())

    def rangeFn(from: Int, to: Int): FreeAp[F, TFunction] =
      lift(RangeFn(from, to))

  }
}

object StatsLifted {
  implicit def apply[F[_]](implicit I: Inject[StatsF, F]): StatsLifted[F] =
     new StatsLifted[F]
}
```

Coproduct (1)

Так как у нас есть набор отдельных алгебр, нам требуется объединить их в одну с помощью Coproduct, который является ни чем иным, как вариантом *Either*

```
/** `F` on the left, and `G` on the right, of [[scalaz.\/]].
  *
  * @param run The underlying [[scalaz.\/]]. */
final case class Coproduct[F[_], G[_], A](run: F[A] \/ G[A]) {
  import Coproduct._

  def map[B](f: A => B)(implicit F: Functor[F], G: Functor[G])
   : Coproduct[F, G, B] =
       Coproduct(run.bimap(F.lift(f), G.lift(f)))

...
```

Coproduct (2)

Определим Coproduct наших алгебр.

```
// coproducts
type TransformsInsF[A] = Coproduct[ScalesF, StatsF, A]
type DataInsF[A] = Coproduct[DataF, TransformsInsF, A]
type VisInsF[A] = Coproduct[MarksF, DataInsF, A]
type CoordInsF[A] = Coproduct[CoordinatesF, VisInsF, A]

type ChartF[A] = Coproduct[VisInsF, CoordInsF, A]
```

Coproduct (3)

Пара функций, которые упрощают комбинирование интерпретаторов.

```
def combineInterpreters[F[_], G[_], H[_]](f: F ~> H, g: G ~> H)
  : Coproduct[F, G, ?] ~> H = new (Coproduct[F, G, ?] ~> H) {
      override def apply[A](fa: Coproduct[F, G, A]): H[A] =
        fa.run match {
          case -\/(ff) => f(ff)
          case \/-(gg) => g(gg)
        }
}

implicit class RichNaturalTransformation[F[_], H[_]](val f: F ~> H) {
  def or[G[_]](g: G ~> H): Coproduct[F, G, ?] ~> H =
   combineInterpreters[F, G, H](f, g)
}
```

Опишем Визуальные свойства (1)

```
/// Axis ///
sealed trait Coordinates
case class Cartesian(x: Double,
                     y: Double) extends Coordinates
case class Polar(radial: Double,
                 angle: Float) extends Coordinates

/// Aes ///
 sealed trait PointShape
case class Circle() extends PointShape
case class Star() extends PointShape
case class Cross() extends PointShape
case class Square() extends PointShape
case class Arrow() extends PointShape

sealed trait MarkStyle
case class PointStyle(
  color: Color,
  borderColor: Color,
  bolderWidth: Double,
  radius: Double,
  shape: PointShape
) extends MarkStyle
```

Опишем Визуальные свойства (2)

```
// marks types
sealed trait Mark
case class Point() extends Mark
case class Text() extends Mark

// marks properties
sealed trait MarkProperties
case class X() extends MarkProperties
case class Y() extends MarkProperties
case class Size() extends MarkProperties
case class Opacity() extends MarkProperties
case class Shape() extends MarkProperties

// relation between mark properties and data columns
case class MarkDataRel(m: Mark,
                       property: MarkProperties,
                       columnId: ColumnId)

// aes description
sealed trait Aes
case class Aes2D(axis: AxisStyle,
                 grid: GridStyle,
                 markStyle: MarkStyle) extends Aes
```

Программа/DSL (1)

Прежде всего нужно описать какие используются алгебры и параметризовать их главным Coproduct-ом.

```
def program(implicit A: DataLifted[ChartF],
      B: ScalesLifted[ChartF],
      C: StatsLifted[ChartF],
      D: MarksLifted[ChartF],
      E: CoordinatesLifted[ChartF]
      ) = {

     import A._, B._, C._, D._, E._

...
```

Программа/DSL (2)

Загрузим данные.

```
// the data source
val in = Vector(
  RawData(1,2.1),
  RawData(2,11.5),
  RawData(3,33.1)
)

// lift data
val dataB: FreeAp[ChartF, Vector[RawData]] =
  data.liftSource(in)
```

Программа/DSL (3)

Определим трансформации данных и views.

```
// apply scales
val scalesB = dataB |@| scales.logFn |@| scales.reverseFn

// create views to scaled data by applying series
// of transformations functions
val stats1: FreeAp[ChartF, Vector[RawData]] =
  (scalesB |@| stats.rangeFn(0,2))(seq3)

val stats2: FreeAp[ChartF, Vector[RawData]] =
  (scalesB |@| stats.meanFn)(seq3)
```

функция seqN применяет трансформации последовательно к друг другу

```
 def seq2[F[_], A](v: F[A],
                   fn: F[A] => F[A],
                   fn2: F[A] => F[A]) = fn2(fn(v))
```

Программа/DSL (4)

Определяем визуальные свойства на основе значений по умолчанию и свойства, которые будут назначаться на данные.

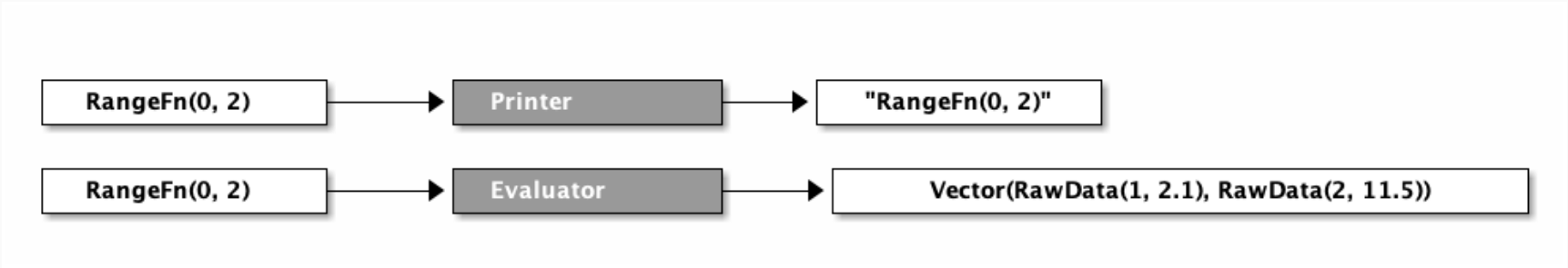
```
// customizing default aesthetic styles
val lStyle = GenLens[LineStyle](_.width)
               .set(2)(defaultAxisStyle.lineStyle)

val pStyle = defaultPointStyle
  .lens(_.color).set(vis.Color.red)
  .lens(_.bolderWidth).modify(_ + 1)
  .lens(_.shape).set(Square())

val pStyle2 = pStyle.lens(_.color).set(vis.Color.blue)

val aStyle = defaultAxisStyle
  .lens(_.lineStyle).set(lStyle)

val gStyle = defaultGridStyle
  .lens(_.lineStyle).set(lStyle)

// defining mark type
val mark = Point()
// define set of mappable properties
val mProps = (X(), Y(), Size())
```

Программа/DSL (5)

Определяем слои и назначаем данные на визуальные свойства.

```
// layer1 definition
val layer1 = (stats1 |@|
  m.setMapping(mark, mProps._1, RawData._t) |@|
  m.setMapping(mark, mProps._2, RawData._v) |@|
  aes.setAes(defaultAes))((x, y, w, a) => (x, y, w, a))

// layer2 definition
val layer2 = (stats2 |@|
  m.setMapping(mark, mProps._1, RawData._t) |@|
  m.setMapping(mark, mProps._2, RawData._v) |@|
  aes.setAes(defaultAes))((x, y, z, w) => (x, y, z, w))

// combing layers to chart
val chart = (layer1 |@| layer2)((x, y) => (x, y))

chart
```

Интерпретатор (1)



Интерпретатор (2)

```
// test printer

type Printer[A] = String

def toDataPrinter[A](): DataF ~> Printer = new (DataF ~> Printer) {
  def apply[A](fa: DataF[A]) = fa match {
    case LiftSource(s) => s"source ${s.mkString(", ")} \n"
  }
}

def toScalesPrinter[A](): ScalesF ~> Printer = new (ScalesF ~> Printer) {
  def apply[A](fa: ScalesF[A]) = fa match {
    case ReverseFn() => "reverse fn\n"
    case LogFn() => "log fn\n"
  }
}

def toStatsPrinter[A](): StatsF ~> Printer = new (StatsF ~> Printer) {
  def apply[A](fa: StatsF[A]) = fa match {
    case MeanFn() => "mean fn 2\n"
    case RangeFn(from, to) => "range fn\n"
  }
}
```

Интерпретатор (3)

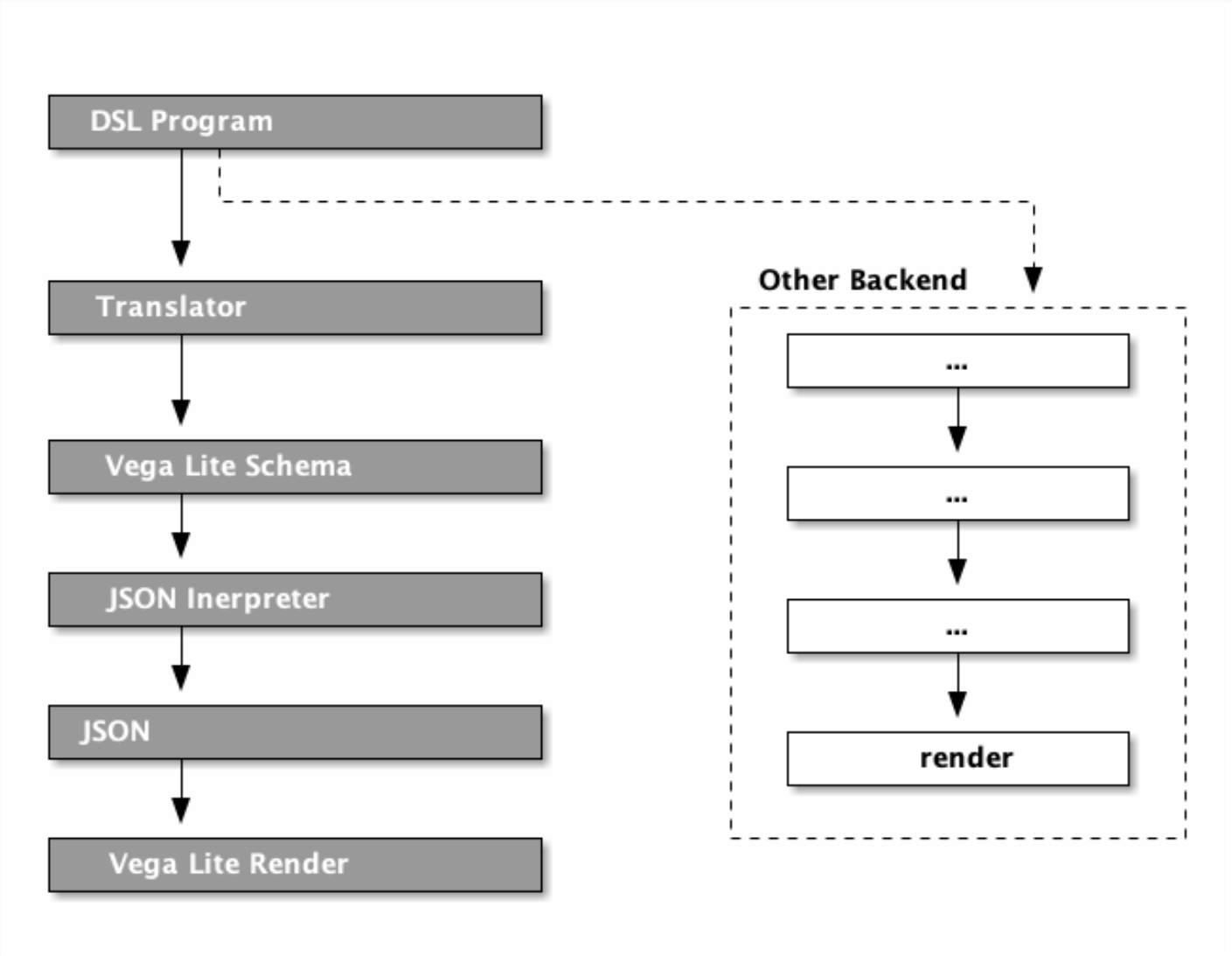
Определяем натуральные трансформации с помощью Coproduct используя те самые 2 функции.

```
def toTransfInsPrinter[A](): TransformsInsF ~> Printer =
  toScalesPrinter or toStatsPrinter

def toDataInsPrinter[A](): DataInsF ~> Printer =
  toDataPrinter or toTransfInsPrinter

def toVisInsPrinter[A](): VisInsF ~> Printer =
  toMarksPrinter or toDataInsPrinter

def toCoordInsPrinter[A](): CoordInsF ~> Printer =
  toCoordsPrinter or toVisInsPrinter

def toChartPrinter[A](): ChartF ~> Printer =
  toVisInsPrinter or toCoordInsPrinter
```

Запускаем программу

```
def run() = {
  println(program.foldMap(toChartPrinter()))
  println(program.foldMap(toChartEvaluator()))
}
```

Что дальше?


DSL Program
Translator
Vega Lite Schema
JSON Inerpreter
JSON
Vega Lite Render
Other Backend
...
...
...
render

Промежуточное представление Schema на Shapeless

```
trait Schema[F[_, _]] {

  def vString: F[String, String]
  def vNumber: F[Double, Double]
  def vInt: F[Int, Int]
  // ...
  def vOpt[A, B](value: GenDsl[A, B]): F[Option[A], Option[B]]
  // ...
}

trait GenDsl[A, B] {
  def apply[F[_, _]](schema: Schema[F]): F[A, B]
}

// example of schema definition
sealed trait FontWeight extends Product with Serializable
object FontWeight {
  case object Bold extends FontWeight
  case object Normal extends FontWeight
  case class Weight(weight: Double) extends FontWeight

  def schema: Dsl[FontWeight] = oneOf(
    "bold".as(Bold) ::
    "normal".as(Normal) ::
    number.imap(Weight)(_.weight) ::
    HNil
  ).to[FontWeight]
}
```

Выводы

Мы немного приблизились к пониманию того как можно в FP стиле реализовать декларативное API, которое позволит описывать широкий спектр визуализаций не жертвуя типизацией и гибкостью композиции.

Многое из того, что было описанно можно сделать проще

Библиотеки Freestyle, Freek, Liberator и д.р. могут сократить определения многих вещей за счет использования макросов и готовых примитивов.

Давайте делать Scalaz-Viz вместе!

Для того, чтобы внести вклад в проект желательно иметь представление об:

- основах FP
- higher-kinded types
- algebraic data types
- type classes
- Monoid, Functor, Applicative, Monad ...
- Free, Tagless Final
- Shapeless

Возможно вам будут интересны и другие проекты Scalaz

- Actors
- Algebra
- CLI
- CT
- Collection
- Config
- Deriving
- Distributed
- Gossip
- HTTP
- IOQueue
- JSON
- MTL
- Metrics
- KleisliIO
- Logging
- Monoidal
- NIO2
- Parsers
- Plugin
- Protocol
- RPC
- Reactive
- STM
- Scalazzi
- ZIO
- AWS
- Crypto
- JDBC
- ML
- Outlaws
- SQL
- SQL-JDBC
- Snippets
- TensorFlow
- YAML

Scalalaz Podcast

Связаться со мной всегда можно через DM twitter:
https://twitter.com/strobegen

Русскоязычный подкаст о Scala.

http://scalalaz.ru