**FUNDAÇÃO GETÚLIO VARGAS**
ESCOLA DE MATEMÁTICA APLICADA

**DAVI SALES BARREIRA**

# Data Visualization from a Category Theory Perspective

Rio de Janeiro
2025

Davi Sales Barreira

# Data Visualization from a Category Theory Perspective

Tese submetida à Escola de Matemática Aplicada como requisito parcial para a obtenção do título de Doutor em Matemática Aplicada e Ciência de Dados.

Área de Concentração: Ciência de Dados

Orientador: Flávio Codeço Coelho
Coorientadora: Asla Medeiros e Sá

Rio de Janeiro
2025

Dados Internacionais de Catalogação na Publicação (CIP)
Ficha catalográfica elaborada pelo Sistema de Bibliotecas/FGV

Barreira, Davi Sales
Data visualization from a category theory perspective / Davi Sales Barreira. – 2024.

119 f.

Tese (doutorado) – Fundação Getulio Vargas, Escola de Matemática Aplicada.
Orientador: Flávio Codeço Coelho.
Coorientadora: Asla Medeiros e Sá.
Inclui bibliografia.

1. Programação (Matemática). 2. Aprendizado do computador. 3. Métodos gráficos. 4. Matemática. I. Coelho, Flávio Codeço,1969-. II. Sá, Asla Medeiros. III Fundação Getulio Vargas. Escola de Matemática Aplicada. IV. Título.

CDD – 511.5

Elaborada por Marcelle Costal de Castro dos Santos – CRB-7-RJ-007517/O

**FUNDAÇÃO GETULIO VARGAS**
**ESCOLA DE MATEMÁTICA APLICADA**
**DOUTORADO EM MATEMÁTICA APLICADA E CIÊNCIA DE DADOS**
**FOLHA DE APROVAÇÃO**

**DAVI SALES BARREIRA**

**"DATA VISUALIZATION FROM A CATEGORY THEORY PERSPECTIVE''.**

**TESE APRESENTADA AO CURSO DE DOUTORADO EM MATEMÁTICA APLICADA E CIÊNCIA DE DADOS QUE CONFERIU O GRAU DE DOUTOR EM MATEMÁTICA APLICADA E CIÊNCIA DE DADOS.**

**DATA DA DEFESA: 13/12/2024**

**ASSINATURA DOS MEMBROS DA BANCA EXAMINADORA**

**PRESIDENTE DA COMISSÃO EXAMINADORA: PROF. FLÁVIO CODEÇO COELHO**

*<ASSINADO ELETRONICAMENTE>*
**PROF. FLÁVIO CODEÇO COELHO**
**ORIENTADOR**

*<ASSINADO ELETRONICAMENTE>*
**PROFª ASLA MEDEIROS E SÁ**
**CO-ORIENTADORA**

*<ASSINADO ELETRONICAMENTE>*
**PROF. ALEXANDRE RADEMAKER**
**MEMBRO INTERNO - FGV EMAP**

*<ASSINADO ELETRONICAMENTE>*
**PROF. MOACYR ALVIM HORTA BARBOSA DA SILVA**
**MEMBRO INTERNO FGV EMAP**

*<ASSINADO ELETRONICAMENTE>*
**PROF. CLAUDIO ESPERANÇA**
**MEMBRO EXTERNO - UFRJ**

*<ASSINADO ELETRONICAMENTE>*
**PROF. JOÃO LUIZ DIHL COMBA**
**MEMBRO EXTERNO - UFRGS**

RIO DE JANEIRO, 13 DE DEZEMBRO DE 2024.

*<ASSINADO ELETRONICAMENTE>*
**PROF. CÉSAR LEOPOLDO CAMACHO MANCO**
**DIRETOR**

*<ASSINADO ELETRONICAMENTE>*
**PROF ANTONIO DE ARAUJO FREITAS JUNIOR**
**PRÓ-REITOR DE ENSINO, PESQUISA E PÓS-GRADUAÇÃO**

ESTE É UM TRABALHO ORIGINAL ONDE FOI VERIFICADA A NÃO EXISTÊNCIA DE PLÁGIO E DE UTILIZAÇÃO DE INTELIGÊNCIA ARTIFICIAL, NÃO EXPLICITADA, NO CORPO DO TRABALHO ATESTADO PELO ALUNO(A) E ORIENTADOR(A). ESTE DOCUMENTO NÃO CONFERE TÍTULO. PARA TAL DEVERÃO SER CUMPRIDOS OS REQUISITOS DO PROGRAMA DE PÓS-GRADUAÇÃO.

## Abstract

In the field of data visualization, there is a persistent challenge in balancing expressiveness and abstraction across different tools and libraries. While some like D3 are highly expressive but lack abstraction, others like Seaborn are highly abstracted but not very expressive.

Visualization grammars have emerged as a solution to this trade-off, attempting to balance abstraction and expressiveness through structured rules and consistent principles. While successful in some respects, these grammars often fall short when handling more complex visualizations such as those involving nested or integrated graphics and custom graphical marks.

To address these limitations, we propose a new theoretical framework that formalizes graphic specification and assembly through a constructive perspective, enhancing expressiveness without compromising abstraction. This approach treats data visualizations as diagrams, integrating diagramming and visualization into a unified framework.

In order to formalize our proposal, we make use of Category Theory (CT). Category Theory excels at modeling compositional structures, which allows us to describe how visual components can be combined and transformed. Moreover, the deep connection between Category Theory and Functional Programming (FP) allows us to translate theoretical concepts into code, through a concept known as Categorical Programming.

To validate the efficacy of our theoretical framework, we introduce a proof-of-concept implementation in the form of a visualization package named Vizagrams. Vizagrams operates as an embedded domain-specific language (DSL), implementing a visualization grammar over a diagramming DSL. We demonstrate its expressiveness through a gallery of visualizations, and evaluate its abstraction by comparing its graphic specifications against other grammars.

# Contents

# Introduction

In data visualization, *abstraction* and *expressiveness* are two fundamental concepts that must be balanced by every visualization tool. Expressiveness refers to the ability of a framework to produce a variety of types of visualizations, while abstraction refers to the process of simplifying descriptions by removing details, thereby making the specification more manageable. These concepts often conflict with each other. Some visualization tools prioritize high expressiveness at the expense of abstraction (e.g. D3 [10], Vega [57]), while others prioritize abstraction over expressiveness (e.g. Matplotlib [26], Seaborn [73], Bokeh [6], Makie.jl [18]). Visualization grammars, on the other hand, are domain-specific languages capable of generating a significant variety of visualizations while maintaining simplicity in their specifications, thus effectively balancing abstraction and expressiveness [41].

Bertin's Semiology of Graphics [4] and Wilkinson's Grammar of Graphics (GoG)[80] stand out as the most influential theoretical contributions to visualization grammars. Bertin [4] pioneered the systematic analysis of the process of graphic construction for data visualization, which served as a foundational influence on the Grammar of Graphics. Wilkinson [80], in turn, provided a theoretical formalization for visualization grammars which has significantly influenced the design of various grammars [78, 75, 57, 58, 38, 49, 81, 35], becoming the predominant theoretical framework in the field. Despite their success, visualization grammars based on GoG have limitations in terms of expressiveness [20]. While proficient for the description of common statistical graphics like scatter plots, bar plots, and histograms, they often lack the descriptive power for more complex graphics, such as composite visualizations and those involving custom marks (glyphs). The recurring limitations in expressiveness across various grammars suggest that the issue may lie within the theoretical foundation itself.

## Motivation

This thesis began by trying to reproduce Figure 1 using the Julia programming language. This figure is originally from Rougier [55], who demonstrates how to reproduce it using Matplotlib[1]. However, the solution given requires advanced techniques, such as the use of the `axisartist` toolkit for creating floating axes. These features demand a deep understanding

[1]The code for reproducing the figure can be found in this link.

of Matplotlib's coordinate systems and transformation capabilities, making the approach complex and challenging for the average user to implement independently.

Setting aside some of the aesthetic details, one might envision recreating the plot in Figure 1 by first generating the scatter plot and histogram independently, and then overlaying the histogram on top of the scatter plot with appropriate adjustments to size, position, and rotation. Yet, despite this seemingly straightforward description, recreating such figure using Julia's available visualization packages proved to be challenging.

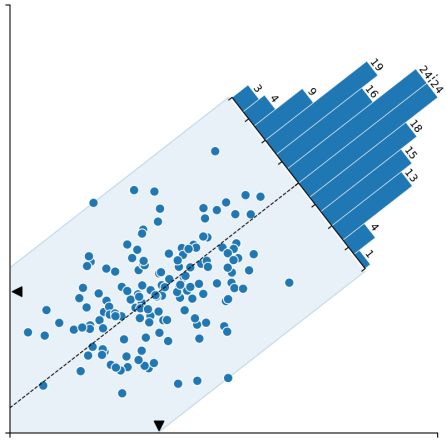



Figure 1: Rotated histogram aligned with second main PCA axis. Figure from Rougier [55].

The difficulty in producing such composite graphic highlighted a gap in existing data visualization tools. Moreover, this lack of expressiveness was not limited to composite visualizations alone, but extended to other valuable visualizations, such as those involving custom graphical marks. This motivated us to develop our own data visualization package.

As we began designing our tool, it became evident that the absence of a formal framework for describing and implementing data visualizations was a significant obstacle. A theoretical foundation was needed to guide both the conceptualization and construction of graphics. Without such foundation, it was unclear how to design a system to be both expressive enough to produce complex plots, and abstract enough to be accessible to a wide range of users.

The need to strike a balance between abstraction and expressiveness led us to the conclusion that our tool should be designed as a visualization grammar. However, the theoretical formalism provided by the Grammar of Graphics [80] seemed insufficient in terms of expressiveness, prompting us to explore alternative theoretical foundations. We required a rigorously defined theory to allow for an a priori analysis of whether our framework could achieve the desired level of expressiveness and abstraction. Additionally, since the theory would serve as the foundation for our package, it was essential that it offer a clear pathway to computational implementation. These considerations ultimately led us to Category Theory, which, with its mathematical rigour, capacity for abstraction and connections to Functional Programming, appeared to be a well-suited foundation for our work.

# Hypothesis and research objectives

The main objective of this thesis is to develop a data visualization framework that extends the expressiveness of visualization grammars while preserving a high level of abstraction. To accomplish this, we propose the following hypotheses, addressing both the root cause of the problem and a potential approach to resolve it:

> **Hypothesis 1 (Cause):** The limitations in expressiveness found in existing visualization grammars stem from a lack of integration between the processes of graphic specification and graphic assembly. This lack of integration restricts users' ability to define new types of visualizations, as existing grammars typically focus on specification while concealing the underlying assembly process. Without incorporating assembly directly within the visualization framework, users are limited in creating complex visualizations — such as those involving composite graphics and custom marks — that require control over both specification and assembly.

> **Hypothesis 2 (Solution):** Achieving this integration calls for a constructive perspective, where the data visualization process begins with graphic assembly and is then connected to graphic specification. This approach redefines assembly as diagramming, as it involves the bottom-up construction of scenes through the incremental combination of graphical primitives. Thus, the key to extending expressiveness lies in integrating diagramming with data visualization specification.

Based on the main goal and the hypotheses, the following objectives are set:

1. Develop a theoretical framework that integrates diagramming and data visualization, satisfying the following criteria:
   (i) Enable users to utilize diagramming operations to specify plots constructively.
   (ii) Enable users to combine and manipulate plots using diagramming operations.
   (iii) Support the creation of custom graphical marks within the framework.
2. Implement this theoretical framework into an open-source package named Vizagrams, written in the Julia programming language, satisfying the following criteria:
   (i) Develop an embedded domain-specific language specifically for diagramming.
   (ii) Implement the visualization grammar on top of the diagramming DSL.
   (iii) Design the graphic specification to closely resemble Vega-Lite, facilitating a comparison of their level of abstraction.
   (iv) Create a gallery of visualizations to assess the framework's expressiveness and flexibility in practice.

# Structure of This Document

Chapter 1 reviews existing theoretical approaches to data visualization. We begin by describing the data visualization process, followed by a discussion on the abstraction and expressiveness trade-off, and finally, we examine visualization grammars as tools to navigate this dichotomy.

Chapter 2 provides an overview of Category Theory, focusing on concepts relevant to our work. This chapter also introduces Applied Category Theory, presenting some applications that influenced our approach to data visualization.

Chapter 3 explores Categorical Programming as a paradigm for bridging Category Theory and Julia programming. This chapter provides an overview of the Julia programming language, detailing how it aligns with both Functional Programming and Category Theory.

Chapter 4 introduces our theoretical framework, which formalizes data visualization through Category Theory, integrating assembly and specification within a single constructive approach. We formalize core concepts such as diagrams, graphical marks, and graphic expressions. These are used to construct plot specifications, achieving a unified approach to diagramming and data visualization.

Chapter 5 introduces Vizagrams, the data visualization tool developed for this thesis. We demonstrate the basic usage of its diagramming and data visualization capabilities. We then provide an evaluation of both its abstraction level and expressiveness. Lastly, we compare it with some related tools, covering diagramming libraries, visualization grammars and visualization authoring systems.

Chapter 6 concludes the thesis by summarizing both theoretical and practical contributions. It also addresses the limitations of our work and outlines potential directions for future research.

# Chapter 1

# Data Visualization Theory

Despite its ubiquity, the field of data visualization still lacks a robust theoretical foundation [14]. The possible reasons for this are varied. First, even with the subject's long history [22], the computational emphasis of visualization is actually quite recent, starting on roughly 1987 [39]. It is, in part, this computational emphasis that forces a more rigorous approach and a better understanding of theoretical aspects. Secondly, the subject involves several fields, such as mathematics, computer graphics, design, cognitive science and semiotics. Thus, any "grand theory" would have to encompass several different perspectives, which is unlikely to be dominated by a single department, even more a single person. Thirdly, the human component in data visualization makes it harder to define what is "right" and what is "wrong".

What do we mean by a theoretical foundation? According to Chen et al. [14]

> ...Theory foundation denotes the set of concepts, theories, and models on which the practice of visualization is based, as well as a system of ideas intended to explain phenomena or observations related to visualization. A concept is an abstract idea or a general notion. A theory provides a description of concepts and their relationships, in order to help us understand a phenomenon or observation. A model is a system or prototype used as an example to follow or imitate; unlike a theory, a model usually involves some meaningful arrangement or sequence of concepts. A theory component is any aspect of a theory foundation above, be it a concept, a theory, or a model...
>
> Grand theories have the broadest scope and present general concepts and propositions or principles... Middle-range theories are narrower in scope than grand theories—they are simple, straightforward, general, and consider a limited number of variables and limited aspect of reality, they present concepts at a lower level of abstraction, and guide theory-based research and visualization practice strategies. The functions of middle-range theories are to describe, explain, or predict phenomena and observations... Practice theories have the most limited scope and level of abstraction and are developed for use within a specific range of

visualization situations. Visualization practice theories may provide guidelines for visualization design and implementation and predict outcomes and the impact of visualization practice.

Following this description, our interest lies mostly in **middle-range** theories that describe the "algorithmic" process of turning data into a computer graphic. We do not to delve into aspects related to visualization design quality or human perception.

## 1.1 What is Data Visualization?

If we are to delve into the theory of data visualization, an essential step is to define our terms. Hence, we start with the field itself.

Camm et al. [13] defines **data visualization** as the graphical representation of data using displays such as charts, graphs and maps. With a slightly different take, Munzner [45] defines data visualization as the computer-based process of providing visual representations of datasets with the goal of helping people carry out tasks more effectively. For our work, the purpose of a visualization is not of first concern, therefore, we adopt a definition more akin to Camm et al. [13] and we incorporate the computational aspect from Munzner [45].

**Definition 1.1.1 (Graphic).** A graphic is a visual presentation on some surface, such as a piece of paper, or a computer screen [56]. A **computer** graphic is any computational representation of a visual presentation, be it static, animated or interactive.

**Definition 1.1.2 (Data Visualization).** Data visualization is the representation of a set of data as a computer graphic.

Note that this definition is actually very similar to how one defines the field of computer graphics. According to de Miranda Gomes and Velho [19], the main task of computer graphics is that of transforming data into images; hence the field of computer graphics consists in the collection of methods and techniques for transforming data into images displayed through a graphics device. This raises the question of how does computer graphics differ from data visualization.

As pointed by Munzner [45], computer graphics takes a scene description given by a geometry dataset, and transforms this scene description in an actual image. Hence, the field is mainly concerned with the process of "decoding" a geometrical description into an image. On the other hand, the field of data visualization is concerned with "encoding" an arbitrary dataset into a geometrical description.

In the context of data visualization, a graphic representation refers to the description of how our data is mapped to a geometrical description. In the context of computer graphics, a graphic representation is more akin to the geometric description itself.

From this discussion, we can see that data visualization and computer graphics meet in the geometrical description of a graphic. It is the job of data visualization to properly transform the data by applying scaling, coordinates and labeling, in order to turn a generic dataset into a meaningful geometrical specification. It is then the job of computer graphics to figure out how to represent geometrical objects computationally and then draw them in the screen according to the specification given.

Wilkinson [80] makes a distinction between statistical graphics, and scientific graphics. Statistical graphics are a specific type of graphic designed to represent data accurately and appropriately with the aid of statistical techniques. Examples are bar plots, scatter plots and so on. Scientific visualizations (or scientific graphics) are those intended to accurately represent some natural phenomena, usually derived from physical simulations. Note that the main difference is that scientific visualizations are closer to the actual geometric descriptions, hence, tend to be more closely associated with computer graphics.

Telea [67] makes a similar distinction when talking about data visualization. Although, he categorizes them as scientific visualization and information visualization, and considers both to be subfields of data visualization.

## 1.2 Abstraction and Expressiveness

Achieving an effective balance between abstraction and expressiveness is essential for any data visualization tool. Expressiveness refers to the range of visualization types a framework can support, while abstraction involves reducing complexity by omitting extraneous details, making visualization specifications clearer and easier to manage. These two principles can often be at odds, as enhancing one can sometimes limit the other.

### 1.2.1 Assessing Visualization Tools

The relation of expressiveness and degrees of abstraction is discussed by Mei et al. [41], where the authors do an extensive survey of information visualization construction tools, i.e. general purpose data visualization tools such as ggplot2 [75], D3 [10], Tableau [66], and many others. In the article, the authors argue that three questions are central when choosing a visualization tool:

- **Can I build it?** Depends on the expressiveness of the tool.
- **Do I know how?** Depends on the user and the accessibility of the tool.
- **How long will it take?** Relates to the cost of designing visualizations, the effectiveness of the rendering process and the responsiveness to user interactions.

These three questions can be summarized in three concepts, namely expressiveness, accessibility and efficiency. Instead of assessing these attributes directly, the survey devises further categories related to each of these attributes and classifies the tools according to them:

- **Degree of Abstraction**: Graphic library, Declarative, Chart Typology.
- **How to Use: Programming**, Configuration, Direct Manipulation or Mixed-initiative.
- **Presentation Medium**: Images, Embedded, or Web Page.
- **Supported Data Source**: Tabular, Hierarchical, Streaming.
- **Customization Allowed**: Dataflow, Interaction, Infographics Design.
- **Interaction**: Fixed, Configurable, Fully Customized.
- **Action Type**: Reverse-engineering, Overlay, Normal.
- **Design Process**: Demonstrational, Exploratory, Confirmatory.

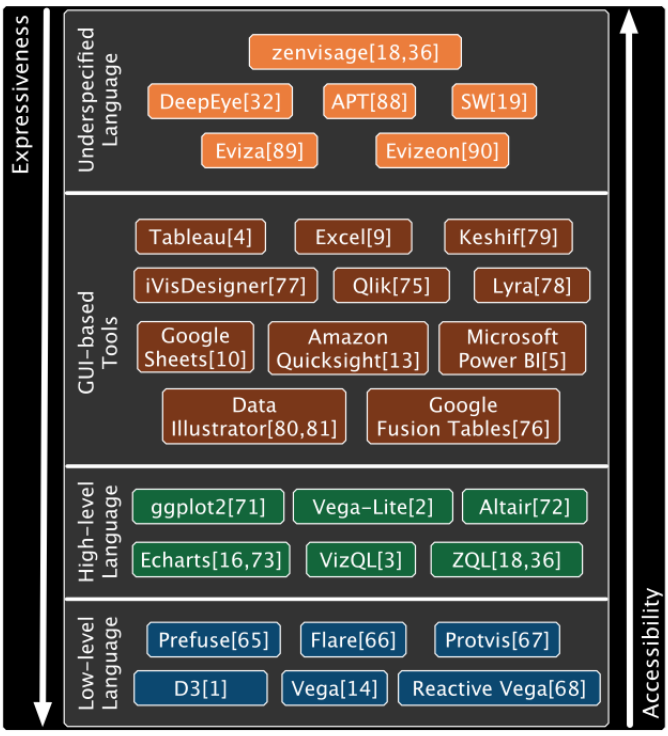



Figure 1.1: Classification of data visualization tools according to expressiveness and accessibility. Image taken from Qin et al. [51].

Mei et al. [41] argues that more abstract tools (chart typology) have less expressiveness, while those based on high-level languages (e.g. Vega-Lite) have a medium degree of abstraction

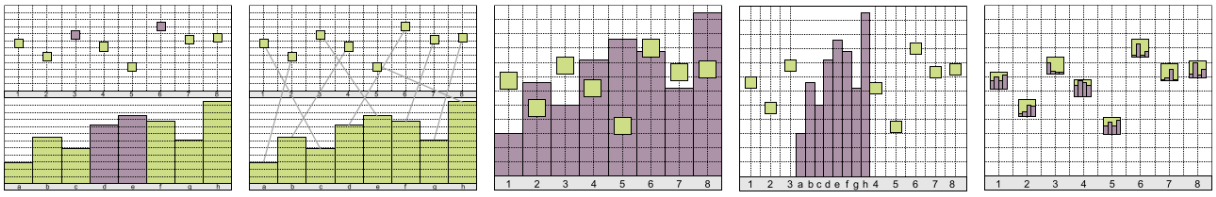
with more expressiveness, and graphic libraries (e.g D3) are the least abstract and have the highest expressiveness. While Mei et al. [41] discusses abstraction as a means of balancing expressiveness and accessibility; in our work, abstraction takes precedence over accessibility.

Qin et al. [51] also did a survey of data visualization tools and explored expressiveness versus accessibility. In this case, degree of abstraction was not discussed. The authors arrived in a similar assessment to Mei et al. [41]. Although, Mei et al. [41] did not directly rank the tools in terms of expressiveness and accessibility, while Qin et al. [51] did. Figure 1.1 shows the classification according to Qin et al. [51].

### 1.2.2 Composite & Glyph-Based Visualizations

Evaluating the expressiveness of a data visualization tool is a challenging task, in part due to the vast and continuously expanding design space of visualizations. Besides the more common statistical graphics (e.g. scatter plots, line plots, bar plots, pie charts, ...), two particularly valuable types of visualizations stand out, namely composite and glyph-based visualizations.

The term *composite visualization* was introduced by Javed and Elmqvist [27] to describe visualizations that consolidate multiple visualizations into a single view using various composition operations such as juxtaposition, integration, overloading, superimposition, and nesting (see Figure 1.2). Deng et al. [20] revisited the categorization defined by Javed and Elmqvist [27], and proposed a new categorization with only juxtaposition, overlay and nesting as the main compositions.




(a) Juxtaposed views. (b) Integrated views. (c) Superimposed views. (d) Overloaded views. (e) Nested views.

Figure 1.2: Example of composite visualizations taken from Javed and Elmqvist [27].

Composite visualizations are widely prevalent in the literature, as demonstrated by the research conducted by Lee and Howe [33], which revealed that 35% of the figures in a corpus of PubMed papers were composite visualizations. Within composite visualizations, the study by Deng et al. [20] found that justaxposition were the most common type, accounting for 51.9% of cases analyzed, while overlay and nested accounted for 23.9% and 24.2%, respectively.

Glyphs, or custom marks, are graphical marks that go beyond basic geometric primitives. In the literature, visualizations that use such marks are called glyph-based visualizations [8]. Glyph-based visualizations often emerge in cases where storytelling is paramount, such as journalistic publications. A classic example is the Chernoff faces plot [17] in which glyphs

representing faces are used (Figure 1.3). Another example is the OECD Better Life Index plot [64], where the custom mark has less of a symbolic meaning, and it is instead used mainly as an abstract way to encode multivariate data into a single view (Figure 1.4).

A visualization framework with the capacity of producing glyph-based visualizations is particularly challenging as it requires the integration of "glyph creation" with data visualization. Such topic has been tackled mainly via visualization authoring systems having graphical user interfaces (GUI) [29, 36, 82, 72, 52, 74, 68].

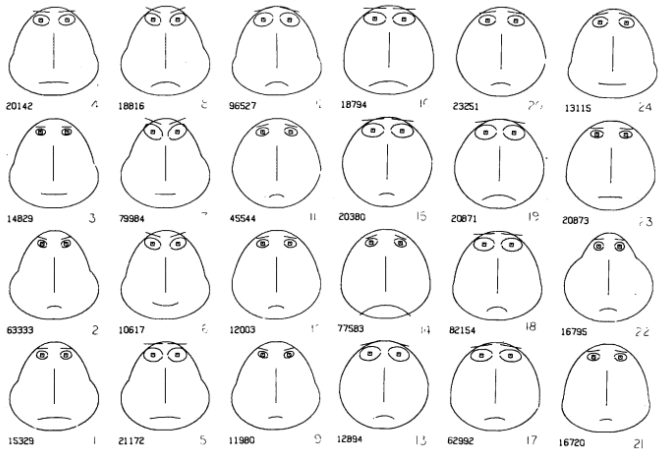



Figure 1.3: Example of Chernoff faces from Chernoff [17].

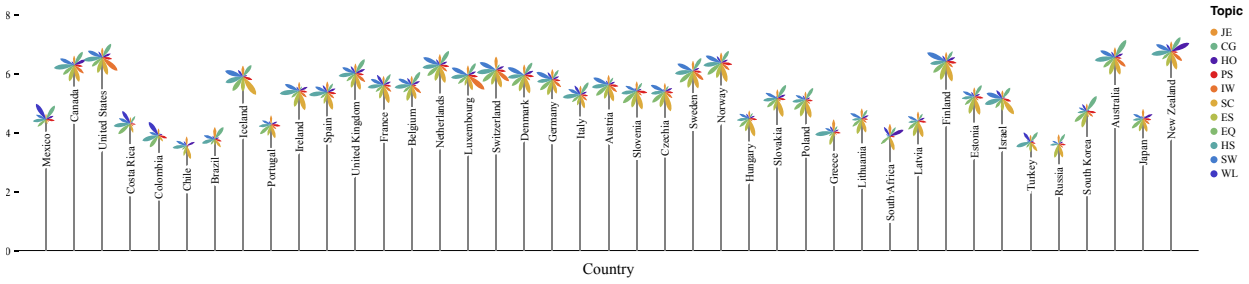



Figure 1.4: Reproduction of the OECD Better Life Index plot [64].

## 1.3 Visualization Grammars

Visualization grammars are specialized domain-specific languages designed to generate data visualizations using a consistent set of rules. One of their key strengths is balancing the trade-off between abstraction and expressiveness inherent in data visualization tools.

The landscape of visualization grammars is extensive, encompassing a wide range of implementations categorized as either low-level or high-level. Low-level implementations prioritize

expressiveness at the expense of increased specification complexity. In contrast, high-level implementations prioritize user-friendly specifications, though reducing expressiveness.

Another important distinction is between specialized and general-purpose grammars. General-purpose high-level visualization grammars often face limitations in expressiveness, such as reduced support for complex composite visualizations [20] and glyph-based visualizations [37]. In contrast, specialized grammars focus on a narrower scope to maximize expressiveness within specific areas.

Examples of low-level general purpose visualization grammar are D3 [10], Protovis [9] and Vega [57], while Vega-Lite [58], ggplot2 [75], Observable Plot [1], Polaris [65], and Gadfly.jl [28] are examples of high-level general purpose grammars. In terms of specialized grammars, there is Gosling [38] for genomics data visualization, ATOM [47] for unit visualizations, GoTree [35] for visualizations with tree layouts, PGoG [49] for visualizations depicting probabilities, Nebula [15] for coordinating visualizations in multiple coordinated views. Beyond the visualization grammars cited here, many others exist, as surveyed and analyzed by McNutt [40], who provides an extensive review of 57 JSON-style domain-specific languages for visualization.

Despite the considerable diversity, two theoretical works stand out as the basis of visualization grammars in general. These are Bertin's Semiology of Graphics [4] and Wilkinson's Grammar of Graphics [80]. Bertin [4] pioneered the systematic analysis of the process of graphic construction for data visualization, and Wilkinson [80] provided a formalization for the specification of statistical graphics. Next, we give a summary of both of these works.

### 1.3.1 Semiology of Graphics

Bertin [4] is one of the first and main references in the field of data visualization that theorizes the concept of a graphic. His usage of the term "graphic" is actually more in line with what Wilkinson [80] calls statistical graphic.

In semiology, a sign-system is a set of signs or symbols that are used to communicate meaning. A system is monosemic if each sign has a known meaning prior to observation, and each sign-system has a perception medium. A graphic representation is a monosemic sign-system where the system of perception is visual. In other words, a graphic representation is a "language for the eye". Figure 1.5 compares different sign-systems.

From this perspective, a graphic encodes "information" in signs which have a preconceived meaning. The graphic representation is the system (language) on which this encoding takes place. The graphic is instantiated in a visual medium (e.g. paper, computer, board), where the viewer interprets the information.

What Bertin refers to as "information" can be interpreted without much confusion as "data". Note that, in order to encode information into a graphic representation, one has to assess the *size* of the information, and the variables available in the sign-system. For example, if

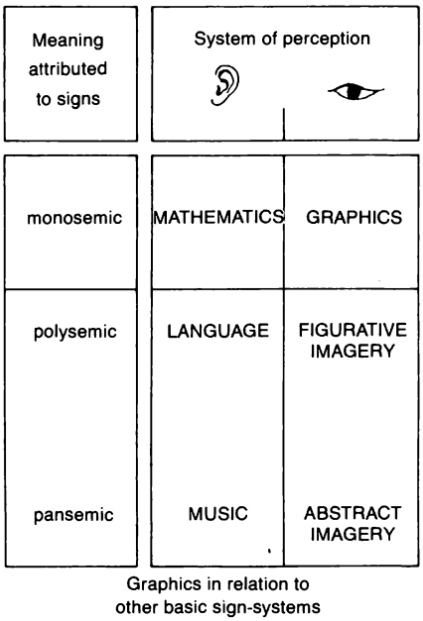



Figure 1.5: Comparison of different sign-system. Image from Bertin [4].

there are 5 people, but my language has only 4 words, then I cannot properly convey the information that there are 5 different individuals.

To deal with this issue, information is classified into three categories: nominal, ordered (ordinal) and quantitative. Each unit ("word") in the graphic sign-system is said to have visual variables. These variables are: position, size, value, texture, color, orientation and shape. This graphical unit is also called "mark", and a mark can have different types, such as a point, line, area, and so on. Bertin separates the positional variables from the rest, which he calls retinal variables. Choosing the appropriate visual variable is crucial to accurately represent different types of data.

To summarize, according to Bertin, data visualization involves representing information through graphical marks by encoding the data into the visual variables of those marks.

### 1.3.2 The Grammar of Graphics

Formulated by Wilkinson [80], the Grammar of Graphics is a framework for constructing statistical graphics via a set of predefined rules. Wilkinson [80] defines a graphic as a physical representation of a graph, where a graph is a set of abstract points. An abstract point together with aesthetic properties defines a notion similar to what Bertin [4] calls a mark. The instantiation of a graphic is done by first *mathematically* constructing a graph and then adding aesthetic properties.

Unlike Bertin [4], the Grammar of Graphics was formulated already with the goal of being implemented computationally, more specifically with the Object-Oriented Programming (OOP) paradigm in mind. In this perspective, graphics are collections of objects that communicate via the grammatical rules.

The GoG defines the data visualization pipeline as a tripartite process consisting of: (1) graphic specification, which outlines the entire structure of the graphic, including data, transformations, scales, and aesthetics; (2) assembly, which integrates these components to form a graphical representation; and (3) display, which renders the final visual output on a screen or other medium.

The GoG focuses primarily on the graphic specification step. A graphic is specified via six components:

1. **DATA**: variables from the dataset that will be encoded in the graphic;
2. **TRANS**: variable transformations (e.g. ordering the data);
3. **SCALE**: scale transformation (e.g. log);
4. **COORD**: coordinate system (e.g. linear, polar);
5. **ELEMENT**: graphs (e.g. circle) and the aesthetic attributes (e.g. color);
6. **GUIDE**: graphical elements that help decoding the visual representation back into the data values (e.g. axes, legends).

The whole process of graphic generation using GoG is shown in Figure 1.6. Wilkinson [80] considers that the ordering of the stages in this pipeline cannot be altered. Yet, different methods can be applied in each stage generating a plethora of different visualizations.

Let us explain the steps in the pipeline above with an example. Consider that we have a dataset of cars as in Figure 1.7. Our goal is to create two pie charts, one for car models that are new and another for car models that are old. The pie must show the proportion according to the origin region.

The first step is to select our variables, i.e. columns `Origin` and `Release`. Note that each variable is like a function where we pass the row id as argument, and it returns a value. For example, `Origin(1) = "Japan"`.

The second step is applying an algebra. In GoG, there are three operators that define the graphics algebra. They are *cross*, *blend* and *nest*. These operators establish how the variables are related. Using the *cross* operator (`*`) for both our variables, we obtain a new variable where the row id input now returns a tuple, e.g. `(Origin * Release)(3) = ("Europe","Old")`. Wilkinson [80] calls this a *varset*, since it is a set of variables.

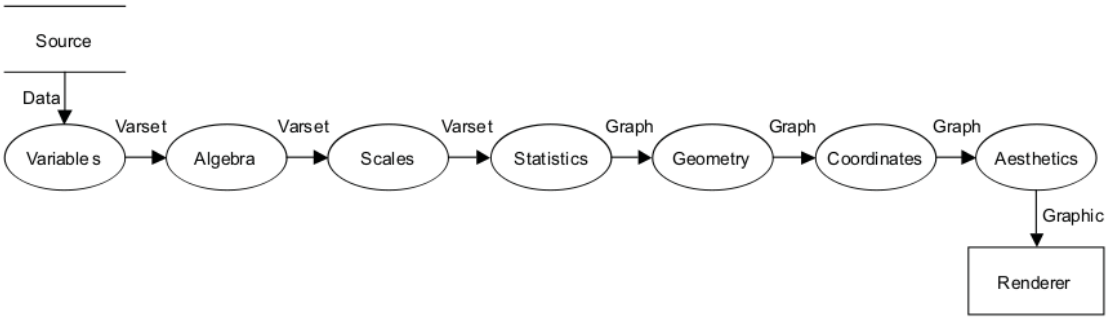



Figure 1.6: Pipeline for turning data into graphics. Image from Wilkinson [80].

Variables `Origin` and `Release` are nominal, thus, the scale function takes a value in the category and returns a number. This number is just a way to index each category. Here is an example.

```
> scale_cat(Origin, ["Europe","Japan", "USA"])
[1,2,3]

> scale_cat(Release, ["New","Old"])
[1,2]
```

For the *pie chart*, we need the proportion of each combined category. For this, we can use a `summary_proportion` function. This function takes a tuple of scaled values, and returns their proportion. The code below shows how this function is applied.

```
> scale_cat(Origin*Release, ("USA","New"))
(1,1)
> summary_proportion(Origin*Release, (1,1))
0.31
```

The next step is to encode this into a geometric graph. In GoG, a geometric graph is an abstract concept to define geometric objects before actually applying a coordinate. Examples of geometric graph functions in GoG are `point`, `line`, `interval`, `area`, `polygon`.

For the *pie chart*, our choice of graphing function is `interval.stack`, since the proportions can be thought as intervals stacked on top of each other. By applying a polar coordinate, an interval then becomes a slice. We then have a collection of slices, one for each tuple of `(Origin,Release)`.

The collection of slices is our *mathematical* graph. This ends the portion of generating an abstract graph. The following step is applying aesthetic functions in order to produce a

| Row | Name | Origin | Release | Miles_per_Gallon | Cylinders |
|---|---|---|---|---|---|
| | String | String | String | Float64 | Int64 |
| 1 | toyota corona mark ii | Japan | New | 24.0 | 4 |
| 2 | datsun pl510 | Japan | Old | 27.0 | 4 |
| 3 | volkswagen 1131 deluxe sedan | Europe | Old | 26.0 | 4 |
| 4 | peugeot 504 | Europe | New | 25.0 | 4 |
| 5 | audi 100 ls | Europe | Old | 24.0 | 4 |
| 6 | saab 99e | Europe | Old | 25.0 | 4 |
| 7 | bmw 2002 | Europe | Old | 26.0 | 4 |
| 8 | datsun pl510 | Japan | New | 27.0 | 4 |
| 9 | chevrolet vega 2300 | USA | New | 28.0 | 4 |
| 10 | toyota corona | Japan | Old | 25.0 | 4 |
| 11 | chevrolet vega (sw) | USA | New | 22.0 | 4 |
| 12 | mercury capri 2000 | USA | Old | 23.0 | 4 |
| 13 | opel 1900 | Europe | New | 28.0 | 4 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Figure 1.7: Example of dataset of cars.

graphic. Examples of aesthetic functions are `position`, `color` and `label`. By calling the position function on the proportion values, and the color and label on the `Origin` variable, we finish the construction of the pie chart. The final chart is shown in Figure 1.8.

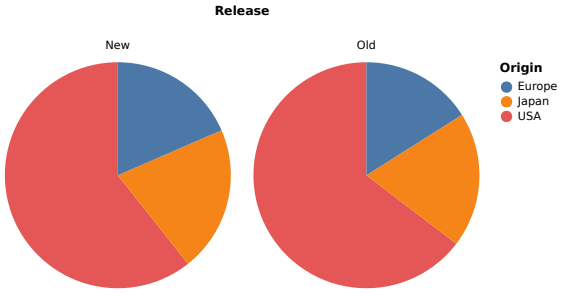



Figure 1.8: Example of pie chart.

The Grammar of Graphics has been an immense contribution to the field of data visualization. It is perhaps the largest treatise on how to formally define statistical graphics. Besides the theoretical contribution, Wilkinson [80] has inspired the creation of some state-of-the-art visualization packages such as ggplot2 and Vega-Lite.

# Chapter 2

# Category Theory and Applications

Category Theory is a branch of mathematics that studies general abstract structures through their relationships. The field originated in the 1940s with the work of mathematicians Samuel Eilenberg and Saunders Mac Lane. Their goal was to bridge the fields of topology and algebra. To do so, it was necessary to develop a theory that could describe the common underlying structures and properties shared by these two fields. Thus, Category Theory was "born".

Informally, Category Theory can be thought of as the "mathematics of analogy". It is able to find relations in seemingly unrelated areas, serving as a bridge between many fields. It has found applications not only within several branches of mathematics (e.g. geometry, probability, algebra, topology, graph theory), but also in fields such as physics, linguistics, computer science and philosophy.

Almost paradoxically, the very abstract nature of Category Theory grants it the ability to "navigate" distinct subjects, even very mundane subjects such as planning and scheduling [12]. In the world of programming, Category Theory is strongly related to the Functional Programming, with many programming patterns taking direct inspiration in categorical concepts (e.g. functors, monoids, monads) [43].

As pointed by Fong and Spivak [21], Category Theory is unmatched in its ability to organize and relate abstractions. We believe that it may serve as a robust framework for bridging mathematics, computer science and data visualization.

In this chapter, our first goal is to introduce the basics of Category Theory, focusing on constructions that will be essential when formalizing data visualization. Our second goal is to show some examples of how CT can and has been applied in subjects other than pure mathematics.

## 2.1 A Brief Introduction to Category Theory

We start with the definition of a category, followed by functors and natural transformations (the "trinity" of Category Theory). We then introduce more specific concepts such as monoids, monads and $F$-algebras. During the exposition, we present some examples to illustrate the concepts.

### 2.1.1 What is a Category?

Let us start with a formal definition of a category.

**Definition 2.1.1 (Category).** A category $\mathcal{C} = \langle \mathrm{Ob}_{\mathcal{C}}, \mathrm{Mor}_{\mathcal{C}} \rangle$ consists of a class of objects $\mathrm{Ob}_{\mathcal{C}}$ and a class of morphisms $\mathrm{Mor}_{\mathcal{C}}$ satisfying the following conditions:

(i) Every morphism $f \in \mathrm{Mor}_{\mathcal{C}}$ is associated to two objects $X, Y \in \mathrm{Ob}_{\mathcal{C}}$ which is represented by $f : X \to Y$ or $X \xrightarrow{f} Y$, where $dom(f) = X$ is called the domain of $f$ and $cod(f) = Y$ is the codomain. Moreover, we define $\mathrm{Mor}_{\mathcal{C}}(X, Y)$ as

$$\mathrm{Mor}_{\mathcal{C}}(X, Y) := \{f \in \mathrm{Mor}_{\mathcal{C}} \ : \ X \in dom(f), \ Y \in cod(f)\};$$

(ii) For any three objects $X, Y, Z \in \mathrm{Ob}_{\mathcal{C}}$, there exists a composition operator

$$\circ : \mathrm{Mor}_{\mathcal{C}}(X, Y) \times \mathrm{Mor}_{\mathcal{C}}(Y, Z) \to \mathrm{Mor}_{\mathcal{C}}(X, Z),$$

(iii) For each object $X \in \mathrm{Ob}_{\mathcal{C}}$ there exists a morfism $id_X \in \mathrm{Mor}_{\mathcal{C}}(A, A)$ called the identity.

The composition operator must have the following properties:

(p.1) *Associative*: for every $f \in \mathrm{Mor}_{\mathcal{C}}(A, B), g \in \mathrm{Mor}_{\mathcal{C}}(B, C), h \in \mathrm{Mor}_{\mathcal{C}}(C, D)$ then

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

(p.2) For any $f \in \mathrm{Mor}_{\mathcal{C}}(X, Y)$, $g \in \mathrm{Mor}_{\mathcal{C}}(Y, X)$,

$$f \circ id_X = f, \quad id_Y \circ g = g.$$

In the literature, many distinct notations are used for the class of morphisms $\mathrm{Mor}_{\mathcal{C}}(X, Y)$, such as $\mathcal{C}(X, Y)$ or $\mathrm{Hom}_{\mathcal{C}}(X, Y)$. The reason for this is that this set is sometimes called hom-set. We will use either $\mathrm{Mor}_{\mathcal{C}}(X, Y)$ or $\mathcal{C}(X, Y)$ when there is no ambiguity.

Another point about conventions. When talking about composition, some authors use the operator $\fatsemi$, which is equivalent to the composition $\circ$, but with the inverted order, i.e. $f \fatsemi g = g \circ f$.

Note that when talking about $\mathrm{Ob}_{\mathcal{C}}$ and $\mathrm{Mor}_{\mathcal{C}}$, we didn't say that they were sets, instead we called them *classes*. The reason for this lies in the foundations of Set Theory. There are concepts in mathematics that are "larger" than sets, e.g. the "set" of all sets, which itself cannot be a set, otherwise it would incur in a paradox (Russell's Paradox). A way to deal with this is making a distinction between classes and sets. This point is quite technical; readers interested in understanding this nuance can check books such as Borceux [7].

Now, due to this distinction between classes and sets, we have the following definitions.

**Definition 2.1.2 (Small and Locally Small Category).** A category $\mathcal{C}$ is *small* if $\mathrm{Ob}_{\mathcal{C}}$ and $\mathrm{Mor}_{\mathcal{C}}$ are sets. A category $\mathcal{C}$ is *locally small* if for any $A, B \in \mathrm{Ob}_{\mathcal{C}}$, then $\mathrm{Mor}_{\mathcal{C}}(A, B)$ is a set. Note that a small category is also locally small.

In this thesis, all the categories we deal with are locally small.

### 2.1.2 Examples of Categories

Let us showcase some examples of categories. To facilitate our exposition, we draw some diagrams to represent the categories. In these diagrams, the arrows symbolize morphisms connecting the objects.

**Example 2.1.1 (Categories 1, 2, 3).** Category $\mathbf{1}$ consists of $\mathrm{Ob}_{\mathbf{1}} := \{A\}$ and $\mathrm{Mor}_{\mathbf{1}} = \{id_A\}$. The diagram for such category is shown below.


Figure 2.1: Category $\mathbf{1}$.

Category $\mathbf{2}$ consists of $\mathrm{Ob}_{\mathbf{2}} := \{A, B\}$ and $\mathrm{Mor}_{\mathbf{1}} = \{id_A, id_B, f\}$, where $f : A \to B$. The diagram for such category is shown below.

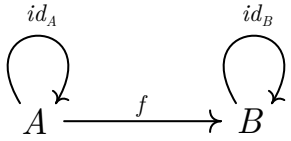



Figure 2.2: Category $\mathbf{2}$.

Category **3** has three morphisms besides the identities. The morphisms are $f$, $g$ and their composition $g \circ f$. The figure below illustrates the category with all its morphisms.

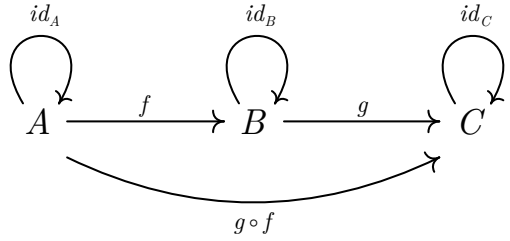



Figure 2.3: Category **3** showing all morphisms.

Drawing all the morphisms can make the diagram become too crowded, specially as the number of objects and morphisms grows. Hence, we simplify the diagram representation by omitting identity morphisms and morphisms given by compositions. These can always be assumed to exist, since they are a necessary condition for every category. Thus, the figure below represents the same diagram as Figure 2.3.

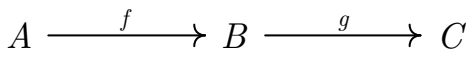



Figure 2.4: Category **3** omitting identities and compositions.

These were very trivial examples. Here are some more interesting categories:

1. **Set** which is the category of sets, where the objects are sets and the morphisms are functions between sets.

2. **Top** is the category where topological spaces are the objects and continuous functions are the morphisms.

3. $\mathbf{Vec}_{\mathbb{F}}$ is the category where vector spaces over field $\mathbb{F}$ are the objects, and linear transformations are the morphisms.

4. **Gr** is the category of directed graphs, where $\mathrm{Ob}_{\mathbf{Gr}} := \{\mathrm{Vertex}, \mathrm{Arrow}\}$, and the morphisms are
$$\mathrm{Mor}_{\mathbf{Gr}} := \{src, tgt, id_{\mathrm{Vertex}}, id_{\mathrm{Arrow}}\}$$
where $src : \mathrm{Arrow} \to \mathrm{Vertex}$ returns the source vertex for each arrow and $tgt : \mathrm{Arrow} \to \mathrm{Vertex}$ returns the target vertex.

It is possible to use existing categories to derive new ones. An example of this is the so called slice category:

**Definition 2.1.3 (Slice Category).** Let $\mathcal{C}$ be a category and $S \in \mathrm{Ob}_{\mathcal{C}}$. A slice category $\mathcal{C}/S$ is such that its objects are tuples $(A, f)$ where $A \in \mathrm{Ob}_{\mathcal{C}}$ and $f : A \to S$.

In $\mathcal{C}/S$, a morphism $\phi_{A,B} : (A, f) \to (B, g)$ is a $\phi \in \mathrm{Mor}_{\mathcal{C}}(A, B)$, such that $f = g \circ \phi$. In other words, a morphism $\phi_{A,B}$ is equivalent to a morphism $\phi$ in the category $\mathcal{C}$, such that going from $A$ to $S$ through $f$ is the same as going to $B$ through $\phi$ and then from $B$ to $S$ through $g$. This is represented in Figure 2.5.

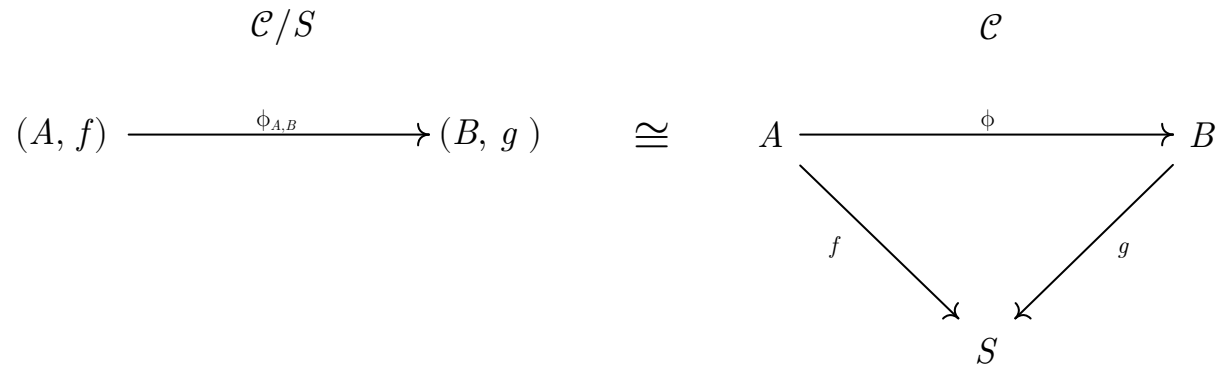



Figure 2.5: On the left we have a morphism in the slice category $\mathcal{C}/S$. On the right we have the equivalent diagram for such representation in the $\mathcal{C}$ category.

Another useful way to construct new categories is via subcategories:

**Definition 2.1.4 (Subcategory).** Let $\mathcal{C}$ be a category. A *subcategory* $\mathcal{S}$ of $\mathcal{C}$ is such that

(i) $\mathrm{Ob}_{\mathcal{S}} \subseteq \mathrm{Ob}_{\mathcal{C}}$;

(ii) For every $A, B \in \mathrm{Ob}_{\mathcal{S}}$, we have $\mathrm{Mor}_{\mathcal{S}}(A, B) \subseteq \mathrm{Ob}_{\mathcal{C}}(A, B)$;

(iii) Composition and identity in $\mathcal{S}$ are the same as in $\mathcal{C}$, restricted to morphisms and objects of $\mathcal{S}$.

A subcategory $\mathcal{S}$ is said to be *wide* if $\mathrm{Ob}_{\mathcal{S}} = \mathrm{Ob}_{\mathcal{C}}$, and it is said to be *full* if for every $A, B \in \mathrm{Ob}_{\mathcal{S}}$, then $\mathrm{Mor}_{\mathcal{S}}(A, B) = \mathrm{Ob}_{\mathcal{C}}(A, B)$. Finally, a subcategory is *thin* if for every $A, B \in \mathrm{Ob}_{\mathcal{S}}$ the set $\mathrm{Mor}_{\mathcal{S}}(A, B)$ has only a single morphism.

### 2.1.3 Special Objects

One of the tenets of Category Theory is that we are not allowed to "peek inside objects". This means that objects are never described by their *intrinsic properties*, but only by how

they relate to other objects in the category. For example, consider the category **Set**. In this category, sets are objects and functions are morphisms. How can we figure out which of these objects is the empty set if we are not allowed to "peek inside" in order to see that "oh, this object has nothing inside, so it must be the empty set"?

We know that the empty set is somewhat special in Set Theory, hence, there must be some way to identify it in the category **Set**. And indeed there is. Since functions are morphisms, we know that for every set $A$, there is only one function $f : \varnothing \to A$. This fact is only true for the empty set. Thus, we can say that the empty set is the object in **Set** where there is a single morphism going from it to every other object in the category. This actually has a special name, we say that the empty set is the initial object of **Set**.

Analogously we can define a terminal object as one that has a unique morphism arriving on it from every other object. Let us give a formal definition.

**Definition 2.1.5 (Zero, Initial and Terminal).** Let $\mathcal{C}$ be a category.

1. An object $I \in \mathrm{Ob}_{\mathcal{C}}$ is *initial* if for every $A \in \mathrm{Ob}_{\mathcal{C}}$, there is exactly one morphism from $I$ to $A$. Thus, from $I$ to $I$ there is only the identity.
2. An object $T \in \mathrm{Ob}_{\mathcal{C}}$ is *terminal* if for every $A \in \mathrm{Ob}_{\mathcal{C}}$, there is exactly one morphism from $A$ to $T$. Thus, from $T$ to $T$ there is only the identity.
3. An object is *zero* if it is both terminal and initial.

Note that in the definitions above, we are defining these objects in terms of existence and uniqueness of morphisms, which is known in Category Theory as **universal constructions**. This suggests that other concepts might also have an analogous "categorical" interpretation, which is in fact true. For example, the notion of isomorphism has a categorical counterpart.

**Definition 2.1.6 (Categorical Isomorphism).** Let $\mathcal{C}$ be a category with $X, Y \in \mathrm{Ob}_{\mathcal{C}}$ and $f \in \mathrm{Mor}_{\mathcal{C}}(X, Y)$.

(i) We say that $f$ is *left invertible* if there exists $f_l \in \mathrm{Mor}_{\mathcal{C}}(Y, X)$ such that $f_l \circ f = id_X$;

(ii) We say that $f$ is *right invertible* if there exists $f_r \in \mathrm{Mor}_{\mathcal{C}}(Y, X)$ such that $f \circ f_r = id_Y$;

(iii) We say that $f$ is invertible if it is both left and right invertible.

When an invertible morphism exists between $X$ and $Y$, we say that they are isomorphic.

**Theorem 2.1.7.** Every *initial* object is unique up to an isomorphism, i.e. if in a category there are two *initial* objects, then they are isomorphic. Similarly, *terminal* objects are unique up to an isomorphism. Moreover, the isomorphism is unique between initial objects, and between terminal objects.

**Example 2.1.2 (Terminal and Initial Objects in Set).** As we have already said, the initial object in **Set** is $\varnothing$. The terminal object is actually all the singletons (sets with only one element). Note that for any set $\{a\}$, there will be only one function $g : A \to \{a\}$, which is $g(x) = a$. Also, every singleton set is isomorphic to each other. Hence, although we have several objects that are terminal, they all represent essentially the same set.

This example highlights the idea that in Category Theory the actual components of an object do not actually matter, only how they relate. This is exactly the case of singleton sets.

Note that a category might not have an initial or terminal objects. Moreover, having both an initial and terminal object does not imply that it has a zero object. In fact, this is the case with **Set**.

### 2.1.4 Functors

Once we have defined what a category is, the next step is to define functors. While morphisms go from objects to objects, functors go from categories to categories.

**Definition 2.1.8 (Functor).** Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A functor $F : \mathcal{C} \to \mathcal{D}$ is a pair of mappings with the following properties:

(i) a mapping between objects

$$F : \mathrm{Ob}_{\mathcal{C}} \to \mathrm{Ob}_{\mathcal{D}},$$

where for each $A \in \mathrm{Ob}_{\mathcal{C}}$, $F(A) \in \mathrm{Ob}_{\mathcal{D}}$.

(ii) a mapping between morphisms

$$F : \mathrm{Mor}_{\mathcal{C}} \to \mathrm{Mor}_{\mathcal{D}},$$

where there are two possibilities:

(a) **Covariant Functor**, in which

$$F : \mathrm{Mor}_{\mathcal{C}}(A, B) \to \mathrm{Mor}_{\mathcal{D}}(F(A), F(B)),$$

hence for a morphism $f : A \to B$, then $F(f) : F(A) \to F(B)$.

(b) **Contravariant Functor**, in which

$$F : \mathrm{Mor}_{\mathcal{C}}(A, B) \to \mathrm{Mor}_{\mathcal{D}}(F(B), F(A)),$$

hence for a morphism $f : A \to B$, then $F(f) : F(B) \to F(A)$.

(iii) Identities morphisms are preserved, i.e. for $A \in \mathrm{Ob}_{\mathcal{C}}$

$$F(id_A) = id_{F(A)}.$$

(iv) Compositions are preserved, i.e. for $f \in \mathrm{Mor}_{\mathcal{C}}(A, B)$ and $g \in \mathrm{Mor}_{\mathcal{C}}(B, C)$,

(a) For a **Covariant Functor**,

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

(b) For a **Contravariant Functor**,

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

It is common for authors to refer to covariant functors only as functors. We follow this practice, i.e. whenever we say that $F$ is a functor, it implies that it is a covariant functor. Also, we will sometimes use $FA$ to mean $F(A)$ and $Ff$ to mean $F(f)$.

Again, the use of diagrams may help understand what is going on. The figure below illustrates the identity and composition preservation of functors.

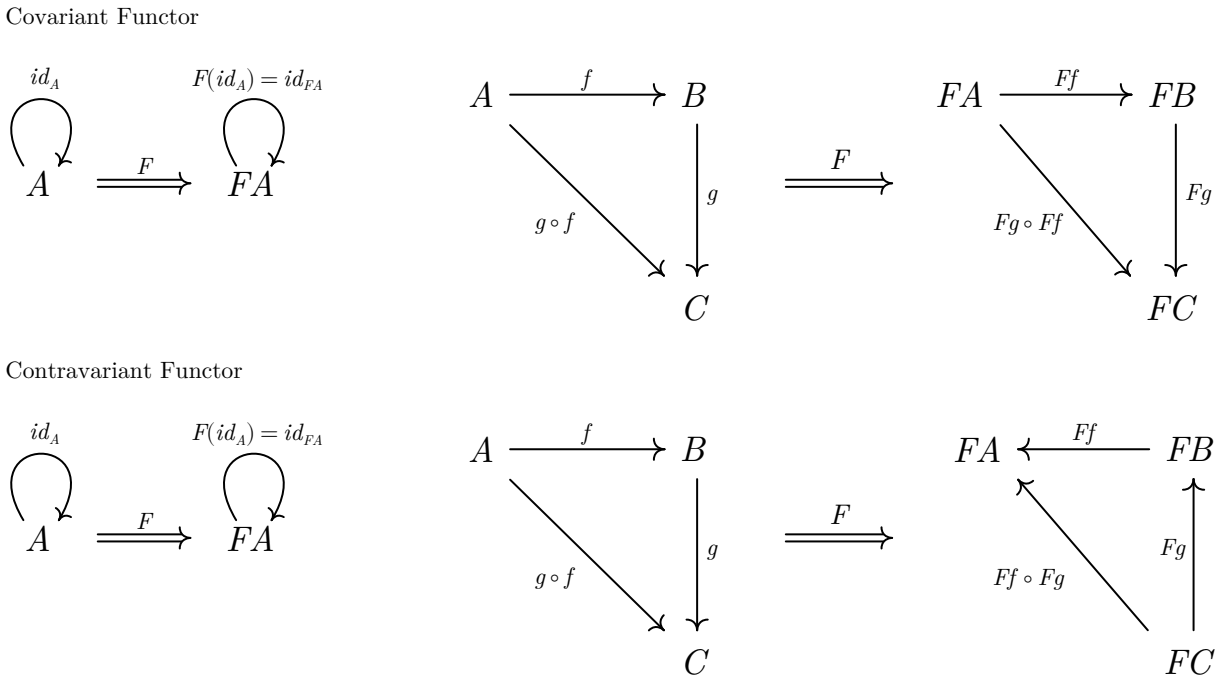



Figure 2.6: Diagrams showcasing the properties of functors.

Bradley [11] points out that a functor $F : \mathcal{C} \to \mathcal{D}$ can be understood as an *interpretation* of $\mathcal{C}$ within $\mathcal{D}$, i.e. $\mathcal{C}$ defines a *syntax*, while $\mathcal{D}$ provides the *semantics*. This observation is useful when trying to apply Category Theory to programming.

**Lemma 2.1.9 (Contravariant Functor).** The definition 2.1.8 of contravariant functor is equivalent to saying that $F$ is a (covariant) functor from $\mathcal{C}^{\mathrm{op}}$ to $\mathcal{D}$.

*Proof.* To prove this, we need to show that for any contravariant functor $F : \mathcal{C} \to \mathcal{D}$ there is an equivalent functor $F' : \mathcal{C}^{op} \to \mathcal{D}$, and vice-versa.

For every $A \in Ob_{\mathcal{C}}$, make $F(A) = F'(A)$. This is fine, since $A \in Ob_{\mathcal{C}} \iff A \in Ob_{\mathcal{C}^{op}}$. Next, for $f \in Mor_{\mathcal{C}}(A, B)$, make $Ff = F'f^{op}$, where $f^{op}$ is the reversed morphism $f$. Again, this is well defined since $f \in Mor_{\mathcal{C}}(A, B) \iff f^{op} \in Mor_{\mathcal{C}^{op}}(B, A)$.

Lastly, note that

$$(g \circ f)^{op} = f^{op} \circ g^{op},$$

hence:

$$F(g \circ f) = F'(f^{op} \circ g^{op}) \iff Ff \circ Fg = F'f^{op} \circ F'g^{op}.$$

Thus, we showed that $F$ is a contravariant functor according to 2.1.8 if and only if $F'$ is a covariant functor from $\mathcal{C}^{\text{op}}$ to $\mathcal{D}$, where $F$ and $F'$ are the same up to an isomorphism from $\mathcal{C}$ to $\mathcal{C}^{op}$. □

Now that we know what a functor is, let us show some examples.

**Example 2.1.3 (Power set functors).** An example of (covariant) functor is the functor $P : \mathbf{Set} \to \mathbf{Set}$, which sends a set $A$ to its power set $2^A$, and sends functions (the morphisms in the case of the category of sets) to their image set, i.e. for $f : A \to B$, we have $Ff : 2^A \to 2^B$ such that for $X \in 2^A$ then $Ff(X) = \{f(x) : x \in X\}$.

An example of **contravariant** functor is the inverse image functor $F : \mathbf{Set} \to \mathbf{Set}$, which sends a set $A$ to its power set $2^A$, but sends $f$ to the inverse image, i.e. $Ff(Y) = \{x \in A : f(x) \in Y\}$. Note that the inverse image satisfy the contravariant property

$$F(f \circ g) = (f \circ g)^{-1} = g^{-1} \circ f^{-1}.$$

**Example 2.1.4 (Group Homomorphism).** In abstract algebra, a group is a triple $(G, \cdot, e)$, where $G$ is a set, $\cdot : G \times G \to G$ is the product mapping which is associative and has an inverse, and $e \in G$ is the identity element. We can *categorify* groups, i.e. we can interpret them as categories. Define a category $\mathbf{G}$ as containing a single object $G$, the elements of $G$ are the morphisms, i.e. for $g \in G$ we have $g : G \to G$. Morphism composition is given by $\cdot$, hence $g_1 \cdot g_2 \equiv g_1 \circ g_2$.

Let $(G, \cdot_G, e_G)$ and $(H, \cdot_H, e_H)$ be two groups. A function $F : G \to H$ is a homomorphism between $G$ and $H$ if $F(g_1 \cdot_G g_2) = F(g_1) \cdot_H F(g_2)$ for every $g_1, g_2 \in G$. Note that this is exactly the definition of a functor $F : \mathbf{G} \to \mathbf{H}$.

**Example 2.1.5 (Identity Functor).** This does what one might expect from the name. The identity functor is $1_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$, such that $1_{\mathcal{C}}(A) = A \in \text{Ob}_{\mathcal{C}}$ and $1_{\mathcal{C}}(f) = f \in \text{Mor}_{\mathcal{C}}$.

Another important aspect of functors is that we can compose them.

**Definition 2.1.10 (Functor composition).** For two functors $F : \mathcal{C} \Rightarrow \mathcal{D}$ and $G : \mathcal{D} \Rightarrow \mathcal{E}$, then $G \circ F$ is a functor from $\mathcal{C}$ to $\mathcal{E}$ where

(i) For any $A \in Ob_{\mathcal{C}}$, $G \circ F(A) = G(F(A))$,

(ii) For any $f \in Mor_{\mathcal{C}}$, $G \circ F(f) = G(F(f))$.

Since we have an identity functor and functors composition, we might wonder whether there exists a category of all categories, where objects are categories and morphisms are functors. The answer is **no**. Similar to the set of all sets, it can be proven that this category does not exist. Yet, the category of all *small categories* does.

**Example 2.1.6 (SmCat).** In category **SmCat**, objects are small categories and morphisms are functors. For each category $\mathcal{C}$, there is an identity functor $1_{\mathcal{C}}$, and composition is given according to Definition 2.1.10.

### 2.1.5 Natural Transformations

Similar to how morphisms define relations between objects in a category, natural transformations define relations between functors. The term "natural" in natural transformations was coined by Eilenberg and MacLane (the founders of Category Theory) due to the fact that these transformations were developed with the aim of explaining why some constructions in mathematics were "natural" while others were not.

**Definition 2.1.11 (Natural Transformations).** Let $\mathcal{C}$ and $\mathcal{D}$ be categories, and let $F, G : \mathcal{C} \to \mathcal{D}$ be functors. A natural transformation $\alpha : F \to G$ is such that:

(i) For all $A \in \mathcal{C}$, there exists $\alpha_A : F(A) \to G(A)$ that is a morphism in $\mathcal{D}$, i.e $\alpha_A \in Mor_{\mathcal{D}}(F(A), G(A))$;

(ii) (Naturality) For all $f \in Mor_{\mathcal{C}}(A, B)$, then

$$F(f) \fatsemi \alpha_B = \alpha_A \fatsemi G(f).$$

Remember that for a morphism $f : A \to B$, we have that $F(f) : F(A) \to F(B)$ and $G(f) : G(A) \to G(B)$. Since $\alpha_A : F(A) \to G(A)$ and $\alpha_B : F(B) \to G(B)$, then $F(f)$ composes with $\alpha_B$ and $G(f)$ composes with $\alpha_A$, and our definition above works.

Another way to represent property (ii) in the definition of natural transformations is to affirm that the diagram below commutes, i.e. that $\alpha_B \circ Ff = Gf \circ \alpha_A$.

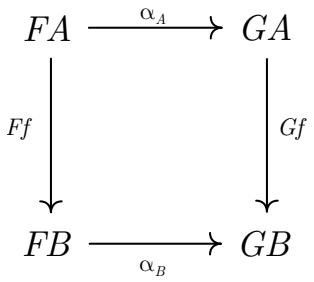



Figure 2.7: Commutative diagram of a natural transformation highlighting the commutative property of the definition.

One can define two types of composition for natural transformations. The vertical composition and the horizontal composition.

**Definition 2.1.12 (Vertical Composition).** Let $\alpha : F \to G$ and $\beta : G \to H$ be natural transformations where $F$, $G$ and $H$ are functors from $\mathcal{C}$ to $\mathcal{D}$. We can define a vertical composition between them by making $(\beta \circ \alpha)_A = \beta_A \circ \alpha_A$ for every object $A \in \mathrm{Ob}_{\mathcal{C}}$. One can check that this composition is associative. The vertical composition is shown in Figure 2.8.

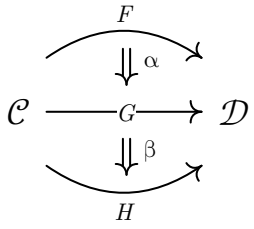



Figure 2.8: Vertical composition of natural transformations.

**Definition 2.1.13 (Horizontal Composition).** Let $\alpha : F \to G$ and $\beta : H \to K$ be natural transformations where $F, G : \mathcal{C} \to \mathcal{D}$ and $H, K : \mathcal{D} \to \mathcal{F}$. We can define a horizontal composition between them by making

$$(\beta * \alpha)_A = \beta_{GA} \circ H(\alpha_A) = K(\alpha_A) \circ \beta_{FA}.$$

Hence, this defines a natural transformation

$$\beta * \alpha : H \circ F \to K \circ G$$

The horizontal composition is illustrated in Figure 2.9.

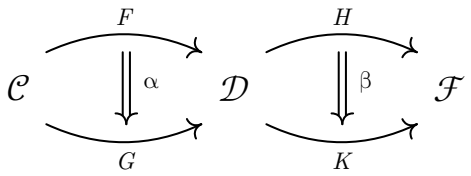



Figure 2.9: Horizontal composition of natural transformations.

Moreover, we can define an identity natural transformation $\mathrm{id}_F : F \to F$ where for any object $A \in \mathrm{Ob}_{\mathcal{C}}$ we have $(\mathrm{id}_F)_A = id_{FA}$. Note that $\mathrm{id}_F$ is a natural transformation and $id_{FA}$ is the identity morphism of an object $FA \in \mathrm{Ob}_{\mathcal{D}}$.

**Definition 2.1.14 (Category of Functors).** For a small category $\mathcal{C}$ and a category $\mathcal{D}$, denote $\mathcal{D}^{\mathcal{C}}$ as the category of functors from $\mathcal{C}$ to $\mathcal{D}$ where objects are functors, morphisms are natural transformations, i.e.

$$\mathrm{Ob}_{\mathcal{D}^{\mathcal{C}}} := \text{Functors from } \mathcal{C} \text{ to } \mathcal{D}$$

$$\mathrm{Mor}_{\mathcal{D}^{\mathcal{C}}}(F, G) := \text{Natural Transformations from } F \text{ to } G.$$

The identity morphism for a functor $F$ is the identity natural transformation $\mathrm{id}_F$, and the composition is the vertical composition of natural transformations.

An endofunctor is a functor $F : \mathcal{C} \to \mathcal{C}$, i.e. a functor that has the same category as domain and codomain. We call $\mathbf{End}_{\mathcal{C}}$ the category of endofunctors in $\mathcal{C}$, i.e. $\mathcal{C}^{\mathcal{C}}$.

### 2.1.6 Monoids and Monads

Monoids and monads are two ubiquitous constructions both in Category Theory and Functional Programming. These two concepts will be used when talking about data visualization.

Let us start with the definition of a monoid in the context of Set Theory.

**Definition 2.1.15 (Monoid - Set Theory).** A monoid is a triple $(M, \otimes, e_M)$ where $M$ is a set, $\otimes : M \times M \to M$ is a binary operation and $e_M$ the neutral element, such that:

1. $a \otimes (b \otimes c) = (a \otimes b) \otimes c$
2. $a \otimes e_M = e_M \otimes a = a$.

An example of a monoid is $(\mathbb{N} \cup \{0\}, +, 0)$. It is easy to check that the summation operator satisfies the associativity and neutrality properties.

Similar to how we did to groups in Example 2.1.4, we can *categorify* this definition of a monoid. Let $(M, \otimes, e_M)$ be a monoid. We define a category $\mathbf{M}$ where there is a single object $M$, and where the morphisms are the elements of $m$. The identity morphism is the neutral element $e_M \in M$, and the composition operator is simply $\otimes$. Figure 2.10 illustrates the category generated by the monoid $(\mathbb{N} \cup \{0\}, +, 0)$.

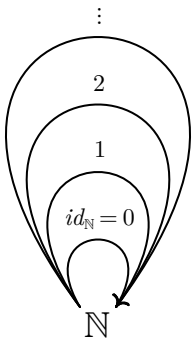



Figure 2.10: Monoid example. Category of natural numbers.

Our definition of monoid, even the categorified version, depended on explicitly referencing the elements of a set $M$. There is a way to define monoids without referring to elements of a set, which we take from Riehl [54].

**Definition 2.1.16 (Monoid in the category Set).** A monoid in **Set** is a triple $(M, \mu, \eta)$, where $M \in \text{Ob}_{\mathbf{Set}}$, $\mu : M \times M \to M$ and $\eta : 1 \to M$ are two morphisms in **Set** satisfying the commutative diagrams in Figure 2.11. Note that 1 is the terminal object in **Set**, i.e. the singleton set (which is unique up to an isomorphism).

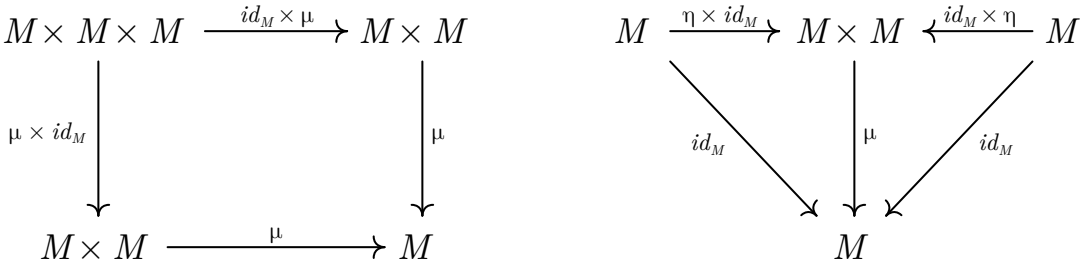



Figure 2.11: Commutative diagram for monoid.

The $\mu$ is acting like the binary product, while $\eta : 1 \to M$ is a way to refer to an element of $M$ without explicitly specifying it. The $f \times g$ morphisms are just the product induced

morphisms, i.e. for $f : A \to B$ and $g : C \to D$, then $f \times g : A \times C \to B \times D$, where $(f \times g)(a, b) = (f(a), g(b))$.

Definition 2.1.15 can be used to generalize the idea of a monoid to other categories. For example:

**Definition 2.1.17 (Monoid in Top).** A monoid in **Top** is a triple $(M, \mu, \eta)$, where $M \in \mathrm{Ob}_{\mathbf{Top}}$, $\mu : M \times M \to M$ and $\eta : 1 \to M$ are two morphisms in **Top** satisfying the commutative diagrams 2.11. Note that 1 is the terminal object in **Top** which is the singleton set with the discrete topology.

Finally, we can define what is commonly know in computer science as a monad. A monad on a category $\mathcal{C}$ is a monoid in the category of endofunctors on $\mathcal{C}$. More specifically:

**Definition 2.1.18 (Monad).** A monad is a monoid $(T, \mu, \eta)$ in $\mathbf{End}_{\mathcal{C}}$, where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\mu : T \circ T \to T$ and $\eta : 1 \to T$ are natural transformations in $\mathbf{End}_{\mathcal{C}}$ satisfying the commutative diagrams 2.12. Note that 1 is the identity functor in $\mathcal{C}$.

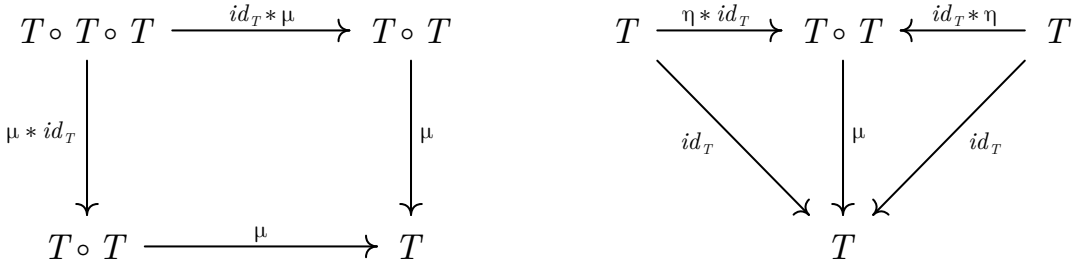



Figure 2.12: Commutative diagram for monad.

Monads induce a category called *Kleisli category*. Such category is of special interest in Functional Programming, where it is used to model computational effects.

**Definition 2.1.19 (Kleisli Category).** Let $(T, \mu, \eta)$ be a monad over a category $\mathcal{C}$. The Kleisli category is the category $\mathcal{C}_T$, where:

(i) $\mathrm{Ob}_{\mathcal{C}_T} = \mathrm{Ob}_{\mathcal{C}}$;

(ii) $\mathrm{Mor}_{\mathcal{C}_T}(A, B) = \mathrm{Mor}_{\mathcal{C}}(A, TB)$;

(iii) the composition of morphisms in $\mathcal{C}_T$ is given by:

$$g \circ_T f := \mu_C \circ Tg \circ f,$$

with $f : A \to TB$ and $g : B \to TC$ as morphisms in $\mathcal{C}$;

(iv) the identity morphism of an object $A$ in $\mathcal{C}_T$ is $\eta : A \to TA$.

In summary, the Kleisli category is defined in such a way that two morphisms $f : A \to TB$ and $g : B \to TC$ can be composed using $g \circ_T f : A \to TC$.

### 2.1.7 F-Algebras, F-Coalgebras

The informal idea of "doing algebra" is that we have an expression, such as $1 + 1$, and we want to evaluate this expression so that we get a single number, i.e. $1 + 1 = 2$. This concept of defining generic expressions and evaluating them can be expressed in Category Theory via the $F$-algebras:

**Definition 2.1.20 ($F$-algebra).** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An $F$-algebra is a tuple $(A, \phi)$, where:

- An object $A \in \mathcal{C}$, called *carrier*;
- A morphism $\phi : FA \to A$, called *structure map*.

Note that for a fixed endofunctor $F$, we can define different algebras by picking different carriers or different structure maps.

**Example 2.1.7.** Consider an endofunctor $F$ as:

$$(\cdot \times \cdot + 1) : \mathbf{Set} \to \mathbf{Set},$$

where $\times$ is the Cartesian product, $+$ is the disjoint union operator and 1 is the terminal object in **Set**, i.e. a generic singleton set $\{e\}$.

We can then define an $F$-algebra $(\mathbb{Z}, \phi)$, where:

$$\begin{gathered}\phi : (\mathbb{Z} \times \mathbb{Z} + \{e\}) \to \mathbb{Z}, \text{ such that} \\ \phi((a, b)) \mapsto a + b \\ \phi(e) \mapsto 0.\end{gathered}$$

What this means is that, for our specific $F$-algebra, an expression is a value of $F\mathbb{Z} = \mathbb{Z}\times\mathbb{Z}+\{e\}$, which is either a tuple of integers, or $e$. Thus, our map $\phi$ evaluates expressions by either summing the tuple of integers or returning 0 for $e$.

Given two $F$-algebras $(\phi, A)$ and $(\psi, B)$, a morphism $f : A \to B$ is an *algebra homomorphisms* between $(\phi, A)$ and $(\psi, B)$ if the diagram in Figure 2.13 commutes.

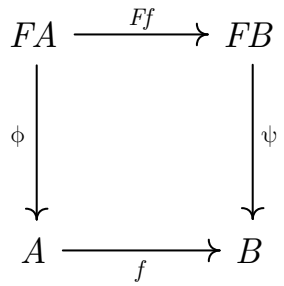



Figure 2.13: Commutative diagram for $F$-algebra morphism.

For a fixed endofunctor $F$, we can now define the category of $F$-algebras:

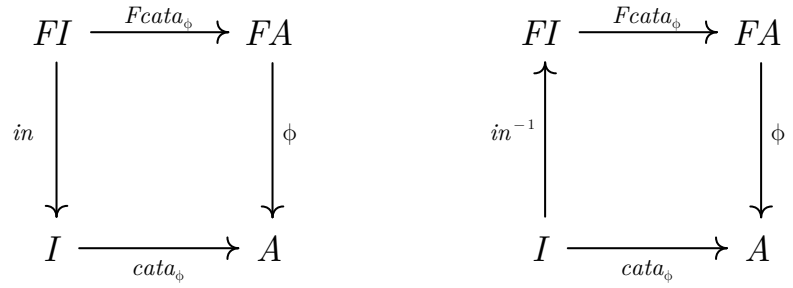
**Definition 2.1.21 (Category of $F$-algebras).** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. The category of $F$-algebras, denoted by $\mathcal{A}lg(F)$, has $F$-algebras as objects, and algebra homomorphisms as morphisms.

Depending on the endofunctor, the $\mathcal{A}lg(F)$ category might have an initial object, called *initial algebra*. As we have shown, initial objects are unique up to an isomorphism, therefore, we can pick a single representative. In this case, we use $(I, in)$ as the initial $F$-algebra. A result known as Lambek's theorem, originally proved in [32], states that if $(I, in)$ is an initial algebra, then $I \cong FI$ via $in$, i.e. $I$ is isomorphic to $FI$. This theorem also implies that $I$ is the least fixed point of $F$.

The fact that $(I, in)$ is an initial object of $\mathcal{A}lg(F)$ means that for any algebra $(A, \phi)$, there exists a unique homomorphism $cata_\phi : I \to A$, commonly called catamorphism [43]. The fact that $in : FI \to I$ is an isomorphism implies that there exists $in^{-1} : I \to FI$. This fact, in conjunction with the fact that $cata_\phi$ is a homomorphism allows us to obtain a universal formula for the catamorphism:

$$cata_\phi = \phi \circ F cata_\phi \circ in^{-1}.$$

The diagrams used in this derivation are in Figure 2.14.




$$\phi \circ Fcata_\phi = cata_\phi \circ in \qquad cata_\phi = \phi \circ Fcata_\phi \circ in^{-1}$$

Figure 2.14: Diagrams for the catamorphism.

In programming, the usefulness of this whole discussion on $F$-algebras and initial algebras is that they help us formalize the idea of recursive data structures. For example, consider an endofunctor $F_{\mathbb{Z}} : \mathbf{Set} \to \mathbf{Set}$, where:

$$F_{\mathbb{Z}}(X) := 1 + \mathbb{Z} \times X.$$

The data type representing a list of integers is equal to the carrier object for the initial algebra for this functor:

$$\text{List}_{\mathbb{Z}} \cong 1 + \mathbb{Z} \times \text{List}_{\mathbb{Z}} = 1 + \mathbb{Z} + (\mathbb{Z} \times \mathbb{Z}) + (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \ldots$$

In this context, the catamorphism is the function that generalizes how to apply an $F_{\mathbb{Z}}$-algebra on $\text{List}_{\mathbb{Z}}$.

If algebras evaluate expressions, then coalgebras produce expressions. In categorical terms, the $F$-coalgebra is simply the dual of an $F$-algebra:

**Definition 2.1.22 ($F$-coalgebra).** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An $F$-coalgebra is a tuple $(U, \varphi)$, where:

- An object $U \in \mathcal{C}$;
- A morphism $\varphi : U \to FU$.

One can also construct a category of $F$-coalgebras and consider its terminal object as $(T, out)$. The dual notion of the catamorphism is called anamorphism, and is given by:

$$ana_\varphi = out^{-1} \circ Fana_\varphi \circ \varphi.$$

We can then think of a map that combines both operations together, i.e. it uses the coalgebra to produce expressions, and then consumes it using the algebra. The name of such map is

hylomorphism, defined as $hylo : (A, alg) \times (B, coalg) \times B \to A$. An example of this is the factorial function, as discussed by Slodičák and Macko [59], where the coalgebra takes an integers and produces a list of integers, which is then consumed by multiplying each value. The hylomorphism is computed by:

$$hylo(alg, coalg) = alg \circ Fhylo \circ coalg.$$

### 2.1.8 Free Constructions

Suppose we have a set $M := \{a, b, c\}$, and someone asks us to define a monoid over this set. What would be the most "generic" monoid possible? Although the question is not rigorously posed, the possible answer would be $(\text{Free}(M), \oplus, e_M)$, where $\text{Free}(M)$ is the set of finite lists of element of $M$, $\oplus : \text{Free}(M) \to \text{Free}(M)$ as list concatenation, and $e_M$ as the empty list. One can check that this indeed satisfies the monoid axioms (associativity and the existence of an identity element). Informally, what makes this the most *free* construction is that it specifies a monoid from the generator set $M$ without imposing any further constraints.

In Category Theory, free constructions can be formally expressed through the concept of adjunctions[1]. These free constructions are characterized by having a universal property in which any map from the generators to another object factors uniquely through the free construction.

**Example 2.1.8 (Free Monoid).** Let **Mon** be the category of monoids, where the objects are monoids and the morphisms are monoid homomorphisms. In this category, $(\text{Free}(M), \oplus, e_M)$ is the free monoid generated by $M$ if there exists a set-function $m : M \to \text{Free}(M)$ such that, for any monoid $(N, \otimes, e_N)$ and any function $f : M \to N$, there exists a unique monoid homomorphism $\phi : \text{Free}(M) \to N$ such that the diagram below commutes.

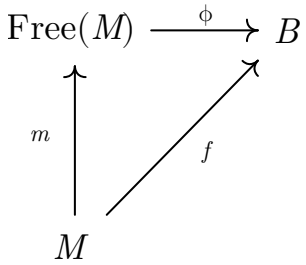



Figure 2.15: Commutative diagram for free monoid.

Free constructions are useful because they provide a canonical way to construct objects from given generators while satisfying structural properties with minimal constraints. For example,

[1]Due to space, we do not discuss adjunctions. The topic is very standard and covered by most introductory texts on Category Theory [7, 54, 34, 43]

in computer science, the idea of lists over a given data type arise naturally from the need of a data structure that satisfies the monoid axioms.

Another quite useful example of free constructions are free monads.

**Example 2.1.9 (Free Monad).** We showed how the free monoid over a set $M$ is equivalent to the set of lists of elements of $M$. Remembering the motto "monads are monoids in the category of endofunctors", a free monad is a free monoid in the category of endofunctors, or, a free monad over an endofunctor $F$ is equivalent to the functor representing a list of applications of $F$.

Let $F : \mathbf{Set} \to \mathbf{Set}$ be an endofunctor. The free monad generated by $F$ is a monad $(\mathbb{T}, \eta, \mu)$, where:

$$\mathbb{T}(X) = X + F(X) + F^2(X) + F^3(X) + \dots$$

where each term $F^n(X) = F(F(...(X)))$ is the $n$-fold application of $F$ to $X$.

A free monad accumulates the functor application in a tree-like structure, which can then be evaluated using an algebra. In this case, the algebra is defined in the category of endofunctors, which means that for an algebra $(G, \alpha)$, the carrier $G : \mathbf{Set} \to \mathbf{Set}$ is an endofunctor, and $\alpha : \mathbb{T}(G) \to F$ is a natural transformation. For a more complete exposition of free monads, see Milewski [44].

## 2.2 Applied Category Theory

In recent years, Category Theory has found applications in a wide range of disciplines, such as chemistry, biology, natural language processing, database theory and so on [11]. Such wide spread applications gave rise to Applied Category Theory (ACT) as a field on its own[2].

In this thesis, we draw on foundational ACT influences, particularly the application of CT to programming. Notable influences include Spivak [61]'s work on CT for databases, which connects categorical concepts to data structures, and Yorgey [85]'s exploration of monoids for diagramming.

In the following sections, we begin by examining the connections between Category Theory and programming. We then review the contributions of Spivak [61] and Yorgey [85], before exploring how Category Theory has been applied in the context of data visualization.

[2]There is no precise data when ACT started to be treated as a field on its own. Yet, we can assume that is a fairly recent since the first conference named "Applied Category Theory Conference" is from 2018.

### 2.2.1 Programming with Category Theory

Perhaps the most influential application of Category Theory has been in programming, specially within Functional Programming. Orchard and Mycroft [46] states that the application of CT to programming can be divided into two distinct approaches, namely *categorical programming* and *categorical semantics*.

Categorical semantics formally interprets programming languages through the structures of Category Theory, as a means to define the precise meaning of programs and their components. In this approach, one commonly interprets types of the programming language as objects in a category, while functions or transformations between types are represented as morphisms. Employing this abstraction gives a more rigorous and principled way of studying programming constructions, allowing for formal proofs of properties such as code correctness and equivalence.

Categorical programming, on the other hand, uses categorical concepts as design patterns for organizing and structuring programs. Rather then using Category Theory to rigorously define the program's semantics, one uses categorical constructions, such as functors and monads, as tools for creating clear abstractions and designing modular, composable code. Thus, categorical programming can be used even within programming languages that are far removed from the Functional Programming paradigm.

By integrating these category-theoretic principles, both approaches offer powerful ways to model, structure, and reason about software. Categorical semantics provides a foundational understanding, while categorical programming applies these abstract concepts in more practical ways. Together, they bring the clarity and rigor of Category Theory into the realm of programming.

### 2.2.2 Category Theory for Databases

Spivak [62] used Category Theory as a way for formalize relational databases. Database schemas are interpreted as categories, and the collection of all database schemas then forms a category **Sch**, which is equivalent to the category of small categories (see Example 2.1.6).

**Definition 2.2.1 (Database Schema).** A *schema* $S$ is a pair $(G, \sim)$ where $G := (V, A, src, tgt)$ is a graph and $\sim$ is an equivalence relation of paths on $G$. From this schema, we can generate a category $\mathcal{S}$ where $\text{Ob}_{\mathcal{S}} := V$, and $\text{Mor}_{\mathcal{S}} := \text{Path}(G)/\sim$, which is the quotient set formed by the paths in the graph $G$[3].

The idea behind this definition is that we can formalize the database schemas as graphs where vertices are tables or data types, and the morphism are columns. Consider the example shown in Figure 2.16. The tables are shown as black vertices while the data types are shown

[3]We do not give a formal definition of Path here. Instead, the interested reader can read Spivak [61], Spivak [62] or Fong and Spivak [21].

as white vertices. We have two tables, Employee and Department. The Employee table has an id column (which is given by the identity morphism), and columns FName, WorksIn and Mngr. The Department table also has an id column, and columns DName and Secr.

In this schema, Mngr, Secr and WorksIn are foreign keys. They are internal references, in the sense that they point to the database table itself. FName and DName are external references, as they point to the data type String [21].

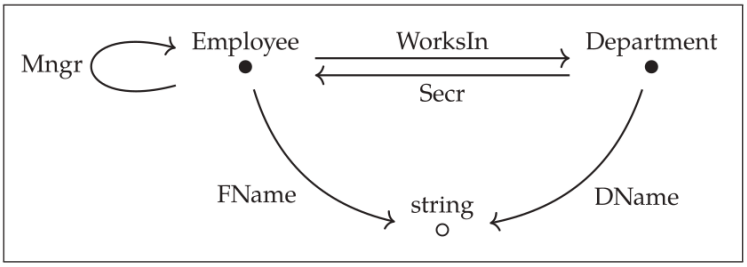



Figure 2.16: Example of a database schema from Fong and Spivak [21].

We could add an equivalence relation such as

$$\forall e \in \text{Employee}, \quad (\text{WorksIn} \circ \text{Mngr})(e) \sim \text{WorksIn}(e).$$

This relation enforces that the manager of every employee works in the same department as the employee.

If a database schema generates a category $\mathcal{S}$, how can one represent an actual instance of this database, i.e. how can one represent an example of this database populated? This can be done with a functor $I : \mathcal{S} \to \mathbf{Set}$.

Note, for example, that our functor $I$ takes a table Employee to a set of employees ids, and it takes type String to the set of all strings. It then takes a morphism FName to a function in **Set**, which is just a subset of Employee $\times$ String, i.e. for each employee id we have a string (FName stands for first name of the employee). Similarly, $I$ takes Mngr to a list of tuples of employee ids. The same interpretation follows for the other morphisms and tables. Figure 2.17 illustrates an example of an instance of this database.

| **Employee** | **FName** | **WorksIn** | **Mngr** |
|---|---|---|---|
| 1 | Alan | 101 | 2 |
| 2 | Ruth | 101 | 2 |
| 3 | Kris | 102 | 3 |

| **Department** | **DName** | **Secr** |
|---|---|---|
| 101 | Sales | 1 |
| 102 | IT | 3 |

Figure 2.17: Instance of database schema $S$. Image taken from Fong and Spivak [21].

Using this formalism, Spivak [61] shows how data migration (the process of passing data from one schema to another) can be rigorously defined as functors. In a more recent work, Spivak [63] has formalized within this functorial perspective the process of querying and aggregating a database.

### 2.2.3 Diagrams as Monoids

Yorgey [85] introduces a domain-specific language which models the process of generic diagram creation, which is used by the Haskell library *Diagrams* [83]. The core idea is to use monoids to model various data structures involved in diagramming. The paper begins with a simple model for defining a diagram and gradually builds upon it, enhancing the expressiveness of the system as complexity increases.

Initially, a diagram is defined as an ordered collection of graphical primitives — geometric elements that "can be drawn," such as circles, ellipses, arcs, lines, and polygons. This leads to defining a monoid `(Diagram,*,[])`, where `Diagram = Array{Prim}`, with `Prim` representing the data type for graphical primitives, `[]` as the identity element (an empty diagram), and `*` as the concatenation operation. The process of rendering a diagram then involves sequentially drawing each primitive in the defined order. Figure 2.18 illustrates a diagram containing three primitives, which are then drawn accordingly.

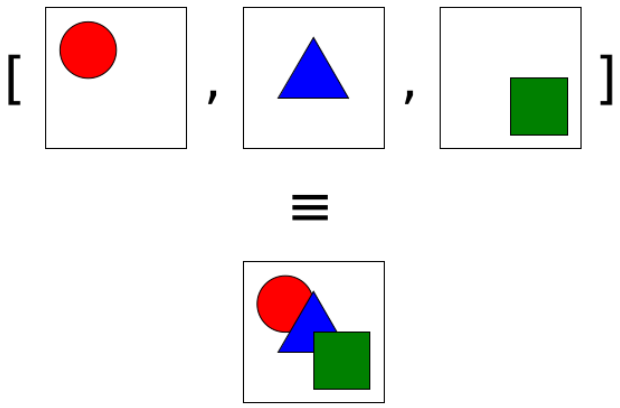


Figure 2.18: Example of how a diagram works. Image from Yorgey [85].

To produce more complex diagrams, one needs to be able to apply transformations to the primitives. Consider `Transformation` to be a type representing affine transformations. We can define a function `transformP(t::Transformation, p::Prim)` that applies transformations to primitives. Note that the triple `(Transformation, ε, ⋄)` also defines a monoid, where `ε` is the identity transformation and `⋄` is the composition of transformations in which `t1 ⋄ t2` simply means applying `t2` then `t1`.

To apply transformations to diagrams, we can define a new function based on `transformP`:

```
transformD(t) = d::Diagram -> fmap(t::transformP, d)
```

While the original diagram type was just an ordered list of primitives, the improved version becomes:

```
Diagram = Tuple{Array{Prim},Transformation}.
```

For two diagrams `([circle,triangle], t1)` and `([square], t2)`, the composition becomes:

```
([circle,triangle], t1) * ([square], t2)

(transformD(t1,[circle,triangle]) * transformD(t2, [square]), ε)

([transformP(t1,circle),transformP(t2,triangle])]*[transformP(t2, square)], ε)

([transformP(t1,circle),transformP(t2,triangle]),transformP(t2, square)], ε)
```

In the most complex definition of a `Diagram`, Yorgey [85] adds aspects such as aesthetic properties, envelope functions and trace functions. Aesthetic functions are for assigning properties such as colors. The envelope function helps placing diagrams beside each other, while the trace function eases the process of drawing lines between diagrams. Figure 2.19 illustrates what we mean by placing diagrams beside each other.

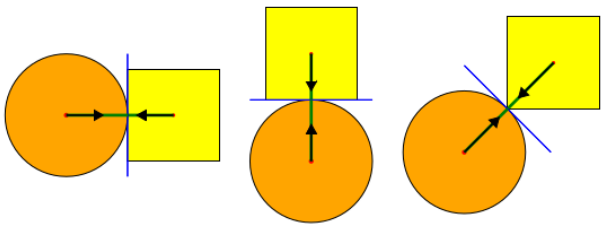


Figure 2.19: Envelope function used to place diagrams beside one another. Image taken from Yorgey [85].

The paper goes into much more details about how to construct diagrams with monoids. It has hugely influenced our work, serving as a basis for our own framework. Before finding out about Yorgey [85], we had already started to think of graphical geometric description as a monoid. Hence, the work of Yorgey [85] gave us confidence on our ideas, and also helped us understand how transformations could be incorporated into this monoidal framework.

### 2.2.4 ACT for Data Visualization

Despite its growth, ACT has made only a few contributions to data visualization. From our literature review, only Vickers et al. [71] explicitly referenced Category Theory for data visualization.

Vickers et al. [71] showed how to describe the process of information visualization within Semiotic Theory, and then formalized this semiotic framework using Category Theory. Although interesting on its own, the theory proposed by Vickers et al. [71] does not tackle the problem of formally defining the process of data visualization from data encoding to graphic representation.

In terms of data visualization tools, Smeltzer et al. [60] developed a Haskell-embedded domain-specific language that incorporates categorical concepts, particularly functors and monads, to build data visualizations. The primary idea behind this DSL is to create visualizations by incrementally applying transformations, rather than a single charting function or specification (see Figure 2.20). In it, visualizations are modeled as a recursive data structure `Vis`, which represents a hierarchy of visual elements. Each `Vis` node can contain either a basic graphical mark (e.g., lines, bars) or a composite structure made up of other `Vis` elements.

Visual transformations in this DSL apply incremental adjustments to `Vis` elements, modifying properties such as position, size, and color. Transformations can be applied sequentially, thus allowing incremental modifications. Visualizations can be composed via spatial operators such as `NextTo`, `Above` and so on.

When it comes to Category Theory, Smeltzer et al. [60] utilizes functors to enable consistent transformations across `Vis` elements through the `fmap` function. Monads are then used to extend this functionality by allowing the composition of functions with type signature `Mark a -> Vis (Mark a)` (see Definition 2.1.19), which can be used for generating additional visual structures based on existing elements. Figure 2.20 shows the difference between the initial chart and the one after the transformations.

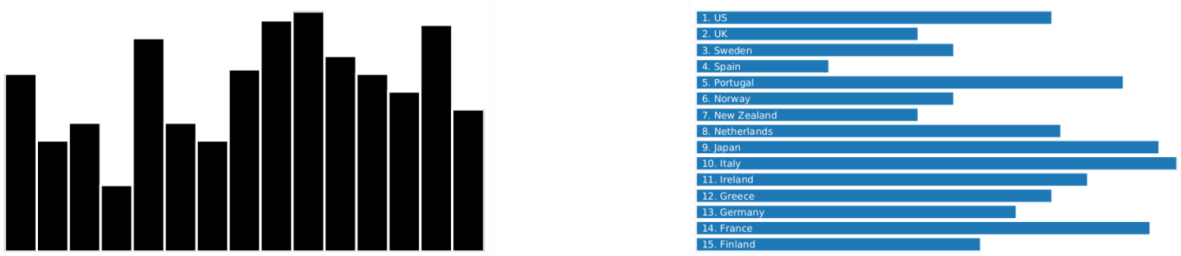



Figure 2.20: The *bar plot* on the left is the original, while the one on the right is after the transformations. Image from Smeltzer et al. [60].

# Chapter 3

# Categorical Programming with Julia

Our data visualization theoretical framework employs categorical programming as a means to bridge theoretical concepts and their implementation. Moreover, our data visualization tool implements this theory using the Julia programming language. Hence, it is paramount to give an account of how categorical programming can applied within the Julia language. To our knowledge, such exposition has not yet been done, as most of the literature on this subject focuses on more purely functional languages, such as Haskell.

Therefore, our goal in this chapter is to introduce the basics of the Julia programming language, and show how it relates both to Functional Programming and Category Theory.

## 3.1 Julia Programming Basics

Julia is a high-level programming language designed with scientific computing in mind. The language was first released in 2012 [5] and has since then gained popularity among the scientific community. Different from most dynamic languages, it uses types extensively, which together with multiple dispatch provide a very elegant way of designing programs.

Another interesting point is that Julia is multi-paradigm, combining features of Imperative, Functional, and Object-Oriented Programming. However, it can be argued that Julia tends towards Functional Programming. Those coming from OOP languages, such as Python, might be surprised to find that there are no classes or objects. Instead, Julia has structs, which allow users to construct new data types containing named fields.

Let us give a brief overview of the basics of the Julia programming with an emphasis on the Functional Programming aspects relevant to the categorical interpretation and to our data visualization framework.

### 3.1.1 Julia's Type System

In Julia, types can be concrete or abstract. Only concrete types can be instantiated, while abstract types are ways of grouping these types. For example, `Int` and `Float64` are concrete types, while `Number` is an abstract type that contains both, i.e. `Int` and `Float64` are subtypes of `Number`. We can check whether a type is a subtype of another type via the `<:` operator, as shown below.

```
julia> Int <: Number, Float64 <: Number
(true, true)
```

Types are enforced in the input of functions, e.g. `f(x::Int, y::String)`. Yet, the output type of a function is not enforced. This type enforcing allows for a feature called multiple dispatch. In Julia, a function has methods, which are specific implementations of the function for different combinations of argument types. Thus, we can have something like:

```
f(x::Int) = x^2
f(x::Int,y) = (x^2, y)
f(x::String) = "a string!"
f(x::Int, y::Int) = x + y

julia> f(2)
4

julia> f(2,"ok")
(4,"ok")

julia> f("test")
a string!

julia> f(1,1)
2
```

Note that the same function `f` was defined several times, one for each dispatching argument. Each of these instances is called a method. When we do not define the type of the argument, Julia uses the type `Any`, which means any type. If a function has a method with a more specific type, the compiler tries to call the more specialized method. In our example, the method `f(x::Int, y::Int)` was more specialized than `f(x::Int, y)`. This idea of a more specialized method is possible due to the fact that types have a hierarchy as we have shown in the example with types `Int`, `Float64` and `Numbers`.

Although the return type of a function is not enforced, Julia provides a syntax for specifying it. This can be done by declaring the function with `::`, and by doing so, Julia tries to convert the return value to the output type.

```
function f(x, y)::Float64
    return x + y
end;

julia> f(1, 1)
2.0
```

Besides the default types that Julia provides, we can create new types. By creating new types, we can define functions that take variables with these new types as arguments and dispatch on them. One way of defining a new type is via a `struct`:

```
struct Point2D
    x::Real
    y::Real
end
julia> p = Points2D(1,1)
Point2D(1, 1)

julia> p.x
1

julia> getproperty(p, :x)
1
```

Structs are by default immutable, which is good for FP. Yet, we can define mutable structs by simply adding the word "mutable" before the "struct". Here is a more interesting example of how to use structs, type hierarchy and multiple-dispatch:

```
abstract type Shape end
struct Point3D
    x::Real
    y::Real
    z::Real
end
struct Square <: Shape
    center::Point3D
    length::Real
end
struct Circle <: Shape
    center::Point3D
    radius::Real
end
area(s::Shape) = 0.0
area(s::Circle) = π * s.radius^2
area(s::Square) = s.length^2

julia> s = Square(10,Point3D(0,0,0));
julia> area(s)
100
```

In our example, we set the default area of a shape to zero by defining our `area(s::Shape)` equal to `0.0`. For the shapes that we know how to compute the area, we write the formulas (e.g. square and circle). Note that we wrote `Square <: Shape` to indicate that both of our structs are subtypes of `Shape`. The type `Square` and the type `Circle` do not have methods themselves. We instead define functions that dispatch on the desired type. This illustrates how Julia is somewhat FP oriented, but not all the way, since we do not have output type enforcing like, for example, in Haskell.

Types can be parametric, e.g. `abstract type MyType{T} end`. Such parametric types carry a family of types, one for each type `T`, with `MyType` also being a type itself. Structs can also be parametric and can use the type parameter for its field values:

```
abstract type MyType{T} end
struct MyStruct{T} <: MyType{T}
    x::T
    y::Int
end
julia> MyStruct("A",1)
MyStruct{String}("A",1)

julia> MyStruct{Int} <: MyType
true

julia> MyStruct{Int} <: MyType{<: Real}
true

julia> MyStruct{Int} <: MyType{Real}
false
```

Note that subtyping for parametric types might be unintuitive. As seen in the code example above, `MyStruct{Int}` is not actually a subtype of `MyType{Real}`, but of `MyType{<:Real}` and of `MyType`. Type parameters can also be used for multiple-dispatch.

### 3.1.2 The Functional Programming Tenets

According to Widman [79], there are three "main" programming paradigms: Imperative programming, Object-Oriented Programming, and Functional Programming. Imperative programming focuses on defining variables and control structures (e.g. loops, conditional), which are executed in a particular order to achieve a desired outcome.

OOP models programs via objects and classes. A class is a template that describes the properties and methods that an object will have. An object is an instance of a class, and it has an internal state, which encapsulates the value mutation in the program. The program runs by instantiating objects that interact with each other to achieve a specific task.

Finally, FP models programs as applying and composing functions. A function takes inputs

and return outputs without mutating any values, which is also referred as not having side effects.

Note that what we call a "function" in programming does not match what we call "function" in mathematics. In mathematics, more specifically Set Theory, a function $f : A \to B$ is just a subset of $A \times B$. In programming, a function is not simply a set, it contains an algorithmic description of how it works, and it can sometimes mutate variables outside of its scope.

The Functional Programming paradigm imposes a series of "restrictions" in order to approximate a programming function to its mathematical counterpart. These "restrictions" are the tenets of FP and usually consist of the following [2]:

1. **Immutability**: once a value is assigned to a variable, it cannot change;
2. **Pure functions**: functions do not have side effects, i.e. do not alter values, they only receive inputs and return outputs.
3. **Referential transparency**: for the same input, a function always return the same output. An example that is not referentially transparent would be a function `rand()` that returns random numbers;
4. **First-class functions**: functions are similar to values, in that that they can be used as arguments, assigned to variables and be returned by other functions;
5. **Higher order functions**: functions can take other functions as arguments;
6. **Composability**: functions can be composed to define new functions;
7. **Lazy evaluation**: expressions can be evaluated only when needed.

The emphasis of FP in controlling side effects makes programming functions similar to functions in Set Theory.

Let's explore how the tenets of FP fit into Julia, e.g. immutability, purity of functions, lazy evaluations and so on.

### Immutability

Variables in Julia are not immutable. We can do `x = 1` and follow with `x = "text"`. This will alter the value of variable `x`. Yet, types are immutable. Once a type is constructed, it cannot be modified. Suppose you define the following type:

```
struct MyType
  a::Int
  b::Int
end
```

Once `MyType` has been created, you cannot modify it. If you try to recreate it by copying the code above, but using `a::String` instead of `a::Int`, you will incur in an error. Another point to note is that structs are by default immutable types, as shown in the example below.

```
julia> x = MyType(1,2);

julia> x.a, x.b
(1,2)

julia> x.a = 1
ERROR: setfield!: immutable struct of type MyType cannot be ...
```

A way around this is to define a `mutable struct`. Although, this is often discouraged due to efficiency concerns.

### Side Effects, Pure Functions and Referential Transparency

Functions in Julia are not necessarily pure. It is up to the user to control the possible side effects of a function. The language convention is to name functions with an exclamation to indicate that they have side effects. This is *just* a notation. A function with an exclamation can still be pure, and a function without can still produce side effects. Consider the example below:

```
x = []

function f!(x)
    push!(x,1)
end

julia> f!(x);
julia> x
Any[1]
```

One can use a macro[1] `@pure` to indicate to the compiler that a function is pure. This can improve performance, but does not actually make the function pure, which is again left to the programmer to do. Hence, functions are not pure, and neither are they referentially transparent. For example:

```
julia> rand(1)
1-element VectorFloat64:
 0.024980886821626025
```

[1] A macro is a function type used for metaprogramming, i.e. macros can generate and manipulate Julia code programmatically.

```
julia> rand(1)
1-element VectorFloat64:
 0.8808740912475495
```

There are (not to our knowledge) any notation recommended by the Julia community to indicate that a function is not referentially transparent.

### First-Class, High Order Functions and Composability

Functions are first-class citizens, meaning that they can be passed around as variables. They can also be anonymous or named:

```
function f(x::Int)
  return x^2
end

julia> g = f;
julia> g(2)
4

julia> h = x::Int -> x^2;
julia> h(3)
9
```

There is an abstract type `Function` for functions. When a function is defined, a new type specific for such function is created. For named functions, the type is actually named `typeof(name_of_function)`. For anonymous function, a type is generated by the compiler.

```
function f(x::Int)
  return x^2
end

julia> typeof(f)
typeof(f) (singleton type of function f, subtype of Function)

julia> typeof(x->x)
var"#3#4"

julia> typeof(x->x)
var"#5#6"
```

Since functions are first-class citizens, we also have high-order functions at our disposal. A common example in FP languages is a function `map(f::Function, A::AbstractArray)`, which takes a function and applies it to the elements of an array. Julia provides a `do`-block as a convenient syntax for passing an anonymous function as the first argument to higher-order functions:

```
# Without do-notation
map(x -> x^2, [1, 2, 3])

# With do-notation
map([1, 2, 3]) do x
    x^2
end
```

We also have function composition at our disposal. Composition is performed with the operator ∘. Julia does not enforce the output type of a function, therefore, it is possible to compose functions that do not have matching types. Again, the responsibility is in the programmer to guarantee that the functions composed actually work together.

```
function f(x::Int)
  x + 1
end

function g(x::String)
  return length(x) * 2
end

julia> h = g ∘ f;

julia> h(1)
ERROR: MethodError: no method matching g(::Int64)...
```

### Lazy Evaluation

In Julia we do not have lazy evaluation as standard, on the contrary, our code is *eagerly* evaluated. This means that once we call a function, it evaluates all the parameters. We can alter our code to try to make it lazy. To do this, we use iterators. Consider the following example:

```
imap = Iterators.map # version of `map` that returns an iterable
take = Iterators.take # returns the `n` first values of an iterable.
squarelazy(nums) = imap(x->x+1,nums)
squareeager(nums) = map(x->x+1,nums)

julia> x = 1:3; # iterator from 1 to 3, represeting a lazy list
2-element VectorInt64:
 1
 2
 3

julia> squarelazy(x)
Base.Generator{UnitRange{Int64}, var"#1#2"}(var"#1#2"(), 1:10)
```

Note that our function `squarelazy` returned `Base.Generator` which works as a lazy iterator. We can use the `collect` function to actually evaluate our iterator.

```
julia> collect(squarelazy(x))
2-element Vector{Int64}:
 1
 2
 3
```

## 3.2 Category Theory in Julia

Our interpretation of Category Theory within Julia is mostly based on Milewski [43]. The starting point is to interpret data types (e.g. `Int`, `String`,`Float`) as sets, subtypes as subsets and programming functions as set-functions, so that we can consider to be working within **Set**. Once this has been established, we can start to to model categorical concepts.

### 3.2.1 Initial and Terminal Objects

We know that in **Set** the $\varnothing$ set is the initial object. In Julia, the initial object corresponds to type `Union{}`. This type is also known as `Base.Bottom`, and it is a subtype of every type in Julia, including itself, just like how $\varnothing$ is a subset of every other set, including itself.

```
julia> Union{} <: Int, Union{} <: Nothing, Union{} <: Union{}
(true, true, true)
```

Note that a function $f : \varnothing \to A$ cannot ever be called, since we cannot provide an element of $\varnothing$. In the same way, a function `f(x::Union{})` cannot be called, because there is no instance of type `Union{}`.

A terminal object in **Set** is any singleton set. Thus, the equivalent for a terminal object is any type with only a single possible instance. Therefore, just like it happens in **Set**, we have various types isomorphic to each other, and all representing the terminal object. For example, type `Nothing` has only a value `nothing`, type `Tuple{}` has only `()` as an element, and so on. We can also define more terminal types using structs:

```
struct Terminal end

julia> Base.issingletontype(Terminal)
true
```

The only element of our type `Terminal` is `Terminal()`.

### 3.2.2 Interpreting Values

Now that we have showed that Julia has terminal objects, we can also formally interpret instances of types. In Category Theory, we cannot talk about what composes an object, only how morphisms act on them. Hence, if we wish to interpret programming within a category, we need to reinterpret what a value of a type is.

The answer for this question is in **Set**. The number of functions going from a singleton set $\{a\}$ to an arbitrary set $A$ is equal to the number of elements in $A$. Therefore, each element $x \in A$ is isomorphic to a morphism $x : \{a\} \to A$. This suggests that an instance of a certain type `T` can be interpreted as a morphism (function) from a singleton type to `T`, e.g. the value `1::Int` is the same as a function `1(::Nothing)`.

When programming, this convoluted representation is not necessary. Yet, it allows us to talk about instances of types without leaving our categorical interpretative framework.

### 3.2.3 Products and Co-products

In **Set**, we have the product of two sets is the Cartesian product, and the co-product is the disjoint union. The same can be applied for types. The `Tuple{Type1, Type2}` is the product of `Type1` and `Type2`, while `Union{Type1,Type2}` is the co-product.

```
julia> ("a", 1) isa Tuple{String,Int}
true

julia> Int <: Union{String,Int}, String <: Union{String,Int}
(true, true)

julia> 1 isa Union{String,Int}, "a" isa Union{String,Int}
(true, true)
```

We can also define types for infinite products using `Vararg`. For example,

```
julia> FinSeq = Tuple{Vararg{Int}};
TupleVarargInt64

julia> (1,2) <: FinSeq, (1,2,3,4,5) <: FinSeq
(true, true)
```

### 3.2.4 Functors

A functor acts on categories, taking objects to objects, morphisms to morphisms, while preserving compositions and identities. Therefore, a functor must take types and return types, take functions and return functions, and it must satisfy the composition and the identity properties. The functors that interests us are the endofunctors over **Set**, i.e. $F : \mathbf{Set} \to \mathbf{Set}$.

This can be modeled by a parametric struct `F{T}` together with a higher-order function `fmap(f::Function, x::F{T})`. Consider the following example:

```
struct F{T}
    x::T
    y::T
end
fmap(f::Function, a::F{T}) where T = F(f(a.x), f(a.y))
F(f::Function) = a -> fmap(f,a);

julia> a = F(1,2)
F{Int64}(1, 2)

id(x) = x
f(x)  = x*2
g(x)  = x^2

julia> fmap(f, a)
F{Int64}(2, 4)

julia> F(f)(a)
F{Int64}(2, 4)

julia> (F(f) ∘ F(g))(a) == F(f ∘ g)(a)
true

julia> F(id)(a)
true
```

So why is this implementation a functor? Because, `F` defines takes types `T` to types `F{T}`, while the `fmap` encodes how `F` acts on functions, i.e. it defines `F(f)`. Moreover, it preserves composition and identity, as shown above.

Note that just being a parametric type with an `fmap` does not guarantee that it is a functor in the categorical sense. It is the user who must check that the specific implementation satisfies the theoretical properties for a functor. Next, let us do some other examples.

#### Identity Functor

The identity functor was introduced in Example 2.1.5. It takes every type `T` to itself, and every function `f` to itself.

```
struct Id{T}
    value::T
end
fmap(f::Function, i::Id{T}) = Id(f(i.value))
```

The `Id` functor just wraps values into the `Id` struct. To apply the `fmap` with a function `f` is effectively the same as applying `f` directly. Note that, technically speaking, our functor `Id` is taking a type `T` to `Id{T}`, which is different leaving it as `T`. Yet, it is easy to see that the `Id{T}` type is isomorphic to type `T`, hence, for our categorical interpretation they are the same object.

### Array as a Functor

There are many other examples of functors. Another very useful one are arrays. Note that an array takes a type `T` and returns a type `Array{T}`. Julia has a function `map` defined on arrays that do what our `fmap` would do.

```
julia> fmap(f::Function, v::Array{T}) where T = map(f,v);

julia> A = [1,2,3];

julia> fmap(x->x^2, [1,2,3])
3-element Vector{Int64}:
 1
 4
 9
```

### Maybe Functor

Let us end this subsection with functor `Maybe` (also called `Option`). This functor is a way of dealing with functions that might return nothing. Think for example of a function that searches a database and returns what it found. The result should then be composed with another function that does some calculation. If the original search returns nothing, the calculation will return an error. We avoid this with `Maybe`.

```
struct Just{T}
    x::T
end
just(J::Just) = J.x

Maybe{T} = Union{Nothing,Just{T}}
Maybe(a::T) where T = Just(a)
```

```
Maybe(::Nothing) = nothing
Maybe() = nothing

fmap(f::Function, a::Just{T}) where T = Just(f(a.x))
fmap(f::Function, a::Nothing) = nothing

Maybe(f::Function) = a::Maybe -> fmap(f,a)
```

In the code above we have created the `Maybe` functor. Note that it consists of a union of types `Nothing` and a parametric type `Just{T}`. Next, let us show an example of how this indeed preserves identities and function compositions.

```
# running our example
julia> Maybe(identity)(Maybe()) isa Nothing
true

julia> f(x::Int)::Int = x^2; g(x::Int)::Int = x + 1

julia> Maybe(f∘g)(Maybe(10)) == (Maybe(f) ∘ Maybe(g))(Maybe(10))
true
```

Our functor worked. Let us do an actual application to show how it can be useful. Consider that we have a function `get_user_from_db` that gets a value from a database by searching for an index. If the index is not in the database, it returns nothing. Hence, our function is of type `Union{User, Nothing}`.

Next, we have a function `get_name(x::User)::String`, that returns the name of the user, if a user is passed. We cannot (in good conscience) compose our function `get_name` with the function `get_user_from_db`, because the types do not match.

We could solve this by extending our `get_name` function to be able to receive `nothing` values, then apply an if statement to check if the `nothing` is passed, and then return `nothing`. But this involves a lot of refactoring and our `get_name` does exactly what it is supposed to do. We do not want to add some extra behavior to deal with incorrect input. In this scenario, we can use `Maybe(get_name)`, which takes `Maybe{User}` and returns `Maybe{String}`.

```
# Create a User type
struct User
    name::String
end

# Populate fictional database
julia> DATABASE = Dict( 4 =>User("James"), 1 =>User("Peter"));

# Define function to search DB
function get_user_from_db(uid::Int)::Union{User, Nothing}
    if uid in keys(DATABASE)
        return DATABASE[uid]
```

```
    end
    return nothing
end

# Definine function to extract name from user
julia> get_name(x::User)::String = x.name;
julia> get_user_from_db(1)
User("Peter")

# Note that we get no errors here even though the user does no exists
julia> fmap(get_name, Maybe(get_user_from_db(10)))
Nothing

# Note that it returns `Just{String}` and not `String`
julia> fmap(get_name, Maybe(get_user_from_db(1)))
Just{String}("Peter")
```

### 3.2.5 Natural Transformations

Once we have shown how to code a functor, we can talk about natural transformations. A natural transformation is a "covariant" mapping between functors, in the sense that we can first apply the transformation to the input value and then the functor, or we can apply the functor and then the transformation.

It is important to note that not every pair of functors have a natural transformation between them. Actually, the existence of a natural transformation between two functors is a relevant information about the similarity of these functors.

In Julia, a natural transformation `α` is a polymorphic function, meaning that it is a function that is defined over many types via parametric types. For example, given two functors `F` and `G`, a natural transformation `α{T}` is a function from type `F{T}` to type `G{T}`, such that for every function `f` from type `A` to `B`, we have `α{B} ∘ F(f)(x) == G(f) ∘ α{A}(x)`.

Note that, just like with functors, it is up to the user to check whether a given implementation indeed satisfies the naturality condition. Below we implement an example:

```
# Defining functors F and G
struct F{T}
    x::T
    y::T
end

struct G{T}
    a::T
    b::T
end

fmap(f,v::F) = F(f(v.x),f(v.y))
```

```
fmap(f,v::G) = G(f(v.a),f(v.b))
F(f::Function) = x::F -> fmap(f,x)
G(f::Function) = x::G -> fmap(f,x)

# Defining natural transformation between F and G
α(v::F{T}) where T = G(v.y,v.x)

julia> α(fmap(f,a)) == fmap(f, α(a))
true
```

### 3.2.6 Monoids and Monads

A monoid in **Set** is a triple $(M, e, \otimes)$ where $M$ is a set, $e \in M$ is the neutral element and $\otimes$ is an associative binary operator. In programming, many types can be given a monoidal structure, for example, `(String, "", *)` is the monoid over strings where the neutral element is the empty string and the binary operation is string concatenation. Remember that the same type can have multiple monoidal structures, such as in `(Int, 0, +)` or `(Int, 1, *)`.

For our data visualization framework, the most important example of monoid are lists (arrays). Given any type `A`, we can use the `Array` functor to define a monoid `(ArrayT, T[], vcat)`, where `vcat` is the list concatenation function and `T[]` is the empty list for values of type `T`.

A monad is more complex. Remember that a monad is a monoid in the category of endofunctors. It consists of a triple $(T, \mu, \eta)$ where $T : \mathcal{C} \to \mathcal{C}$ is an endofunctor, $\mu : T \circ T \to T$ is a natural transformation called join, and $\eta : Id \to T$ is a natural transformation called unit. They must satisfy the commutative diagram shown in Figure 2.12.

This suggests that, computationally, a monad is defined in the level of parametric types together with a pair natural transformations. Instead of trying to explain how a monad works, it is better to dive into an example[2]. We are going to define a monad `(Writer, μ, η)`, and we start by creating the functor `Writer`.

```
struct Writer{T}
    a::T
    log::String
end
fmap(f::Function, x::Writer) = Writer(f(x.a),x.log)
Writer(f::Function) = x->fmap(f,x);

julia> fmap(x->x^2, Writer(10,"ok"))
WriterInt64(100, "ok")
```

The functor embellishes functions by adding a string to them. This is useful if one wishes to

[2]The example we are going to present is highly inspired in Milewski [43].

attach a log to a function, and it can be done by simply wrapping the original function in our `Writer` functor. See the example below:

```
square(x::Int) = x^2
square_log(x::Int) = Writer(square(x),"Value squared. ")

addone(x::Int) = x + 1
addone_log(x::Int) = Writer(addone(x), "+1. " );
```

Imagine now a situation where we have a variable `x` and that we apply a sequence of transformations to it. As we apply the transformations, we want to update the log, so that by the end we have the final result together with the log of all operations applied. In other words, we want to be able to compose `square_log` and `addone_log`. The problem is, their types do not match. The question is then, how can we compose two functions with type signature `Int -> Writer{Int}`?

Monads solve this question for us via the Kleisli category 2.1.19, denoted by $\mathbf{Set}_{\texttt{Writer}}$. Note that, in the definition of the Kleisli category we have a special composition that uses the monadic operator $\mu$. Hence, in order to compose our functions `Int -> Writer{Int}`, we must specify a monad over the endofunctor `Writer` by implementing two polymorphic functions η and μ. This is done in the code below:

```
η(x::T) where T = Writer(x, "")
μ(w::Writer{Writer{T}}) where T = Writer(w.a.a,w.log * w.a.log )
```

The η represents the neutral element, while μ is the monadic binary operator. The neutrality of η is shown by composing it with μ:

```
x = Writer(1,"Ok.");
julia> μ(η(x)) == x
true
```

Next, note that if we try to apply the `fmap` to our embellished function, we get a `Writer{Writer{Int}}`. Using the monadic μ we can "flatten" it into a `Writer{Int}`.

```
julia> fmap(addone_log, Writer(10,"")
WriterWriterInt64(WriterInt64(11, "+1."), "")
```

We can now implement the Kleisli composition using ⊗ to represent it. With this operator, we can finally compose `square_log` and `addone_log` even though their types don't match:

```
⊗(g::Function, f::Function) = μ ∘ Writer(g) ∘ f

julia> (square_log ⊗ addone_log)(1)
Writer{Int64}(4, "+1. Value squared. ")

julia> (addone_log ⊗ square_log ⊗ square_log)(1)
Writer{Int64}(2, "Value squared. Value squared. +1. )
```

### 3.2.7 F-Algebras and F-Coalgebras

The implementation of $F$-algebras and $F$-coalgebras is quite straightforward. Given a functor `F{T}`, an algebra is simply a function such as `alg(a::F{Int})::Int`, where `Int` here is the carrier. Similarly, a coalgebra would be `coalg(a::Int)::F{Int}`.

Let us do an example. Consider the following functor `RingF`:

```
struct RZeroF end
struct ROneF end
struct RAddF{T}
    _1::T
    _2::T
end
struct RMultF{T}
    _1::T
    _2::T
end
RingF{T} = Union{RZeroF,ROneF,RAddF{T}, RMultF{T}}

fmap(f::Function, x::RZeroF) = RZeroF()
fmap(f::Function, x::ROneF)  = ROneF()
fmap(f::Function, x::RAddF{T}) where T = RAddF(f(x._1),f(x._2))
fmap(f::Function, x::RMultF{T}) where T = RMultF(f(x._1),f(x._2))
```

The functor `RingF` is a abstract parametric type, where each concrete subtype is a functor with a corresponding `fmap`. This functor represents the algebraic structure for a ring.

Note that this functor is "shallow", in the sense that we cannot construct nested expressions. We can create expression trees for `RingF` using a recursive data type `Ring`. In categorical terms, this `Ring` data type corresponds to the initial algebra for our functor `RingF`:

```
abstract type Ring end

struct RZero <: Ring end
struct ROne <: Ring end
struct RAdd <: Ring
```

```
    _1::Ring
    _2::Ring
end
struct RMult <: Ring
    _1::Ring
    _2::Ring
end

unfix(x::RZero) = RZeroF()
unfix(x::ROne)  = ROneF()
unfix(x::RAdd) = RAddF{Ring}(x._1,x._2)
unfix(x::RMult) = RMultF{Ring}(x._1,x._2)
```

The `Ring` data type represents expression trees, which can be evaluated using an algebra in conjunction with the catamorphism. Remember that the initial algebra has an inverse function $in^{-1}$ that transforms values of the initial algebra, which, in our code is represented by the `unfix` function. Using the `unfix`, we can define the catamorphism as:

```
cata(alg::Function, a::Ring) = alg(fmap(x->cata(alg,x), unfix(a)))
```

The power of this abstraction is that we use `Ring` to store expressions, which can then be evaluated by combining algebras over `RingF` with the catamorphism. Different algebras can be used in order to evaluate the expressions differently.

```
# Simple algebra using regular sum and multiplicat
alg1(a::RZeroF) = 0
alg1(a::ROneF)   = 1
alg1(a::RAddF{Int})   = a._1 + a._2
alg1(a::RMultF{Int})  = a._1 * a._2

# Special algebra for strings
alg2(a::RZeroF) = " hello "
alg2(a::ROneF)   = " bye "
alg2(a::RAddF{String})   = a._2 * a._1
alg2(a::RMultF{String})  = a._1 * a._2

# Expression representing ((1+1) + (1*0))
expression = RAdd(RAdd(ROne(),ROne()),RMult(ROne(),RZero()))

julia> cata(alg1, expression)
2

julia> cata(alg2, expression)
" bye  hello  bye  bye "
```

A similar implementation can be devised for co-algebra. The interested reader should check Milewski [43].

# Chapter 4

# Data Visualization with Category Theory

Despite their success, visualization grammars based on the Grammar of Graphics have limitations in terms of expressiveness [20]. While proficient for the description of common statistical graphics like scatter plots, bar plots, and histograms, they often lack the descriptive power for more complex graphics, such as composite visualizations and those involving custom marks (glyphs). The occurrence of expressiveness challenges across visualization grammars suggests that the limitation may be intrinsic to the theoretical foundation itself.

In the Grammar of Graphics [80], the data visualization pipeline consists of three main components, namely (1) graphic specification, (2) assembly and (3) display, as shown in Figure 4.1. The GoG tends to focus on specification, leaving assembly mostly as a technical aspect to be dealt by those implementing visualization grammars.

Our hypothesis is that a significant portion of the expressiveness limitations originates from a lack of integration between assembly and specification. Why would this lack of integration cause loss of expressiveness? Because without incorporating the assembly within the visualization framework, it is not clear how users can define new types of visualizations. In other words, to define a new type of visualization one must not only specify how to draw the graphical mark, but also how to assemble the graphic containing such mark. Yet, visualization grammars often hide the assembly process from users.

To integrate assembly and specification, we approach the data visualization process from a constructive perspective. From such perspective, *assembly* becomes synonymous with *diagramming*, which consists in the process of describing a scene by incrementally combining graphical primitives. Therefore, our theory can be seen as combining diagramming and data visualization in a single framework. Despite the similarities, these two subjects are often treated separately, with distinct tools and libraries. We instead treat data visualization as a subset of diagramming, where data is used in a structured manner to generate plots as diagrams containing the commonly present visual guides such as legends and axes.

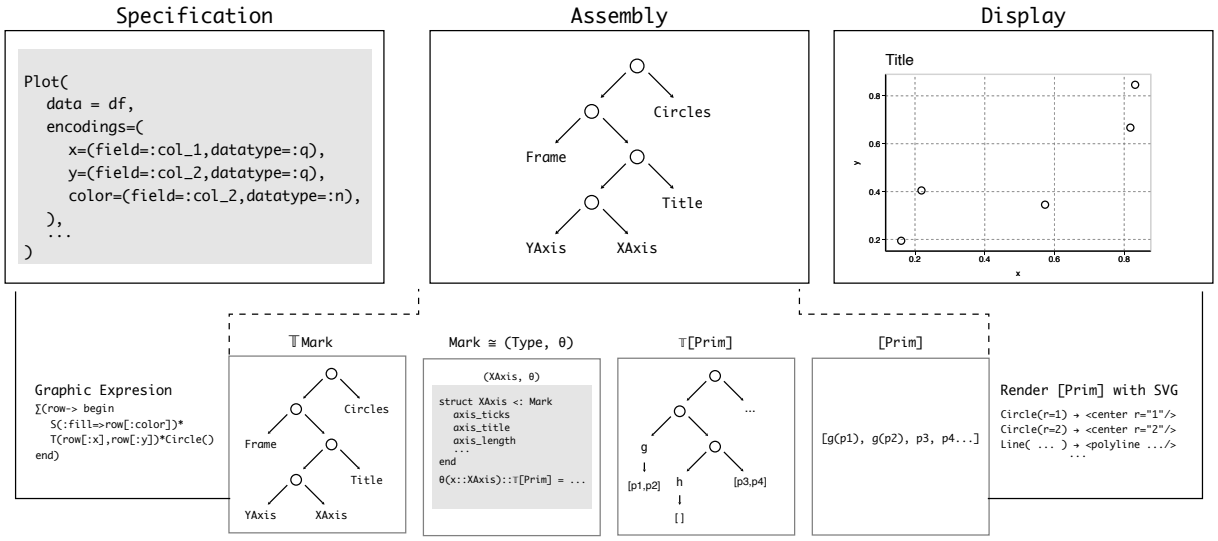



Figure 4.1: Overview of the data visualization pipeline for graphic construction. In the top, we have the tripartite division of specification, assembly and display per Wilkinson [80]. In our framework, the assembly process is divided into "diagrams", "diagram trees" (𝕋`[Prim]`), "marks" and "graphics" (𝕋 `Mark`). The specification of a data visualization is done via "graphic expressions". To display an image, each primitive in a diagram is rendered in the screen. Instead of implementing a renderer, Vizagrams uses SVG as a backend by transcribing primitives into SVG code.

The constructive approach led us to reformulate the concept of graphical marks, and to develop the concept of graphical expressions. Graphical marks work as abstractions over collections of primitives, while graphic expressions describe how data is mapped into graphics. These two concepts are at the core of how our framework integrates diagramming and data visualization, and thus, they are given special attention.

We formalize our proposal using Category Theory. The language of categories offers not only precision in our descriptions, but also establishes a novel bridge between data visualization and mathematics, potentially leading to new insights. Furthermore, by formalizing our theory within CT, we gain access to *design patterns* that can be directly translated into code implementations.

This chapter follows the structure delineated in Figure 4.1. We start from the concept of graphical primitives, which are the units that bind the assembly and the display. We then move on to diagramming, where different representations are introduced, each with a higher degree of complexity. We begin from the simplest representation, which is that of a list of primitives (`[Prim]`), and gradually build up to a representation where diagrams are modeled as trees of graphical marks. At last, we present graphic specification and how it connects to the assembly via graphic expressions.

## 4.1 Primitives

Graphical primitives are the building block of diagrams. They bridge the assembly process and the display. Primitives are simple geometric shapes which can be drawn in the screen. Although this description captures the main use of primitives, it is too broad and does not explain what exactly would be a "simple geometric shape" or how to model it in a computer. In fact, the need for a more precise definition came once we started to implement our diagramming DSL.

In order to define primitives, we start by formalizing the concepts of geometry, geometric objects and geometric primitives. After that, we move on to graphical spaces, graphical objects and lastly to graphical primitives.

### 4.1.1 Geometry and Geometric Objects

The first question to be addressed is "what is geometry"? This question has different possible answers. The one that interests us was proposed by the German mathematician Felix Klein (1849–1925) in his famous *Erlanger Programm* [19]:

> **A geometry is determined by its symmetry group.**

Following this precept, we get the following definitions:

**Definition 4.1.1 (Geometry).** A geometry is a tuple $(X, G)$ where $X$ is the space of points and $G$ is a group of transformations that act on this space.

**Definition 4.1.2 (Geometric Object).** Given a geometry $(X, G)$, a *geometric object* is a subset $S \subset X$. We define the application of $g \in G$ to $S$ as:

$$gS := \{gx \ : \ x \in S\}.$$

Moreover, we say that a geometric object $S$ has a certain *geometric property* $P$ if $P$ is invariant under all actions $g \in G$ [19].

For example, let $X$ be $\mathbb{R}^2$ and $G$ be the group of rigid transformations, and consider a set $S \subset \mathbb{R}^2$ given by:

$$S := \{(x, y) \in \mathbb{R}^2 \ : \ (x-1)^2 + (y-1)^2 = 1\}.$$

This set defines the shape of a circle. Moreover, we can say that $P :=$ *"having a radius equal to* $1$*"* is a geometric property of such object, as the radius is invariant under rigid transformations. Yet, the property $Q :=$ *"centered at (1,1)"* is not a geometric property of $S$, since we can apply a translation to $S$ which will make $Q$ false. This is illustrated in Figure 4.2.

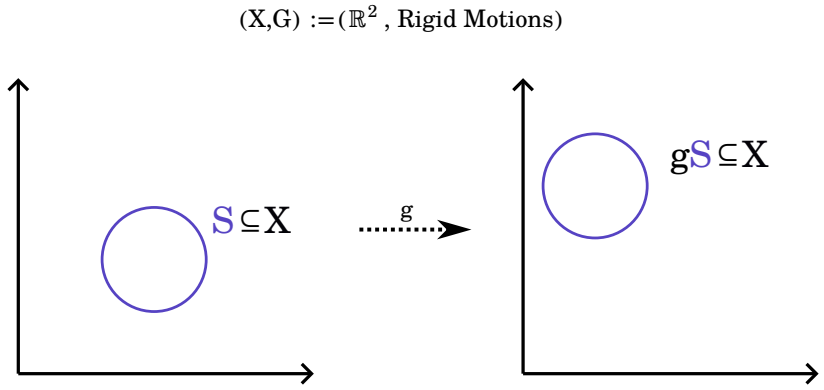



Figure 4.2: Example of the application of a translation $g$ to a geometric object $S$ consisting of a circle.

### 4.1.2 Geometric Primitives

The next question we need to address is how to represent these geometric objects. Note that explicitly listing the set of points is not feasible computationally, as a set might have an infinite number of elements. Instead, we want to find a finite parametrization that we can use to represent $S$.

**Definition 4.1.3 (Parametric Representation).** Let $(X, G)$ be a geometry. A parametric representation is a set-valued map $p : \Theta \to \mathcal{P}(X)$, where $\Theta$ is a parameter space and $\mathcal{P}(X)$ is the power-set of the geometric space $X$.

The above definition states that $p(\theta)$ returns a returns a geometric object $S \subset X$. This definition is similar to that of a primitive object in Computational Solid Geometry, where a primitive solid object is selected from a collection of shapes, and is instantiated by picking values for certain parameters that control the shape of such object [25]. Despite the similarity, we avoid saying that a parametric representation is a geometric primitive, as we will require other properties when defining geometric primitives.

An example of a parametric representation for a circle is $p : \mathbb{R}_+ \times \mathbb{R}^2 \to \mathcal{P}(\mathbb{R}^2)$, where

$$p(r, (c_1, c_2)) := \{(x, y) \in \mathbb{R}^2 \ : (x - c_1)^2 + (y - c_2)^2 = r^2\}.$$

The representation above consists in obtaining a circle by providing the radius and the center point. This way of representing circles is very natural, yet, it introduces a problem. It is not clear how a transformation $g \in G$ can be applied to such representation. Consider for example that applying a uniform scaling transformation will alter the radius of a circle, while a translation will not. If we have $p(r, (c_1, c_2))$, the only way to apply a generic transformation $g \in G$ would be by turning our representation into the actual geometric object $C \subset \mathbb{R}^2$ and

applying the transformation to each individual element. This is exactly what we wish to avoid, as applying $g$ infinite times is not computationally feasible.

The problem would be solved if our representation was covariant with respect to transformations $g \in G$, i.e. if we had a representation $q(\theta_1, ..., \theta_n)$ such that

$$g\ q(\theta_1, ..., \theta_n) = q(g\theta_1, ..., g\theta_n).$$

This leads us to the following definition:

**Definition 4.1.4 (Covariant Parametric Representation).** Let $(X, G)$ be a geometry. We say that a parametric representation $p : \Theta \to X$ is covariant in this geometry if $\Theta = X^n$ for $n \in \mathbb{N}$ and

$$gp(x_1, ..., x_n) = p(gx_1, ..., gx_n).$$

The requirement for $\Theta = X^n$ comes from the fact that transformations $g \in G$ are only well-defined to act on points of $X$.

Note that our initial representation for circles was not covariant, since the radius is not a point in $\mathbb{R}^2$. An example of covariant parametric representation would be using a pair of points collinear to the center of the circle, as illustrated in Figure 4.3.

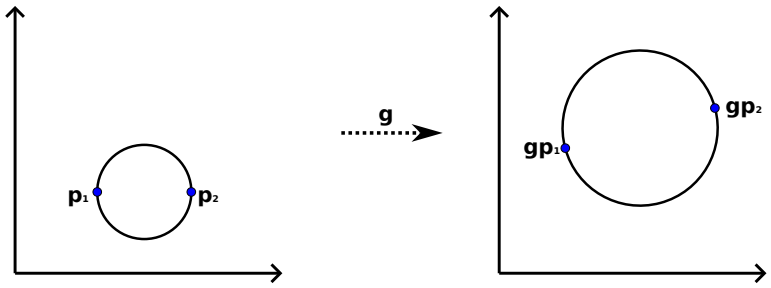



Figure 4.3: Example of covariant parametrization for circles.

Although this new representation is covariant, it is less intuitive then the previous representation. Describing a circle by providing its radius and center point is the standard in many vector graphics formats, e.g. SVG. Hence, we want to move between representations, so that we can use more common ones, while still being able to apply geometric transformations. This is where reparametrization comes in.

**Definition 4.1.5 (Reparametrization).** Let $(X, G)$ be a geometry, with parametric representations $p : \Theta \to \mathcal{P}(X)$ and $p' : \Theta' \to \mathcal{P}(X)$. We say that a pair of functions $(\phi : \Theta \to \Theta', \psi : \Theta' \to \Theta)$ defines a reparametrization between $p$ and $p'$ if:

(i) For any $\theta \in \Theta$ we have $p(\theta) = p'(\phi(\theta))$;

(ii) For any $\theta' \in \Theta'$ we have $p'(\theta') = p(\psi(\theta'))$.

Moreover, a parametric representation $p : \Theta \to \mathcal{P}(X)$ is said to be *well-specified* under geometry $(X, G)$ if there exists a covariant representation $p' : X^n \to \mathcal{P}(X)$ and a reparametrization pair $(\phi, \psi)$ between $p$ and $p'$.

When a parametric representation is well-specified, the existence of the reparametrization pair induces a way to apply transformations $g \in G$ to $p$, which is given by:

$$gp(\theta) := gp'(\phi(\theta)) = gp'(x_1, ..., x_n) = p'(gx_1, ..., gx_n) = p(\psi(gx_1, ..., gx_n)) = p(\theta').$$

Note that the reparametrization guarantees that we can move back and fourth between parametrizations, yet, this does *not* imply that $\phi$ and $\psi$ are inverses. Consider again the example of circles with parametric representation $p : \mathbb{R}_+ \times \mathbb{R}^2 \to \mathcal{P}(\mathbb{R}^2)$ where the first argument is the radius and the second argument is the center point. Although not covariant, this representation is in fact well-specified.

We can define $\phi : \mathbb{R}_+ \times \mathbb{R}^2 \to \mathbb{R}^2$ and $\psi : \mathbb{R}^2 \to \mathbb{R}_+ \times \mathbb{R}^2$ as

$$\phi(r, \mathbf{c}) := (\mathbf{c} - (r, 0),\ \mathbf{c} + (r, 0))$$

$$\psi(\mathbf{x_1}, \mathbf{x_2}) := \left( \frac{\sqrt{(\mathbf{x_1} - \mathbf{x_2})^2}}{2}, \frac{(\mathbf{x_1} + \mathbf{x_2})}{2} \right).$$

With the reparametrization functions, we can then apply transformations to such geometric objects. This process is illustrated in Figure 4.4, where we apply a transformation $g$ to a circle by first turning it into a covariant representation, applying $g$ to its parameters, and then returning to the original representation.

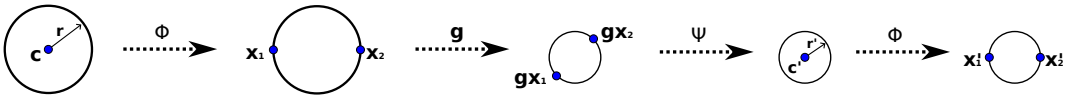



Figure 4.4: Illustration of the application of a transformation $g$ to a circle representation using radius and center.

The core idea of a geometric primitive is that we know a priori how to represent it and manipulate it. For us, being able to properly "represent" a collection of geometric objects means having a parametric representation, while being able to "manipulate" this object means having a well-specified representation. Thus, we give the following definition for geometric primitives:

**Definition 4.1.6 (Geometric Primitive).** Let $(X, G)$ be a geometry and $p : \Theta \to \mathcal{P}(X)$ be a well-specified parametric representation. A geometric primitive $P := p(\Theta)$ is the collection of geometric objects generated by the well-specified representation. A geometric object $x \in \mathcal{P}(X)$ is an instance of a geometric primitive $P$ if $x \in P$.

Note that the same geometric object can belong to different geometric primitives. For example, one can define a primitive `Line`, and another `Bézier`. Since Bézier curves can represent lines, then a line would belong to both classes.

By requiring that geometric primitives have well-specified parametric representation, this guarantees not only that we can transform primitives by transforming the parameters, but also that geometric primitives are closed under transformations $G$. This last property guarantees that for a geometry $(X, G)$, if we implement a render function for a certain geometric primitive $P$, our system will be closed under every combination of geometric object $S \in P$ and transformation $g \in G$, i.e. every $gS$ is still in $P$ thus it can be rendered. This is formalized in the following proposition:

**Proposition 4.1.7.** Let $(X, G)$ be a geometric space. Every geometric primitive $P$ is closed under transformations $g \in G$.

*Proof.* Let $P$ be a geometric primitive. This means that there exists a well-specified representation $p : \Theta \to \mathcal{P}(X)$ such that $p(\Theta) = P$. Since $p$ is well-specified, there exists a covariant representation $q : X^n \to \mathcal{P}(X)$ and a reparametrization pair $\phi : \Theta \to X^n$ and $\psi : X^n \to \Theta$.

Let $p(\theta) = S$ and $g \in G$. We have that

$$gS = gp(\theta) = g(q(\phi(\theta)) = gq(x_1, ..., x_n) = q(gx_1, ..., gx_n) = p(\psi(gx_1, ..., gx_n)) = p(\theta').$$

Since $gS = p(\theta')$, then $gS \in P$. □

### 4.1.3 Graphical Space, Graphical Objects and Graphical Primitives

In computer graphics, a primitive encodes more than geometric information, it also contains style properties such as color, opacity, stroke, an so on. When we increment a geometric primitive with these non-geometric properties, we obtain a *graphical* primitive. From now on, we use the term "primitive" alone to mean "graphical primitive". Analogously, by adding style properties to geometric objects, we get *graphical* objects.

What makes these style properties non-geometric is the fact that they are invariant under geometric transformations. By assuming that every geometric object has the same collection of possible style properties, we can define a *graphical space* $(X, G, \mathcal{S})$, where $(X, G)$ is the geometry and $\mathcal{S}$ is the style space. Similar to geometric transformations, we can pose the existence of style transformations, which alter the style properties of an object without modifying its geometry. Moreover, we call *graphical transformations* the collection of both geometric and style transformations.

At last, let us formally define graphical objects and primitives:

**Definition 4.1.8 (Graphical Object).** Let $(X, G, \mathcal{S})$ be a graphical space. A graphical object is a tuple $(p, s)$ where $p \subset X$ and $s \in \mathcal{S}$.

**Definition 4.1.9 (Graphical Primitive).** Let $(X, G, \mathcal{S})$ be a graphical space and $P$ a geometric primitive. A graphical primitive `P` is:

$$\texttt{P} := \{(p, s) \ : \ p \in P, s \in \mathcal{S}\}.$$

Given $(p, s) \in \texttt{P}$ and a geometric transformation $g$, then

$$g((p, s)) := (g(p), s).$$

Similarly, given a style transformation $a$, then

$$a((p, s)) := (p, a(s)).$$

In summary, turning a geometric primitive into a graphical primitive consists in appending the style properties to its geometry. With these two components, one can then define a render function in order to display a given graphical object.

### 4.1.4 Primitives Computationally

We have discussed the concept of primitives in a mathematical sense using mostly Set Theory. We can translate this set theoretic language into a computational implementation using categorical programming. The idea is to interpret our concepts as types, subtypes, and functions.

Overall, primitives are modeled as instances of a type `Prim`, which contains two fields: *geom* for geometric objects, and *s* for style properties. In the Julia language, this can be implemented as a *struct*:

```
struct Prim
    geom::GeometricPrimitive
    s::S
end
```

Code 4.1: Implementing `Prim` type using structs.

Note that *geom* is an instance of `GeometricPrimitive`, which is an abstract type with subtypes such as `Circle`, `Line` and `Polygon`. Similarly, style properties are modeled as instances of a type `S`, which are modeled as collections of key-value pairs (dictionary).

Type `G` is used for geometric transformations. In our framework, we limit the geometric transformations to translations (`T(x,y)`), rotations (`R(ang)`), uniform scaling (`U(s)`) and reflection (`M(p)`).

Following Yorgey [85], we model style transformations as right-biased union of style properties. Thus, values of type `S` can be used both to represent style properties and style transformations.

For example, the code snippet 4.2 illustrates how a style dictionary `s` is transformed by merging it with the style dictionary `a`. The right-bias can be seen from the fact that the output style dictionary retains the `:fill=>:red` from the original style dictionary `s`.

Graphical transformations are represented via type `H:=Tuple{G,S}`. Note that both geometric and style transformations can be made into graphical transformations simply by appending identity transformations, that is $\mathtt{g} \cong (\mathtt{g}, \mathtt{id_S})$ and $\mathtt{s} \cong (\mathtt{id_G}, \mathtt{s})$. For geometric transformations, the identity is just the identity function, while for style transformations is `S()`, the empty collection of style properties.

```
s = S(:fill=>:red)
a = S(:fill=>:blue,:stroke=>:green)
a(s) # S(:fill ⇒ :red,:stroke ⇒ :green)
```

Code 4.2: Example of applying style transformation.

In order for our geometric primitives to be properly functional, we need to be able to manipulate them, i.e. apply geometric transformations. This means that every geometric primitive `P` must implement a function `act(g::G,p::P)` that tells how to apply a transformation `g` to `p`.

Note that if `P` has a covariant representation, applying a geometric transformation consists in simply applying the transformations to each parameter. Otherwise, one can implement the pair of reparametrization functions $\phi$ and $\psi$, thus transforming `p::P` into a covariant representation, applying the geometric transformation to each parameter and then reverting back to the original representation. Consider the Code 4.3, where we show how to implement the geometric primitive for the circle.

```
# Creating the type Circle as a subtype of GeoemtricPrimitive
struct Circle <: GeometricPrimitive
    r::Real
    c::Vector
end

# Creating a type CovCircle for the covariant representation
struct CovCircle
    _1::Vector
    _2::Vector
end
ψ(p::CovCircle) = Circle(norm(p._1 - p._2) / 2, (p._1 + p._2) / 2)
ϕ(p::Circle) = CovCircle(p.c - [p.r, 0], p.c + [p.r, 0])

act(g::G, x::CovCircle) = CovCircle(g(x._1), g(x._2))
act(g::G, x::Circle) = ψ(act(g, ϕ(x)))
```

Code 4.3: Implementing the `Circle` geometric primitive.

Once a geometric primitive is specified, its graphical primitive counterpart is already available, since we only need to append the style properties to it. Graphical transformations can be applied to graphical primitives by applying the geometric transformation part to the *geom*

and the style part to $s$. Code 4.4 showcases how to define a graphical primitive and to transform it using a graphical transformation. Note that we overload the multiplication operator `*` to perform the transformation application.

```
p = Prim(Circle(r=1,c=[0,0]),S(:fill=>:red))
h = (Translate(2,1),S(:stroke=>:green))
h * p # Prim(Circle(r=1,c=[2,1]),S(:fill⇒:red, :stroke⇒:gree))
```

Code 4.4: Applying a graphical transformation to a primitive.

Besides knowing how to apply geometric transformations, primitives must also have a render function. We can avoid directly implementing this function by instead defining a function that returns a primitive in a ready to user vector graphics format, such as SVG. And, in fact, this is how we implement our diagramming DSL. For example, instead of defining a function `render(c::Circle)`, we implement `primtosvg(c::Circle)`, which returns the SVG tag corresponding to the circle primitive.

## 4.2 Diagram $\cong$ `[Prim]`

Following Yorgey [85], we define **diagrams** as ordered lists of primitives, where the order in the list defines the order for rendering (see 4.5). Such representation can be seen as the "simplest" way of representing diagrams from primitives. Indeed, from a Category Theory perspective, ordered lists are the same as free monoids [85], meaning that lists are the most "generic" monoid to be constructed over a given set. Thus, we can formally define diagrams as:

**Definition 4.2.1 (Diagrams).** Let `Prim` be the set (type) of graphical primitives and (`[Prim]`, $+$, `[ ]`) the free monoid over `Prim`. Diagrams are values of `[Prim]`, where `[ ]` is the neutral element representing an empty diagram, and $+ :$ `[Prim]` $\times$ `[Prim]` $\to$ `[Prim]` is list concatenation:

$$[p_1, ..., p_i] + [q_1, ..., q_j] = [p_1, ..., p_i, q_1, ..., q_j].$$

We have shown how graphical transformations can be applied to primitives. Since diagrams are lists of primitives, we can also apply transformations to diagrams by simply applying the transformation to each element in the list (Fig. 4.5). Note that this has a clear interpretation within Category Theory. As shown in Chapter 3, lists can be seen as functors, which, computationally, can be thought of as types endowed with an `fmap` function that *lifts* functions to the functorial type. Another way of stating this is that given any function $f :$ `Prim` $\to$ `Prim`, we can use `fmap` in order to obtain a new function from `[Prim]` to `[Prim]`. In the case of the list functor, the `fmap` does exactly the element-wise application:

```
fmap(g, [p1,p2,p3]) # [g(p1),g(p2), g(p3)]
```

Code 4.5: Example of how `fmap` works for lists.

We overload the multiplication operator so that $* : \texttt{H} \times \texttt{[Prim]} \to \texttt{[Prim]}$

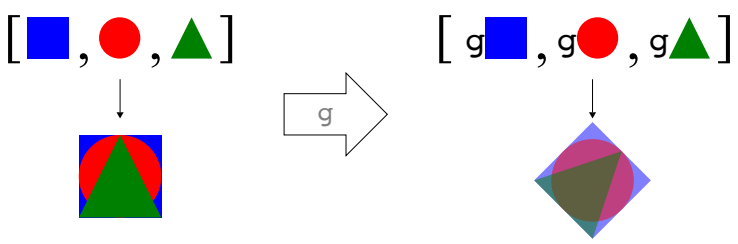



Figure 4.5: Applying graphical transformation to a diagram.

More can be said about the monoidal interpretation of diagrams, such as the fact that graphical transformations also define a monoid. Yet, we instead defer to the work of Yorgey [85], whose focus is on exploring how monoids can be useful in Functional Programming.

## 4.3 Diagram Tree $\cong \mathbb{T}$ [Prim]

Representing a diagram as an ordered list can be thought of as the *normal form* of a diagram[1]. Working directly with this representation might lead to inefficiency when constructing complex diagrams [85]. Hence, Yorgey [85] proposes the use of monoidal trees, a polymorphic data structure where the leaves contain monoidal structures.

Instead of monoidal trees, we use *free monads*, which is a concept from Category Theory commonly used in Functional Programming to build embedded domain-specific languages. Free monads formalize the idea of expression trees, i.e. binary trees where each node contains an operation and each leaf contains an operand. These trees are used in Constructive Solid Geometry (CSG) to represent geometric objects, where the nodes contain either Boolean operations (e.g., union, intersection) or geometric transformations, and the leaves contain geometric primitives [53].

Akin to CSG, the idea is to represent diagrams using trees where the nodes contain either graphical transformations or diagram composition (+), and the leaves contains lists of primitives. While representing a diagram directly as a list of primitives requires eagerly evaluating graphical transformations and composition, expression trees provide a lazy data structure by storing these operations inside the tree structure.

As stated, this tree structure can be given a formal interpretation within Category Theory via free monads. While a free monoid is defined over a given set (e.g. `Prim`), a free monad is defined over an endofunctor $F$. This endofunctor $F$ is used to represent the node operations of the tree, which in our case is composition and graphical transformations.

[1] *Normal form* is a term from lambda calculus that refers to an expression that cannot be further reduced via $\beta$-reduction.

```
abstract type F{a} end
struct Comp{a} <: F{a}
    _1::a
    _2::a
end
struct Act{H, a} <: F{a}
    _1::H
    _2::a
end
fmap(f::Function, x::Comp) = Comp(f(x._1),f(x._2))
fmap(f::Function, x::Act) =  Act(x._1,f(x._2))
```

Code 4.6: Implementing functor `F` representing composition and graphical transformation.

In Code 4.6, it is shown how the `F` functor can be implemented in the Julia language as a parametric type. Note that a value of type `F{[Prim]}` is either a composition of two lists of primitives, i.e. `Comp([Prim],[Prim])`, or the application of a graphical transformation to a list of primitives, i.e. `Act(H,[Prim])`. Note that `F[Prim]` is not yet the actual tree, since the values are *shallow*, in other words, values of type `F[Prim]` represent just a single node. For example, trying to compose a list of primitives with the node containing a graphical transformation, e.g. `Comp([p1,p2], Act(g,[p3,p4]))`, is not well defined (does not type-check), since `Comp` must have two values of the same types.

The free monad is the structure that enables us to compose the nodes (values of type `F`) into tree expressions. Its implementation is shown in the code snippet 4.7. We define the free monad as a parametric type `𝕋` having three possible subtypes: `Pure`, `FreeComp` and `FreeAct`. The `fmap` function is implemented in order to make `𝕋` into a functor. The `free` and `unfree` functions are used to take values from `F` to `𝕋`, and from `𝕋` to `F`, respectively.

The monadic structure of `𝕋` is given by `η` and `μ`. We can think of `η` as the operation that takes a value of a type `A` and puts it into a leaf, which is represented by the `Pure` constructor. The `μ` function takes a tree of trees and flattens into a single tree, similar to how a list of list can be flattened into a single list.

```
abstract type 𝕋{a} end
struct Pure{a} <: 𝕋{a}
    _1::a
end
struct FreeComp{a} <: 𝕋{a}
    _1::𝕋{a}
    _2::𝕋{a}
end
struct FreeAct{H, a} <: 𝕋{a}
    _1::H
    _2::𝕋{a}
end

fmap(f::Function, x::Pure) = Pure(f(x._1))
fmap(f::Function, x::𝕋) = free(fmap(y -> fmap(f, y), unfree(x)))
```

```
η(x) = Pure(x)
μ(x::Pure{<:𝕋}) = x._1
μ(x::𝕋{<:𝕋}) = free(fmap(μ, unfree(x)))
```

Code 4.7: Implementing the free monad data structure.

By applying `𝕋` over `[Prim]`, we obtain diagram trees, i.e. a tree expression with leaves containing lists of primitives and nodes containing either composition or graphical transformations.

Our claim is that a value of `𝕋[Prim]` is a valid representation for a diagram. In order to show that this is the case, we need to be able to turn diagram trees into single diagrams, that is, into single lists of primitives. This can be done by devising an evaluation function that flattens the tree into a single list of primitives, i.e. `eval` : $\mathbb{T}$`[Prim]` $\rightarrow$ `[Prim]`.

One of the advantages of interpreting our data structure as a free monad is that CT already provides guidance on how to implement this evaluation function. To evaluate a free monad, we only need to apply the catamorphism functional pattern to an specific $F$-algebra function.

The catamorphism implementation (see Code 4.8) is valid for any free monad independent of the underlying endofunctor.

```
cata(algebra::Function, x::Pure) = x._1
cata(algebra::Function, x::𝕋) = algebra(fmap(y -> cata(algebra, y), unfree(x)))
```

Code 4.8: Catamorphism for the free monad.

The $F$-algebra function `alg(d::F{[Prim]})` simply applies the node operation to the leaf, as shown in Code 4.9. With the catamorphism and the $F$-algebra, we then flatten the diagram trees using `eval = cata ∘ alg`. Figure 4.6 illustrates how a diagram tree

This is illustrated in Figure 4.6, where the tree on the left when evaluated produces the list of primitives on the right.

```
alg(x::Comp{Vector{Prim}}) = x._1 + x._2
alg(x::Act{Vector{Prim}})::Vector{Prim} = fmap(y -> act(x._1, y), x._2)
```

Code 4.9: Algebra for functor `F`.

Just like with the list representation, diagrams as trees can also be composed and transformed. In this case, composition means joining two trees by creating a new node that connects both sub-trees. For transformations, we just append the transformation as a node above the original tree. Both of these operations are illustrated in Figure 4.7.

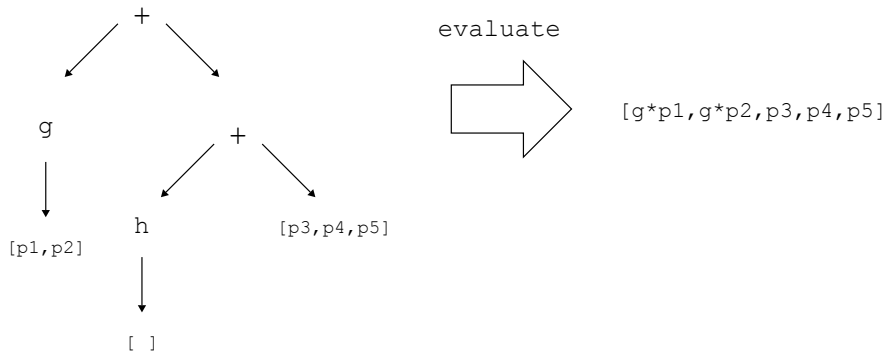



Figure 4.6: Diagram tree on the left, which when evaluated produces the list of primitives on the right.

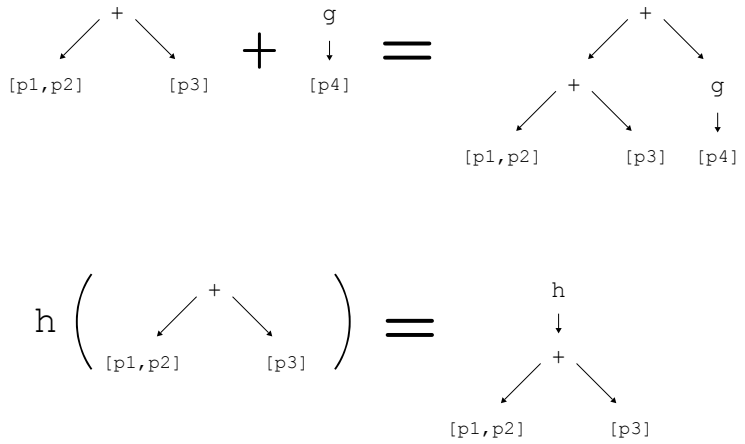



Figure 4.7: On the top, two diagram trees are composed together. On the bottom, a graphical transformation `h` is applied to a diagram tree.

## 4.4 Graphical Marks

We started from the definition of diagrams as lists of primitives, and then showed how free monads over lists of primitives can also be used to represent diagrams. Our final and most abstract representation for a diagram is that of a tree of graphical marks. Thus, to make this representation precise, we must first formalize the concept of graphical marks.

In this section, we revisit the current definition of graphical marks and then introduce a novel definition for marks that extends on the current one. Additionally, we illustrate the incremental construction of increasingly complex marks based on preceding ones.

### 4.4.1 Current Definition

According to Bertin [4], data visualization consists of the process of representing set of data as an image using a graphic sign-system. This sign-system must be *monosemic*, which means that each sign possesses a single meaning. Thus, ensuring that each graphic prompts an unambiguous interpretation of the underlying data. Moreover, the graphic sign-system is composed of graphical marks, which are the "*graphical unit*" used to represent the data. In the Grammar of Graphics [80], the term "*geometric graph*" is employed in place of "*mark*".

Munzner [45] defines graphical marks as primitive geometric objects that depict items or links. Similarly, Vega's documentation [57] states that marks visually encode data through geometric primitives. Marks are then said to be able to vary their *visual variables* (position, size, value, texture, color, orientation, and shape), also referred as *visual channels* [45]. Notably, this conception of mark is basically indistinguishable from that of a graphical primitive.

A related concept is that of a glyph or custom mark, which are used to denote collections of geometric shapes [8]. Our goal is to merge these concepts by keeping marks as the basic graphic unit, while expanding their definition to encompass glyphs. Instead of restricting marks to primitive geometries, we allow them to be any "properly defined" glyph - a term which will be made precise further in the text. In this way, graphical marks function as abstract representations for collections of primitives, maintaining their role as graphical units in data visualization.

### 4.4.2 A New Definition

The main idea we wish to capture with our novel definition is that a graphical mark must be an abstraction over a collection of primitives with parameters to allow us to manipulate its visual properties. Moreover, from our discussion in 1.2, we want marks to cover primitives, glyphs and even plots. Why? Because if marks are to be the graphical units in our system, and if we want to be able to manipulate plots using the same diagramming operations we use in graphical marks, then plots must somewhat be interpreted also as marks. Based on this, we propose the following definition:

**Definition 4.4.1 (Graphical Mark).** A **graphical mark** is a tuple $(\mathtt{A}, \theta_{\mathtt{A}})$, where $\mathtt{A}$ is a type and $\theta_{\mathtt{A}}$ is a function $\theta_{\mathtt{A}} : \mathtt{A} \to \mathbb{T}\mathtt{[Prim]}$.

This definition entails that a mark is a pair consisting of a type and a recipe to turn this type into a diagram. Note that we could have used `[Prim]` as the codomain of $\theta_{\mathtt{A}}$ instead of $\mathbb{T}\mathtt{[Prim]}$, since both are valid representations for diagrams.

Now, let us show how this new definition fits the necessary requirements previously stated. Note that every primitive can be interpreted as a mark. To do so, we only need to define $\theta_{\mathtt{Prim}} : \mathtt{Prim} \to \mathbb{T}\mathtt{[Prim]}$ as:

$$\theta_{\mathtt{Prim}}(\mathtt{p}) = \eta_{\mathtt{[Prim]}}(\mathtt{[p]}).$$

Analogously, `[Prim]` can be interpreted as a mark $(\texttt{[Prim]}, \eta_{\texttt{[Prim]}})$ and $\mathbb{T}\texttt{[Prim]}$ as a mark $(\mathbb{T}\texttt{[Prim]}, \texttt{id})$, where `id` is the identity function. This suggests primitives and even diagrams can be interpreted as marks.

This definition also covers custom marks (glyphs), we only need to define the corresponding type together with the $\theta$ function that gives the diagram representing the collection of primitives for that glyph. This same mechanism used to create custom marks can also be used to create marks representing plotting functions. For example, one can define a mark `Histogram`, where the input field is a set of data and the $\theta$ function returns a diagram of rectangles stacked together. Therefore, plots can also be interpreted as marks.

In a programming language with subtypes, such as Julia, marks can be represented as subtypes of an abstract type (or supertype) called `Mark`. Each specific mark subtype defines a method of a function, denoted here as $\theta : \texttt{Mark} \to \mathbb{T}\texttt{[Prim]}$, to generate its corresponding collection of primitives. This approach relies on ad-hoc polymorphism, which in Julia is achieved through multiple dispatch, allowing each mark subtype to implement its own distinct version of the $\theta$ method.

For example, consider that we want to create a mark `Petal` that can have different heights, widths and colors. We can specify such mark by first creating a struct with the chosen attributes, together with the $\theta$ function that describes how to represent such mark as a diagram. This implementation is shown in Code 4.10, where the $\theta$ function for `Petal` returns a leaf containing a list with a single Bézier polygon. Figure 4.8 illustrates an instance of this `Petal` mark.

In contrast, Functional Programming languages without subtypes, such as Haskell, can use type classes to represent marks. Here, a mark is an instance of the `Mark` type class, with the constraint given by the presence of the $\theta$ method. Each mark type then provides its own implementation of $\theta$, fulfilling the type class requirement.

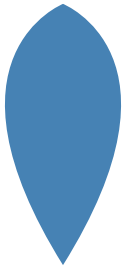


Figure 4.8: Example of `Petal` mark implemented in Code 4.10.

```
struct Petal <: Mark
    height::Real
    width::Real
    color::String
end
```

```
function θ(p::Petal)
    (; height, width, color) = p

    geom = CBezierPolygon(
        [[0, 0], [0, height], [0, 0]],
        [[-width, height*0.8],[0, height],[0, height],[width, height*0.8],[0, 0],[0, 0]])

    s = S(:fill => color, :strokeWidth => 0)
    return n([Prim(geom, s)])
end
```

Code 4.10: Creating a `Petal` mark.

## 4.5 Mark Tree $\cong \mathbb{T}$ `Mark`

With the concept of graphical marks firmly defined, we arrive at our final and most abstract representation for diagram, that of mark trees. A **mark tree** is a value of $\mathbb{T}$`Mark`, that is, a tree expression where the leaves are marks. Since every mark has a corresponding $\theta$ function, we can therefore transform values of $\mathbb{T}$`Mark` into values of $\mathbb{T}$`[Prim]`, which can then be evaluated into values of `[Prim]`.

Using mark trees to represent diagrams allows us to form diagrams by directly combining marks, without having to first turn them into lists of primitives. For example, a diagram for a person could be constructed by composing a mark for head and a mark for body translated by a certain value (Figure 4.9). By using marks instead of lists of primitives, mark trees become a more condensed, albeit more abstract, way of representing diagrams. Thus, $\mathbb{T}$`Mark` can be seen as adding one more layer of abstraction and laziness over the other representations.

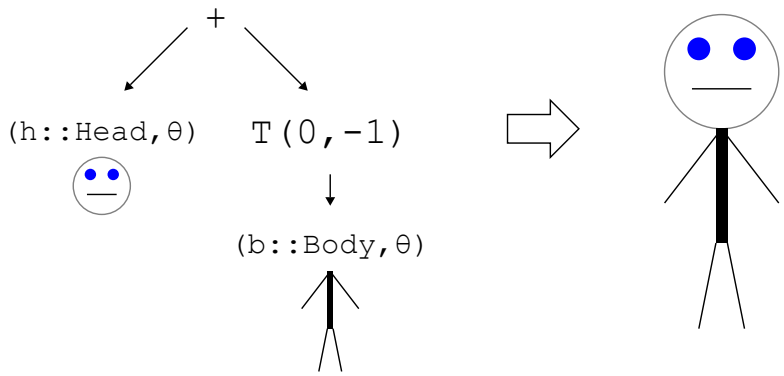



Figure 4.9: Example of mark tree representation. The tree representation is shown on the left, and the respective diagram is shown on the right.

This way of representing diagrams is somewhat convoluted, since it uses marks, which themselves encode diagrams. However, this introspective approach enables the use of diagrams to define new marks and marks to define new diagrams. Despite its power, this introspection carries the risk of circular definitions, which must be carefully managed.

## 4.6 Building Marks from other Marks

Our construction suggests that if we want to define a mark, we must pick a type and a function that turns this type into a $\mathbb{T}$`[Prim]`. This works fine for simple marks where one can easily describe it by combining primitives, but as marks become more complex, this can become too cumbersome. Alternatively, a better strategy would be to use predefined marks to construct new ones, thus increasing the complexity incrementally. In other words, we need to find a way to define a graphical mark $(\texttt{A}, \theta_{\texttt{A}})$ where instead of implementing $\theta_{\texttt{A}} : \texttt{A} \to \mathbb{T}\texttt{[Prim]}$, we can implement a function $\zeta_{\texttt{A}} : \texttt{A} \to \mathbb{T}\texttt{Mark}$ and derive $\theta_{\texttt{A}}$ "for free".

To do so, we must prove two properties are valid according to our definition of graphical marks: (1) that we can derive a mark $(\mathbb{T}\texttt{A}, \theta_{\mathbb{T}\texttt{A}})$ from a previously defined mark $(\texttt{A}, \theta_{\texttt{A}})$; (2) that given two marks $(\texttt{A}, \theta_{\texttt{A}})$ and $(\texttt{B}, \theta_{\texttt{B}})$, we can derive a mark $(\texttt{A + B}, \theta_{\texttt{A+B}})$. The way these two properties matter will be made clear in a following use case.

For the first property, recall that our tree constructor is a free monad $(\mathbb{T}, \mu, \nu)$. Given a mark $(\texttt{A}, \theta_{\texttt{A}})$, we can construct trees $\mathbb{T}\texttt{\{A\}}$, where the leaves are values of type `A`. We then claim that $(\mathbb{T}\texttt{\{A\}}, \mu \circ \mathbb{T}\theta_{\texttt{A}})$ is a valid mark, where:

$$\mathbb{T}\theta_{\texttt{A}}(t :: \mathbb{T}\texttt{\{A\}}) = \texttt{fmap}(\theta_{\texttt{A}}, t).$$

Note that $\mathbb{T}\theta_{\texttt{A}}$ takes a value of type $\mathbb{T}\texttt{\{A\}}$ and returns a value of type $\mathbb{T}\texttt{\{}\mathbb{T}\texttt{[Prim]\}}$, i.e. a tree of diagram trees. We can then flatten it into $\mathbb{T}$`[Prim]` via the monadic operator $\mu$, thus proving that $(\mathbb{T}\texttt{\{A\}}, \mu \circ \mathbb{T}\theta_{\texttt{A}})$ is indeed a mark.

For the second property, we use what is known in Functional Programming as sum types. Given two types `A` and `B`, a sum type $\texttt{A} + \texttt{B}$ contains values that are either of type `A` or type `B` (akin to disjoint unions in sets). For example, a list of type `[Int + String]` is a list that can have both integers and strings. Similarly, $\mathbb{T}\texttt{\{A + B\}}$ are trees where leafs can have values of type `A` and type `B`. Sum types can also be used to construct new marks. For two marks $(\texttt{A}, \theta_{\texttt{A}})$ and $(\texttt{B}, \theta_{\texttt{B}})$, we can define a sum mark $(\texttt{A + B}, \theta_{\texttt{A + B}})$, where $\theta_{\texttt{A + B}}$ applies $\theta_A$ if the value is of type `A`, and applies $\theta_B$ if the value is of type `B`. Thus, we proved that sum types also define marks, or, in set theoretic terms, that our collection of marks is closed under disjoint unions.

We can now demonstrate how to construct new marks from previously defined ones. Suppose that we want to define a new mark `Flower` using the already implemented `Petal`. Instead of specifying a $\theta_{\texttt{Flower}} : \texttt{Flower} \to \mathbb{T}\texttt{[Prim]}$, we can specify a function $\zeta :: \texttt{Flower} \to \mathbb{T}\texttt{\{Petal\}}$, which is done in Code 4.11. The idea is that we want to draw a flower by drawing multiple petals, allowing for different colors and sizes, as illustrated in Figure 4.10.

```
struct Flower <: Mark
    heights::Vector{Real}
    widths::Vector{Real}
    colors::Vector{String}
end

function ζ(flower::Flower)
    (; heights, widths, colors) = flower
    numberpetals = length(heights)
    angles = range(0, 2π, numberpetals + 1)[begin:(end - 1)]
    return reduce(
        +,
        map(x -> R(-x[1]) * Petal(x[2], x[3], x[4]), zip(angles, heights, widths, colors)),
    )
end
```

Code 4.11: Creating a `Flower` mark.

Figure 4.10: Instance of `Flower` mark.

Using the fact that mark trees are also marks, we can obtain $\theta_{\texttt{Flower}}$ by:

$$\theta_{\texttt{Flower}} = \theta_{\mathbb{T}\texttt{\{Petal\}}} \circ \zeta_{\texttt{Flower}} = \mu \circ \mathbb{T}\theta_{\texttt{Petal}} \circ \zeta_{\texttt{Flower}}.$$

The equation above states that to turn a flower mark into a diagram, first we apply $\zeta_{\texttt{Flower}}$ to obtain a $\mathbb{T}\texttt{Petal}$. We then apply $\theta_{\texttt{Petal}}$ to each leaf, which is done by $\mathbb{T}\theta_{\texttt{Petal}}$. This returns a value of $\mathbb{T}\mathbb{T}\texttt{[Prim]}$, which is flattened into a single tree via the monadic operator $\mu$.

Next, suppose we want to create an even more complex mark named `Plant`, that is constructed by combining a flower with a stem. Using our mark `Flower` and assuming the existence of a mark `Stem`, we can create `Plant` again using a $\zeta$ function, implemented in Code 4.12, and illustrated in Figure 4.11.

Using the fact that sum types define marks, and that mark trees are also marks, we compute $\theta_{\texttt{Plant}}$ by:

$$\theta_{\texttt{Plant}} = \theta_{\mathbb{T}\texttt{\{Flower+Stem\}}} \circ \zeta_{\texttt{Plant}} = \mu \circ \mathbb{T}\theta_{\texttt{Flower+Stem}} \circ \zeta_{\texttt{Plant}}.$$

We can generalize this construction rule by defining $\zeta : \texttt{Mark} \to \mathbb{T}\texttt{\{Mark\}}$ and $\theta : \texttt{Mark} \to \mathbb{T}\texttt{[Prim]}$ as a polymorphic functions over a supertype `Mark`. Thus, we can specify any mark `A` by passing $\zeta$, and inferring $\theta$ as:

$$\zeta(\texttt{a::A}) = \ldots \implies \theta(\texttt{a::A}) = \theta \circ \zeta(\texttt{a::A})$$

When specifying marks from $\zeta$ we must be careful not to create circular (non-terminating) constructions. For example, if we define $\zeta_A : A \to B$, $\zeta_B : B \to C$ and $\zeta_C : C \to A$, then when we try to compute $\theta_A$, we will fall into a non-terminating loop. Hence the prescription is that to induce $\theta_A$ from $\zeta_A : A \to B$, one must ensure that $B$ is already a "well-defined" mark, i.e. a mark with a terminating $\theta$ function.


Figure 4.11: Instance of `Plant` mark.

```
struct Plant <: Mark
    flower_heights::Vector{Real}
    flower_widths::Vector{Real}
    flower_colors::Vector{String}
    stem_height::Real
    stem_text::String
end

function ζ(plant::Plant)
    (;flower_heights, flower_widths, flower_colors, stem_height, stem_text) = plant
    flower = Flower(flower_heights, flower_widths, flower_colors)
    stem = Stem(stem_height, stem_text)
    d = stem + T(0,stem_height) * flower
end
```

Code 4.12: Creating a `Plant` mark.

### 4.6.1 The Category of Marks

We have shown that it is possible to build marks from previously defined marks via the $\zeta$ function, which can be used to derive the appropriate $\theta$ function. This derivation of $\theta$ from $\zeta$ has a category-theoretic origin, which can be used to rigorously formalize our construction.

First, recall that our tree constructor $\mathbb{T}$ is modeled as a free monad $(\mathbb{T}, \nu, \mu)$ over the category

**Set**. Thus, we can define the Kleisli Category $\mathbf{Set}_{\mathbb{T}}$[2], where $\mathrm{Ob}_{\mathbf{Set}_{\mathbb{T}}} = \mathrm{Ob}_{\mathbf{Set}}$, $\mathrm{Mor}_{\mathbf{Set}_{\mathbb{T}}}(A, B) = \mathrm{Mor}_{\mathbf{Set}}(A, \mathbb{T}B)$, and composition is given by:

$$g \circ_{\mathbb{T}} f := \mu_C \circ \mathbb{T}g \circ f,$$

with $f : A \to \mathbb{T}B$ and $g : B \to \mathbb{T}C$.

From $\mathbf{Set}_{\mathbb{T}}$, we construct the slice category $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$, in which an object is a pair $(A, \theta_A)$ with $\theta_A \in \mathrm{Mor}_{\mathbf{Set}_{\mathbb{T}}}(A, \texttt{[Prim]})$, and a morphism between $(A, \theta_A)$ and $(B, \theta_B)$ is equivalent to a function $\zeta_A \in \mathrm{Mor}_{\mathbf{Set}_{\mathbb{T}}}(A, B)$, such that:

$$\theta_A = \theta_B \circ_{\mathbb{T}} \zeta_A = \mu_B \circ \mathbb{T}\theta_B \circ \zeta_A.$$

Note that this formula is exactly the one used in section 4.6 when deriving $\theta_{\texttt{Plant}}$. Thus, the $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$ category very closely resembles how we are modelling marks, yet, it still misses some important points.

In $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$, every set (type) is an object, which is not true for marks, since only subtypes of `Mark` are in fact valid. Moreover, for any mark type `A`, there must exist only a single function (morphism) $\theta_{\texttt{A}}$ taking `A` to $\mathbb{T}\texttt{[Prim]}$, otherwise, this would imply that the same mark could be drawn differently, which is again not true in $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$. In summary, we need smaller category.

To address these constraints, we pose the existence of a **category of marks**, denoted by $\mathcal{M}$. This category is constructed as a *thin* subcategory of $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$, where only subtypes of `Mark` are objects. Furthermore, the fact that $\mathcal{M}$ is thin implies that at most one morphism exists between any two objects. This captures the principle that each mark type must be associated with a unique morphism $\theta$. Another consequence of this definition is that the $\zeta$ functions are also unique. This uniqueness of morphisms is what ensures that our system of graphical marks is consistent, in the sense that a mark cannot be drawn in more than one distinct way.

## 4.7 Plot Specification

A data visualization specification is a description, or blueprint, for how to draw a visualization. The process can be divided into two components: (1) specifying guides and encodings; (2) specifying the graphic.

### 4.7.1 Guides and Encodings

Guides are the visual aids which enable users to decode information regarding the visualization. They are comprised of graphical elements such axes, legends and titles. Encodings are

[2]Go to chapter 2 for a more in depth explanation of the Kleisli Category.

dictionaries which describe how to scale data attributes, with each encoding having a channel, a field and a scale. The encoding applies the scale to the specified field variable and assigns the result to a designated channel variable. Consequently, we can think of the encoding as constructing a scaled dataset where a column, identified by the value in the field, is transformed by the specified scale function, resulting in a new column named according to the channel (Figure 4.12).

| Topic | Value | Country |
|---|---|---|
| JE | 0.42 | Australia |
| CG | 0.88 | Australia |
| ... | ... | ... |
| SW | 0.53 | Costa Rica |
| WL | 0.63 | Costa Rica |

encoding →

| x | y | color |
|---|---|---|
| 22.73 | 84.0 | #E69537 |
| 22.73 | 176.0 | #6EB38A |
| ... | ... | ... |
| 185.7 | 106.0 | #447CCD |
| 185.7 | 126.0 | #4039C4 |

Figure 4.12: Example of dataset encoding, where the original dataset is shown in the left, and the scaled data is shown in the right. Columns `x, y` and `color` correspond to the channels in the `encodings` component.

In visualization grammars the *channel* usually represents visual variables such as *position* (x,y), *color*, *size* and *shape*. In our framework, the channels are not limited to visual variables. Note that to avoid long specifications, visualization grammars try to infer the scales in an encoding. This is done via smart defaults that use information such as the semantic meaning of the channel variable, the data type for a given field, and so on.

### 4.7.2 Graphic Expressions

The second component of the data visualization specification is the graphic specification. We can think of a graphic as a diagram generated by a collection of data. The graphics that are useful for data visualization are those that produce diagrams in a "structured" manner, such that those viewing the graphic can recover information regarding the data that produced it. This suggests that we want to create graphics by "orderly" processing the data, and "orderly" drawing the diagram. For example, to generate a scatter plot we can iterate over the rows in a dataset, drawing a circle for each row while mapping the values in dataset to attributes in the circle. This process is captured in the following equation:

$$\sum_{row \in D} \text{Circle(r=row[:size],center=[row[:x],row[:y]])},$$

where $D$ represents the dataset, `:size`, `:x` and `:y` are column names, and the summation operator $\sum$ is "adding" (+) circles, which in our framework means that it is composing diagrams.

We can generalize this equation by substituting the graphical primitive `Circle` for a generic `m::Mark`. However, not every mark has a "center" field or "radius". Thus, we can instead

use the translation transformation, and do the same for the radius, by passing an uniform scaling. At last, we can also add rotation, resulting in the following function:

$$\sum_{row \in D} \text{T(row[:x],row[:y])R(row[:angle])U(row[:size])*m}$$

We can think of this formula as the expression representing a generic scatter plot. By changing the graphical mark `m`, one can produce a plethora of visualizations, all of which would still fall within the idea of a scatter plot.

One might wonder whether every "relevant" graphic in data visualization is just a scatter plot, but with distinct marks. The answer is no. In fact, we cannot use the scatter plot equation to draw a line plot. In a line plot, we need to break the lines according to the color attribute. Moreover, while each circle requires only a single x and y value, the line requires a vector of x and y as input. With these considerations, we might write the equation for a line plot as follows:

$$\sum_{rows \in D_{color}} \text{Line(rows[:x],rows[:y],color=rows[1,:color])}$$

The equation above has significant differences when compared to the equation for the scatter plot. First, we are not iterating over each row, instead, we are grouping the data by values of column *color*. Each *rows* is not a single row, but a subset of $D$ where all the values of column *color* are equal. Also, note that `Line` receives a vector of values corresponding to column $x$, another vector for column $y$, and a single value for *color*. Since each row in *rows* has the same *color*, we can just pick the color from the first row in *rows*.

In both examples, we have iterated through the data, created several graphic elements, and then summed them together to obtain a single graphic. Summing the elements together is not our only option. Depending on the situation, we might instead wish to combine the graphics by stacking them on top of each other. This is the case, for example, in stacked bar plots.

All these considerations lead us to formulate following concept graphic expressions:

**Definition 4.7.1 (Graphic Expression).** A graphic expression is a triple $(expr, alg, coalg)$, where:

$$\sum_{coalg}^{alg} expr \cong \begin{bmatrix} expr : D \to \mathbb{T}\texttt{\{Mark\}} \\ alg : [\mathbb{T}\texttt{\{Mark\}}] \to \mathbb{T}\texttt{\{Mark\}} \\ coalg : D \to \texttt{[}D\texttt{]} \end{bmatrix}$$

The *expr* is the expression inside the summation that maps data values to attributes in a diagram. The *alg* is the function that takes a list of diagrams and combines them into a single diagram. The *coalg* is the function that takes the dataset $D$ and iterates over it producing a list of subsets of $D$. In categorical terms, *alg* is an $F$-algebra and *coalg* is an $F$-coalgebra where $F$ is the list functor `[ ]` : **Set** $\to$ **Set**.

Graphic expressions are similar to the recursion structure known in Functional Programming as hylomorphism [42], with the difference that it includes *expr* as a separate mapping function

as an additional parameter. Thus, they can be computed by the following formula:

$$\sum_{coalg}^{alg} expr = alg \circ \texttt{fmap}(expr) \circ coalg$$

In summary, graphic expressions are a formal description of the process of breaking down a dataset into chunks, using each chunk to produce a diagram, and finally combining each of these diagrams into a single diagram; a pattern also known as split-apply-combine [76]. In this sense, a graphic is a diagram that was produced by applying a graphic expression to a dataset. Figure 4.13 presents an schematic drawing of how the graphic expressions work.

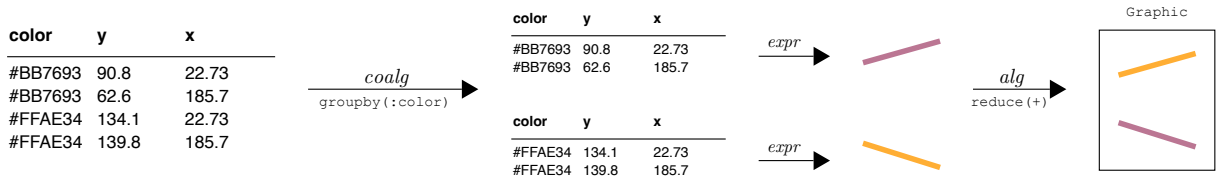



Figure 4.13: Example of a graphic expression that produces lines. The coalgebra used is a `groupby` over the `:color` column. The expression function takes the data subset and produces lines. The algebra combines the lines using the diagram composition operator (+). The final graphic consists in two lines of different colors.

Note that a graphic expression is itself a function from a dataset to a diagram, that is $\sum_{coalg}^{alg} expr : D \to \mathbb{T}\{\texttt{Mark}\}$. Thus, we can use it as an $expr$ for another graphic expression:

$$\sum_{coalg_2}^{alg_2} \sum_{coalg_1}^{alg_1} expr = alg_2 \circ \texttt{fmap} \left( \sum_{coalg_1}^{alg_1} expr \right) \circ coalg_2.$$

By coupling summands, we model the process of sequentially splitting the data and then sequentially combining it, akin to a nesting of for-loops. This kind of coupling is useful to express graphics such as vertical stacked-bar plots, where the data is split both vertically inside the bar and horizontally across the axis.

### 4.7.3 Combining Guides, Encodings and Graphic Expressions

The complete data visualization specification is composed of a dataset, guides, encodings and a graphic expression. These four components are combined into a mark `Plot`, where the $\zeta$ function first draws the guide elements (e.g. axes, legends), then, scales the dataset using the encodings and applies this scaled dataset to the graphic expression, thus drawing the graphic. The process is illustrated in Figure 4.14.

Note that our main contributions when it comes to data visualization specification is twofold. First, the use of graphic expressions as a way to specify graphics using diagramming tools. Second, by treating a plot as a mark, and by making our diagramming description introspective, we can then manipulate and combine plots as any other diagram.

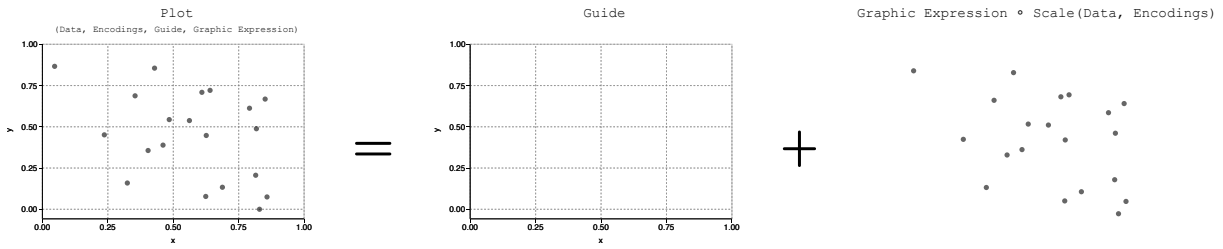



Figure 4.14: Overview of the process to draw a plot from a specification.

## 4.8 Theory Overview

Our goal in this chapter was to formalize the data visualization process from a constructive perspective integrating the assembly and specification. We used categorical programming as a means of tying the mathematical formalization with the computational implementation. In this regard, sets were equated with types, subsets were equated with subtypes, and mathematical functions were equated with programming functions.

We started by formalizing graphical primitives, which serve as the bridge between assembly and display. Graphical primitives are the basic geometric shapes used to draw figures. They are comprised of a geometric shape and style properties. Primitives can be manipulated via geometric transformations and style transformations, which together are called graphical transformations. Computationally, primitives, geometric transformations and style properties are modeled using types `Prim`, `G` and `S`, respectively.

Diagrams are collections of primitives. There are different ways to represent these collections. The most basic representation is using an ordered list of primitives, which in Category Theory is equivalent to a free monoid over the set of primitives. A second representation is that of a tree of lists of primitives, i.e. values of type $\mathbb{T}$`[Prim]`. The tree constructor $\mathbb{T}$ is formally defined as a free monad over an endofunctor `F`, which encodes the composition and graphical transformation operators. This means that $\mathbb{T}$ denotes an expression tree where the nodes consists in either a composition of two leafs or the application of a graphical transformation to a leaf. Our final representation for a diagram is as a tree of graphical marks, i.e. values of type $\mathbb{T}$`Mark`. In this representation, graphical marks become the basic unit for expressing diagrams. In order to make such representation precise, we then took a detour into formalizing the concept of graphical marks.

While a primitive type is a parametrization of a collection of basic geometric shapes, a

mark type is a parametrization of a collection of diagrams (lists of primitives). Formally, a graphical mark is defined as tuple $(A, \theta_A)$ where $A$ is the type representing the abstraction and $\theta_A : A \to \mathbb{T}\texttt{[Prim]}$ is function decoding how to turn a value of type $A$ into a diagram. This new definition generalizes the concept of mark to cover both primitives and glyphs.

Instead of defining a mark $A$ by specifying $\theta_A$, one can instead specify $\zeta_A : A \to \mathbb{T}\texttt{Mark}$ and use it to infer $\theta_A$. The use of $\zeta$ allows one to create graphical marks from previous ones, with the downside that it opens the possibility of circular definitions. The way graphical marks interact is made precise again through the use of Category Theory. Graphical marks are objects of a category $\mathcal{M}$, which is called *category of marks* and consists in a thin subcategory of $\mathbf{Set}_{\mathbb{T}}/\texttt{[Prim]}$.

With diagrams (assembly) formalized, the next step is to connect it with specification. A data visualization is fully specified by combining a dataset, guides, encodings and the graphic expression. Guides establish the visual aids commonly present in a plot (e.g. axes, legends), while the encoding specifies which portions of the dataset are to be used and how to scale them. Lastly, graphic expressions determine how to map the data into diagrams in order to produce the different graphics (e.g. scatter plots, line plots, bar plots, histograms). Formally, graphic expressions are comprised of a *List*-coalgebra, $expr : \texttt{D} \to \mathbb{T}\texttt{Mark}$ function and *List*-coalgebra, which are used to split the data, turn the bits of data into diagrams, and then combine the diagrams into the final graphic.

Computationally, a data visualization specification is modeled as an instance of a mark type `Plot`. By modeling plots as marks, we can combine them and other graphical marks into diagrams. Consider, for example, Figure 4.15, which composes a scatter plot, a rotated histogram and a line mark. Representing this type of idiosyncratic visualization is often unfeasible in visualization grammars, yet, it becomes possible in our framework.

```
d = scatter_plot +
    T(x,y)R(θ)*histogram +
    S(:strokeDasharray=>2)*Line([p1,p2])
```

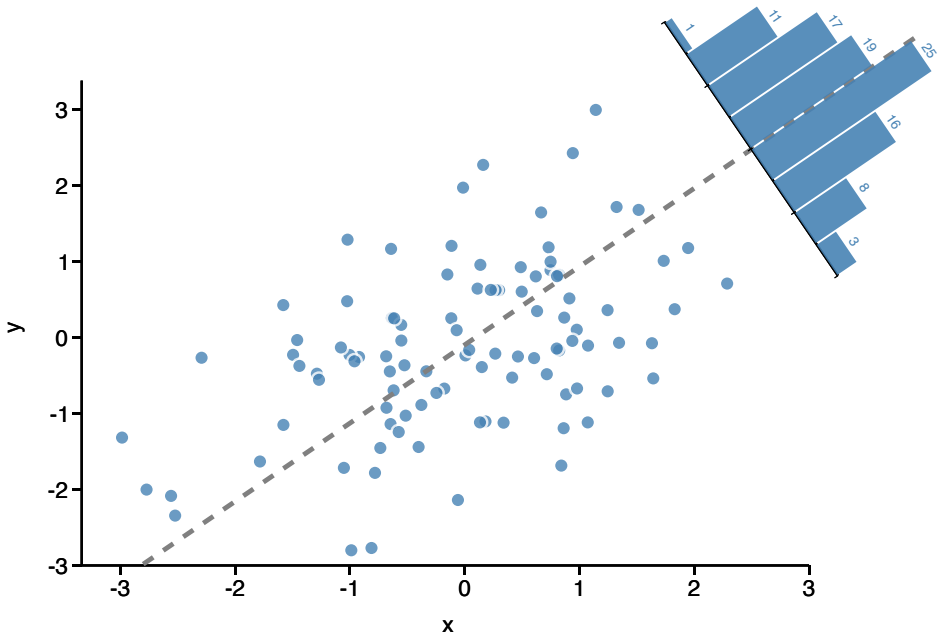



Figure 4.15: Example of PCA plot inspired on Rougier [55]. At the top, the Vizagrams code combines the scatter plot diagram with the transformed histogram and adds a line. The complete code to reproduce this visualization is present in Vizagrams documentation (https://davibarreira.github.io/Vizagrams.jl/dev/).

# Chapter 5

# Vizagrams

Based on the theoretical work described in Chapter 4, we developed Vizagrams, a data visualization framework written in the Julia programming language. The package implements a diagramming DSL with a visualization grammar built on top of it. This integration allows one to create and manipulate data visualizations using diagramming operations.

In this chapter, we aim to demonstrate how Vizagrams can be used to create complex visualizations. We begin with some technical details regarding the package, such as installation, documentation and so on. We then introduce the basics of diagramming, including the manipulation and composition of marks. From there, we move on to the creation of custom graphical marks, providing users with greater flexibility. We then explore how to construct basic data visualizations using the framework's visualization grammar. Building on this, we dive into the use of graphic expressions to generate more complex plots. We also highlight some of the low-level functionalities available in Vizagrams that can further enhance the creation of custom visualizations. Finally, we discuss how Vizagrams balances abstraction and expressiveness.

## 5.1 Getting Started

Vizagrams is an open-source package implemented in the Julia programming language under the MIT Licence [3]. The code is hosted at `https://github.com/davibarreira/Vizagrams.jl`, where it has been actively maintained.

The choice for the Julia language was based on three main considerations: first, Julia is a high-level language focused on scientific computing, thus quite suitable for data science tasks; second, it supports the functional programming paradigm which eases the process of implementing our theoretical framework; lastly, it is dynamically typed, which allows the use of interactive environments such as notebooks.

The package is registered in the Julia General Registry, which means that the installation can be done directly via the Julia package manager. One can install Vizagrams latests version by entering into the Julia REPL and running the commands in code 5.1.

```
julia> using Pkg
julia> Pkg.add("Vizagrams")
```

Code 5.1: Installing Vizagrams from a Julia REPL.

The package is documented at https://davibarreira.github.io/Vizagrams.jl/dev/, where several tutorials can be found, together with a gallery of examples. The documentation focuses on the usability of Vizagrams, not covering the theoretical aspects discussed in Chapter 4. Some of the codes presented here are also present in the package's documentation with the complete implementation steps, importing dependencies, data and so on.

Vizagrams is specially suited for interactive programming environments, such as computational notebooks [30, 69]. These notebooks are an essential tool for data analysts, as they allow for an iterative workflow that combines code, visualizations, and narrative seamlessly [50].

## 5.2 The Diagramming DSL

Vizagram is implemented according to the theory laid out in Chapter 4, therefore, diagrams are modeled as values of type `𝕋{Mark}`, which represents expression trees with leaves containing graphical marks and nodes containing either the composition operation or graphical transformations.

The package provides several ready to use marks, ranging from primitive geometric shapes, such as circles, lines, polygons, to more complex ones such as arrows, faces, grids, legends, and plots. As an example, consider the Code 5.2, where we define a diagram `d` as a single circle centered at `[0,0]` with radius equal to 1.

```
d = Circle() # By default, Circle() is equal to Circle(r=1,c=[0,0])
draw(d)      # Converts d into an SVG vector graphic
```

Code 5.2: Simple diagram using Vizagrams

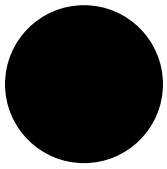


Figure 5.1: Diagram produced from Code 5.2.

Here a technical note is important to mention. Internally, a circle is of type `Circle`, which itself is a subtype of `GeometricPrimitive`. Hence, it would be incorrect to say that `d` is a diagram, since it is not a value of type `𝕋{Mark}`. Yet, as was shown in Chapter 4, one can *lift* a geometric primitive into a diagram, i.e. there is a canonical way of turning a value of `GeometricPrimitive` into `𝕋{Mark}`. Using this fact, Vizagrams can treat single geometric primitives as if they were diagrams, thus reducing the boilerplate that would be necessary to make `d` into a "formal" diagram. This same strategy is used for other data types, such as singular marks (`Mark`) and graphical primitives (`Prim`).

Note that in Code 5.2, we call function `draw` in order to display the diagram in the screen. In fact, `draw` is actually turning the diagram into an SVG vector graphic. Vizagrams does not implement a renderer for diagrams, instead, it relies on existing graphical formats. The default, and only currently implemented backend, is the SVG format, yet, the code is modularized as to allow different backends. The SVG format is well suited for 2D graphics and can be rendered by browsers, and common data science tools such as Jupyter Notebook [30] and VSCode.

Marks and diagrams can both be composed together using the summation operator (`+`). Summing two objects creates a new diagram where each object is lazily stored in the mark tree. This summation operation is not commutative, since the order of the sum determines the order which the objects will be rendered in the screen, as shown in Figure 5.2.

```
d = Face() + Circle(c=[1,0],r=1)
```

Code 5.3: Diagram illustrating how the order of composition affects the rendering order.

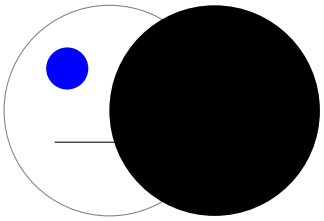


Figure 5.2: Diagram represented in Code 5.3 showcasing the non-commutative of `+`.

Vizagrams implements both geometric transformations (`G`) and style properties (`S`), which can be used to alter diagrams geometry and aesthetics, respectively. The geometric transformations available are translation (`T(x,y)`), rotation (`R(ang)`), uniform scaling (`U(s)`) and reflection (`M(ang)`). Style properties are written using key-value pairs, i.e. `S(attribute_name => attribute_value,...)`, and can be used as style transformations via right-biased union. Since Vizagrams uses SVG as backend, the attribute names match those from SVG, e.g. `fill` represents fill color, `stroke` represents stroke color, and so on. Any style attribute existent in SVG can be used in Vizagrams, with the only caveat that the attribute name must be written in *camel case*, instead of hyphenated, for example, `strokeWidth` instead of `stroke-width`.

Graphical transformations are modeled as tuples of geometric transformations and style transformations (`H:=Tuple{G,S}`). Similar to primitives and marks, geometric and style transformations can be lifted into graphical transformations, which enable us to talk about them without making a distinction. Transformations are applied to diagrams using the multiplication operator (`*`), and they can be composed by placing them side by side. It is also possible to compose transformations using the same multiplication operator, yet, this can make the code less readable. Moreover, the multiplication operation is distributive over composition, hence, one can use, for example, `T(1,1) * (a + b)` as a way to apply translation to both `a` and `b`. The use of graphical transformations is exemplified in Code 5.4.

```
d = R(π/4)*Square(l=3) +
    S(:fill=>:green)*Circle() +
    T(5,0)U(2)S(:fill=>:purple,:stroke=>:orange,:strokeWidth=>10)*RegularPolygon(n=8)
```

Code 5.4: Applying graphical transformations to diagrams.

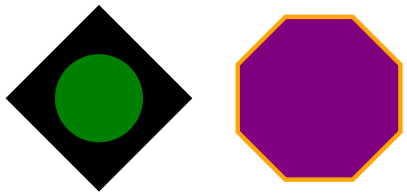


Figure 5.3: Exemplifying the use of graphical transformations from Code 5.4.

We can now combine all these operations in order to create more complex diagrams. Using interactive environments such as computational notebooks, users can store diagrams into variables and combine them together, as demonstrated in Code 5.5.

```
d1 = S(:fill=>:red,:strokeWidth=>10,:stroke=>:black)Circle(r=1.5) + T(1.5,0)R(π/4)*Square(l=2)
d2 = S(:fill=>:blue)T(0,2.5)*TextMark(text="my diagram",fontsize=0.5)
d3 = U(4)S(:opacity=>0.1)Rectangle(h=1,w=2)
d = d3 + T(-1,0)*d1 + d2
```

Code 5.5: Example of diagram operations.

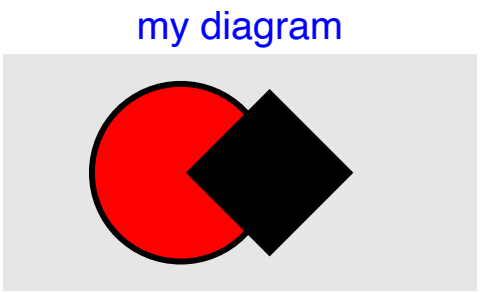



Figure 5.4: Diagram produced from Code 5.5.

Composition and graphical transformations are the main way of combining diagrams, yet, Vizagrams also implements some other convenient functions. For example, diagrams can be stacked together using the arrow unicode (see Code 5.6). To stack diagrams, Vizagrams computes the envelope for each diagram, and applies a translation transformation. Envelopes are a concept developed by Yorgey [85] which resembles the idea of bounding boxes.

```
d = Circle() → Circle() ↑ Square()

draw(d)
```

Code 5.6: Example of diagram stacking.

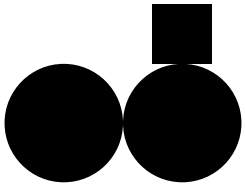


Figure 5.5: Illustration of diagram stacking produced from Code 5.6.

The fact that Vizagrams is an embedded DSL means that users can use regular programming logic to create diagrams (see Code 5.7. Again interactive environments such as notebooks shine, as users can quickly implement functions, loops, if-else statements, and see the results.

```
function sierpinski(n::Int)
    if n == 0
        return Polygon([(1,0), (0,0), (1/2, 1)])
    else
        t = sierpinski(n - 1)
        return U(0.5) * (T(-1/2,0)t + T(0,1)t + T(1/2,0)t)
    end
end
d = sierpinski(5) # fractal diagram
```

Code 5.7: Using programming logic to create fractal diagram.

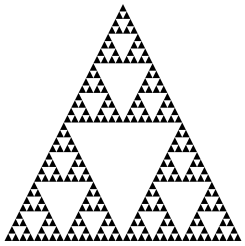


Figure 5.6: Sierpinski's triangle diagram produced from Code 5.7.

## 5.3 Creating Custom Marks

In Vizagrams, users can create their own graphical marks by making use of Julia's type system and multiple dispatch functionality. The process of mark creation is exactly the one explained in Chapter 4. First, one must create a struct to represent the new mark, where the struct creates a new type which must be a subtype of `Mark`. Second, the user must define a function $\zeta$ where the input is the new mark, and the output is the diagram representing this mark.

Again, the code implementation is the same as the one used in Chapter 4, yet, we provide here another example for the sake of completeness. In Code 5.8, we create a simple mark emulating the famous Chernoff faces [17].

```
struct Chernoff <: Mark
    headshape::Real
    eyebrow::Real
    smile::Real
end

function ζ(face::Chernoff)
    eye = S(:fill=>:white,:stroke=>:black)Circle(r=0.3) + Circle(r=0.05)
    eyes = T(-1,1)*eye + T(1,1)*eye

    eyebrow = T(-1,1.4)R(atan(face.eyebrow))*Line([[-0.3,0],[0.3,0]])
    eyebrows = eyebrow + M(π/2)*eyebrow

    smile = QBezier(bpts=[[-1.3, -0.5], [1.3, -0.5]], cpts=[[0, -face.smile]])

    rx = 2 - min(0,face.headshape)
    ry = 2 + max(face.headshape,0)
    head = S(:fill=>:white,:stroke=>:black)Ellipse(rx=rx,ry=ry)

    return head + eyes + eyebrows + smile
end
d = Chernoff(-0.5,-0.5,0) + T(5,0)*Chernoff(0,0,0.5) + T(10,0)*Chernoff(0.5,0.5,1)
```

Code 5.8: Implementing graphical mark inspired in Chernoff [17].

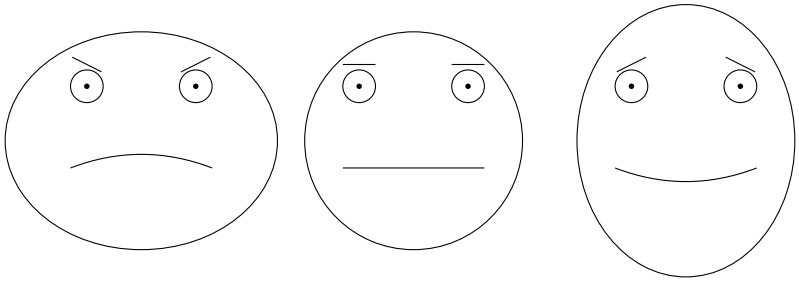


Figure 5.7: Diagrams with custom mark produced from Code 5.8.

Note that once the struct and the $\zeta$ function are defined, the new mark becomes instantly available to the user. The dynamic nature of Julia combined with the interactive computing environment greatly facilitates the process of creating and testing custom marks. Users can experiment with new graphical elements swiftly, without relying on external packages or extensions. This streamlined workflow not only accelerates the development cycle, but also simplifies the sharing of implementations between users. By simply copying and pasting code examples, users can easily exchange and build upon each other's work.

## 5.4 The Visualization Grammar

Vizagrams implements a visualization grammar with a plot specification similar to the one in Vega-Lite [58]. The grammar is built on top of the diagramming DSL, with graphical marks being used to model both the specification and visual guides such as ticks, axes, legends and grids. The plot specification is modeled as a mark `Plot` with the following structure:

```
Plot(
    config = (...),
    encodings = (
        channel_1 = (field,scale,...),
        channel_2 = (field,scale,...),
        ...),
    graphic = (...)
)
```

Code 5.9: Structure of `Plot` mark.

The `config` parameter holds information related to the visual guides, the `encodings` holds the encoding dictionaries, and the `graphic` contains the graphic expression for the plot. Since `Plot` is a mark, Vizagrams implements a $\zeta$(`p::Plot`) function responsible for turning the plot specification into a diagram, i.e. performing the assembly process. In order to avoid long specifications, Vizagrams contains smart defaults that allows users to provide "incomplete" (more succinct) specifications. This is exemplified in Code 5.10, where it shows a minimalistic specification for a scatter plot. In this example, the `config` and `encodings` are omitted, and a `Circle` mark is used in the `graphic` parameter, instead of the graphic expression.

```
plt = Plot(
    data=df,
    x=(field=:Horsepower,),
    y=(field=:Miles_per_Gallon,),
    color=(field=:Origin,),
    graphic=Circle(r=5)
)
```

Code 5.10: Structure of `Plot` mark.

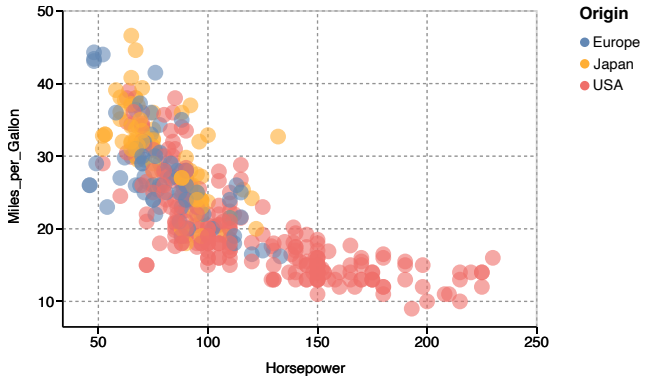



Figure 5.8: Scatter plot using Vizagrams specification shown in Code 5.10. The `df` variable is a dataframe containing the Stanford Cars dataset from Krause et al. [31].

The integration between diagramming and the visualization grammar enables users to incorporate diagrams into plots. For example, one can create a diagram and use it as a mark within the plot specification. As shown in Code 5.11, users can create diagrams, and use them as marks inside plots.

```
d = S(:fill=>:white,:stroke=>:black)*Circle(r=2) + Circle() + T(2,0)*Square()

new_plt = Plot(
    data=df,
    x=(field=:Horsepower,),
    y=(field=:Miles_per_Gallon,),
    color=(field=:Origin,),
    size=(field=:Acceleration,),
    graphic=Mark(d)
)
```

Code 5.11: Plot specification with diagram as mark.

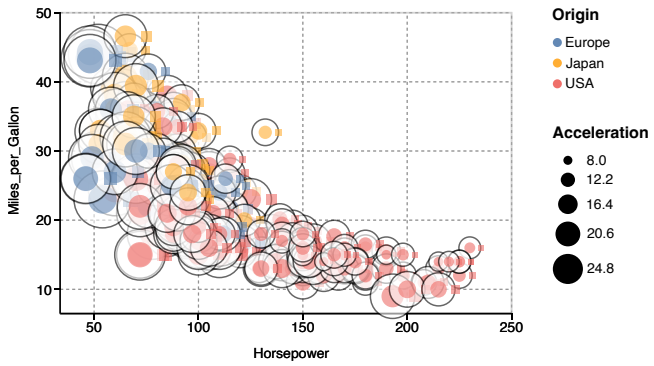



Figure 5.9: Plot from Code 5.11 where a diagram is used as a graphical mark for a scatter plot.

This integration of diagramming and data visualization actually runs both ways. Vizagrams models plots as graphical marks, which means that we can use the same diagramming operations to combine and manipulate plots. This can be seen in Code 5.12, which combines the plots defined in Code 5.10 and 5.11.

```
viz_title = TextMark(text="Creating a New Viz",anchor=:c,fontsize=20)
viz_frame = S(:fillOpacity=>0,:stroke=>:black)T(400,100)*Rectangle(h=370,w=1000)

new_viz = new_plt + T(470,0)plt + viz_frame + T(400,250)*viz_title
```

Code 5.12: Manipulating plots as diagrams.

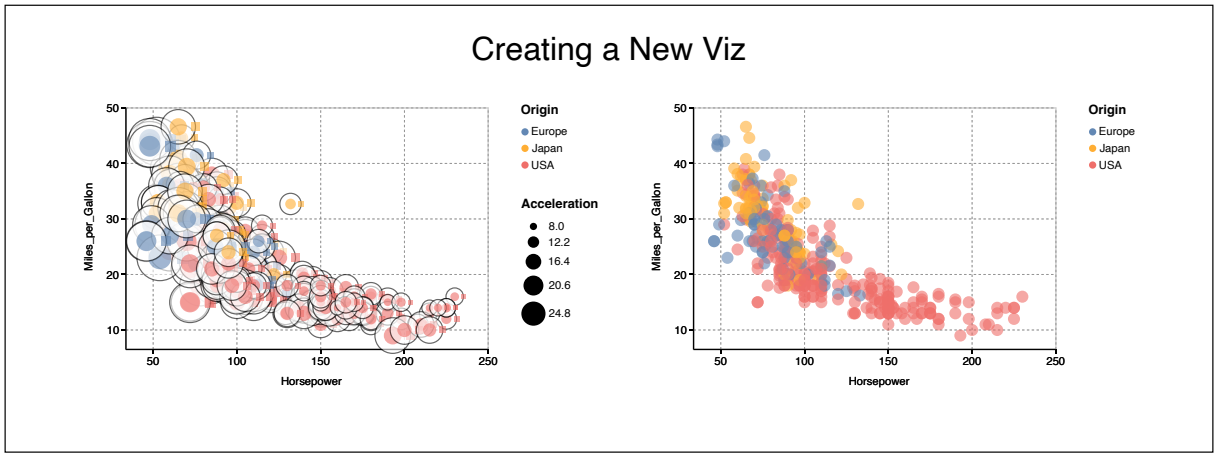



Figure 5.10: Plot from Code 5.12 which combines plots using diagramming operations.

Similar to how is done in Vega-Lite, distinct graphics can be produced by picking distinct graphical marks. For example, a line plot can be obtained using the `Line` mark, a bar plot can be obtained using a `Bar` mark, and a boxplot can be obtained using a `BoxPlot` mark (see Code 5.13). Internally, Vizagrams automatically infers the corresponding graphic expressions for these marks, minimizing the amount of specification needed from the user. However, as will be demonstrated in the next section, users can override this inference by directly supplying their own graphic expressions.

```
boxplot = Plot(
    config=(title="BoxPlot",xgrid=NilD(),ygrid=NilD()),
    data=df,
    encodings=(
        x=(field=:Origin,),
        y=(field=:Miles_per_Gallon,),
        color=(field=:Origin,)
    ),
    graphic=BoxPlot()
)
```

Code 5.13: Specifying a boxplot.

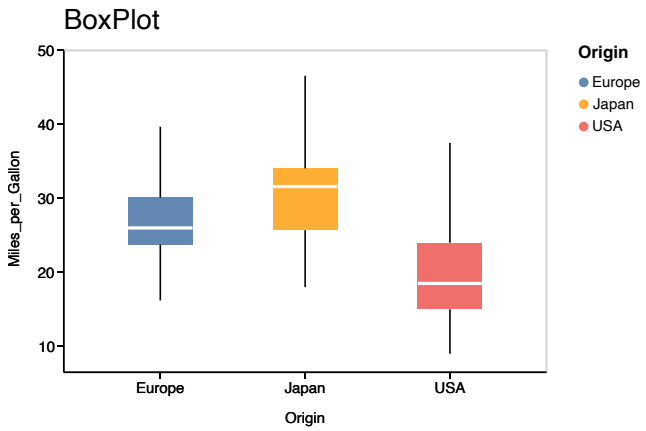



Figure 5.11: Boxplot from Code 5.13.

## 5.5 Graphic Expressions

Graphic expressions are used to specify how to construct graphics from the data by making explicit the process of breaking down the data and mapping it into diagrams. As discussed in Chapter 4, graphic expressions are modeled as a struct containing fields `expr`, `coalg` and `alg`, where a function is stored in each one. In Vizagrams, graphic expressions can be defined using the function shown in Code 5.14.

```
∑(expr; op=+,i=nothing, orderby=nothing, descend=false, kwargs...)
```

Code 5.14: Graphic expression function from Vizagrams.

This is a function that produces graphic expressions by constructing algebras from the `op` operator, and coalgebras from the index `i`. The algebra is the `reduce(op)` function that iteratively applies `op` to the elements in the list, e.g. `reduce(→, [a,b,c]) := a → b → c`. The coalgebra is a `groupby(i)` function that returns a list of sub-datasets grouped by index `i`, where `i` is a column name. If no `i` is passed, then the function groups by the row index, i.e. it iterates row by row. The other arguments, such as `orderby` and `descend`, are ways to control the `groupby` function.

The most *ergonomic* way of writing graphic expressions in Julia is using the *do-block* syntax, which is a concise and clean way to pass anonymous functions as the final argument to a function call. For example, the graphic expression for a scatter plot can be written as:

```
∑() do row
    S(:fill=>row.color)Circle(r=5,c=[row.x,row.y])
end
```

Code 5.15: Graphic expression for scatter plot.

In this example, the graphic expression summation operator (∑) is not receiving any index `i`, thus, it is iterating over each row in the dataframe, which are represented by `row`. Inside the do-block, we create a diagram using the values in each `row`.

The use of graphic expressions grants finer control over how to construct visualizations. Users can control how to split the data, and how each chunk of the data becomes a diagram. For instance, while a scatter plot iterates over each row to plot individual points, a line plot groups rows based on their color and then draws a continuous line for each group, as demonstrated in Code 5.16.

```
Plot(
    config=(xaxis=(ticktextangle=π/2,),),
    data=gdf,
    x=(field=:Year,),
    y=(field=:Horsepower_mean,),
    color=(field=:Origin,),
    graphic= ∑(i=:color) do rows
        S(:stroke=>rows.color[1],:strokeWidth=>3)Line(rows.x,rows.y)
    end
)
```

Code 5.16: Line plot using graphic expression. The `gdf` variable is a dataframe containing the Stanford Cars dataset [31] grouped by year and origin.

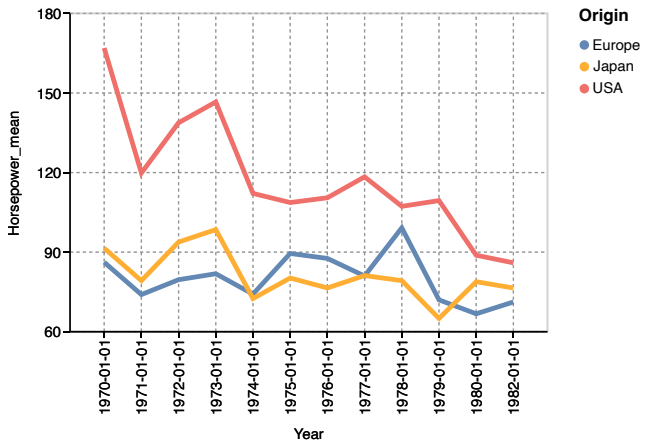



Figure 5.12: Line plot from specification using graphic expression in Code 5.16.

Once we have the explicit graphic expression, it is easy to modify the graphic according to one's needs. For example, one could easily change the line mark for any other similar mark, such as an arrow or a polyline.

Another property of graphic expressions is that they can be nested inside each other, which is useful when one needs to group a dataset more than once and apply a different algebra depending on the grouping. An example of this is the graphic expression for a stacked bar plot in Code 5.17, which can be used to produce the plot in Figure 5.17.

```
∑(i=:x,op=+,
  ∑(i=:color,op=↑,orderby=:color,
    ∑() do row
      S(:fill=>row[:color])*Bar(h=row[:y],c=[row[:x],0], w = 40)
    end
  )
)
```

Code 5.17: Graphic expression for vertical stacked bar plot.

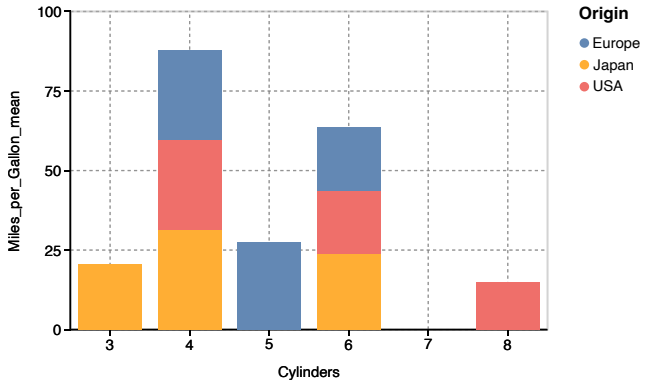



Figure 5.13: Vertical stacked bar plot using the graphic expression in Code 5.17.

Note that in Code 5.17 there are three nested graphic expressions, each with a distinct algebra and coalgebra operator. In summary, what this graphic expression does is the following:

```
Group data by values of x
    Group data by values of color
        Iterate over each row drawing a bar
    Stack bars vertically
Sum the stacked bars
```

This highlights how graphic expressions translate the construction logic of a graphic in a fairly direct manner. Moreover, just like in the case of the line plot, once we have the expression behind a graphic, we can easily modify and increment it, adding elements such as text, modifying the width of the bars and so on. An example of this kind of modification can be seen in Figure 5.14 (G).

The key takeaway is that graphic expressions, despite their abstract mathematical notation, operate on a straightforward principle. The process involves splitting data into chunks, transforming these chunks into individual diagrams, and then combining these diagrams to create the final graphic. The true expressiveness lies in the flexibility and the variety of ways these elements can be combined.

## 5.6 Extra Functionalities

We have shown two basic strategies for producing complex visualizations with Vizagrams. One consists in using graphic expressions to imbue diagrams into plot specifications. The other consists in manipulating plots as diagrams. A third strategy involves using extra functionalities in order to gain even more expressiveness.

These extra functionalities are functions available in Vizagrams that fall somewhat outside our theoretical exposition, but that are nonetheless quite useful. For instance, Vizagrams has function `getscale` to extract the scales from a plot, allowing users to reuse them to transform other datasets. Similarly, function `getmarkpath` can be used to extract marks and graphical transformations from diagrams.

An example of how these extra functions are useful is demonstrated in visualization (B) in Figure 5.14, which is a composite visualization known as integrated, a term coined by Javed and Elmqvist [27]. Producing integrated visualizations is specially challenging because they involve rendering graphical elements across different scales. In order to produce visualization (B), one can generate each bar plot separately, use the diagramming tools to stack them, extract their scales, and then use those scales to draw lines across each plot. The code for this example is available in the Vizagrams documentation under the tutorial section on composite visualizations.

Some other useful functions are `align_transform` and `centralize_graphic`. These functions work similarly to the stacking operation in the sense that they use they use the envelope of the diagram in order to produce geometric transformations. The `align_transform` is used to align diagrams according to some rule (e.g. align vertically or horizontally), while the `centralize_graphic` is used to center the diagram in the canvas coordinates.

## 5.7 Evaluation

In this section, we evaluate Vizagrams according to its expressiveness and abstraction. These were pivotal attributes that led to the development of our framework, as our goal was to extend the capability to generate a wide range of complex visualizations while retaining the abstraction inherent to high-level specifications. To assess the expressiveness of Vizagrams, we showcase a gallery of examples that include composite visualizations and plots with custom marks. Additionally, we compare Vizagrams' level of abstraction with that of another well-established visualization grammar, Vega-Lite, which served as a foundation for Vizagrams' own specification syntax.

### 5.7.1 Expressiveness

Our primary goal with Vizagrams was to enhance the expressiveness of visualization grammars. Specifically, we aimed to enable the creation of composite visualizations and plots with custom marks. To achieve this, we proposed an integration of diagramming and data visualization, which was done in three ways. First, we defined graphical marks in terms of diagramming operations, enabling the creation of custom marks within Vizagrams diagramming DSL. Second, we modeled plots as marks, allowing the use of diagramming tools to manipulate them. Third, we incorporated graphic expressions into the data visualization specification, thereby connecting data encoding and diagramming.

To demonstrate the expressiveness of Vizagrams, we produced a gallery of visualizations, which can be found in Vizagrams documentation. Figure 5.14 illustrates 10 selected examples. These examples cover composite visualizations (e.g. B,C,D,E and F), visualizations with custom marks (e.g. A, G, I, J) and plots with polar coordinates (e.g. H).

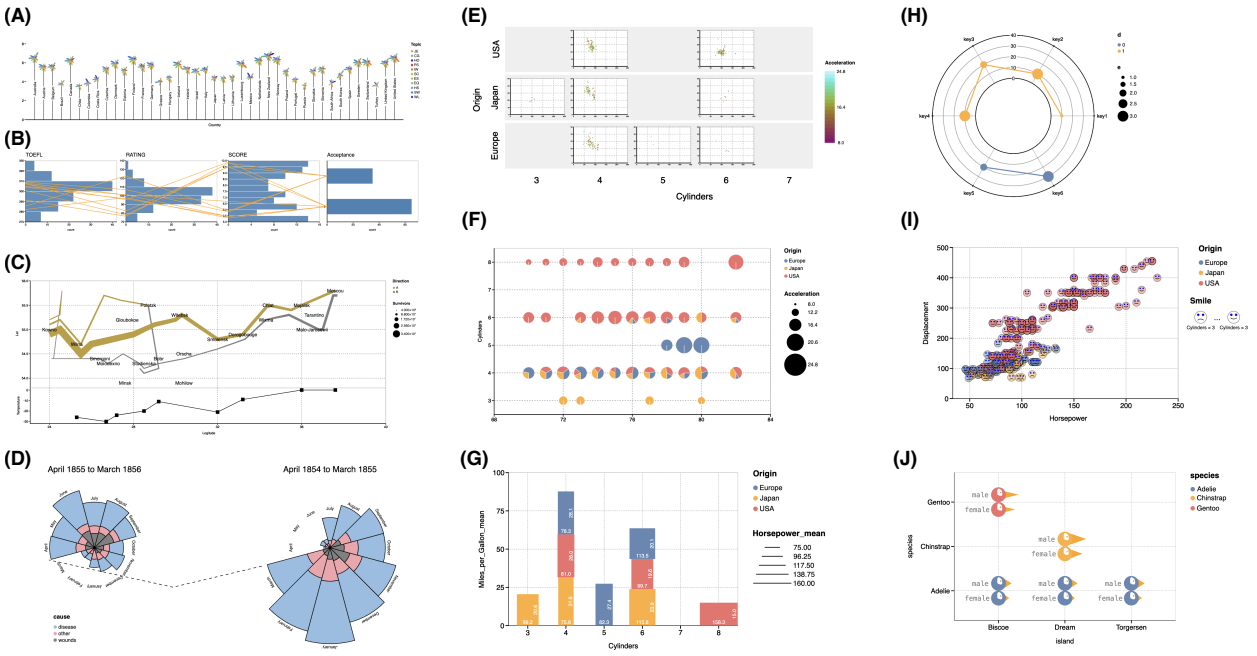



Figure 5.14: Gallery of examples produced with Vizagrams. (A) is a reproduction of the OECD Better Life Index plot [64]; (B) is inspired on the visualizations from Cheng et al. [16]; (C) is Minard's chart of Napoleon's Russian campaign; (D) is Nightingale's rose chart. The other examples are composite and custom mark visualizations designed by the authors. All these examples are present in Vizagrams documentation, with the full code for reproduction.

### 5.7.2 Abstraction

A second goal with Vizagrams was to devise a grammar specification that maintained the high-level abstraction present in other visualization grammars. In order to achieve this, its specification was developed based on Vega-Lite. Despite the similarities, there are several notable design differences. For instance, Vizagrams separates guide information from encodings, and some attribute names differ. In terms of abstraction, the most significant differences are: (1) the use of *graphic* instead of *mark*; (2) a lack of *facet* channel.

In high-level visualization grammars like Vega-Lite [58] and ggplot2 [75], graphics are typically specified using a single mark (*geom*). These grammars infer the overall graphic from the selected mark. For instance, a line plot is generated by choosing the line mark, while a scatter plot is created by selecting the point mark.

Vizagrams introduces a different approach by using the *graphic* parameter instead of *mark*. The *graphic* parameter contains graphic expressions, which describe how to map the dataset onto a diagram. To maintain the same level of abstraction as other visualization grammars, Vizagrams allows users to specify graphics by passing graphical marks to the `graphic` parameter. Through polymorphism, Vizagrams can infer the corresponding graphic expression from the provided mark. This enables users to enjoy the simplicity of mark-based specifications while also having the option to write explicit graphic expressions when greater expressiveness is required.

The second major distinction is the lack of a *facet* channel. Faceting consists in grouping the dataset by a given data column, and then creating a collection of stacked plots sharing the same axes and scales. The *facet* channel is quite distinct from other encoding channels, which led us to avoid implementing it in Vizagrams. This was a design choice that prioritized consistency over user convenience. Note that, despite not having a *facet* channel, Vizagrams can reproduce its behavior due to its diagramming capabilities. Consider, for example, Code 5.18, where we use the graphic expression function on the plots themselves in order to reproduce the faceting behavior.

```
∑(i=:Origin,op=→) do gdf
    Plot(
        title=gdf.Origin[1],
        data=gdf,
        x=(field=:Horsepower,scale_domain=(40,250)),
        y=(field=:Miles_per_Gallon,scale_domain=(0,50)),
        graphic=Circle(),
    )
end(df)
```

Code 5.18: Example of facet plot in Vizagrams.

In Code 5.18, the dataset `df` is grouped by column `:Origin` and each grouped dataset `gdf` is used to produce a scatter plot. These plots are then stacked horizontally using the stack operator. This illustrates a way of producing the facet plot, but several others are possible.

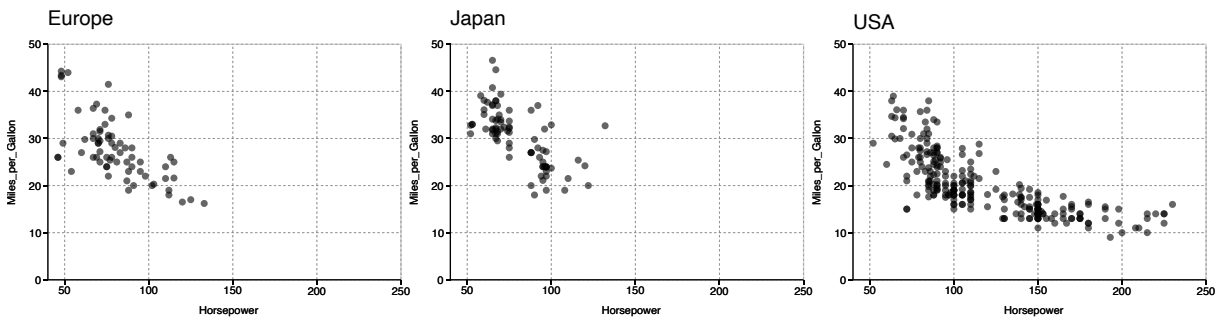



Figure 5.15: Facet plot from Code 5.18.

For example, one could have used a simple for-loop to produce the scatter plots, and then combine them applying translation transformations.

## 5.8 Related Works

### 5.8.1 Diagramming

Yorgey [85] provided a theoretical foundation for diagram construction through the use of monoids within Functional Programming. His ideas were used to develop the Haskell Diagrams library [83], which has inspired other diagramming libraries such as Diagrammar [24] and Compose [23], as well as our own work.

There are diagramming libraries that are not based on Yorgey's conception of diagrams, such as Bluefish [48] and Penrose [84]. Penrose focuses on mathematical diagrams, and does diagram construction by automatically translating math-like statements into visual representations via constrained numerical optimization. Bluefish models diagrams using a *relational grammar*, where primitive shapes are combined via perceptual groupings (e.g., distribute, align, link). This relational grammar also covers statistical graphics, thereby integrating diagramming and data visualization within the same framework. In this sense, Bluefish is similar to Vizagrams, but it takes a different approach on how to integrate both subjects. While Vizagrams uses diagramming as a foundation to build visualizations, Bluefish's relational grammar is what allows for the description of both diagrams and statistical graphics within a single framework.

The idea of constructing data visualization tools on top of diagramming frameworks has been done in other packages, although the focus tends to be drawing the plot rather than an actual integration between diagramming and data visualization. An example of this is the Gadfly.jl visualization package, which is built on the diagramming package Compose.jl [23].

### 5.8.2 Visualization Grammars

In the diverse ecosystem of visualization grammars, Vizagrams can be classified as a general purposed high-level grammar, resembling grammars such as Vega-Lite, ggplot2, Plot Observable and Gadfly.jl, all of which are based on the Grammar of Graphics. Among these grammars, ggplot2 is the only one to have implemented mark creation, which is available via a lower-level system called ggproto [77].

The Atlas[1] [37] grammar addresses this issue of mark creation through a *graphics-centric approach*, where graphical objects are treated as first-class citizens. Users can create graphical objects (primitives and glyphs), which can further be manipulated and combined with data in order to generate visualizations. This constructive approach is similar to Vizagrams', where graphical marks are used to build visualizations, yet, the construction mechanism of both frameworks are quite distinct. First, Atlas defines visualizations via a procedural syntax instead of a declarative specification. Second, Atlas joins data and graphics through rules such as *repeat*, *divide* and *densify*, while Vizagrams uses graphic expressions and diagramming operations. Third, the mark (glyph) creation in Atlas consists in grouping primitives instead of the creation of a new datatype containing new parameters.

Smeltzer et al. [60] introduces a Haskell-embedded DSL that takes a transformational approach to data visualization. It focus on incrementally modifying visuals through functional transformations, rather than using a declarative specification, as in most visualization grammars. While it shares with Vizagrams a foundation in the Functional Programming paradigm, the overall approach and resulting visualizations are quite distinct.

### 5.8.3 Visualization Authoring Systems

A visualization authoring system is a software platform or tool that enables users to create, design, and customize data visualizations. These systems typically feature graphical user interfaces (GUIs), which fundamentally distinguish them from visualization grammars. Despite this difference, some of these authoring systems incorporate internal grammars and design principles that are closely aligned with those of Vizagrams. The authoring systems that most closely resemble Vizagrams are those that offer glyph creation capabilities, allowing users to design and manipulate custom graphical marks [29, 36, 82, 72, 52, 74, 68]. Notably, Improvise [74], Data Illustrator [36], Charticulator [52], and StructGraphics [68] share several similarities with Vizagrams. These systems, like Vizagrams, emphasize the importance of customizable glyphs (marks) in building expressive and flexible visualizations, though they do so within the context of a GUI rather than a domain-specific language.

Improvise [74] is a system implemented in Java that enables users to create and coordinate multiple visualizations. Central to the system is a visual abstraction language called Coordinated Queries, which enhances expressiveness by allowing users to define how data is

[1] Currently this package has been renamed to Mascot.js.

filtered, projected, and visually encoded into glyphs using expressions. These expressions are structured as trees of operators, which compute the value of output fields based on the fields of input records. Vizagrams and Improvise share several striking similarities, particularly in how they extend expressiveness through user-defined expressions. However, the primary distinction lies in their underlying languages: Coordinated Queries in Improvise employs a structured, operator-driven approach, whereas Vizagrams utilizes graphic expressions and diagramming operations.

Data Illustrator [36] and Charticulator [52] break down the visualization specification into two distinct levels: the glyph-level and the chart-level. At the glyph-level, users can design custom glyphs (marks) using an editor where anchor points are defined and can be bound to data variables. At the chart-level, the two tools differ in their approaches to constructing the overall visualization. Data Illustrator employs repetition and partitioning in order to create the final graphic. In contrast, Charticulator utilizes the concept of *plot segments*, which distribute glyphs across the chart using scaffolds such as horizontal stacking or axes. These plot segments also apply transformations based on the coordinate system, providing a flexible way to arrange the glyphs within the chart.

StructGraphics [68] is another authoring system that enables users to create custom glyphs and manipulate their properties to produce visualizations. The way StructGraphics models visualizations internally shares similarities with Vizagrams. In StructGraphics, visualizations are structured as trees, where the leaf nodes represent graphical primitives. These primitives can be grouped together into *groups*, which are conceptually similar to *marks* in Vizagrams, and groups can be further combined into *collections*, analogous to the concept of *graphics* in Vizagrams. Through *repetition*, *variation*, and *grouping*, users can manipulate collections to create visualizations in StructGraphics.

# Chapter 6

# Conclusion

## 6.1 Summary of Contributions

The goal of this work was to enhance the expressiveness of visualization grammars while maintaining a high level of abstraction. We addressed this challenge by developing a theoretical model that redefined the graphic assembly process from a constructive perspective, effectively integrating diagramming principles with data visualization. We formalized our theory in the language of Category Theory, and tied it to computational implementation via the concept of categorical programming. Throughout our theoretical exposition, the following ideas were established:

- Developing a theory of diagram construction that builds on the foundations laid by Yorgey [85];
- Redefining graphical marks as abstract representations of diagrams, enabling greater flexibility and expressiveness;
- Incorporating the creation of custom marks within the visualization framework;
- Introducing graphic expressions as a method to encode data into graphics;
- Formalizing plots as graphical marks, thus enabling the use of diagramming operations to manipulate plots.

Based on this theory, we developed Vizagrams, an open-source package in Julia that combines a diagramming DSL with a visualization grammar. This package serves as both a proof-of-concept and a practical tool for data visualization.

As a proof-of-concept, Vizagrams was used to illustrate the claim that our framework could expand expressiveness while balancing abstraction. These were shown via a gallery of examples and comparisons with graphic specifications.

As a practical tool, Vizagrams can be seen as a general-purpose, high-level visualization grammar similar to packages such as ggplot2, Altair [70], and Gadfly, as it is embedded within a dynamic programming language. However, unlike these other packages, Vizagrams also supports diagramming, making it uniquely versatile in the data visualization ecosystem. Another notable feature is that users can create new marks and define custom types of visualizations through graphic expressions, all within the Vizagrams framework, without requiring external packages or plugins. This self-contained approach simplifies and encourages sharing and collaboration among users.

## 6.2 Limitations and Future Work

In this section, we reflect on the current limitations of our visualization framework and explore directions for future research and development.

**Beyond Dataframes.** Our theory was formulated from the premise that datasets were structured as dataframes, i.e. single row-indexed table. We believe that it is possible to generalize the theory to other structures, such as relational databases. This would be interesting not only from a theoretical perspective, but also from a practical one. Currently, most visualization grammars limit the data input to dataframes, leaving it up to users to perform data transformations in order to turn their specific dataset into a singular dataframe. An example of this are graphs (network data), which is commonly stored as two related tables (edges and vertices).

**Learning Curve.** Vizagrams is primarily focused on users with programming skills. Users are required to learn both the diagramming DSL and the visualization specification syntax. Although the diagramming DSL has been designed to be simple, the complexity of graphic expressions may present an entry barrier for many users. Lowering this entry barrier is a key consideration for future development.

**Embedded DSL.** As an eDSL, Vizagrams is tightly coupled with the Julia programming language, which restricts the framework's use to those already familiar with Julia. However, by delineating the theory and core ideas behind Vizagrams, the framework's main concepts can be incorporated into other visualization frameworks and programming environments. Additionally, a possible development would be the creation of a GUI-based authoring system that leverages Vizagrams as a backend, broadening its usability beyond programmers.

**Interactivity.** Currently, Vizagrams lacks support for interactive visualizations, which limits its applicability in fields where user interaction with data is crucial. Incorporating interactivity is a priority for future development. This feature would not only enhance the user experience with the framework, but it would also bring Vizagrams in line with other modern visualization tools that emphasize interactive data exploration.

**3D Visualization.** An important direction for future work is extending Vizagrams to

support 3D visualizations. While 3D graphics are less commonly addressed by visualization grammars, they hold significant potential, particularly in scientific and engineering domains. Currently, Vizagrams is built on an SVG backend, which limits its capacity for rendering 3D content. Thus, a possible development involves exploring alternative rendering backends that provide 3D support.

**Empirical Studies.** It would be interesting to perform empirical studies with users in order to gain new insights into the overall usability of Vizagrams. Such studies could clarify how intuitive users find concepts like graphic expressions or mark creation in practice.

# Bibliography


[1] Observable plot. https://github.com/observablehq/plot, 2024.

[2] C. Allen, J. Moronuki, and S. Syrek. *Haskell Programming from First Principles*. Lorepub LLC, 2016. ISBN 9781945388033. URL https://books.google.com.br/books?id=5FaXDAEACAAJ.

[3] Davi Sales Barreira. davibarreira/vizagrams.jl: v0.2.3, August 2024. URL https://doi.org/10.5281/zenodo.13331908.

[4] Jacques Bertin. *Semiology of graphics*. University of Wisconsin press, 1983.

[5] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[6] Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2018. URL https://bokeh.pydata.org/en/latest/.

[7] Francis Borceux. *Handbook of categorical algebra: volume 1, Basic category theory*, volume 1. Cambridge University Press, 1994.

[8] Rita Borgo, Johannes Kehrer, David HS Chung, Eamonn Maguire, Robert S Laramee, Helwig Hauser, Matthew Ward, and Min Chen. Glyph-based visualization: Foundations, design guidelines, techniques and applications. In *Eurographics (state of the art reports)*, pages 39–63, 2013.

[9] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1121–1128, 2009.

[10] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. $D^3$ data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[11] Tai-Danae Bradley. What is applied category theory? *arXiv preprint arXiv:1809.05923*, 2018.

[12] Spencer Breiner, Peter Denno, and Eswaran Subrahmanian. Categories for planning and scheduling. *Notices of the American Mathematical Society*, 67(11), 2020.

[13] Jeffrey D Camm, James J Cochran, Michael J Fry, and Jeffrey W Ohlmann. *Data visualization: exploring and explaining with data*. Cengage Learning, 2021.

[14] Min Chen, Helwig Hauser, Penny Rheingans, and Gerik Scheuermann. *Foundations of data visualization*. Springer, 2020.

[15] Ran Chen, Xinhuan Shu, Jiahui Chen, Di Weng, Junxiu Tang, Siwei Fu, and Yingcai Wu. Nebula: A coordinating grammar of graphics. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):4127–4140, 2021.

[16] Furui Cheng, Yao Ming, and Huamin Qu. Dece: Decision explorer with counterfactual explanations for machine learning models. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1438–1447, 2020.

[17] Herman Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American statistical Association*, 68(342):361–368, 1973.

[18] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia . *Journal of Open Source Software*, 6(65):3349, 2021. doi: 10.21105/joss.03349. URL https://doi.org/10.21105/joss.03349.

[19] Jonas de Miranda Gomes and Luiz Velho. *Fundamentos da computação gráfica*. Impa, 2008.

[20] Dazhen Deng, Weiwei Cui, Xiyu Meng, Mengye Xu, Yu Liao, Haidong Zhang, and Yingcai Wu. Revisiting the design patterns of composite visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2022.

[21] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019. doi: 10.1017/9781108668804.

[22] Michael Friendly and Howard Wainer. *A history of data visualization and graphic communication*. Harvard University Press, 2021.

[23] Samuel Giovine. Compose.jl: Declarative vector graphics for julia, 2022. URL https://github.com/GiovineItalia/Compose.jl. Accessed: March 11, 2024.

[24] Pontus Granström. Diagrammar: Simply make interactive diagrams. Talk, 2022. URL https://www.thestrangeloop.com/2022/diagrammar-simply-make-interactive-diagrams.html.

[25] Christoph M Hoffmann. *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., 1989.

[26] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[27] Waqas Javed and Niklas Elmqvist. Exploring the design space of composite visualization. In *2012 ieee pacific visualization symposium*, pages 1–8. IEEE, 2012.

[28] Daniel C. Jones, Ben Arthur, Tamas Nagy, Mattriks, Shashi Gowda, Godisemo, Tim Holy, Andreas Noack, Avik Sengupta, Darwin Darakananda, Adam B, Iain Dunning, Simon Leblanc, Rik Huijzer, Keno Fischer, David Chudzicki, Morten Piibeleht, Alex Mellnik, Dave Kleinschmidt, Tom Breloff, Yichao Yu, Joey Huchette, Mike J Innes, inkyu, john verzani, Artem Pelenitsyn, Calder Coalson, Ciarán O'Mara, and Elliot Saba. Giovineitalia/gadfly.jl: v1.3.4, October 2021. URL https://doi.org/10.5281/zenodo.5559613.

[29] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. Data-driven guides: Supporting expressive design for information graphics. *IEEE transactions on visualization and computer graphics*, 23(1): 491–500, 2016.

[30] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[31] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 554–561, 2013.

[32] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[33] Po-Shen Lee and Bill Howe. Dismantling composite visualizations in the scientific literature. In *ICPRAM (2)*, pages 79–91. Citeseer, 2015.

[34] Tom Leinster. *Basic category theory*, volume 143. Cambridge University Press, 2014.

[35] Guozheng Li, Min Tian, Qinmei Xu, Michael J McGuffin, and Xiaoru Yuan. Gotree: A grammar of tree visualizations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[36] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[37] Zhicheng Liu, Chen Chen, Francisco Morales, and Yishan Zhao. Atlas: Grammar-based procedural generation of data visualizations. In *2021 IEEE Visualization Conference (VIS)*, pages 171–175. IEEE, 2021.

[38] Sehi LYi, Qianwen Wang, Fritz Lekschas, and Nils Gehlenborg. Gosling: A grammar-based toolkit for scalable and interactive genomics data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):140–150, 2021.

[39] Bruce H McCormick. Visualization in scientific computing. *Acm Sigbio Newsletter*, 10 (1):15–21, 1988.

[40] Andrew M McNutt. No grammar to rule them all: A survey of json-style dsls for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 29(1): 160–170, 2022.

[41] Honghui Mei, Yuxin Ma, Yating Wei, and Wei Chen. The design space of construction tools for information visualization: A survey. *Journal of Visual Languages & Computing*, 44:120–132, 2018.

[42] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on functional programming languages and computer architecture*, pages 124–144. Springer, 1991.

[43] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018.

[44] Bartosz Milewski. The dao of functional programming. https://github.com/BartoszMilewski/DaoFP, 2019. Accessed: 2024-11-10.

[45] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[46] Dominic Orchard and Alan Mycroft. Categorical programming for data types with restricted parametricity. *Unpublished note*, 2012.

[47] Deokgun Park, Steven M Drucker, Roland Fernandez, and Niklas Elmqvist. Atom: A grammar for unit visualizations. *IEEE transactions on visualization and computer graphics*, 24(12):3032–3043, 2017.

[48] Josh Pollock, Catherine Mei, Grace Huang, Daniel Jackson, and Arvind Satyanarayan. Bluefish: A relational grammar of graphics. *arXiv preprint arXiv:2307.00146*, 2023.

[49] Xiaoying Pu and Matthew Kay. A probabilistic grammar of graphics. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[50] Xiaoying Pu and Matthew Kay. How data analysts use a visualization grammar in practice. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–22, 2023.

[51] Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. Making data visualization more efficient and effective: a survey. *The VLDB Journal*, 29:93–117, 2020.

[52] Donghao Ren, Bongshin Lee, and Matthew Brehmer. Charticulator: Interactive construction of bespoke chart layouts. *IEEE transactions on visualization and computer graphics*, 25(1):789–799, 2018.

[53] Aristides G Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys (CSUR)*, 12(4):437–464, 1980.

[54] Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

[55] Nicolas P Rougier. *Scientific Visualization: Python+ Matplotlib*. 2021.

[56] David Salomon. *The computer graphics manual*. Springer Science & Business Media, 2011.

[57] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega: A visualization grammar. http://trifacta.github.io/vega, April 2014.

[58] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[59] Viliam Slodičák and Pavol Macko. Some new approaches in functional programming using algebras and coalgebras. *Electronic Notes in Theoretical Computer Science*, 279 (3):41–62, 2011.

[60] Karl Smeltzer, Martin Erwig, and Ronald Metoyer. A transformational approach to data visualization. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 53–62, 2014.

[61] David I Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012.

[62] David I Spivak. *Category theory for the sciences*. MIT Press, 2014.

[63] David I Spivak. Functorial aggregation. *arXiv preprint arXiv:2111.10968*, 2021.

[64] Moritz Stefaner and OECD. Oecd better life index. http://www.oecdbetterlifeindex.org/, 2012. Accessed on 14 Oct. 2012.

[65] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[66] Tableau Software. Tableau, 2024. URL https://www.tableau.com/. Version 2024.

[67] Alexandru C Telea. *Data visualization: principles and practice*. CRC Press, 2014.

[68] Theophanis Tsandilas. Structgraphics: Flexible visualization design through data-agnostic and reusable graphical structures. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):315–325, 2020.

[69] Fons van der Plas, Michiel Dral, Paul Berg, Π Γ , Rik Huijzer, Mikołaj Bocheński, Alberto Mengali, Connor Burns, Hirumal Priyashan, Benjamin Lungwitz, Jerry Ling, Eric Zhang, Felipe S. S. Schneider, Ian Weaver, Xiu zhe (Roger) Luo, Shuhei Kadowaki, Gabriel Wu, Timothy, Luis Müller, Zachary Moon, Supanat, Sergio A. Vargas, Rok Novosel, Jelmar Gerritsen, Vlad Flore, Jeremiah, Ciarán O'Mara, and Michael Hatherly. fonsp/pluto.jl: v0.20.3, October 2024. URL https://doi.org/10.5281/zenodo.14006492.

[70] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: interactive statistical visualizations for python. *Journal of open source software*, 3(32): 1057, 2018.

[71] Paul Vickers, Joe Faith, and Nick Rossiter. Understanding visualization: A formal approach using category theo. *IEEE transactions on visualization and computer graphics*, 19(6):1048–1061, 2013.

[72] Yun Wang, Haidong Zhang, He Huang, Xi Chen, Qiufeng Yin, Zhitao Hou, Dongmei Zhang, Qiong Luo, and Huamin Qu. Infonice: Easy creation of information graphics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[73] Michael L. Waskom. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi: 10.21105/joss.03021. URL https://doi.org/10.21105/joss.03021.

[74] Chris Weaver. Building highly-coordinated visualizations in improvise. In *IEEE Symposium on Information Visualization*, pages 159–166. IEEE, 2004.

[75] Hadley Wickham. ggplot2. *Wiley interdisciplinary reviews: computational statistics*, 3 (2):180–185, 2011.

[76] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of statistical software*, 40:1–29, 2011.

[77] Hadley Wickham. *Advanced r*. chapman and hall/CRC, 2019.

[78] Hadley Wickham and Maintainer Hadley Wickham. The ggplot package. *URL: https://cran. r-project. org/web/packages/ggplot2/index. html*, 2007.

[79] J. Widman. *Learning Functional Programming*. O'Reilly Media, 2022. ISBN 9781098111700. URL https://books.google.com.br/books?id=_c-AEAAAQBAJ.

[80] Leland Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer, 2012.

[81] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics*, 22(1): 649–658, 2015.

[82] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. Dataink: Direct and creative data-oriented drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[83] Ryan Yates and Brent A Yorgey. Diagrams: a functional edsl for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 4–5, 2015.

[84] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)*, 39(4):144–1, 2020.

[85] Brent A Yorgey. Monoids: theme and variations (functional pearl). *ACM SIGPLAN Notices*, 47(12):105–116, 2012.