UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GIUSEPPE MORONI RAMELLA

An introduction to Category Theory and a web-based visual editor for categorical concepts

Advisor: Prof. Dr. Rodrigo Machado

Porto Alegre
December 2018

"Running out of chips, you got no line in an eight bit town."

— AC/DC

AGRADECIMENTOS

Agradeço aos meus pais todo esforço voltado à minha educação, à minha irmã a assistência com as fotografias, aos familiares e amigos que me incentivaram a terminar este trabalho e às amizades feitas durante a graduação. Sou grato ao meu orientador por depositar em mim a responsabilidade de realizar este trabalho, espero que ele possa usá-lo durante as aulas e que este deixe as aulas ainda melhores. Regracio o povo brasileiro que, através de impostos me trouxe até aqui. Por fim, agradeço também a todos aqueles que nos permitiram chegar ao estado atual do conhecimento humano e espero que este trabalho possa contribuir para que outros o evoluam.

ABSTRACT

Category Theory is an area of mathematics with multiple applications in computing. However, because it is too abstract, students consider it very challenging, in particular when studying the discipline for the first time. One of the interesting aspects of Category Theory is the utilization of diagrams to express concepts and properties. Because of this, tools that allow the visualization and manipulation of the categorical concepts are of great value.

This work presents a survey on Category Theory and an auxiliary tool for teaching the subject, called CatViz. The survey aims to be an introduction to categorical concepts usually presented in a first course on Category Theory. The purpose of CatViz is to allow the edition of categorical diagrams and their manipulation through operations that reflect the axioms of the theory, making it possible to visualize demonstrations of properties on the tool. CatViz is being constructed using web technologies, with the purpose of being accessible and easily integrated with learning environments such as Moodle.

Keywords: Category Theory. Visual Editor. Web-based Technology.

Uma introdução a Teoria das Categorias e um editor visual baseado em Web para conceitos categoriais

RESUMO

Teoria das Categorias é uma área da matemática com diversas aplicações em computação. Contudo, por ser muito abstrata, estudantes a consideram bastante desafiadora, em particular ao estudarem a disciplina pela primeira vez. Um dos aspectos interessantes de Teoria das Categorias é a utilização de diagramas para expressar conceitos e propriedades. Por conta disso, ferramentas que permitam a visualização e manipulação dos conceitos categoriais são de extrema valia.
Este trabalho apresenta uma revisão de conceitos básicos de Teoria das Categorias e uma ferramenta de auxílio ao ensino desta disciplina, denominada CatViz. A revisão visa apresentar construções categoriais habitualmente estudadas em um curso introdutório a Teoria das Categorias. Ademais, o propósito de CatViz é permitir a edição de diagramas categoriais e a manipulação dos mesmos através de operações que reflitam os axiomas da teoria, tornando possível visualizar demonstrações de propriedades na ferramenta. CatViz está sendo construído utilizando tecnologias web, com o propósito de ser acessível e facilmente integrada a ambientes de ensino tais como o Moodle.

Palavras-chave: Teoria das Categorias. Editor Visual. Tecnologia Web.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AJAX	Asynchronous Javascript and XML
API	Application Programming Interface
CDN	Content Delivery Network
CERN	European Organization for Nuclear Research
CRUD	Create, Read, Update, and Delete
CSS	Cascading Style Sheets
D3.js	Data-Driven Documents
DHTML	Dynamic HTML
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
MVC	Model–view–controller
OOP	Object-oriented Programming
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language

LIST OF SYMBOLS

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{I}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, $\mathbb{O}$ and $\mathbb{S}$	naturals, integers, rationals, irrationals, reals, complexes, quaternions, octonions and sedenions
$f : A \to B$	function from set A to set B
$g \circ f$	composition of functions f and g
$a \in A$	a is an element of set A
$\forall$	for all
$\exists$	exists
F^k	set of k-ary functions
$(S_0, S_1 ..., S_m, F^n, F^{n-1}, ..., F^0)$	algebra with a collection of sets S_i and a collection of operations F^j with different arities
$*$	operation of group-like algebra
i	identity element of group-like algebra
G_*^{-1}	set of inverse elements of operation $*$ of group-like algebra
$(G, *)$	groupoid or semigroup
$(G, *, i)$	monoid
$(G, *, i, G_*^{-1})$	group or abelian group
$(x_1, x_2, ..., x_n)$	tuple with n elements
$+$	addition operation of ring-like algebra
$\times$	multiplication operation of ring-like algebra
R_+^{-1}	set of inverse elements of $+$ operation of ring-like algebra
0	identity element of addition operation of ring-like algebra
1	identity element of multiplication operation of ring-like algebra

(R, $+$, $\times$, R_{+}^{-1}, 0, 1)	ring-like algebra
$A \times B$	cartesian product of sets A and B
π_1 and π_2	projections of cartesian product
$A \cup B$	union of sets A and B
$A + B$	disjoint union of sets A and B
$A \subseteq B$	A is a subset of B
$A \supseteq B$	A is a superset of B
A/R	quotient set of relation R on A
$[x]_R$	an equivalence class of relation R, represented by element x
mod	modulo operation
$succ$	successor operation
$\neg P$	not P
$P \Rightarrow Q$	if P, then Q
$\perp$	bottom element
$\top$	top element
$\mid$	divisibility relation
$P \vee Q$	P or Q
$P \wedge Q$	P and Q
$h : A \to B$	homomorphism from algebra A to algebra B
$M = (\Sigma, S, s_0, \delta_{in}, F)$	state machine M
$C = \langle Ob_C, Mor_C, \partial_0, \partial_1, \imath, \circ \rangle$	category C
id_A	identity morphism of object A
$g \circ f$	composition of morphisms f and g
$M_{*,\mathrm{i}}$	monoid (M, $*$, i) seen as a category
$G_{*,\mathrm{i}}$	group (G, $*$, i, G_{*}^{-1}) seen as a category

f^{op}	dual morphism of f
$f : A \rightarrowtail B$	monomorphism from object A to object B
$f : A \twoheadrightarrow B$	epimorphism from object A to object B
$f : A \leftrightarrow B$	isomorphism from object A to object B
f^{-1}	inverse morphism of f
$!$	unique morphism
π_1 and π_2	projections
ι_1 and ι_2	immersions
(S, π_1, π_2)	span
$(A \times B, \pi_1,\ \pi_2)$	product of objects A and B
$(A + B, \iota_1, \iota_2)$	coproduct of objects A and B
$f \times g$	product of morphisms f and g
$f + g$	coproduct of morphisms f and g
$(K, \{k_1, \ldots, k_\mathsf{n}\})$ or $(K, k_1, \ldots, k_\mathsf{n})$	cone
$(L, \{l_1, l_2, ..., l_\mathsf{n}\})$	limit
(E, e)	equalizer
(E^{op}, e^{op})	coequalizer
$(A \times_C B, \pi_1, \pi_2)$	pullback of objects A and B to object C
$(A +_C B, \iota_1, \iota_2)$	pushout of objects A and B from object C
C^{op}	dual category of C
$C \times D$	product category
$(Fob, FMor)$ and F	functor
$g \circ_C f$	composition of C-morphisms f and g
$F\ X$	functor applied to object X
$F\ f$	functor applied to morphism f

CONTENTS

1 INTRODUCTION

Category Theory is a relatively recent area of mathematics used to study structure-preserving transformations in a very abstract way. It resorts heavily on visual representations (diagrams) for both definitions and proofs. Due to its abstract nature, Category Theory may be considered a challenging topic for students and teachers, even considering its visual appeal. We believe that this difficulty could be eased by the usage of visual editors, letting the student experiment with the diagram manipulations, allowed by the Category Theory axioms, to construct valid proofs.

This work has two objectives:

- present an introductory survey on Category Theory;
- develop a visual editor, named CatViz, for creating and manipulating categorical diagrams.

We expect CatViz to be used in both traditional and online courses on Category Theory, helping students to construct proofs based on diagram manipulation. Since CatViz is built upon web-based technology (HTML, CSS, JS, jQuery and D3.js), it can be easily integrated with modern learning platforms such as Moodle. The main characteristics of CatViz currently include:

- easy-to-configure user preference variables;
- text localization for worldwide availability;
- brush for drawings and annotations on top of diagrams;
- keyboard and mouse manipulation of data restricted to axioms and operations of Category Theory;
- undoing and redoing operations.

The following text is structured as follows. Chapter 2 reviews some basic mathematical background. Chapter 3 presents an introduction to Category Theory, focusing on the most common categorical concepts and definitions. Chapter 4 reviews the web-based technologies used in the construction of CatViz. Chapter 5 presents the CatViz visual editor, discussing its features and architecture. Finally, Chapter 6 concludes the text.

2 MATHEMATICAL BACKGROUND

This chapter reviews mathematical concepts that are going to be used throughout the text, covering ideas from Mathematics, like functions, relations and algebras, and state machines from Computer Science. We review definitions reproduced from many references, in particular (MENEZES; HAEUSLER, 2008), (MENEZES, 2005), (MENEZES; TOSCANI; LóPEZ, 2009), (MENEZES, 2011), (PIERCE, 1991), (BURRIS; SANKAPPANAVAR, 1981) and (BRONSHTEIN et al., 2007).

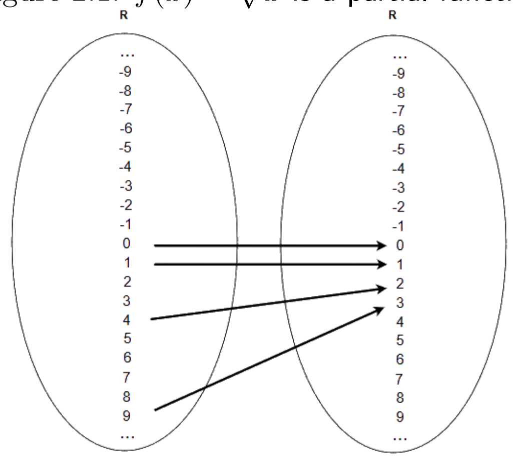
2.1 Set Theory and Functions

Definition 1 (Set Theory)**.** Theories, like Set Theory and Category Theory, contain a finite number of valid statements about something that they study. Set Theory was developed by Georg Cantor and studies sets, which are collections of elements. It is used in many branches of mathematics. A set contains elements that are grouped together due to some relationship among them and receives a name that emphasizes their relationship. At school, as a general rule, people learn about the different sets of numbers: naturals, integers, rationals, irrationals, reals and complex. These sets are named $\mathbb{N}$, $\mathbb{Z}$ (from the german word Zahlen, meaning numbers), $\mathbb{Q}$ (from the italian word quoziente, meaning quotient), $\mathbb{I}$, $\mathbb{R}$, $\mathbb{C}$, respectively. There are some abstractly higher sets of numbers: quaternions ($\mathbb{H}$, in honor of William Rowan Hamilton), octonions ($\mathbb{O}$) and sedenions ($\mathbb{S}$). But a set does not need to contain just numbers, any collection of abstract or concrete things can form a set. The set of fruits contains apples, bananas, cherries and all other things that are considered fruits. Of all sets that can be created, there is at least one that is special: the empty set which contains no element.

Definition 2 (Partial Function)**.** A partial function f from a set A to a set B, represented by $f : A \to B$, is a projection, also called mapping, of each element a from a subset of A (called *domain of definition* of f) to one, and only one, element b in B. It is denoted by $f(a) = b$. The sets A and B are called *domain* and *codomain* of f, respectively. The subset of the codomain that contains all outputs of the function is the *image* of f. We say f is *undefined* for elements of A that are not in the domain of definition of f.

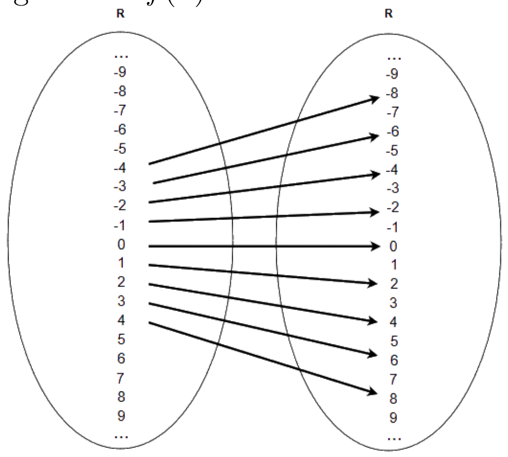
Example 1 (Partial Function)**.** Figure 2.1 indicates how $f : \mathbb{R} \to \mathbb{R}, f(x) = \sqrt{x}$ is only defined for $x \geq 0$ by showing the mapping of a portion of the domain of definition,

Figure 2.1: $f(x) = \sqrt{x}$ is a partial function



Source: The Author

Figure 2.2: $f(x) = 2x$ is a total function



Source: The Author

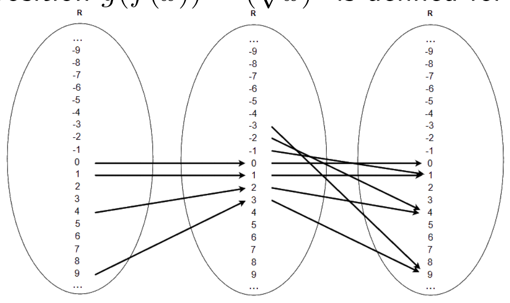
namely that of the integers.

Definition 3 (Total Function)**.** A total function f from A to B, denoted by $f : A \to B$ is a partial function whose domain of definition is equal to A, that is, it is defined for the whole set, or, equivalently, all elements of A have a projection in B.

Example 2 (Total Function)**.** Figure 2.2 indicates how $f : \mathbb{R} \to \mathbb{R}, f(x) = 2x$ is defined for all real numbers by showing the mapping of a portion of the domain of definition, namely that of the integers.

Definition 4 (Function Composition)**.** The composition of partial functions $f : A \to B$ and $g : B \to C$ is the function $g \circ f : A \to C$ such that $\forall a \in A$, $g \circ f(a) = g(f(a))$ if $f(a)$ and $g(f(a))$ are defined, otherwise it is undefined for a. The composition of total

Figure 2.3: The composition $g(f(x)) = (\sqrt{x})^2$ is defined for non-negative numbers



Source: The Author

functions $f : A \to B$ and $g : B \to C$ is the function $g \circ f : A \to C$ such that $\forall a \in A$, $g \circ f(a) = g(f(a))$.

Example 3 (Function Composition)**.** The composition $g(f(x))$ of functions $f : \mathbb{R} \to \mathbb{R}$, $f(x) = \sqrt{x}$ and $g : \mathbb{R} \to \mathbb{R}$, $g(x) = \sqrt{-x}$ is only defined for $x = 0$. The composition $g(f(x))$ of functions $f : \mathbb{R} \to \mathbb{R}$, $f(x) = \sqrt{x}$ and $g : \mathbb{R} \to \mathbb{R}$, $g(x) = x^2$ is only defined for $x \geq 0$. The composition $g(f(x))$ of functions $f : \mathbb{R} \to \mathbb{R}$, $f(x) = 2x$ and $g : \mathbb{R} \to \mathbb{R}$, $g(x) = x/2$ is defined for all $x \in \mathbb{R}$. Figure 2.3 indicates how $(\sqrt{x})^2$ is only defined for non-negative numbers by showing the mapping of a portion of the domain of definition, namely that of the integers.

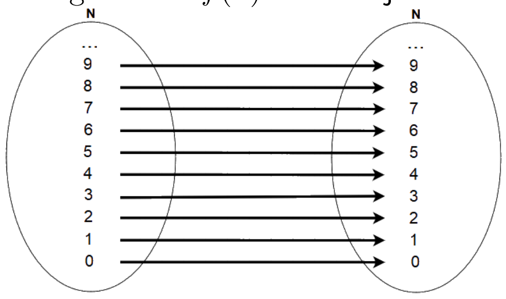
Definition 5 (Injective Function)**.** A function $f : A \to B$ is injective if for every $a_1, a_2 \in A$, if $a_1 \neq a_2$ then $f(a_1) \neq f(a_2)$. That is, a value in B is the result of f applied to necessarily at most one value in A.

Definition 6 (Surjective Function)**.** A function $f : A \to B$ is surjective if for every $b \in B$, there is an $a \in A$ such that $f(a) = b$. That is, the codomain and the image of f are equal.

Definition 7 (Bijective Function)**.** A function $f : A \to B$ is bijective if and only if there exists a $g : B \to A$ such that $g \circ f = id_A$ and $f \circ g = id_B$.

A function is bijective if it is injective and surjective. If a function is injective, then it has a left inverse function that goes from its image to its domain of definition. If it is surjective, then it has a right inverse that goes from its domain of definition to its image. With this, the inverse maps each element of the image to a unique element in the domain of definition. Consequently, the composition of a function with its inverse is the identity, a function that maps an element from the domain of definition to that element. Conversely, the composition of the inverse with the original function returns the identity of an element in the image.

Figure 2.4: $f(x) = x$ is bijective



Source: The Author

Bijective functions can map a unique element from one set to a unique element from another. These sets are then called *isomorphic* sets and, abstractly, one of them can be the representative of both and even a whole collection of isomorphic sets. Figure 2.4 shows how the identity function is bijective.

Example 4 (Injective, Surjective and Bijective Functions)**.** $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x$ is injective and surjective, and so, bijective, while $g : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not injective neither surjective and so, far from being bijective.

2.2 Algebras

Definition 8 (Algebra)**.** An algebra, also called algebraic structure, ($\mathbb{S}_0$, $\mathbb{S}_1$, .., $\mathbb{S}_m$, $\mathbb{F}^n$, $\mathbb{F}^{n-1}$, ..., $\mathbb{F}^0$) is a finite collection of sets $\mathbb{S}_i$ and a finite collection of finitary operations on the sets $\mathbb{F}^j$, where an n-ary operation is a function that takes n elements from the sets and returns one single element.

The most common algebras are limited to at most binary operations on one or two sets, that is: ($\mathbb{S}_0$, $\mathbb{F}^2$, $\mathbb{F}^1$, $\mathbb{F}^0$) and ($\mathbb{S}_0$, $\mathbb{S}_1$, $\mathbb{F}^2$, $\mathbb{F}^1$, $\mathbb{F}^0$). Functions in $\mathbb{F}^0$ are often called constants. Some of the terms used in algebras are extensions of terms learned at school. These include addition (binary operation represented by $+$), multiplication (binary operation represented by $\times$), identity (nullary operation represented by i where $e * i = e = i * e$), inverse element (unary operation represented by $^{-1}$ where $x * x^{-1} = i = x^{-1} * x$), associativity ($(x * y) * z = x * (y * z)$) and commutativity ($x * y = y * x$), for some binary operation $*$.

Depending on how specific the algebras are, different names are given to them: group-like structures have one binary operation, ring-like structures have two binary operations, called addition and multiplication, where multiplication distributes over addition, that is, $a \times (b + c) = (a \times b) + (a \times c)$ and $(a + b) \times c = (a \times c) + (b \times c)$. Furthermore, lattices are structures that have at least one of two special binary operations, called *meet* and *join* but can also have other binary operations, all operating on one set. The operations are closed, in that their result gives an element in the set, though this is usually not explicit when describing these algebras.

We review some important group-like algebras that will be referred to in future examples:

- Groupoid $(G, *)$: one set and one closed binary operation.
- Semigroup $(G, *)$: a groupoid where the operation is associative.
- Monoid $(G, *, i)$: a semigroup with an identity element.
- Group $(G, *, i, G_*^{-1})$: a monoid with inverse elements.
- Abelian group $(G, *, i, G_*^{-1})$: a group where the operation is commutative.

Table 2.1 summarizes the properties of the most common group-like algebras.

Table 2.1: Properties of group-like structures

Algebra	*Closed*	*Associative*	*Identity*	*Inverse Element*	*Commutative*
Groupoid	✓				
Semigroup	✓	✓			
Monoid	✓	✓	✓		
Group	✓	✓	✓	✓	
Abelian group	✓	✓	✓	✓	✓

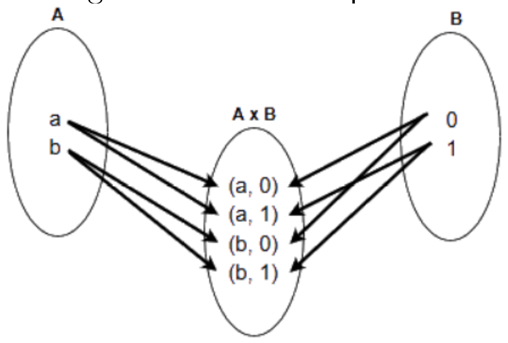
Source: The Author

2.3 Tuples and Operations on Sets

Definition 9 (Tuple)**.** A finite sequence of n components, called *ordered n-uple* consists of n objects (not necessarily distinct) in a fixed order. An n-uple in which the components are, in order, $x_1, x_2, ..., x_\mathrm{n}$ is denoted by $\langle x_1, x_2, ..., x_\mathrm{n} \rangle$ or $(x_1, x_2, ..., x_\mathrm{n})$. The order is important, so $(x, y) \neq (y, x)$, if $x \neq y$. Some sizes of tuples have names. A 2-uple can be called a *pair* and a 3-uple can be called a *triple*, but it is usual to call a higher sized tuple *n-uple* and the ordering is implicit by the representation, so a 2-uple (a, b) contains 2 elements and a comes before b.

Example 5 (Tuple)**.** The description of an algebraic structure like a ring (R, $+$, $\times$, R_+^{-1}, 0, 1) is an n-uple, in this case a 6-uple.

Figure 2.5: Cartesian product



Source: The Author

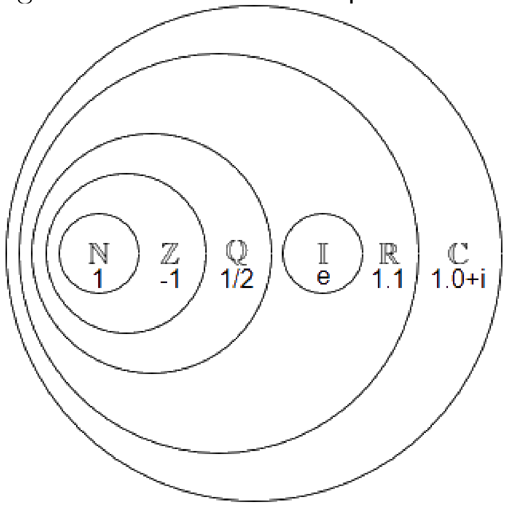
Definition 10 (Cartesian Product)**.** The cartesian product of sets A and B, denoted by $A \times B$ is defined as $A \times B = \{(a, b) \mid a \in A, b \in B\}$. In general, $A \times B \neq B \times A$. It consists of another set that contains pairs in which the first component is a copy of an element of A and the second component is a copy of an element of B. Two functions called projections, and denoted by π_1 and π_2, are associated with a cartesian product, where $\pi_1\colon A \times B \to A$ and $\pi_2\colon A \times B \to B$ are such that, for every $(a, b) \in A \times B$, $\pi_1((a, b)) = a$ and $\pi_2((a, b)) = b$. At school, some may find the cartesian product associated with immersions to it, instead of projections.

Example 6 (Cartesian Product)**.** Consider the sets $A = \{a, b\}$ and $B = \{0, 1\}$. Their cartesian product is defined as $A \times B = \{(a, 0), (a, 1), (b, 0), (b, 1)\}$. Applying π_1 and π_2 to each pair gives back A and B. Figure 2.5 shows an example of cartesian product of two small sets.

Definition 11 (Disjoint Union)**.** As seen at school, the union of sets A and B, denoted by $A \cup B$ defines that elements with the same identification on both sets are to be considered once in the resulting set. Disjoint union, however, defines that each element is different from their equivalent in the other set, so the operation becomes reversible and it is possible to reconstruct the original sets from the resulting set. It can be formally defined as $A + B = \{(a, A) \mid a \in A\} \cup \{(b, B) \mid b \in B\}$ and consists of pairs where each element is combined with an identifier of its original set. Two functions called immersions, the duals of projections, are associated with a disjoint union. $\iota_1\colon A \to A+B$ and $\iota_2\colon B \to A + B$ are such that, for every $a \in A$ and $b \in B$, $\iota_1(a) = (a, A)$ and $\iota_2(b) = (b, B)$.

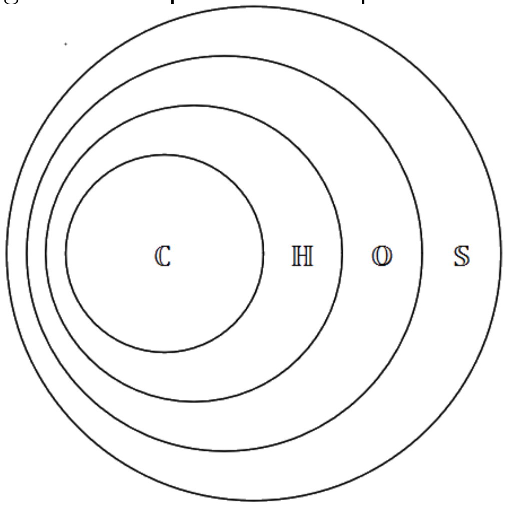
Example 7 (Disjoint Union)**.** Now consider the sets $A = \{a, b\}$ and $B = \{a, c\}$, $A + B = \{(a, A), (b, A), (a, B), (c, B)\}$. For the regular union, $A \cup B = \{a, b, c\}$.

Figure 2.6: Subsets of complex numbers



Source: The Author

Figure 2.7: Supersets of complex numbers

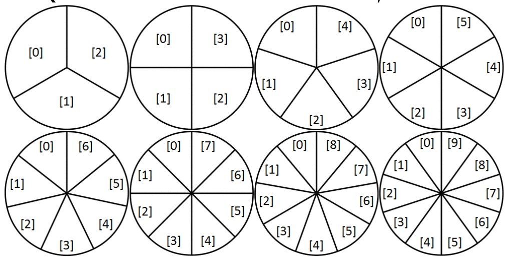


Source: The Author

Definition 12 (Subset and Superset)**.** If all elements of a set A are also elements of a set B, A is said to be a subset of B, denoted by $A \subseteq B$. On the other hand, B is said to be a superset of A, denoted by $B \supseteq A$.

Example 8 (Subset and Superset)**.** Of all numbers that exist, forming the set of numbers, some form the subset of complex numbers, which itself contains the subset of real numbers. This contains the subsets with all rational and irrational numbers. Rationals contain integers which contain naturals. On the other hand, sedenions are supersets of octonions, which are supersets of quaternions which in turn are supersets of complex numbers. Figures 2.6 and 2.7 show the subsets and supersets of complex numbers.

Figure 2.8: Quotient sets of $\mathbb{Z}$ mod n, for n from 3 to 10



Source: The Author

Definition 13 (Quotient Set)**.** For a given set A, the quotient set of A with respect to a relation R (the definition of relation is the next) is the set of equivalence classes of R in A, which are themselves sets. It is denoted by A/R. Formally, $A/R = \{[x]_R \mid x, y \in A \wedge x\ R\ y \Rightarrow x, y \in [x]_R\}$. All equivalence classes of A/R are disjoint, so performing a simple set union on them is enough to find A: $A = \bigcup_{[x]_R \in A/R}$. If the relation R is implied, $[x]_R$ is simply denoted by $[x]$.

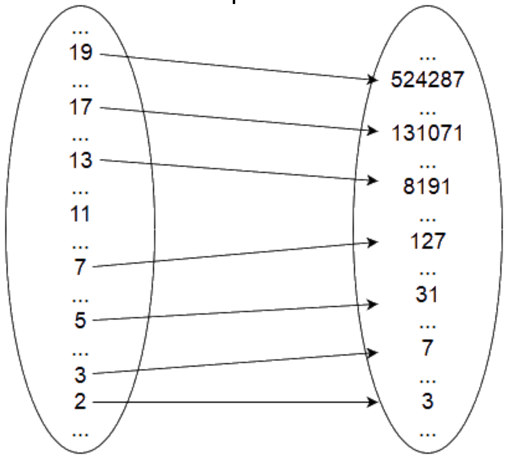
Example 9 (Quotient Set)**.** There is an operation frequently used in Computer Science called modulo operation and denoted by $mod\ n$. It takes an integer number as input and returns the remainder of the division of that number by another number n, previously set as the divisor, so for instance, 7 $mod\ 3 = 1$ because 7 = 3 × k + 1 and k = 2. By expanding this reasoning, $\mathbb{N}\ mod\ 2 = \{[0], [1]\}$, $\mathbb{N}\ mod\ 3 = \{[0], [1], [2]\}$, and in general, $\mathbb{N}\ mod\ n = \{[0], [1], [2], ..., [n-1]\}$. For integers, $\mathbb{Z}\ mod\ z = \{[0], [1], [2], ..., [z-1]\}$. When z = 3, $\mathbb{Z}\ mod\ 3 = \{[0], [1], [2]\}$ where $[0] = \{-9, -6, -3, 0, 3, 6, ...\}$, $[1] = \{-8, -5, -2, 1, 4, 7, ...\}$ and $[2] = \{-7, -4, -1, 2, 5, 8, ...\}$, composed of those integers that are achieved by adding or subtracting 3 from 0, 1 and 2, respectively. Figure 2.8 shows how the set of integer numbers is partitioned in classes according to the modulo operation.

2.4 Relations

Definition 14 (Relation)**.** A relation R of set A in set B is a subset of the cartesian product of those sets. It contains pairs $(a, b) \in R$ which can also be denoted by $a\ R\ b$. The relation $R \subseteq A \times B$ is defined by the domain A, the codomain B and the set of pairs (a, b) where $a \in A, b \in B$ and $a\ R\ b$. Modifications in any of these three parts produce a new relation.

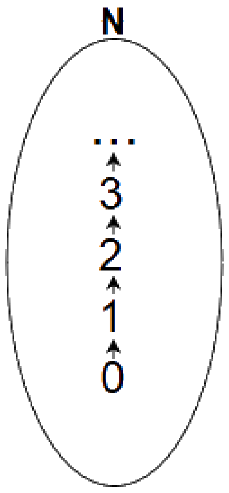
Example 10 (Relation)**.** The subset of natural numbers that are called *Mersenne primes*

Figure 2.9: Mersenne primes as a relation on $\mathbb{N}^2$



Source: The Author

Figure 2.10: Successor relation on natural numbers is an endorelation



Source: The Author

is given by the relation $p = 2^n - 1$ between the natural numbers that are the predecessor of a power of 2 and the natural numbers that are primes. It is a subset of the cartesian product $\mathbb{N} \times \mathbb{N}$ that contains $\{(2,3),(3,7),(5,31),(7,127),...)\}$. In a pair (a,b) of this relation, b is a Mersenne prime. Figure 2.9 shows how Mersenne primes can be defined as a relation on $\mathbb{N}^2$.

Definition 15 (Endorelation)**.** An endorelation R is a relation of set A in the same set A. It is a subset of the cartesian product $(R \subseteq A \times B)$ and is often denoted by (A, R).

Example 11 (Endorelation)**.** $(\mathbb{N}, succ)$ is an endorelation because every natural number has a successor, which is a natural number itself. Figure 2.10 shows how the successor relation is and endorelation on $\mathbb{N}$.

Definition 16 (Reflexive Endorelation)**.** An endorelation R is reflexive if every element is related to itself: $(\forall a \in A)\ (a\ R\ a)$.

Definition 17 (Irreflexive Endorelation)**.** An endorelation R is irreflexive if every element is not related to itself: $(\forall a \in A)\ \neg(a\ R\ a)$.

Definition 18 (Symmetric Endorelation)**.** An endorelation R is symmetric if, for every two elements in the set, if one relates to another, the other relates to the first: $(\forall a, b \in A)\ (a\ R\ b \Rightarrow b\ R\ a)$.

Definition 19 (Anti-Symmetric Endorelation)**.** An endorelation R is anti-symmetric if for every two elements in the set, if one relates with another and vice-versa, then they are equal, which implies that the endorelation cannot be inverted when the elements are different: $(\forall a, b \in A)\ (a\ R\ b \wedge b\ R\ a \Rightarrow a = b)$.

Definition 20 (Asymmetric Endorelation)**.** An endorelation R is asymmetric if for every pair of elements, if one relates with another, the other does not relate to the first: $(\forall a, b \in A)\ (a\ R\ b \Rightarrow \neg(b\ R\ a))$.

Definition 21 (Transitive Endorelation)**.** An endorelation R is transitive if for every three elements in the set, if the first relates to the second and the second to the third, then the first relates to the third: $(\forall a, b, c \in A)\ (a\ R\ b \wedge b\ R\ c \Rightarrow a\ R\ c)$.

Example 12 (Endorelations that are Reflexive, Anti-Symmetric and Transitive)**.** Endorelations that are reflexive, anti-symmetric and transitive are called *(non-strict) partial orders*. The endorelation $(\mathbb{N}, \leq)$ is reflexive (because a number is smaller or equal to itself), anti-symmetric (because a number is smaller or equal to another but that is not smaller or equal to the first) and transitive (because when a number is smaller or equal to another which is smaller or equal to a third, the first is also smaller or equal to the third).

The endorelation $(\mathbb{N}, =)$ is reflexive (because a number is equal to itself), anti-symmetric (because a number is equal to another, the other is equal to that, but both are the same number) and transitive (because when a number is equal to another which is equal to a third, the first is also equal to the third). Thus, $(\mathbb{N}, \leq)$ and $(\mathbb{N}, =)$ are partial orders.

Example 13 (Endorelations that are Reflexive, Symmetric and Transitive)**.** Endorelations that are reflexive, symmetric and transitive are called equivalence relations.

$(\mathbb{N}, mod\ 2)$, the parity of a number (if its even or odd), is a reflexive endorelation (because a number has the same parity as itself), anti-symmetric (because if a number has the same parity of another, the other also has the same parity of the first) and transitive (because when a number has the same parity of another which has the same parity of a third, the first has the same parity of the third).

The endorelation $(\mathbb{N}, =)$ is reflexive (because a number is equal to itself), symmetric (because a number is equal to another and the other is equal to the first, though they are the same) and transitive (because when a number is equal to another which is equal to a third, the first is also equal to the third, though they are the same). Thus, $(\mathbb{N}, \leq)$ and $(\mathbb{N}, =)$ are equivalence orders.

Definition 22 (Equivalence Classes)**.** Equivalence relations divide their original set in classes called equivalence classes. Such classes are denoted by one of their elements which most of the times best represents the relationship between the elements inside each class. It follows that the quotient set is the collection of all equivalence classes of that relation. For parity in naturals $(\mathbb{N}, mod\ 2)$, the equivalence classes are $[0]$ and $[1]$, so the quotient set of the relation is $\{[0], [1]\}$.

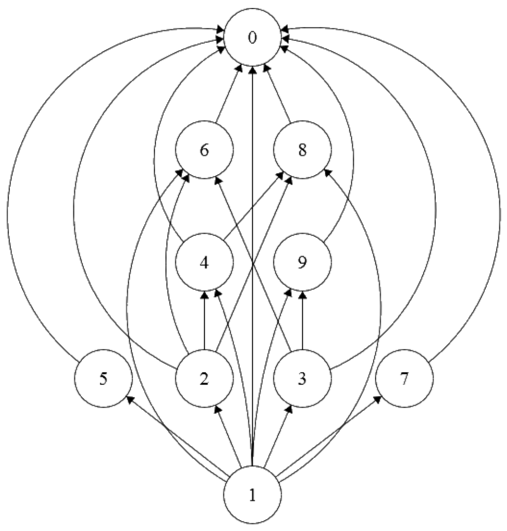
Example 14 (Endorelations that are Irreflexive and Asymmetric)**.** The endorelation $(\mathbb{N}, <)$ is irreflexive (because a number is not smaller than itself) and asymmetric (because if a number is smaller than another, this is not smaller than that).

Definition 23 (Partially Ordered Set (Poset))**.** A partially ordered set $(A, \leq)$ is such that A is a set and $\leq$ is a partial order in A. An element $a \in A$ is a minimal element if there is no other element $b \in A$ such that $b \leq a$. An element $a \in A$ is a maximal element if there is no other element $b \in A$ such that $a \leq b$. There can be more than one minimal or maximal element. An element $a \in A$ is the least element if for <u>every</u> other element $b \in A$, $a \leq b$. An element $a \in A$ is the greatest element if for <u>every</u> other element $b \in A$, $b \leq a$. The least and greatest elements can be denoted as $\perp$ and $\top$, respectively.

Example 15 (Partially Ordered Set (Poset))**.** The endorelation $|$, representing the natural division, containing pairs $a \mid b$ where the remainder of the division of b by a is strictly equal to 0, implying $b = ka$ (for some $k \in \mathbb{N}$), that is, b is a multiple of a:

- $|$ is reflexive: $a \mid a$, because $a = 1a$
- $|$ is anti-symmetric: if a divides b, $b = pa$, if b divides a, $a = qb$, then $b =$

Figure 2.11: Representation of divisibility poset (up to 9 only and reflexive arrows omitted)



Source: The Author

$p(qb) = (pq)b$, so $pq = 1$, therefore p and q must be 1, from that, $b = 1a = a$ and $a = 1b = b$

- $|$ is transitive: if a divides b, $b = pa$, if b divides c, $c = qb$, then $c = q(pa) = (qp)a$, so a divides c

Since it is divided by every other number, 0 is the greatest element. The opposite can be said of 1, so it is the least element.

Figure 2.11 shows the divisibility relation on a portion of the natural numbers.

Definition 24 (Connex Endorelation)**.** In a connex endorelation (A, R), every element of A relates to at least one other: $(\forall a, b \in A)\ ((a\ R\ b) \vee (b\ R\ a))$.

Example 16 (Connex Endorelation)**.** The endorelation $(\mathbb{N}, succ)$ is connex because any natural number is either a predecessor or a successor of another.

Definition 25 (Ordered Set)**.** A non-strict partial order relation $(A_1, \leq_1)$ is reflexive, anti-symmetric and transitive. A strict partial order relation $(A_2, \leq_2)$ is irreflexive, anti-symmetric and transitive. A connex non-strict relation $(A_3, \leq_3)$ is a connex and non-strict partial order relation. A connex strict relation $(A_4, \leq_4)$ is a connex and strict partial order relation. In all four order relations, the relation is anti-symmetric and transitive and their sets (A_1, A_2, A_3 and A_4) are called ordered sets.

Example 17 (Ordered Set)**.** Since it is the set of the relation $(\mathbb{N}, |)$ where $|$ is a partial order, $\mathbb{N}$ is an ordered set.

Definition 26 (Homomorphism)**.** An homomorphism $h : (A, *_1) \to (B, *_2)$ is a function that maps an algebra into another, preserving the structure and properties (associativity, commutativity, identity and inverse element). The algebraic structures (groups, rings, lattices, etc) must be the same for that to make sense, as the prefix *homo* indicates:

- homomorphisms of semigroups preserve associativity
- homomorphisms of monoids preserve associativity and identity elements
- homomorphisms of groups preserve associativity, identity and inverse elements
- homomorphisms of abelian groups preserve associativity, identity, inverse elements and commutativity

This concept has been generalized, under the name of morphism, to many other structures that do not have sets or are not algebraic. Later on we see that morphisms are the main concern in Category Theory.

Homomorphisms are composable like regular functions: $f : (A, *_1) \to (B, *_2), g : (B, *_2) \to (C, *_3) \Rightarrow g \circ f : (A, *_1) \to (C, *_3)$.

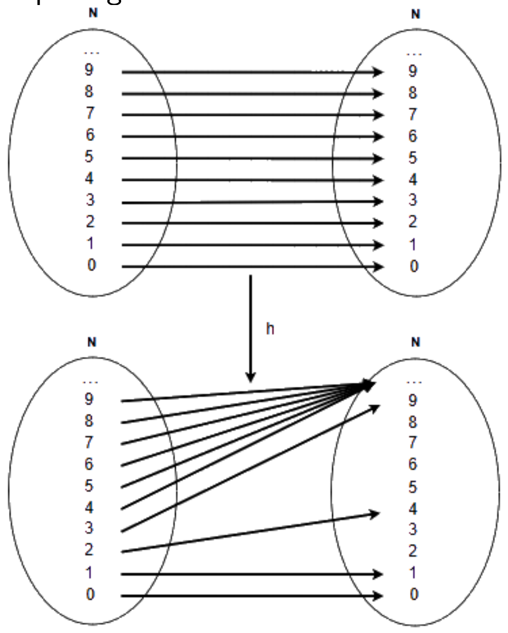
- A monomorphism is an injective homomorphism
- An epimorphism is a surjective homomorphism
- An isomorphism is a bijective homomorphism
- An endomorphism is an homomorphism from an algebra to itself
- An automorphism is an homomorphism which is also an isomorphism

Example 18 (Homomorphism)**.** $h : (G, +, 0) \to (G, \times, 1)$ is a homomorphism that maps addition's identity, which is zero, to multiplication's identity, which is one.

Definition 27 (Monotonic Function)**.** A monotonic function h (also called homomorphism of partially ordered sets) denoted by $h : (A, R) \to (B, S)$ is a function $h : A \to B$ such that $(\forall a_1, a_2 \in A)\ (a_1\ R\ a_2 \Rightarrow h(a_1)\ S\ h(a_2))$. It is a function that maps elements of A to elements of B, preserving the order structure.

Example 19 (Monotonic Function)**.** $h : (\mathbb{N}, x) \to (\mathbb{N}, x^2)$ is monotonic because for all natural numbers, when $x < y$, it is also true that $x^2 < y^2$. Figure 2.12 shows how the square function on natural numbers is monotonic.

Figure 2.12: Squaring a natural number is a monotonic function



Source: The Author

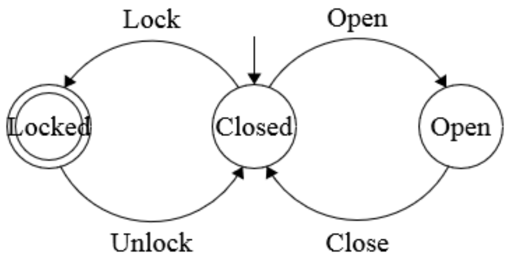
2.5 State Machines

Definition 28 (State Machine)**.** A (finite-)state machine, or finite automaton, is defined by the tuple $M = (\Sigma, S, s_0, \delta_{in}, F)$ where:

- Σ is the input alphabet that contains symbols the machine considers to be valid;
- S is the (finite) non-empty set of states;
- $s_0 \in S$ is the initial state, from which the computation starts;
- $\delta_{in} : S \times \Sigma \to S$ is the transition function, mapping input symbols and states to a subset of states, that is, it is a partial function δ whose inputs are pairs (s_1, σ) and whose outputs are s_2, for $s_1, s_2 \in S$ and $\sigma \in \Sigma$;
- $F \subseteq S$ consists of final states.

The user provides the machine with a tape containing a string (list of symbols) with (possibly valid) input symbols. The machine reads each symbol of the tape and transitions between states until it reaches the end of the tape, the current symbol is invalid, or there is not a transition defined for the current state and symbol. All the machine can say is if it accepted the input string or not by checking if its current state is a final state after all the computation is performed. Figure 2.13 shows a common representation of a state machine.

Figure 2.13: Example of a state machine with initial state, no final state and two transitions



Source: The Author

3 CATEGORY THEORY

In Set Theory, the most important objects of concern are sets and the elements that compose them. In Category Theory, objects, which are the equivalent of elements, are not that important: the relationships between the objects, represented by morphisms, are the elements that matter most.

A category is an abstract structure composed of objects and arrows between those objects. The fundamental property of a category is the composition operation on arrows. In particular, sets and functions can be seen, respectively, as the objects and the arrows of a specific category. Any modification on the objects, the arrows or the composition results in a new category. For instance, the replacement of total functions by partial functions changes the properties of the category.

For that reason, objects, arrows and composition are the three basic components of a category. The last one may have important interpretations: if a state machine is represented by a category where the states are the objects and transitions are the arrows, composition plays the role of the computation.

It is worth mentioning that these three components do not need to have structures resembling those used by Set Theory: for a partially ordered set viewed as a category, the elements of the set are objects but do not have any structure. Besides, the arrows link pairs of the relation and the composition is given by the transitivity of a relation of this kind.

Some structures can be a whole category or just a single object in a bigger category: in the category of partially ordered sets, the objects are the partially ordered sets, the arrows are monotonic functions and the composition is the composition of monotonic functions. Because of that, this category can be considered a category of categories. Likewise, monoids and groups are objects in the categories of monoids and groups, respectively, and looking deeper, a monoid and a group can each be a category, which often contain a single element and multiple arrows from and to itself.

Category Theory is relatively recent, being created by Samuel Eilenberg and Saunders Mac Lane in 1945. Since then, due to its abstractness, it influenced many areas as a revolutionary way to understand and approach things. Menezes and Haeusler (2008) point to five important characteristics of Category Theory that motivate its use in Computer Science:

Independence of Implementation. Objects and arrows are not necessarily collections of elements and functions as the usual interpretation in Set Theory. In fact, as the text will point out, sets can be called 0-categories and categories seen in this text are 1-categories. Thus, Category Theory is an adequate formalization to treat properties of objects and their relations in an abstract way, so abstract that categories may contain lower order categories as their objects.

Duality. In Category Theory, the main concepts and constructions happen in pairs, according to the generic notion of duality, which divides the work in half. This notion of duality, though obvious in Category Theory, often is not so intuitive in other theories. One example that will be shown in the text is that the concepts of cartesian product and disjoint union are not intuitively dual from the point of view of Set Theory, but are clearly dual when extended to Category Theory.

Inheritance of Results and Comparison of Expressivity of Formalisms. Category Theory lets us construct new categories from existing ones, as well as go from one kind of mathematical structure to another. In both cases, it is usually possible to inherit results already proven in a category, simplifying and dismissing the verification of some results. Also, transiting between mathematical structures allows for comparing the expressivity of apparently uncomparable formalisms by checking what is gained and lost after doing that.

Graphical Notation. A category is visually expressed with diagrams representing the relations between objects. That way, we can more easily understand its components and their relationship.

Expressivity of its Constructions. Constructions in Category Theory often have nothing similar in other theories, letting us formalize complex ideas in a simple way.

3.1 Category Theory in Computer Science

Menezes and Haeusler (2008) also point that the expressivity of categorical constructions is perhaps the main motivation for using Category Theory in Computer Science. As the systems get more complex, developing solutions for proposed problems becomes limited to our capacity to express them. We expect to advance more in knowledge when we use expressive formalisms. With those, we have better and more understandable specifications and simpler and clearer proofs.

3.2 Formal Definition of a Category

In the text that follows, it is easy to realize that the definition of a category is related to a particular algebra. It has two sets, that of the objects and that of the morphisms between pairs of objects, and four operations: Source, Target, Identity and Composition.

Definition 29 (Category)**.** A category C is a 6-tuple $C = \langle Ob_C, Mor_C, \partial_0, \partial_1, \imath, \circ\rangle$ where:

1. Ob_C is a collection (set, multiset, proper class, or even some other kind of collection) of Objects.

2. Mor_C is a collection of Morphisms (until now they were called Arrows).

3. $\partial_0, \partial_1 : Mor_C \to Ob_C$ are operations called Source and Target, respectively. A morphism f such that $\partial_0(f) = A$ and $\partial_1(f) = B$ is usually denoted by $f : A \to B$.

4. $\imath : Ob_C \to Mor_C$ is an operation called Identity such that each object A is associated to a morphism $id_A : A \to A$. The identity operation must satisfy the properties of identity, according to which for any morphism $f : A \to B \in Mor_C$, $f \circ id_A = f = id_B \circ f$ is true.

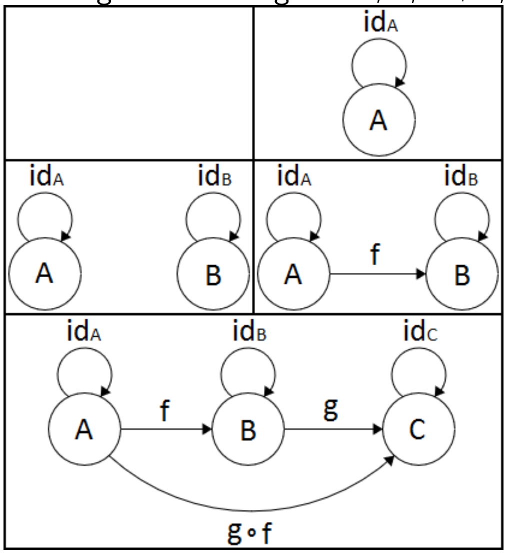
5. $\circ : Mor_C \times Mor_C \to Mor_C$ is a partial operation called Composition such that each pair of morphisms $\langle f : A \to B, g : B \to C\rangle$ is associated to a morphism $g \circ f : A \to C$. The composition operation must satisfy the associative property, according to which for any morphisms $f : A \to B$, $g : B \to C$ and $h : C \to D$ in Mor_C, $(h \circ g) \circ f = h \circ (g \circ f)$ is true.

Definition 30 (Small and Large Categories)**.** $C = (Ob_C, Mor_C, \partial_0, \partial_1, \imath, \circ)$ is a small category if Ob_C and Mor_C are sets, otherwise it is a large category.

3.3 Examples of Categories

Definition 31 (Category 0)**.** The smallest category is the category that has no objects and morphisms. Formally, it is defined by the 6-tuple $(\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$

Figure 3.1: Diagrams of categories 0, 1, $1+1$, 2 and 3



Source: The Author

Definition 32 (Category 1)**.** The second smallest category has just one object and its identity.

Definition 33 (Category 1+1)**.** The third smallest category has two objects and their identities.

Definition 34 (Category 2)**.** The fourth smallest category has two objects, their identities and a morphism between the objects.

Definition 35 (Category 3)**.** The category 3 has three objects, their identities, a morphism from the first to the second, another from the second to the third and their composition, from the first to the third.

A visual representation of the categories defined so far is presented in Figure 3.1.

Definition 36 (Category Set)**.** Set is defined by the 6-tuple $(Ob_{Set}, Mor_{Set}, \partial_0, \partial_1, \imath, \circ)$ where:

1. Ob_{Set} is the collection of all sets.
2. Mor_{Set} is the collection of all total functions.
3. $\partial_0, \partial_1 : Mor_{Set} \rightarrow Ob_{Set}$ are such that, for every $f : A \rightarrow B$, $\partial_0(f) = A$ e $\partial_1(f) = B$.
4. $\imath : Ob_{Set} \rightarrow Mor_{Set}$, that defines that each set A is associated to its identity function $id_A : A \rightarrow A$.
5. $\circ : Mor_{Set} \times Mor_{Set} \rightarrow Mor_{Set}$ is the partial operation of function composition.

Definition 37 (Category Cat)**.** Cat is defined by the 6-tuple $(Ob_{Cat}, Mor_{Cat}, \partial_0, \partial_1, \imath, \circ)$ where:

1. Ob_{Cat} is the collection of all small categories.
2. Mor_{Cat} is the collection of all morphisms between small categories, called functors (a formal definition of functor comes later on the text, in its subsection as part of the categorical constructions).
3. $\partial_0, \partial_1 : Mor_{Cat} \rightarrow Ob_{Cat}$ are such that, for every $f : A \rightarrow B$, $\partial_0(f) = A$ e $\partial_1(f) = B$.
4. $\imath : Ob_{Cat} \rightarrow Mor_{Cat}$, that defines that each category A is associated to its identity functor $id_A : A \rightarrow A$.
5. $\circ : Mor_{Cat} \times Mor_{Cat} \rightarrow Mor_{Cat}$ is the partial operation of functor composition.

Definition 38 (Category Pos)**.** Pos is defined by the 6-tuple $(Ob_{Pos}, Mor_{Pos}, \partial_0, \partial_1, \imath, \circ)$ where:

1. Ob_{Pos} is the collection of all posets $(A, \leq)$.
2. Mor_{Pos} is the collection of all monotonic functions for the ordered sets $(A, \leq)$ e $(B, \leq)$.
3. $\partial_0, \partial_1 : Mor_{Pos} \rightarrow Ob_{Pos}$ are such that, for every monotonic $f : (A, \leq) \rightarrow (B, \leq)$, $\partial_0(f) = (A, \leq)$ and $\partial_1(f) = (B, \leq)$.
4. $\imath : Ob_{Pos} \rightarrow Mor_{Pos}$, that defines that each $(A, \leq)$ is associated to its identity function $id_A : (A, \leq) \rightarrow (A, \leq)$.
5. $\circ : Mor_{Pos} \times Mor_{Pos} \rightarrow Mor_{Pos}$ is the partial operation of function composition.

Definition 39 (Category $P_{\leq}$)**.** A single poset can be seen as a category. The poset $(P, \leq)$ is represented by the category $P_{\leq}$ defined by the 6-tuple $(P, \leq, \partial_0, \partial_1, \imath, \circ)$ where:

1. P is the set of elements in the relation $\leq$.
2. $\leq$ is the set of pairs $(a\ R\ b)\ \in \leq$, for all $a, b \in P$.
3. $\partial_0, \partial_1 : \leq \rightarrow P$ are such that, for every pair $r = (a\ R\ b) \in \leq$, $\partial_0(r) = a$ and $\partial_1(r) = b$.
4. $\imath : P \rightarrow \leq$ is such that $id_a = (a\ R\ a)$.
5. $\circ : \leq\ \times\ \leq \rightarrow \leq$ is the operation of composition of the transitive relation $\leq$.

Definition 40 (Category Mon)**.** Mon is defined by the 6-tuple $(Ob_{Mon}, Mor_{Mon}, \partial_0, \partial_1, \imath, \circ)$ where:

1. Ob_{Mon} is the collection of all Monoids $(M, *, i)$.
2. Mor_{Mon} is the collection of all monoid homomorphisms $h : (M_1, *_1, i_1) \rightarrow (M_2, *_2, i_2)$ such that for any $a, b \in M_1$, $h(i_1) = (i_2)$ and $h(a *_1 b) = h(a) *_2 h(b)$.
3. $\partial_0, \partial_1 : Mor_{Mon} \rightarrow Ob_{Mon}$ are such that, for every homomorphism $\partial_0(h) = (M_1, *_1, i_1)$ and $\partial_1(h) = (M_2, *_2, i_2)$.
4. $\imath : Ob_{Mon} \rightarrow Mor_{Mon}$, that defines that each $(M, *, i)$ is associated to its identity function $id_M : (M, *, i) \rightarrow (M, *, i)$.
5. $\circ : Mor_{Mon} \times Mor_{Mon} \rightarrow Mor_{Mon}$ is the partial operation of function composition.

Definition 41 (Category $M_{*,\mathrm{i}}$)**.** A single monoid can be seen as a category. The monoid (M, $*$, i) is represented by the category $M_{*,\mathrm{i}}$ defined by the 6-tuple $(\{\Diamond\}, M, \partial_0, \partial_1, \imath, \circ)$ where:

1. $\{\Diamond\}$ is a singleton.
2. M is a set of morphisms.
3. $\partial_0, \partial_1 : M \rightarrow \{\Diamond\}$ are such that, for every $m \in M$, $\partial_0(m) = \Diamond = \partial_1(m)$.
4. $\imath : \{\Diamond\} \rightarrow M$ is such that $id_{\Diamond} = i$.
5. $\circ : M \times M \rightarrow M$ is the operation of composition.

Definition 42 (Category Grp)**.** Grp is defined by the 6-tuple $(Ob_{Grp}, Mor_{Grp}, \partial_0, \partial_1, \imath, \circ)$ where:

1. Ob_{Grp} is the collection of all groups (G, $*$, i, G_*^{-1}).
2. Mor_{Grp} is the collection of all group homomorphisms $h : (G_1, *_1, i_1, G_{*1}^{-1}) \rightarrow (G_2, *_2, i_2, G_{*2}^{-1})$ such that for any $a, b \in G$, $h(i_1) = (i_2)$ and $h(a *_1 b) = h(a) *_2 h(b)$.
3. $\partial_0, \partial_1 : Mor_{Grp} \rightarrow Ob_{Grp}$ are such that, for every homomorphism $\partial_0(h) = (G_1, *_1, i_1, G_{*1}^{-1})$ and $\partial_1(h) = (G_2, *_2, i_2, G_{*2}^{-1})$.
4. $\imath : Ob_{Grp} \rightarrow Mor_{Grp}$, that defines that each (G, $*$, i, G_*^{-1}) is associated to its identity function $id_G : (G, *, i, G_*^{-1}) \rightarrow (G, *, i, G_*^{-1})$.
5. $\circ : Mor_{Grp} \times Mor_{Grp} \rightarrow Mor_{Grp}$ is the partial operation of function composition.

Definition 43 (Category $G_{*,\mathrm{i}}$)**.** A single group can be seen as a category. The group (G, $*$, i, G_*^{-1}) is represented by the category $G_{*,\mathrm{i}}$ defined by the 6-tuple $(\{\Diamond\}, G, \partial_0, \partial_1, \imath, \circ)$ where:

1. $\{\lozenge\}$ is a singleton.
2. G is a set of morphisms.
3. $\partial_0, \partial_1 : G \to \{\lozenge\}$ are such that, for every $g \in G$, $\partial_0(g) = \lozenge = \partial_1(g)$.
4. $\imath : \{\lozenge\} \to G$ is such that $\imath(\lozenge) =$ i.
5. $\circ : G \times G \to G$ is the operation of composition.

Definition 44 (Endomorphism)**.** An endomorphism is a morphism that has the same object as source and target. If A is the object of the endomorphism f, $f : A \to A$ is such that $\partial_0(f) = \partial_1(f)$.

Example 20 (Endomorphism)**.** Besides the obvious example of the identity morphisms, any morphism of a monoid seen as a category is trivially an endomorphism.

Definition 45 (Duality)**.** In a category, duality is created by inverting all morphisms. The dual of a statement is achieved by replacing "domain" by "codomain", "codomain" by "domain" and "h is the composite of g with f" by "h is the composite of f with g" throughout it. Clearly, the dual of the dual is the original statement.

With the duality principle, if a statement is a consequence of the axioms, so is the dual statement. For complicated theorems, this is a handy way to have the dual theorem and so, no proof of it is needed. Figure 3.2 shows the diagrams of the duals of the first five categories. Table 3.1 summarizes the properties of the dual category and the equivalences between a category and its dual.

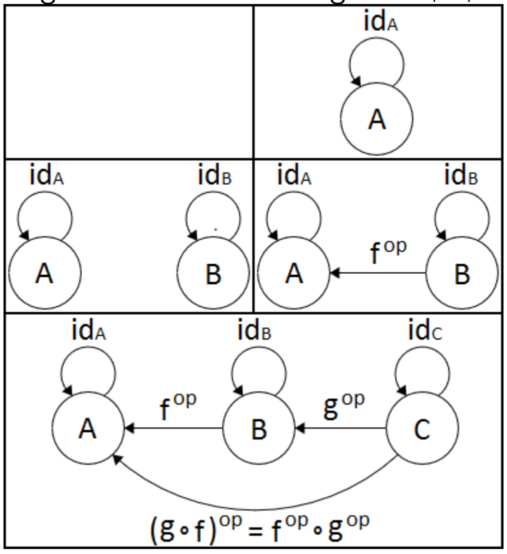
Table 3.1: Categorical duality

Category	*Dual*	*Equivalences*
$f : A \to B$	$f^{op} : B \to A$	$(f^{op})^{op} = f$
$\partial_0(f) = A$	$\partial_0(f^{op}) = B$	$\partial_0(f^{op}) = \partial_1(f)$
$\partial_1(f) = B$	$\partial_1(f^{op}) = A$	$\partial_1(f^{op}) = \partial_0(f)$
$g : B \to C$	$g^{op} : C \to B$	$(g^{op})^{op} = g$
$\partial_0(g) = B$	$\partial_0(g^{op}) = C$	$\partial_0(g^{op}) = \partial_1(g)$
$\partial_1(g) = C$	$\partial_1(g^{op}) = B$	$\partial_0(g^{op}) = \partial_1(g)$
$g \circ f : A \to C$	$(g \circ f)^{op} : C \to A$	$(g \circ f)^{op} = f^{op} \circ g^{op}$
$\partial_0(g \circ f) = A$	$\partial_0((g \circ f)^{op}) = C$	$\partial_0((g \circ f)^{op}) = \partial_1(g \circ f)$
$\partial_1(g \circ f) = C$	$\partial_1((g \circ f)^{op}) = A$	$\partial_1((g \circ f)^{op}) = \partial_0(g \circ f)$

Source: The Author

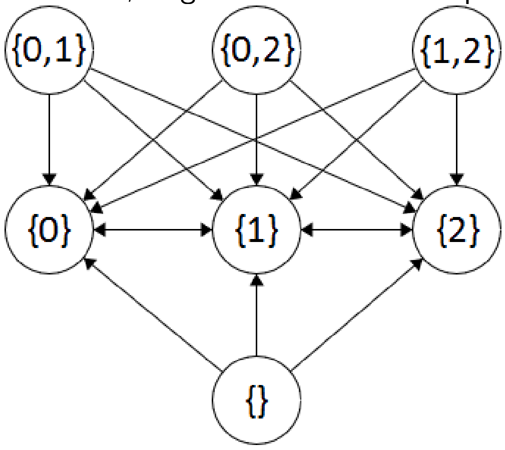
Definition 46 (Isomorphic Objects)**.** Two objects A and B in a category are said to be isomorphic if there is an isomorphism $f : A \leftrightarrow B$ (isomorphisms will be defined soon

Figure 3.2: Diagrams of duals of categories 0, 1, $1+1$, 2 and 3



Source: The Author

Figure 3.3: In Set, singleton sets are isomorphic objects



Source: The Author

on the text, in the subsection about monomorphism, epirmorphism and isomorphism as categorical constructions). In Category Theory, isomorphic objects are considered essentially the same object. Figure 3.3 shows a diagram of category Set that shows that singleton sets are isomorphic.

3.4 Diagrams

Definition 47 (Multiset)**.** A multiset, also called mset or bag, differs from a set by allowing for multiple instances of its element. Each element has an associate non-zero natural number, the multiplicity, which is just the number of occurences of the element.

A set is a special kind of multiset, with multiplicities always equal to 1.

Definition 48 (Diagram)**.** A diagram is a visual representation of a multiset of objects and morphisms of a category such that, for any morphism in the diagram, its source and target objects are in the diagram as well. This means that a diagram has no "floating" morphism.

Sometimes the composability of morphisms is relevant enough to appear in a diagram and sometimes not. When it is, they are called commutative diagrams and are often used to represent important concepts of Category Theory. It is said that the diagram commutes when all the routes between two objects reduce to the same morphism via composition. It is better to say "a diagram in a category" rather than "a diagram of a category" because not all elements (objects and morphisms) need to be in the diagram. Identities and compositions are often not represented in diagrams because they pollute it, making it harder to understand the structure of the category.

3.5 Categorical Constructions

Considering that, in Category Theory, objects are primitive entities, it is not possible to specify the universal constructions by analyzing solely the objects. They must be specified in terms of morphisms and composition. The first constructions are monomorphism, epimorphism and isomorphism, followed by initial and terminal object.

While epimorphism is the dual construction of monomorphism, terminal object is the dual construction of initial object. They generalize the notions of injective and surjective function and empty and unit set.

It is interesting to notice that, in Set Theory, it is not obvious nor intuitive the fact that the concepts of surjective function and unit set are the duals of injective function and empty set, respectively. Such fact shows the advantage of treating the properties by looking to the relationship of the objects (morphisms), and not to their internal structures (objects, which in this case, are set elements).

Similarly, isomorphism is a generalization of bijective function, so it is possible to extend, in a precise way, this types of functions to morphisms of any category based or not in sets. That is, it allows for defining precisely the notions of injection, surjection and bijection not only for set based category, but also to categories whose objects and morphisms do not have any relation to sets and functions, like categories 1, 1+1 and 2.

The next construction are product and coproduct, which generalize the notion of cartesian product and disjoint union of Set Theory, respectively.

It should be noticed that, when it comes to Set Theory, the fact that disjoint union is the dual concept of cartesian product (and vice-versa) is not an intuitive notion. Hence, not only the notion of duality "divides the work in half", but also correlates apparently distinct concepts. Such correlation is possible due to the abstraction of Category Theory when treating the properties "independently of implementation".

In an analogous manner, the concepts of initial object and terminal object and product and coproduct are universal constructions, that is, are defined in terms of existence and unicity of arrows. In particular, terminal and initial objects are particular cases of products and coproducts (zero-ary), respectively. Again, correlating apparently distinct concepts.

A product of a diagram constituted exclusively by objects (without arrows), when it exists, satisfies a special property, called Universal Property of Product, which, for any span of the same objects, is defined in terms of existence and unicity of an arrow.

The generalization of this notion for any diagrams (not limited to object, like in products) is called Limit, being Colimit its dual concept. Therefore, products and coproducts are special limits and colimits, respectively. The same can be affirmed to terminal and initial objects, respectively.

Beyond the general concept of limit and colimit, there are special limits (and colimits) called Equalizer (and Coequalizer) and Pullback (and Pushout), being the concepts of limits and colimits defined on top of diagrams, Cones and Cocones.

When Category Theory was originally proposed by its authors, a category was an auxiliary concept to the concepts of functor and natural transformation.

In reality, along its development as Mathematical Theory and as a mathematical tool applied to other sciences, this original perspective was modified when other categorical constructions proved themselves fundamentally important. Indeed, many developments can be done without the need for functors nor natural transformations.

In the meantime, as previously introduced, one of the main applications of Category Theory is the Unification of Mathematical Structures. Indeed, some of the most important functors are those that describe the passing from one type of mathematical structure to another. In this context, functors and natural transformations are of fundamental importance.

3.5.1 Monomorphism, Epimorphism and Isomorphism

Monomorphism, also called mono, is a generalization of an injective function in set theory. What defines an injective function can be redefined using the existence and equality of functions without referencing the elements of the sets. Next, the generalization of this redefinition to any category induces the concept of monomorphism.

A function $f : A \to B$ is called injective when, for any $a, b \in A$, if $f(a) = f(b)$, then $a = b$. In order to define an injective in terms of sets and functions, we replace "any $a, b \in A$" with "any morphisms $g, h : X \to A$". Since g and h are arbitrary, we can affirm that $g(x) = a$ and $h(x) = b$ and if f is injective, for any $x \in X$, if $f(g(x)) = f(h(x))$, then $g(x) = h(x)$, which implies that g and h are equal.

Now, a function $f : A \to B$ is injective if and only if, for any $g, h : X \to A$, if $f \circ g = f \circ h$, then $g = h$.

In a category, a morphism $f : A \to B$ is a monomorphism, denoted by $f : A \rightarrowtail B$, such that, for any object X and any $g, h : X \to A$, if $f \circ g = f \circ h$, then $g = h$. Because of this, f is said to be left cancellable.

With this new vision, we can now say that in category Set, every injective function is an example of monomorphism. Besides, in category 2, the morphism between the objects is a monomorphism, since identity is the only morphism that reaches the source object. It can be easily seen: in the definition "if $f \circ g = f \circ h$, then $g = h$", g and h are that identity. The same reasoning applies to all morphisms in a poset seen as a category.

Epimorphism, also called epi, is a generalization of a surjective function in set theory. It is the dual concept of a monomorphism.

In a category, a morphism $f : A \to B$ is an epimorphism, denoted by $f : A \twoheadrightarrow B$, such that for any object X and any $g, h : B \to X$, if $g \circ f = h \circ f$, then $g = h$. Because of this, f is said to be right cancellable.

As before, now we can say that in category Set, every surjective function is an example of epimorphism. Category Theory lets us see that, curiously, injection and surjection are, actually, dual concepts in Set Theory.

In category Mon, the homomorphism $(\mathbb{N}, +, 0) \to (\mathbb{Z}, +, 0)$ is an epimorphism.

Isomorphism, also called iso, is a generalization of a bijective function in set theory.

In a category, a morphism $f : A \to B$ is an isomorphism, denoted by $f : A \leftrightarrow B$

if there is an inverse $f^{-1} : B \to A$ such that $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$. It is easy to see that an isomorphism is auto dual, that is, its dual is itself.

Now, we can say that in category Set, every bijective function is an example of isomorphism. It is curious that not every morphism that is both mono and epi is iso, however, every iso is mono and epi, which contradicts the intuition created in Set Theory.

In category $M_{*,\mathrm{i}}$, every morphism is mono and epi, but only the identity morphism, which is the arrow of number 0, is iso. Its inverse is itself. In category $G_{*,\mathrm{i}}$, by the definition of group, every morphism has an inverse morphism, so, it is iso. Its inverse is itself.

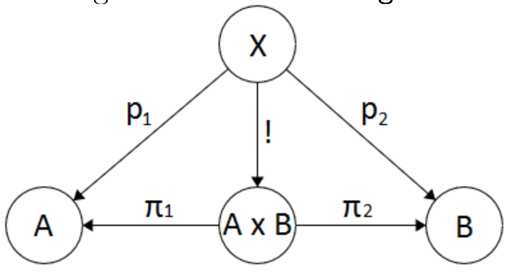
3.5.2 Initial Object, Terminal Object and Zero Object

Initial object is a generalization of empty set in set theory. In a category, the initial object is an object that is the source of exactly one arrow to each other object. Because of that, clearly not all categories have an initial object. Such object is often called 0. In the category of categories, the category 0, previously defined as the tuple (∅, ∅, ∅, ∅, ∅, ∅) is the initial object. In Set, the empty set is the initial object. In $\mathbb{N}_{\leq}$, the categorical equivalent of poset $(\mathbb{N}, \leq)$, 0 is the initial object. Formally, object 0 is an initial object if for any other object A, there is a unique morphism $!_0 : 0 \to A$.

Terminal object is a generalization of singleton set in set theory. It is the dual of initial object in that it is the target of exactly one arrow from each other object. It is often called 1 and not all categories have it. In the category of categories, the category 1 is the terminal object. Now it is possible to see that the naming of these categories was well thought. In Set, a singleton set is the terminal object up to isomorphism, that is, all singleton sets are terminal objects and there is an isomorphism between each. In $\mathbb{N}_{\leq}$, there is no terminal object. Formally, object 1 is a terminal object if for any other object A, there is a unique morphism $!_1 : A \to 1$.

A zero object of a category is an object that is simultaneously initial and terminal in the category. Any monoid (i, $*$, i) with a singleton set is a zero object of category Mon because !: (i, $*$, i) $\to$ (i', $*$, i') defined as !(i) $=$ i' must be unique and !: (i', $*$, i') $\to$ (i, $*$, i) defined as !(i') $=$ i is unique because monoid homomorphisms are total functions. The symbol ! is commonly used to represent a unique morphism.

Figure 3.4: Product diagram



Source: The Author

3.5.3 Product and Coproduct

Definition 49 (Span)**.** Three objects A, B and S and two morphisms p_1 and p_2 form a span if $p_1 : S \to A$ and $p_2 : S \to B$. It can be represented with a 3-uple (S, p_1, p_2).

Product is a generalization of cartesian product in set theory. Like with monos, epis and isos, it is necessary to adapt the definition of product so it uses morphisms only. A cartesian product offers two functions called projections, so it could be seen as the span $(A \times B, \pi_1, \pi_2)$, but this is still a broad definition since any span (X, p_1, p_2) where $p_1 : X \to A$ and $p_2 : X \to B$ could be a product. But also, cartesian product is a revertible operation because the projections are capable of returning the original sets. That is possible because the product has the precise number of elements ($|A \times B| = |A| \times |B|$). If another candidate span for product has less than $|A| \times |B|$ elements, it cannot give back the original sets and also cannot map all of its elements to another candidate. On the other hand, if another candidate for product has more than $|A| \times |B|$ elements, it cannot give back the original sets properly and also cannot $\underline{uniquely}$ map all of its elements to another candidate.

Definition 50 (Product)**.** A candidate span $(A \times B, \pi_1, \pi_2)$ is the product of A and B if, for all other candidate spans (X, p_1, p_2), where $p_1 : X \to A, p_2 : X \to B$:

- $! : X \to A \times B$ such that, for any $x \in X$, $!(x) = (p_1(x), p_2(x))$
- $p_1 = \pi_1 \circ !$
- $p_2 = \pi_2 \circ !$

Figure 3.4 shows the common diagram for a product.

A product is an universal construction because for all span candidates there is a unique ! morphism from them to it, making it a terminal object of a category formed by all span candidates. Also, a product is unique up to isomorphism. It is important to mention that a categorical product is not an object, but a 3-uple consisting of an object and two morphisms. Like in set theory, the product is a revertible operation. In Set, a product $(A \times B, \pi_1, \pi_2)$ consists of the cartesian product along with two projections $\pi_1 : A \times B \to A$ such that $\pi_1(a, b) = a$ and $\pi_2 : A \times B \to B$ such that $\pi_2(a, b) = b$ for any $(a, b) \in A \times B$. In $\mathbb{N}_{\leq}$, the product of two elements is the smaller.

Coproduct, also called sum, is a generalization of disjoint union in set theory. It is formally defined as a 3-uple $(A + B, \iota_1, \iota_2)$, $\iota_1 : A \to A + B$, $\iota_2 : B \to A + B$, such that $\iota_1(a) = (a, A)$ and $\iota_2(b) = (b, B)$, for every $a \in A$ and $b \in B$.

Formally, a candidate cospan $(A + B, \iota_1, \iota_2)$ is indeed the sum if, for all other candidate cospans (X, i_1, i_2), where $i_1 : A \to X, i_2 : B \to X$:

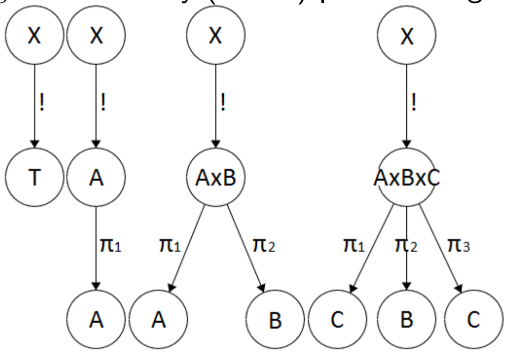
1. $! : A + B \to X$ such that, for any $x \in X$, $!((\iota_1(x), \iota_2(x))) = x$
2. $i_1 = ! \circ \iota_1$
3. $i_2 = ! \circ \iota_2$

Being the dual of the product, it is also an universal construction (but it is an initial object in the category of candidate cospans), unique up to isomorphism and represented by a 3-uple (with morphism reversed). Like in set theory, the coproduct is a revertible operation. In Set, a coproduct $(A + B, \iota_1, \iota_2)$ consists of the disjoint union along with two immersions $\iota_1 : A \to A+B$ such that $\iota_1(a) = (a, A)$ and $\iota_2 : B \to A+B$ such that $\iota_2(b) = (b, B)$ for any a ∈ A and b ∈ B. In $\mathbb{N}_{\leq}$, the coproduct of two elements is the greater.

The construction of (co)products can be extended to have a different number of projections. A nullary (co)product has a unique morphism (from)to an object and 0 (immersions)projections, but that is the definition of (initial)terminal object. An unary (co)product has a unique morphism (from)to an object and 1 (immersion)projection. That would be the object itself with the identity morphism. Another way to verify that, is to remember that, in Set, the binary product of A and B is a 2-uple (A, B). So an unary product would be an 1-uple (A) which has an isomorphism to A and a nullary product would be an 0-uple, or, the empty set, the terminal object. This idea of creating tuples in Set can be done for any size to better understand n-ary products. Figure 3.5 shows diagrams for the first n-ary products.

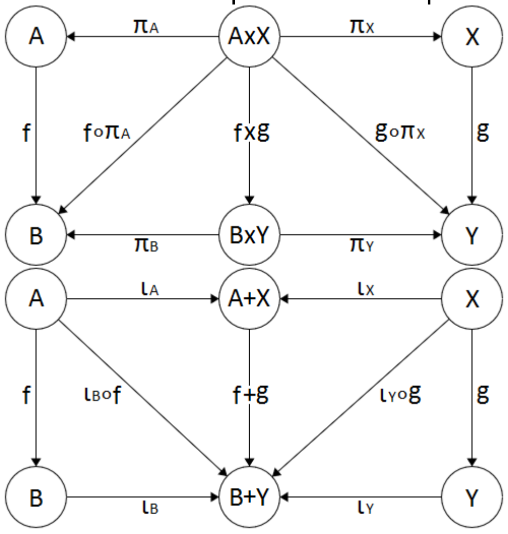
Another extension of products are the product $f \times g : A \times B \to B \times Y$ of two morphisms $f : A \to B$ and $g : X \to Y$ and the coproduct $f + g : A + B \to B + Y$ of two morphisms $f : A \to B$ and $g : X \to Y$. Figure 3.6 shows the diagrams for binary

Figure 3.5: N-ary (0 to 3) product diagrams



Source: The Author

Figure 3.6: Product and coproduct of morphisms diagrams



Source: The Author

product and coproduct.

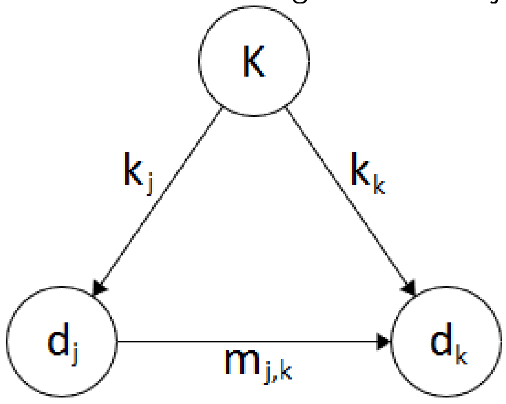
3.5.4 Cone and Cocone

For a diagram D with a multiset $\{d_1, d_2, ..., d_n\}$ of objects, a commutative cone, or simply cone, of D is an object K and a multiset of n morphisms k_i such that $k_i : K \to d_i$.

Formally, a cone of a diagram D of objects $\{d_1, d_2, ..., d_n\}$ is defined either by a 2-uple $(K, \{k_1, k_2, ..., k_n\})$ or by an (n+1)-uple $(K, k_1, k_2, ..., k_n)$ such that, for every $k_i : K \to d_i$ and $d_i \in D$, if there is a morphism $m_{j,k} : d_j \to d_k$ in D, then $k_k = m_{j,k} \circ k_j$, that is, the diagram commutes. Figure 3.7 shows the cone of a diagram with 2 objects.

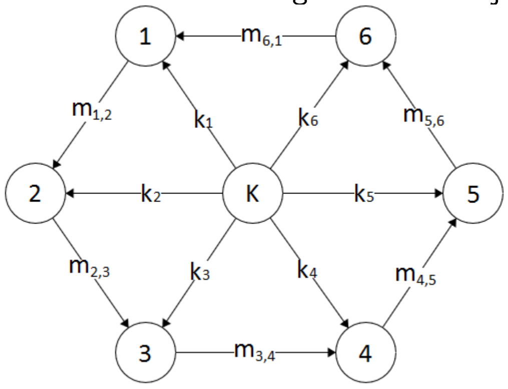
A diagram with 2 objects, a morphism from one to the other and a cone forms

Figure 3.7: Cone of a diagram with 2 objects



Source: The Author

Figure 3.8: Cone of a diagram with 6 objects



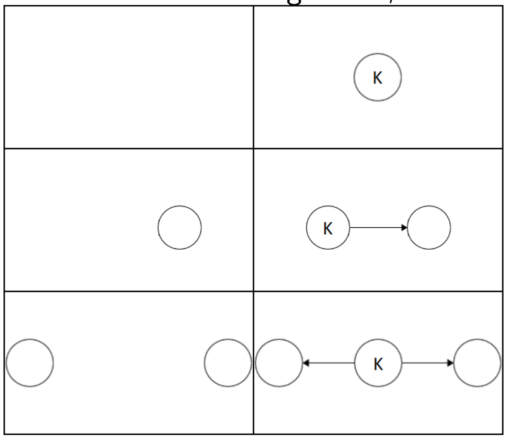
Source: The Author

a triangle. If there were 3 objects and a sequence of 2 morphisms from the first to the second and then from the second to the third, there would be 2 triangles. In general, if there were n objects and a sequence of n-1 morphisms similarly constructed, there would be n-1 triangles. The cone structure is named that way because in three dimensions it is possible to draw the objects and morphisms by forming a plane and letting the cone's object be the apex of a polygon whose shape resembles a cone. In two dimensions, the cone's object would be in the center and the other objects in the radius of a circle. Figure 3.8 shows why the cone gets its name.

It is easy to see that the cones of categories 0 and 1 are categories 1 and 2, respectively. Figure 3.9 shows this.

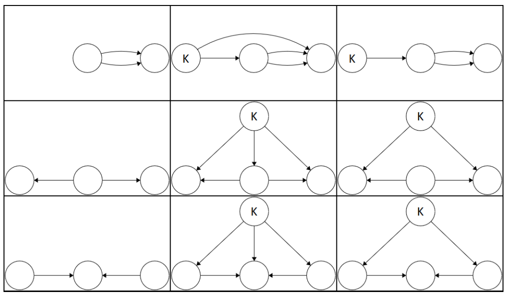
Almost every cone projection turns out to be a composition of other morphisms in the diagram. In that case, one can choose to show a projection like that or not. The reason is similar to that of not drawing identities and compositions due to pollution. Figure 3.10 shows this process of composition omission, which I find very dangerous to do in didactic books.

Figure 3.9: Cones of categories 0, 1 and $1+1$



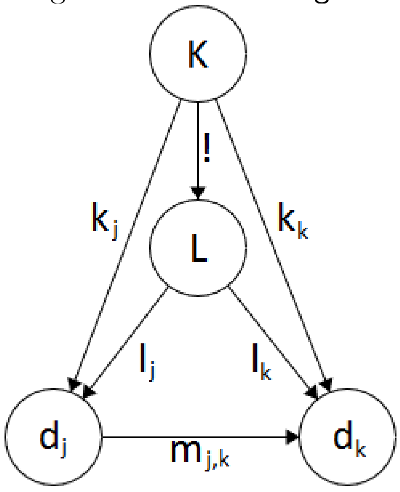
Source: The Author

Figure 3.10: Cones (colums: diagram, diagram with cone, diagram with projection omission)



Source: The Author

Figure 3.11: Limit diagram



Source: The Author

The cocone is the dual concept of the cone. Not much needs to be said about it except that all objects in the diagram are the source to a unique morphism to the cocone's object.

3.5.5 Limit and Colimit

The limit of a diagram is a cone that satisfies the universal property of the limit, that is, it contains all essential information of the cone structure, with no redundancy or lack of something. In the category of all cone candidates for some diagram D, the limit is the terminal object. Like any cone, it is formally defined as $(L, \{l_1, l_2, ..., l_n\})$ such that, for any cone candidate $(K, \{k_1, k_2, ..., k_n\})$ where $k_j = l_j \circ\, !$ and $k_k = l_k \circ\, !$. Figure 3.11 shows that the limit of a diagram is the best candidate cone. Dually, the colimit is the best cocone.

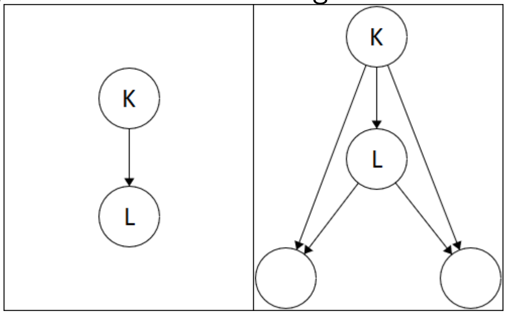
One can notice that the limits of categories 0 and 1+1 are equivalent to the definition of terminal object and product, respectively.

One can notice that the limits of the empty diagram and that with exactly two objects, which have the shape of category 0 and 1+1, respectively, are equivalent to the definition of terminal object and product. Figure 3.12 shows this.

Also, since the colimit is the dual concept, initial object and coproduct are kinds of colimits.

After mono, epi and iso, we started presenting (and will continue to) several categorical constructions that rely on the idea of universal construction (represented

Figure 3.12: Limits of categories 0 and $1+1$



Source: The Author

by limits and colimits). Roughly speaking, the universal construction is composed of an object (and a morphism, in some cases) among all said "candidate" objects to be that construction. The universal construction is said to be the "best" candidate in the sense that it contains a morphism from other candidates to it (and dually, from it to the other candidates). If we see the collection of candidates as a category, the universal construction is the terminal object in that category (dually, initial). That unique morphism that makes it a terminal object is usually denoted by ! and is frequently the composition of other morphisms related to the universal construction being studied.

In the diagram of a product, in figure 3.4, we now see that X and $A \times B$ are candidates to be the product of A and B. $A \times B$ is considered the best candidate because it is the target of a unique morphism ! from (all) other candidates such that the composition of ! with one of the product's projections is equivalent to the equivalent projection of the other candidates.

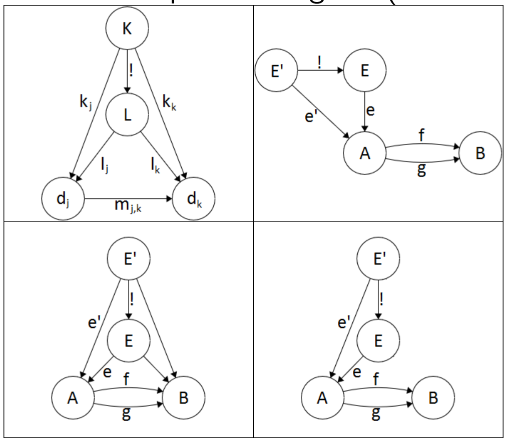
Similarly, in figure 3.11, we now see that the limit is the best cone because it is the target of a morphism from another cone, hence we say that that object satisfies the universal construction of the limit, as mentioned briefly in the beginning of this subsection.

3.5.6 Equalizer and Coequalizer

Equalizer is a generalization of a specific subset in set theory which contains the common elements of two functions such that, for each element of the subset, the image under both functions is the same. It does not have an equivalent operation in set theory.

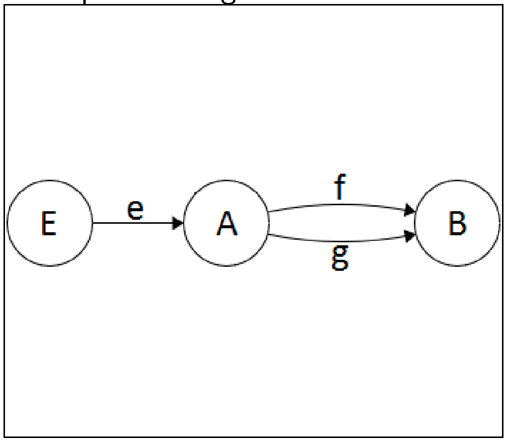
For a diagram D with two objects A and B and two morphisms $f, g : A \to B$, the equalizer is formally defined as the 2-uple (E, e) which consists of an object E and

Figure 3.13: Derivation of the equalizer diagram (anti-clockwise from top left)



Source: The Author

Figure 3.14: Equalizer diagram resembles monomorphism



Source: The Author

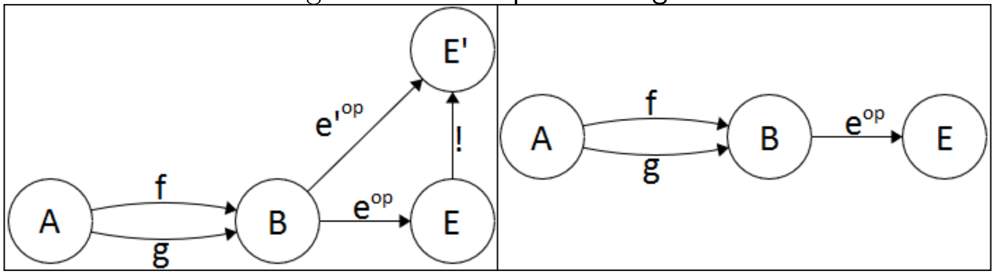
a morphism $e : E \to A$ which is the terminal object for all equalizer candidates (E', e'), $e' : E' \to A$.

Equalizer structure is derived from limit: the diagram morphism becomes two different ones and then morphisms to the second object are omitted because they are compositions of others. Figure 3.13 shows the derivation of the equalizer from limit.

The diagram without other equalizer candidates resembles a monomorphism diagram. Indeed, the projection of the equalizer (the second component of the 2-uple that represents it) is always a monomorphism. Figure 3.14 shows this.

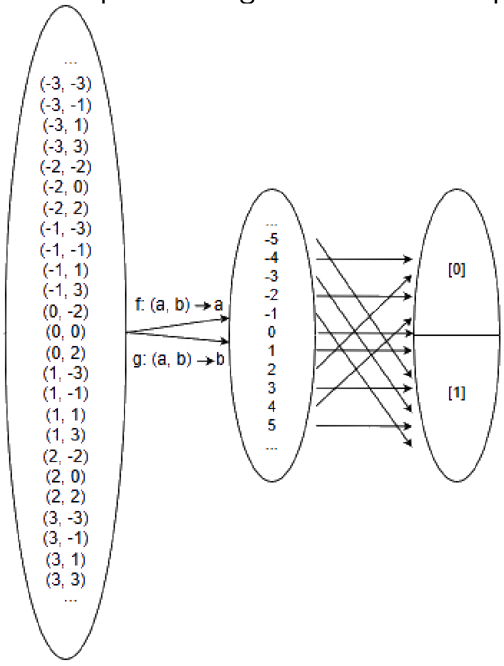
In Set, for two sets A and B and two parallel functions from A to B, an equalizer (E, e) is such that E is the greatest subset of A that makes f and g coincide, that is, $f(x) = g(x)$, for any $x \in E$. For $x, y, r \in \mathbb{R}$ and $f, g : \mathbb{R}^2 \to \mathbb{R}$ (it is implied that A is $\mathbb{R}^2$ and B is $\mathbb{R}$), if $f(x, y) = x^2 + y^2$ and $g(x, y) = r^2$ the equalizer of f and g is the set of pairs $(a, b) \in \mathbb{R}^2$ that define a circle of radius r.

Figure 3.15: Coequalizer diagram



Source: The Author

Figure 3.16: Coequalizer diagram of modulo 2 quotient set

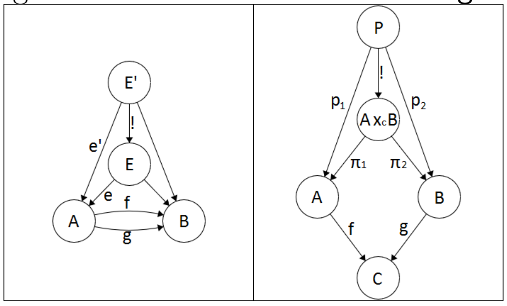


Source: The Author

The coequalizer produces a slightly different diagram that resembles that of the epimorphism. Object B is the source of the coequalizer's immersion e^{op} into E^{op}, being a dual, the immersion of the coequalizer is always an epimorphism. Figure 3.15 shows these affirmations.

In Set, for two sets A and B and two parallel functions from A to B where A is an equivalence relation R over B, a coequalizer (E^{op}, e^{op}) is such that E^{op} is the quotient set of R. For the modulo 2 relation over $\mathbb{Z}$, the coequalizer of $f, g : (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}$, such that $f((a, b)) = a$ and $g((a, b)) = b$ is the set of sets $\{[0] = \{-4, -2, 0, 2, 4, ...\}, [1] = \{-3, -1, 1, 3, 5, ...\}\}$. Figure 3.16 shows the coequalizer diagram of the quotient set for the modulo 2 relation.

Figure 3.17: Pullback derivation diagram



Source: The Author

3.5.7 Pullback and Pushout

Pullback is a generalization of the equalizer. Instead of an object being the source of two morphisms to the equalizer's object, in the pullback there are two objects and each is the source of one morphism and the pullback is the product of these objects. Inversely, an equalizer is a specific equalizer in which the two objects are the same and so the two morphisms have the same source. This way, the product gets a restriction. The restriction is that the second projection is a composition of the first, which is a mono. The pullback is a specific kind of limit. In the category of pullback candidates, the pullback is the terminal object. Figure 3.17 shows the difference between equalizer and pullback diagrams.

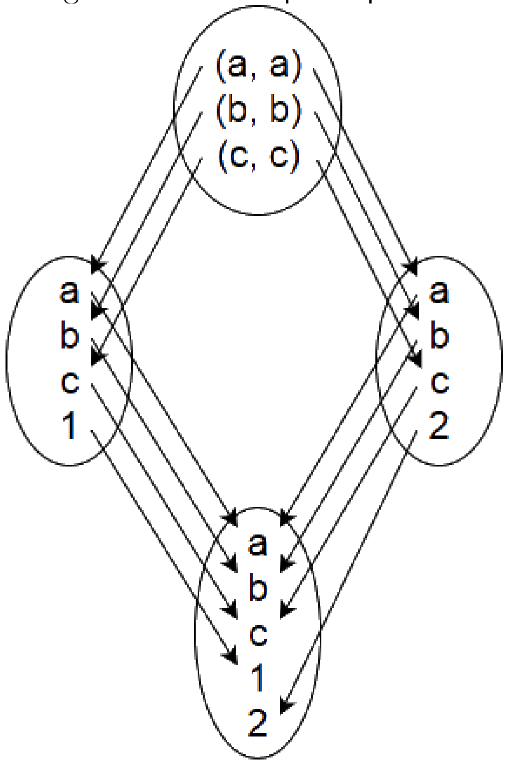
The formal definition of a pullback is a 3-uple $(A \times_C B, \pi_1, \pi_2)$ for two objects A and B and two morphisms $f : A \to C$ and $g : B \to C$ where $\pi_1 : A \times_C B \to A$ and $\pi_2 : A \times_C B \to B$ such that for any pullback candidate (P, p_1, p_2) and $! : P \to A \times_C B$ where $p_1 : P \to A$ and $p_2 : P \to B$, it is a fact that $p_1 = \pi_1 \circ !$ and $p_2 = \pi_2 \circ !$. The notation makes it clear that the pullback's object is the product of A and B and each has a morphism to C.

A pullback is an equalized product. If $(A \times B, \pi_1, \pi_2)$ is a product of A and B and (E, e) is an equalizer of parallel morphisms $f \circ \pi_1$ and $g \circ \pi_2$, then $(E, \pi_1 \circ e, \pi_2 \circ e)$ is a pullback of f and g.

In the category of all pullback candidates, the pullback is the terminal object with a unique morphism ! from each candidate to it. And, being a limit, the pullback has another projection from its object to C, the target of the two morphisms ($\pi_3 : A \times_C B \to C$), but this third projection is the composition of the two other projections with the two diagram's morphisms ($f \circ \pi_1 = \pi_3 = g \circ \pi_2$) and so, it is ignored for simplification.

In Set, the product $A \times_C B$ is a set of pairs (a, b) such that the projec-

Figure 3.18: Example of pullback



Source: The Author

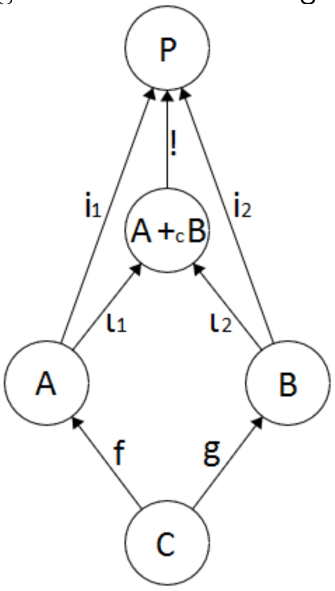
tions $\pi_1((a,b)) = a$ and $\pi_2((a,b)) = b$ and also $f(a) = g(b)$. All of this is equivalent to $f(\pi_1((a,b))) = g(\pi_2((a,b)))$. For sets $A = \{x_1, x_2, ..., x_n, y_1, y_2, ..., y_p\}$ and $B = \{x_1, x_2, ..., x_n, z_1, z_2, ..., x_q\}$ consisting of a collection of similar elements (x_i) and a collection of different elements (y_j and z_k), the pullback $A \times_C B = \{(x_1, x_1), (x_2, x_2), ..., (x_n, x_n)\}$ contains pairs of the similar elements. Figure 3.18 shows that the pullback resembles set intersection in Set Theory.

Pushout is a generalization of the coequalizer. Formally, it is defined by $(A +_C B, \{\iota_1, \iota_2\})$ where $\iota_1 : A \to A +_C B$ and $\iota_2 : B \to A +_C B$ for $f : C \to A$ and $g : C \to B$ such that for any pushout candidate (P, i_1, i_2), $i_1 : A \to P$, $i_2 : B \to P$, it is a fact that $i_1 =! \circ \iota_1$ and $i_2 =! \circ \iota_2$. Figure 3.19 shows the pushout diagram.

In Set, for two sets A and B and an equivalence relation R between A and B ($R \subseteq A \times B$), the pushout is a quotient set of $A +_R B$. Figure 3.20 shows that the pushout resembles quotient set in Set Theory.

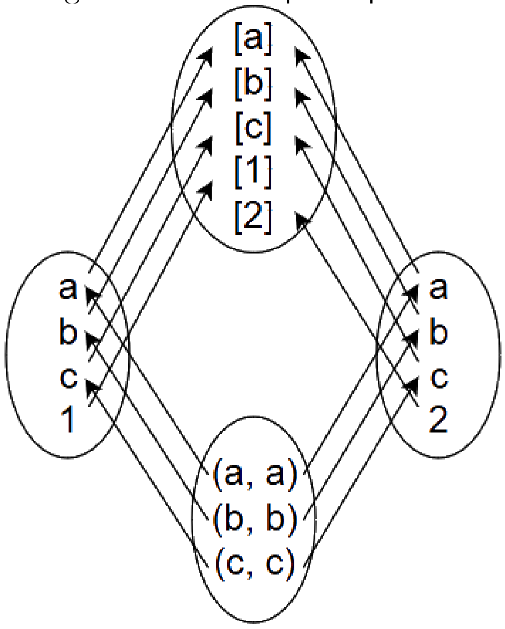
Dually, a pushout is a coequalized coproduct $(E^{op}, e^{op} \circ \iota_1, e^{op} \circ \iota_2)$ for a coproduct $(A + B, \iota_1, \iota_2)$ of A and B and a coequalizer (E^{op}, e^{op}) of parallel morphisms $\iota_1 \circ f$ and $\iota_2 \circ g$.

Figure 3.19: Pushout diagram



Source: The Author

Figure 3.20: Example of pushout



Source: The Author

3.5.8 Functor

Definition 51 (Dual Category)**.** For a category $C = \langle Ob_C, Mor_C, \partial_0, \partial_1, \imath, \circ\rangle$ and two morphisms $f : A \to B$ and $g : B \to C \in Mor_C$ such that $g \circ f$ is a composition, there is a dual categorial $C^{op} = \langle Ob_C, Mor_C, \partial_1, \partial_0, \imath, \circ^{op}\rangle$ where $f^{op} : B \to A$ and $g^{op} : C \to B$ such that $f^{op} \circ g^{op}$ is a composition.

Definition 52 (Subcategory)**.** A subcategory S of C contains some of the objects and morphisms of C. For it to be considered a category itself, it needs to have all the sources and targets of its morphisms and also have all the compositions and identities.

Definition 53 (Product Category)**.** A product category $C \times D$ consists of:

- pairs (A, B) of objects from C and D, respectively, as objects;
- pairs (f, g) of morphisms from (A$_1$, B$_1$) to (A$_2$, B$_2$) as morphisms, where f: A$_1$ $\to$ A$_2$ is a morphism from C and g: B$_1$ $\to$ B$_2$ is a morphism from D;
- pairs of identities id$_{(A,B)}$ = (id$_A$, id$_B$);
- and compositions (f$_2$, g$_2$) o (f$_1$, g$_1$) = (f$_2$ o f$_1$, g$_2$ o g$_1$) where f$_i$ come from C and g$_i$ come from D.

Definition 54 (C-Object and C-Morphism)**.** A C-object is an object from category C. Similarly, a C-morphism is a morphism from category C. Any set is a Set-object, for example.

There are categories that can be seen as categories of categories because their objects can be considered categories themselves. Cat, Pos and Mon are the categories of small categories, posets and monoids, respectively.

When observing such categories strictly, mappings between their objects are view as morphisms, but if one zooms into the objects, they also have objects and morphisms.

A functor is a morphism between categories that preserves the categorical structure, similar to what a homomorphism does to abstract algebras. That is the word used to describe mappings between categories.

Back when this Theory was created, categories were an abstraction of mathematical structures, so what was studied were the homomorphisms between these structures, which now are functors, but it turned out that constructions inside categories are now recognized as equally important.

Because of this importance, there are several types of functors, below are some of the easiest to understand.

Definition 55 (Covariant Functor)**.** For categories $C = (Ob_C, Mor_C, \partial_{0_C}, \partial_{1_C}, \imath_C, \circ_C)$ and $D = (Ob_D, Mor_D, \partial_{0_D}, \partial_{1_D}, \imath_D, \circ_D)$, a covariant functor, or simply functor, is a homomorphism represented by a 2-uple $F = (F_{Ob}, F_{Mor})$ where $F_{Ob} : Ob_C \to Ob_D$ and $F_{Mor} : Mor_C \to Mor_D$ such that for C-objects A, B, C, X, C-morphisms $f : A \to B$ and $g : B \to C$ and D-object Y, $F_{Mor}(g \circ_C f) = F_{Mor}(g) \circ_D F_{Mor}(f)$. A functor preserves identities and composition, so $F_{Mor}(id_X) = id_{F_{Ob}(X)} = id_Y$. For readability, it is better to denote $F_{Ob}(X)$ and $F_{Mor}(f)$ as $F\ X$ and $F\ f$. It does not cause much trouble because it is easy to know if the input of the functor is an object or a morphism if objects are represented by uppercase and morphisms by lowercase.

A covariant functor can be more technically defined as $F\ f : (\partial_0(f)\ \to\ \partial_1(f)) \to (F\ \partial_0(f)\ \to\ F\ \partial_1(f))$ and $F\ g : (\partial_0(g)\ \to\ \partial_1(g)) \to (F\ \partial_0(g)\ \to\ F\ \partial_1(g))$, being also implied that $F\ g \circ f : (\partial_0(g \circ f)\ \to\ \partial_1(g \circ f)) \to (F\ \partial_0(g \circ f)\ \to\ F\ \partial_1(g \circ f))$, so sources and targets are equivalently mapped, including identity morphisms.

Since what matters in a category are the morphisms and not really the objects, a functor can be a synonym to F_{Mor} and F_{Ob} is just a consequence of the rules of identity and composition.

Definition 56 (Contravariant Functor)**.** Closely recalling the concept of dual category, a contravariant functor maps morphisms to morphisms that are in the opposite direction. It extends the concept of isomorphism to functors.

For morphisms f, g of category C, a contravariant functor F is such that the mapping of objects is the same as a covariant, but $F\ f : (\partial_1(f)\ \to\ \partial_0(f)) \to (F\ \partial_1(f)\ \to\ F\ \partial_0(f))$ and $F\ g : (\partial_1(g)\ \to\ \partial_0(g)) \to (F\ \partial_1(g)\ \to\ F\ \partial_0(g))$, being also implied that $F\ g \circ f : (\partial_1(g \circ f)\ \to\ \partial_0(g \circ f)) \to (F\ \partial_1(g \circ f)\ \to\ F\ \partial_0(g \circ f))$, so sources and targets are swapped in the mapping.

Covariant functors are simply called functors because a contravariant functor $F : C \to D$ is equivalent to a covariant functor $G : C^{op} \to D$ whose source is the dual category of the contravariant. The source category of a contravariant functor is assumed to be the dual of the one being studied.

Definition 57 (Identity Functor)**.** For an object A and morphism f of a category C, the identity functor $id_C : C \to C$ is such that $id_C\ A = A$ and $id_C\ f = f$.

Definition 58 (Composite Functor)**.** Like regular morphisms, functors can also be composed. For categories C, D, E and functors $F : C \to D, G : D \to E$, there is a functor $G \circ F : C \to E$ whose operation is associative. The identity functor represents the identity of functor composition.

Definition 59 (Dual Functor)**.** The dual functor of $F : C \to D$ is $F^{op} : C^{op} \to D^{op}$ which has an identical to F but causes restrictions when composing. Namely, for $F : C \to D$ and $G : D^{op} \to E$, $G \circ F$ is not possible, but $G \circ F^{op}$ and $G^{op} \circ F$ are.

Definition 60 (Faithful Functor)**.** Extending the concept of injection to functors, a functor $F : C \to D$ is faithful when for any parallel C-morphisms f and g, if $F\ f = F\ g$, then it is true that $f = g$.

Definition 61 (Full Functor)**.** Extending the concept of surjection to functors, a functor $F : C \to D$ is faithful when for any D-morphism $g : F\ A \to F\ B$, there is a C-morphism $f : A \to B$ so that $F\ f = g$.

Definition 62 (Endofunctor)**.** An endofunctor is a functor whose source and target are the same category. meaning that $F : C \to C$ where $F\ A = B$ and $F\ f = g$, for some C-objects A, B and C-morphisms f, g. The identity functor is the most obvious endofunctor.

Definition 63 (Constant Functor)**.** A constant functor maps every object from the source category to a single object $\Diamond$ in the target and every morphism from the source to $id_\Diamond$.

For object A, B in the source category C, the target category D and the terminal category 1 (1 object with identity) along with functors $! : C \to 1$ and $d_\Diamond : 1 \to D$ where $\Diamond$ is the result of $d_\Diamond$, a constant functor $\Delta_\Diamond$ is equivalent to the composition $d_\Diamond \circ !$.

Another way of saying that is that $\Diamond$ is the result of mapping the object from the source category to the only object in 1 and then applying ! from the terminal category 1 to some element of the target category ($\Diamond\ = !\ A \circ d_\Diamond$). Without considering the whole process of composition we would think that the target category acts like a black hole, but the composition shows that actually the terminal category is a funnel that points to a single element of the target category.

This definition is good for understanding the composition, but it is based on the objects. We rather use morphisms for that, so for $f \in C$, $\Delta_\Diamond\ f : (\partial_0(f)\ \to\ \partial_1(f)) \to (\Delta_\Diamond\ \partial_0(f)\ \to\ \Delta_\Diamond\ \partial_1(f))$ where $\Delta_\Diamond\ \partial_0(f) = \Delta_\Diamond\ \partial_1(f)$.

Definition 64 (Diagonal Functor)**.** A (binary) diagonal functor of category C is the functor $\Delta : C \to C \times C$ so that $\Delta\ A = (A, A)$ and $\Delta\ f = (f, f)$ where $f : A \to B$ and $\Delta\ f : (A, A) \to (B, B)$.

Definition 65 (Inclusion Functor)**.** For a subcategory S of C, there is a pretty obvious functor from S to C that maps the objects and morphisms of S to themselves in C.

For $f \in S$, $inc\ f : (\partial_0(f) \rightarrow \partial_1(f)) \rightarrow (inc\ \partial_0(f) \rightarrow inc\ \partial_1(f))$ but $inc\ \partial_0(f) = \partial_0(f)$ and $inc\ \partial_1(f) = \partial_1(f)$.

Definition 66 (Forgetful Functor)**.** This functor "forgets" some structure of the source's objects so that is is not considered anymore and if the morphisms take that structure into consideration, after applying this functor, they no longer do it. A simple example is a functor that takes a collection of algebras and returns their sets, mapping algebras to sets and homomorphisms to functions, so the target category in this example is always Set:

- Pos to Set: posets $(A, \leq_1)$ and $(B, \leq_2)$, and a homomorphism $h_{AB} : (A, \leq_1) \rightarrow (B, \leq_2)$ are mapped to A, B and $f_{AB} : A \rightarrow B$, respectively
- Mon to Set: monoids $(M, *_1, i_1)$ and $(N, *_2, i_2)$, and a homomorphism $h_{MN} : (M, *_1, i_1) \rightarrow (N, *_2, i_2)$ are mapped to M, N and $f_{MN} : M \rightarrow N$, respectively
- Grp to Set: groups $(G, *_1, i_1, G^{-1}_{*_1})$ and $(H, *_2, i_2, G^{-1}_{*_2})$, and a homomorphism $h_{GH} : (G, *_1, i_1, G^{-1}_{*_1}) \rightarrow (H, *_2, i_2, G^{-1}_{*_2})$ are mapped to G, H and $f_{GH} : G \rightarrow H$, respectively

Definition 67 (Free Functor)**.** On the other hand, a free functor takes objects of a category to form a structure. As an example, consider the free functor that creates monoids from sets. Sets M and N, and function $f_{MN} : M \rightarrow N$ are mapped to $(M^*, \bullet, \varnothing)$, $(N^*, \bullet, \varnothing)$ and $h^*_{MN} : (M^*, \bullet, \varnothing) \rightarrow (N^*, \bullet, \varnothing)$, respectively, restricted to:

- $h^*_{MN}(\varnothing) = \varnothing$
- $h^*_{MN}(m) = f_{MN}(m)$
- $h^*_{MN}(m_1 m_2 ... m_n) = f_{MN}(m_1) f_{MN}(m_2) ... f_{MN}(m_n)$

These are called free monoids. Their operation, $\bullet$, is the concatenation and $\varnothing$, the empty set, is their identities.

Definition 68 (Multifunctor)**.** The source of a multifunctor is a product category. A bifunctor F: $C \times D \rightarrow E$ is a functor from a binary product category to another category.

Definition 69 (Strict Higher Categories)**.** A category of categories can also be seen as a 2-category, which consists of objects, morphisms (called 1-morphisms) and morphisms of morphisms (called 2-morphisms). This increase in level of abstraction can be extended to reach an n-category. Particularly, 0-categories are sets, which demonstrates that Category Theory evolves from Set Theory.

- Set is the 1-category of all small 0-categories (sets).
- Cat is the 2-category of all small 1-categories (categories).
- 2Cat is the 3-category of all small 2-categories.

3.5.9 Natural Transformation

Similar to functors, natural transformation are another higher abstraction over morphisms. In fact, they are morphisms between functors. Category Theory allows for these abstractions to be extended infinitely, if desired. Because these concepts are beyond the scope of this text, the reader is suggested to check the bibliography for a deeper approach on natural transformations and what comes next.

3.6 General Review

All universal constructions were seen with a bottom-up approach in the sense that each new construction was a generalization of the previous one. In the literature there is also the possibility of seeing them with a top-down approach, starting with limits towards terminal object. In this manner, each construction is a specification of the previous one. It is possible to verify that almost all constructions, including the very definition of object, morphism and identity morphism, are a specification of a limit/colimit, a cone/cocone or a morphism of a limit/colimit (projection/immersion):

- a cone is an object which is the source of many morphisms whose targets are objects of a diagram, which is simply a representation of a part of a category, or even it all;
- a limit is the best cone, so that from each similar cone there is a unique morphism whose target is the limit;
- a pullback is the limit of a diagram with three objects where one of them is the target of two morphisms whose sources are each of the other objects. It is the most general diagram described in this text;
- an equalizer is the limit of a diagram with two objects where one of them is the

target of two morphisms whose sources are the other object. It is a pullback where those objects that are sources become one object;

- a product is the limit of a diagram with a number of objects such that it is the source of morphisms whose targets are the other objects. It is an equalizer where the morphisms between the two objects can be ignored;
- a terminal object is the limit of an empty diagram. It is a product that has no projections;
- a monomorphism is the immersion of a limit whose source is an object that is target of two parallel morphisms so that their source is a same second object. In a pullback, if the morphisms that go from the two objects to the third are the same, then that repeated morphism is a monomorphism;
- an identity morphism is the projection of an unary product, which is a limit;
- an object is a cone of an empty diagram;
- a morphism is the projection of a cone of a diagram with one target object.

And dually:

- a cocone is an object which is the target of many morphisms whose sources are objects of a diagram;
- a colimit is the best cocone;
- a pushout is the colimit of a diagram with three objects where one of them is the source of two morphisms whose targets are each of the other objects;
- a coequalizer is the colimit of a diagram with two objects where one of them is the target of two morphisms whose sources are the other object. It is a pushout where those objects that are targets become one object;
- a coproduct is the colimit of a diagram with a number of objects such that it is the target of morphisms whose sources are the other objects. It is a coequalizer where the morphisms between the two objects can be ignored;
- an initial object is the colimit of an empty diagram. It is a coproduct that has no projections;
- an epimorphism is the projection of a colimit whose target is an object that is source of two parallel morphisms so that their target is a same second object. In a pushout, if the morphisms that go from the third object to the two objects are the same, then that repeated morphism is an epimorphism;

- an identity morphism is the immersion of an unary coproduct, which is a colimit;
- an object is a cocone of an empty diagram;
- a morphism is the immersion of a cocone of a diagram with one source object.

About functors and natural transformations, we can see that a category is such an abstract structure that even functors between two categories can form their own category, where the morphisms are the natural transformations. This gives rise to a higher level of abstraction that would let us define limits as the best cone of a diagram by using functors and natural transformations.

It appears to me that the top-down approach makes things clearer, so I recommend doing so, or at least performing a top-down analysis of what was learned bottom-up. When the bibliography is not clear enough, (NLAB, 2018) is an excellent online reference for the concepts presented in this chapter.

4 TECHNOLOGY REVIEW

The CatViz system intended to be easily accessible across platforms and easily integrated into web-based learning platforms such as Moodle. Because of this, CatViz was built upon web-based technologies. This chapter reviews the main technologies and concepts used in its construction.

4.1 Web-based Technologies

In 1989, Tim Berners-Lee created the main tecnologies behind the World Wide Web (WWW). He also wrote the first World Wide Web server, "httpd" and the first client program, "WorldWideWeb", which was both a browser and an editor. He wrote the first version of the HyperText Markup Language (HTML), a markup language with the capability for hypertext links. His initial specifications for URIs, HTTP, and HTML were refined and discussed in larger circles as Web technology spread. In October 1994, he founded the World Wide Web (W3C) Consortium at the Massachusetts Institute of Technology in collaboration with CERN, where the Web originated. The W3C is responsible for standardizing the World Wide Web specifications.

4.1.1 HTML

HTML (Hypertext Markup Language) is one of the core technologies for building Web pages. It provides the structure of the page and is the basis of building Web pages and Web Applications. With it, we can:

- publish online documents with headings, text, tables, lists, photos, etc;
- retrieve online information via hypertext links;
- design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc;
- include spreadsheets, video clips, sound clips, and other applications directly in the documents.

With HTML, the structure of pages are described using markup. The elements of the language label pieces of content such as paragraphs, lists, tables and so on with the so called tags. These tags are written between < and > signs. A website is always structured in a tree-like manner where the index.html is the head. This file is analogous to the main file used for implementing a regular software, except that it is visible to the user.

Example 21. Basic HTML Page with Obligatory Tags

```html
<html>
    <head>
    </head>
    <body>
    </body>
</html>
```

This HTML page would show nothing to the user and, depending on the browser, would either result in an error or warning due to the lack of a meta tag to indicate the charset used on the page. Also, if wanted, the page title goes between `<title>` and `</title>` tags. The `<html>` tag tells the browser that this is an HTML document, representing the root of the document and is the container for almost all other HTML elements. The <head> element is a container for all the head elements, including a title for the document, scripts, styles, meta information, and more. The <body> tag defines the document's body and contains all the contents of an HTML document, such as text, hyperlinks, images, tables, lists, etc.

4.1.2 CSS

One of the other core technologies, CSS (Cascading Style Sheets), provides the visual layout. It is the language for describing the presentation of Web pages, including colors, layout, and fonts. It is also used to adapt the presentation to different types of devices that vary in screen size. It is independent of HTML, which makes it easier to maintain sites, share style sheets across pages and use existing files. While HTML provides the structure, CSS provides the presentation of a page.

A style sheet is not obligatory to exist, but when it does, the browser reads it and formats the HTML document according to the information in it.

There are three ways of inserting a style sheet:

- with an **external style sheet**, it is possible to change the look of an entire website by changing just one file. Each page must include a reference to the external style sheet file inside the <link> element. The <link> element goes inside the <head> section;

- an **internal style sheet** may be used if the page needs a specific style. It is defined within the <style> tag, inside the <head> section of an HTML page;
- an **inline style** is used to apply a unique style for a single element by adding the style attribute to the element. It is applied to every occurrence of the tag in the file. If a property is defined multiple times for the same element in different style sheets, the value from the last read style sheet will be used.

Example 22. External Style Sheet

```
<link rel="stylesheet" type="text/css" href="filename1.css">
<link rel="stylesheet" type="text/css" href="filename2.css">
...
```

Example 23. Internal Style Sheet Tagging

```
<style>
body {
    background-color: linen;
}

h1 {
    color: maroon;
    margin-left: 40px;
}

...
</style>
```

Example 24. Inline Style Tagging

```
<h1 style="color:blue;margin-left:30px;">Header</h1>
```

4.1.3 JavaScript

A script is a program code that does not need pre-processing (commonly a synonym for compilation) before being run. In the context of a Web browser, scripting usually refers to program code written in JavaScript that is executed by the browser when a page is downloaded, or in response to an event triggered by the user.

Scripting can make Web pages more dynamic. For example, without reloading a new version of a page it may allow modifications to the content of that page, or allow content to be added to or sent from that page. The former has been called DHTML (Dynamic HTML), and the latter AJAX (Asynchronous JavaScript and XML). This additional interactivity makes Web pages behave like a traditional software application and these Web pages are often called Web applications.

Scripting offers a great opportunity to develop new interfaces and new user interactions. Over time, if these are greatly used by the community, they become new HTML tags.

Example 25. JavaScript

```
<html>
    ...
    <script src="scriptFile.js"></script>
    ...
    <script>
        //JavaScript code
    </script>
    ...
</html>
```

JavaScript code can be added either from an external file or between script tags in the file where it is used. Among the advantages that external files provide, they separate HTML and code, making it easier to read and maintain. Moreover, cached JavaScript files can speed up page loads. The source of an external script can be a file stored in the server or even in another place. In this case, the source is a full URL link.

4.1.4 DOM

The DOM (Document Object Model) is a tree-like hidden structure where the nodes are the elements of the page. It allows JavaScript scripts to dynamically access and update the content of HTML and CSS elements, making formally static documents highly dynamic.

Example 26. DOM

```
<html>
    ...
    <body>
        <p id="p1" class="p"></p>
        <p id="p2" class="p"></p>
        <script>
            document.getElementById("p1").innerHTML = "Hello World!
                ";
            document.getElementById("p2").innerHTML = "Hello World!
                ";
            document.getElementsByClassName("p")[0].innerHTML += "
                In both paragraphs!";
            document.getElementsByClassName("p")[1].innerHTML += "
                In both paragraphs!";
        </script>
    </body>
</html>
```

The getElementById() acts as a pointer to the element that contains a desired id. Similarly, getElementsByClassName() returns a collection of elements of a desired class. With them, it is possible to access elements defined in the HTML file through the JS script and change their content dynamically.

4.1.5 SVG

Scalable Vector Graphics, or SVG, is a W3C recommendation used to define vector-based graphics for the Web. Every element and every attribute in SVG code can be animated. SVG code can be directly written in HTML and CSS files or be dynamically created with JS code. It defines shapes that have attributes similar to HTML tags and visual attributes similar to CSS styles to those tags. SVG uses the painter's algorithm, so the first shapes in the code are rendered first and the last are drawn on top, overlapping those first. If the alpha channel of the color is not fully opaque, the resulting color is calculated with Alpha Blending. Good explanations for the painter's algorithm and Alpha Blending can be easily found.

To create a canvas, we need to create a *svg* tag with attributes width and height defined. SVG integrates with other W3C standards such as the DOM. If necessary, shapes can be grouped inside the *g* tag. One problem in the tags for SVG shapes is that equivalent attributes may have different names and this is not accused by the code interpreter. This can be undermined by defining JS scripts to create shapes statically or dynamically by passing values to those shapes' svg attributes via those scripts' parameters.

Example 27. SVG

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
<title>Flag of Brazil</title>
</head>
<body>
<script>
var svgNS = "http://www.w3.org/2000/svg";
var scale = 40;
var width = 20 * scale;
var height = 14 * scale;
var offset = 1.7 * scale;
var radius = 3.5 * scale;
document.body.outerHTML += '<svg xmlns=' + svgNS + ' id="canvas"
    width="' + width + '" height="' + height + '"></svg>';

function createRect(id, x, y, width, height, fill, stroke,
    stroke_width) {
  var rect = document.createElementNS(svgNS, "rect");
  rect.setAttributeNS(null, "id", id);
  rect.setAttributeNS(null, "x", x);
  rect.setAttributeNS(null, "y", y);
  rect.setAttributeNS(null, "width", width);
  rect.setAttributeNS(null, "height", height);
  rect.setAttributeNS(null, "fill", fill);
  rect.setAttributeNS(null, "stroke", stroke);
  rect.setAttributeNS(null, "stroke", stroke_width);
  document.getElementById("canvas").appendChild(rect);
}

function createPolygon(id, points, fill, stroke, stroke_width) {
  var polygon = document.createElementNS(svgNS, "polygon");
  polygon.setAttributeNS(null, "id", id);
  polygon.setAttributeNS(null, "points", points);
  polygon.setAttributeNS(null, "fill", fill);
  polygon.setAttributeNS(null, "stroke", stroke);
  polygon.setAttributeNS(null, "stroke", stroke_width);
  document.getElementById("canvas").appendChild(polygon);
}

function createCircle(id, cx, cy, r, fill, stroke, stroke_width) {
  var circle = document.createElementNS(svgNS, "circle");
  circle.setAttributeNS(null, "id", id);
  circle.setAttributeNS(null, "cx", cx);
  circle.setAttributeNS(null, "cy", cy);
  circle.setAttributeNS(null, "r", r);
  circle.setAttributeNS(null, "fill", fill);
  circle.setAttributeNS(null, "stroke", stroke);
  circle.setAttributeNS(null, "stroke", stroke_width);
  document.getElementById("canvas").appendChild(circle);
}

createRect("rect", 0, 0, width, height, "#009C3B", "none", 0);

var points = "";
points += (width / 2).toString() + "," + (offset).toString() + " ";
points += (width - offset).toString() + "," + (height / 2).toString
    () + " ";
points += (width / 2).toString() + "," + (height - offset).toString
    () + " ";
points += (offset).toString() + "," + (height / 2).toString() + " "
    ;
createPolygon("polygon", points, "#FFDF00", "none", 0);

createCircle("circle", width / 2, height / 2, radius, "#002776", "
    none", 0);
</script>
</body>
</html>
```

4.1.6 jQuery

jQuery is a fast, small, and feature-rich JavaScript library that makes things like HTML document (DOM) traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across browsers.

Example 28. jQuery

```
<html>
    <head>
        ...
        <script src="http://code.jquery.com/jquery-2.1.1.js"></script>
        ...
    </head>
    <body>
        ...
        <p id="p1" class="p"></p>
     <p id="p2" class="p"></p>
     ...
     <script>
            $("#p1")[0].innerHTML = "Hello World!";
            $("#p2")[0].innerHTML = "Hello World!";
            $(".p")[0].innerHTML += " In both paragraphs!";
            $(".p")[1].innerHTML += " In both paragraphs!";
        </script>
    </body>
</html>
```

The `$("#")` method points to an element by id. The `$("̇")` method creates a list of elements by class. Despite having a similar notation to regular DOM access methods, both produce jQuery objects. To access the desired element, it is a matter of using array access notation. By using jQuery, it is much easier and readable to select DOM elements by id or class, which is frequently done in CatViz.

4.1.7 CDN

CDNs (content delivery networks) are large networks of servers that store and deliver information (data, software, API code, media files, etc) making them easily accessible on the Web. When these nodes are strategically placed around the Internet they can increase the speed at which information is delivered. Windows Azure and Amazon CloudFront are examples of traditional CDNs. In an HTML file, links to CDN files must be referenced using an external reference like a regular external file.

Example 29. CDN

```
<html>
    <head>
        ...
        <script src="https://ajax.googleapis.com/ajax/libs/jquery
           /3.3.1/jquery.min.js"></script>
        <script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery
           -3.3.1.min.js"></script>
        <script src="http://code.jquery.com/jquery-2.1.1.js"></
           script>
        ...
    </head>
    ...
</html>
```

Google, Microsoft and even jQuery's webpage provide CDNs that deliver copies of jQuery versions. When users visit webpages that use Google's or Microsoft's CDNs to deliver jQuery, it is stored in the browser cache, which speed up loading time. Also, most CDNs make sure that user requests are served from the server closest to them, which also leads to faster loading time. On the other hand, if the user does not have it in cache and is not connected to the internet, he cannot load the library. That is why, in CatViz, there is a copy of the library.

4.1.8 D3.js

D3.js is a JavaScript library for manipulating documents based on data. It is similar to jQuery in that it is another approach to DOM manipulation. At first, it would be used to manage the visualization of objects and morphisms but that turned out to be more than necessary for this application. Actually, it could be replaced by jQuery but the objects that its methods return are much easier to work with.

Example 30. D3.js

```
...
var canvasPtr = d3.select("body").append("svg").attr("id", "canvas"
    ).attr("width", this.canvasWidth).attr("height",
    this.canvasHeight);
...
var object = d3.select("#circle"+id);
console.log(d3.select("#circle"+id) == $("#circle"+id)); //false
console.log(d3.select("#circle"+id).attr("cx") == $("#circle"+id).
    attr("cx")); //true
...
var morphism = d3.select(".curve"+id);
console.log(d3.select(".curve"+id) == $(".curve"+id)); //false
console.log(d3.select(".curve"+id).attr("d") == $(".curve"+id).attr
    ("d")); //true
...
var morphism = d3.select(".line"+id);
console.log(d3.select(".line"+id)[0] == $(".line"+id)); //false
console.log(d3.select(".line"+id)[0][0] == $(".line"+id)[0]); //
    true
...
```

In CatViz, mostly for debugging purposes, the user can view the values that the data structures store for objects and morphisms. In the method used to print this information, there is a loop going through the list of objects and morphisms. With D3.js, accessing the attributes is as readable as it would with jQuery but pointing just to the objects and morphisms' structures is not directly equivalent.

4.2 Object-Oriented Programming

The basic idea of OOP is that real world are expressed by so called objects, inside of which are related data and code, representing information about the modeled part of the world, along with its functionality. Data is grouped in attributes and code, in methods.

A real world person, for instance, would be represented by a class called Person and would contain different sorts of attributes and methods, depending on what needs to be reproduced. A first example to introduce OOP frequently uses name, age and gender. Also, an exemplary method would be to display the person's attributes.

4.2.1 Access Modifiers

Such methods are good cases for showing what access modifiers are. They allow for attributes and methods to be hidden to other classes, unless they are used to give an accessing interface for those other classes. Three access modifiers are commonly seen in most programming languages:

- private access to an attribute or method limits such access to the class that contains them. For safety reasons, attributes and auxiliary methods, those used by other methods inside the same class, are often private;
- on the other hand, access to a public attribute or method is unlimited. If another class needs to access another class's private attributes, that access would be possible through the use of methods called getters and setters, which act as interfaces from the outside world to the class. Getters return the current value of an attribute. Contrarily, setters change the value of an attribute;
- protected access is the least frequently used modifier, an explanation to its use would require much effort, as different terms, such as inheritance, would need to be defined, so it will not be found here.

4.2.2 Object-Oriented Programming in JavaScript

One of the possible ways to create classes in JS, the one used in CatViz, is to use a function as a class. Being interpreted, it is harder to fix bugs in JS, but if a class is defined this way, trying to access a private method crashes the application and so, time taken for bug correction is reduced. Also, creating classes with the keyword class makes it quite difficult and polluting to use the class' attributes.

Example 31. Object-Oriented Programming in JavaScript

```
function Class() {
 var private = "private";
 this.public = "public";
 this.getPrivate = function() {return private;}
 this.setPrivate = function(p) {private = p;}

 function getPrivateAux() {return private;};
 this.getPrivate2 = function() {return getPrivateAux();}
}

var instance = new Class();
console.log(instance.private); //undefined
console.log(instance.public); //public
console.log(instance.getPrivate()); //private
instance.setPrivate("visible");
console.log(instance.getPrivate2()); //visible
```

Trying to access a private attribute directly in JS results in an undefined value, to achieve the goal, a getter (and, if necessary a setter) method is required. Depending on the point of view, JS returning an undefined can be seen as a good thing or not. Due to the language being interpreted, and not compiled, that is what can be done to avoid crashing the application, but if the developer is not aware of that undefinition, or if he does not verify that the value is correct, it can lead to moments of despair searching for the source of errors.

In the example above, trying to call getPrivateAux() from outside the class results in a crash.

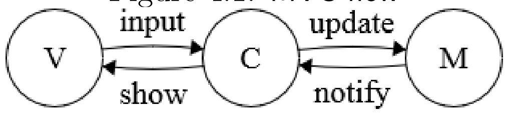
4.3 Event-Driven Programming

In event-driven programming, the software responds to external and internal events, which are stimuli from user input (clicks and key presses) or from methods of classes in the application. The program is assumed to contain a finite number of states and the events may cause transitions from one state to another. This kind of programming is particularly appropriate for real-time systems and is commonly associated with the MVC pattern. The problem with state-based modeling is that the number of possible states increases rapidly. In event-driven programming, user interaction trigger parts of the program to be executed. An event handler is a script that is implicitly executed in response to the appearance of an event.

4.3.1 Event-Driven Programming in JavaScript

In JavaScript, one way to handle events is to attach the addEventListener method to an element. There are dozens of available events that can be listened to, as the

Figure 4.1: MVC flow



Source: The Author

following code snippet presents.

Example 32. Event-Driven Programming in JavaScript

```
window.addEventListener("blur", function(e) {...});
window.addEventListener("focus", function(e) {...});

window.addEventListener("keypress", function(e) {...});
window.addEventListener("keydown", function(e) {...});
window.addEventListener("keyup", function(e) {...});

window.addEventListener("dragstart", function(e) {...});
$(document).bind("contextmenu", function(e) {...});

window.addEventListener("mousedown", function(e) {...});
window.addEventListener("mousemove", function(e) {...});
window.addEventListener("mouseup", function(e) {...});
```

4.4 MVC Pattern

When specifying the architecture of a system that allows user interaction via graphical user interface, one soon notices that the graphical representation and the stored data that it represents can be separated because the core of the system is based on functional requirements while the interface is subject to change and adaptation that do not modify that data.

MVC, short for Model–View–Controller, is a pattern frequently used in such architectures, dividing it into three components. The model contains the core functionality and data, the view displays it and the controller handles input. User interacts directly with the view, which triggers the controller to modify the model. After the modification, the model notifies the controller that it was completed, which makes the controller tell the view to show the new value to the user. Figure 4.1 shows this flow.

4.5 CRUD

CRUD is an acronym for the four basic operations associated not only with persistent storage (databases) but also with data structures during the execution of a software. C stands for the operation of creating new entries in a table. R is for reading the current value of data. U, for updating the value of data. D, for deleting data from the storage. In the context of CatViz, they are used to manage the data structures of objects and morphisms. These operations are closely associated with the MVC flow of CatViz as the user creates, reads, updates and deletes elements. Also, by using the right combination of keys to modify the system state stack, he undoes and redoes these operations. By providing a state history, the editor can be improved to support step-by-step proof of theorems assigned as exercises by the instructor and correction of these exercises.

5 THE CATVIZ TOOL

5.1 General Overview

The initial ideas for CatViz envisioned it as a visual editor with the following features:

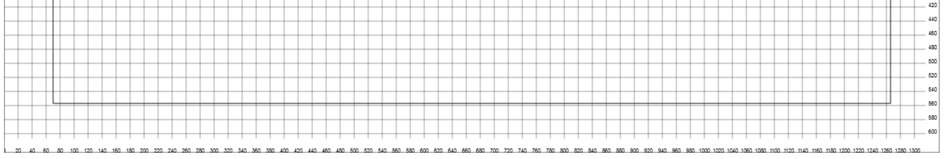
- free from any installation;
- usable in any platform;
- specialized in Category Theory.

These features make it different from previous projects that relied mostly in Java's visual capabilities.

Nowadays, the best and easiest way to achieve the first two requirements is for the application to be built upon web-based technologies. That is why CatViz is presented as a webpage. The user can download it once and run it just by opening the index.html file in a browser, or use it directly in the web. After that, the user is able to create diagrams for categorical concepts, as the third requirement demands. Figure 5.1 shows how the tool is presented to the user. The canvas contains a grid with a spacing of 10 pixels horizontally and vertically, with a numbering to visually show the location of the intersection of the grid's lines. There is also an inner area inside of which the objects are free to be moved.

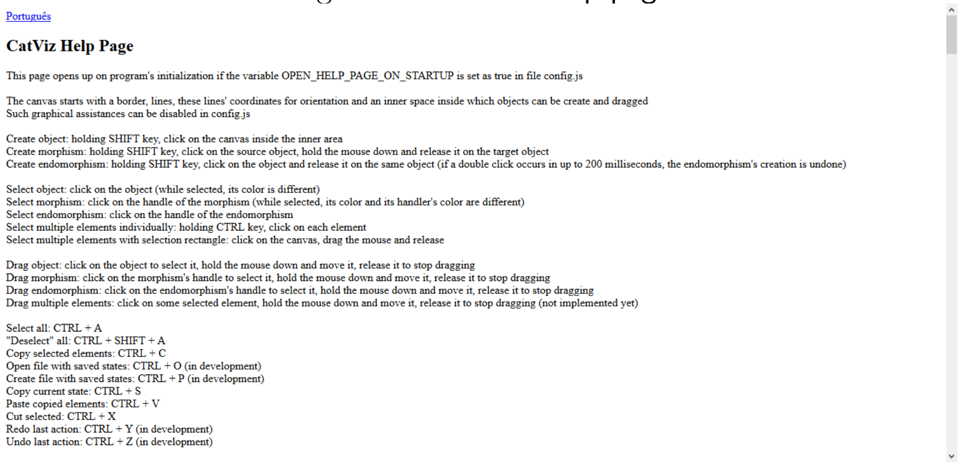
Specially for first time users, there is a help page (another html file) that can be opened and describes how the software can be used. It lists the combinations of keys

Figure 5.1: CatViz screen



Source: The Author

Figure 5.2: CatViz help page



Português

CatViz Help Page

This page opens up on program's initialization if the variable OPEN_HELP_PAGE_ON_STARTUP is set as true in file config.js

The canvas starts with a border, lines, these lines' coordinates for orientation and an inner space inside which objects can be create and dragged
Such graphical assistances can be disabled in config.js

Create object: holding SHIFT key, click on the canvas inside the inner area
Create morphism: holding SHIFT key, click on the source object, hold the mouse down and release it on the target object
Create endomorphism: holding SHIFT key, click on the object and release it on the same object (if a double click occurs in up to 200 milliseconds, the endomorphism's creation is undone)

Select object: click on the object (while selected, its color is different)
Select morphism: click on the handle of the morphism (while selected, its color and its handler's color are different)
Select endomorphism: click on the handle of the endomorphism
Select multiple elements individually: holding CTRL key, click on each element
Select multiple elements with selection rectangle: click on the canvas, drag the mouse and release

Drag object: click on the object to select it, hold the mouse down and move it, release it to stop dragging
Drag morphism: click on the morphism's handle to select it, hold the mouse down and move it, release it to stop dragging
Drag endomorphism: click on the endomorphism's handle to select it, hold the mouse down and move it, release it to stop dragging
Drag multiple elements: click on some selected element, hold the mouse down and move it, release it to stop dragging (not implemented yet)

Select all: CTRL + A
"Deselect" all: CTRL + SHIFT + A
Copy selected elements: CTRL + C
Open file with saved states: CTRL + O (in development)
Create file with saved states: CTRL + P (in development)
Copy current state: CTRL + S
Paste copied elements: CTRL + V
Cut selected: CTRL + X
Redo last action: CTRL + Y (in development)
Undo last action: CTRL + Z (in development)

Source: The Author

Figure 5.3: Category 0 represented in CatViz

Source: The Author

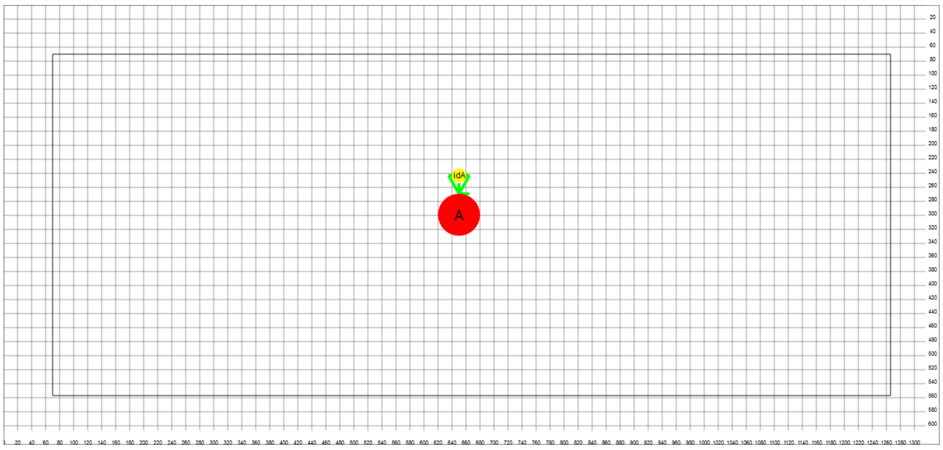
and sequences of mouse buttons to perform the available operations. Figure 5.2 shows a screenshot of that page.

Simply by opening the application, the user is already representing category 0. The first simplest thing he can do is to create objects, morphisms and endomorphisms. Figures 5.3, 5.4, 5.5, 5.6 and 5.7 show some of the simplest categories created this way.

In the context of CatViz, when it is irrelevant to distinguish between objects, morphisms and endomorphisms, they are all together called <u>elements</u>. They can be selected individually or collectively and actions can be performed over them, some of them require that the elements be selected beforehand while some require the opposite. Figure 5.8 shows a morphism that was selected and figure 5.9 shows that it was deleted.

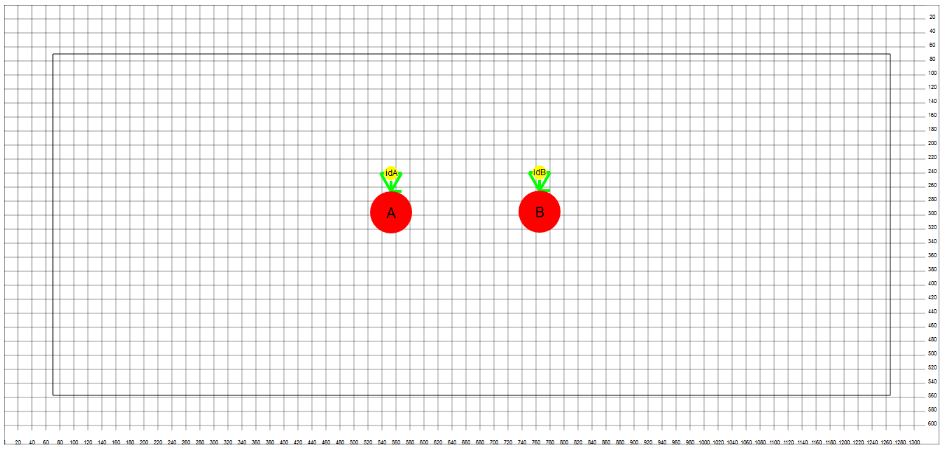
The right mouse button lets the user perform new actions, some of which are similar to those performed with the left mouse button. Some of them let the user have

Figure 5.4: Category 1 represented in CatViz



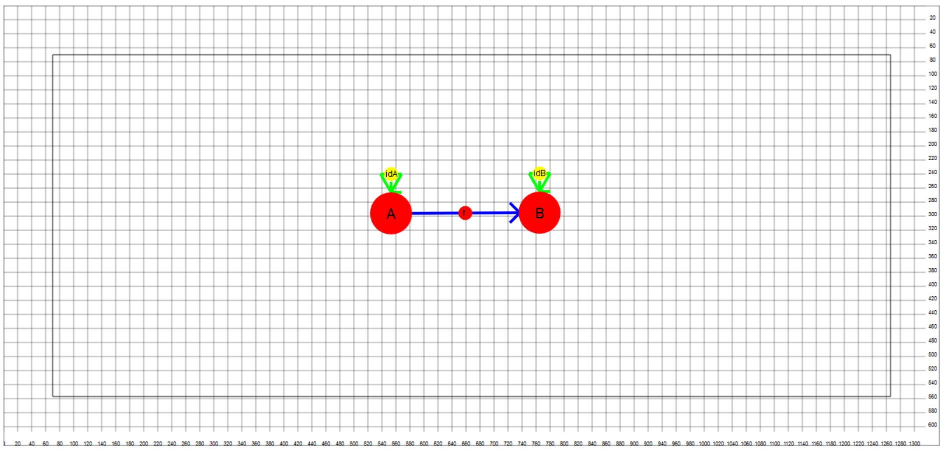
Source: The Author

Figure 5.5: Category 1+1 represented in CatViz



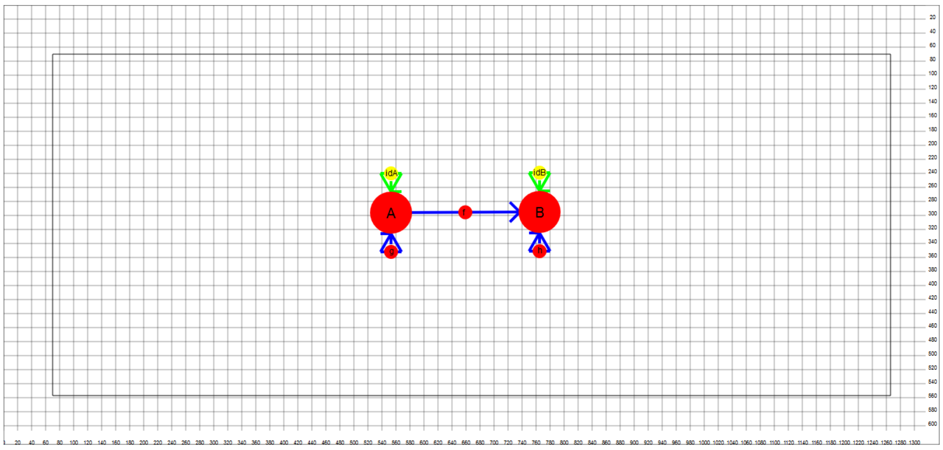
Source: The Author

Figure 5.6: Category 2 represented in CatViz



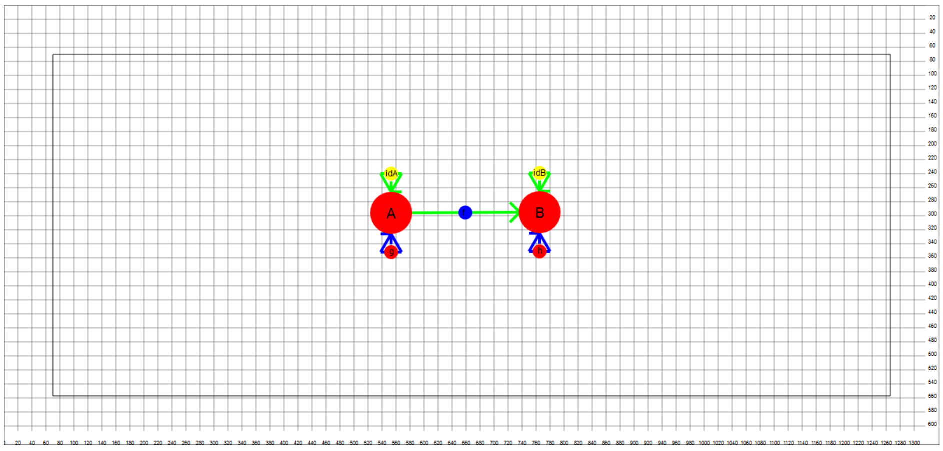
Source: The Author

Figure 5.7: A category with 2 objects, a morphism between them and one endomorphism in each object represented in CatViz



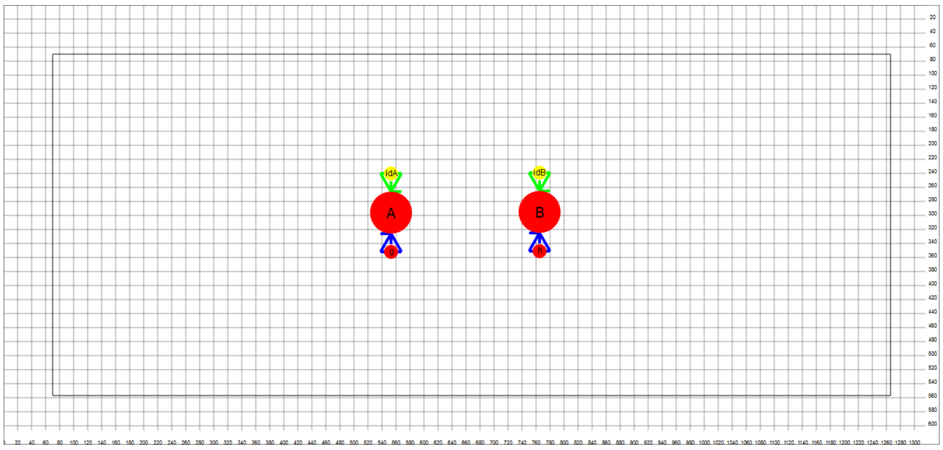
Source: The Author

Figure 5.8: CatViz morphism selected



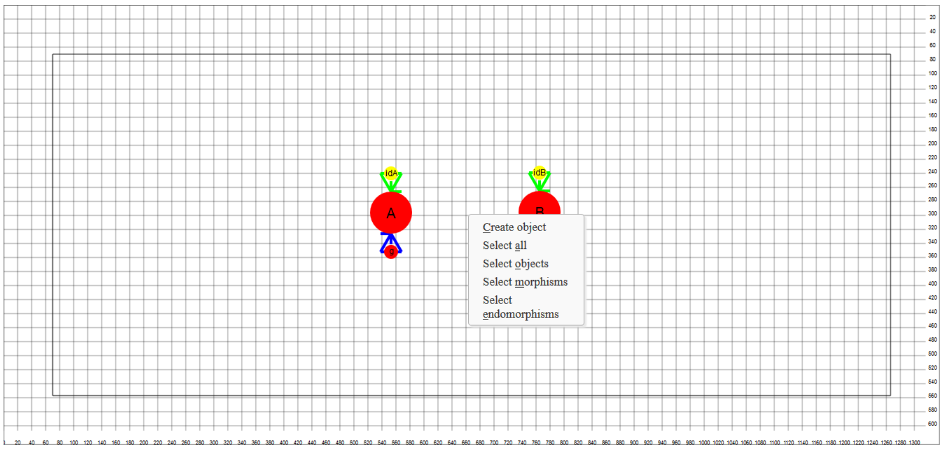
Source: The Author

Figure 5.9: CatViz morphism deleted



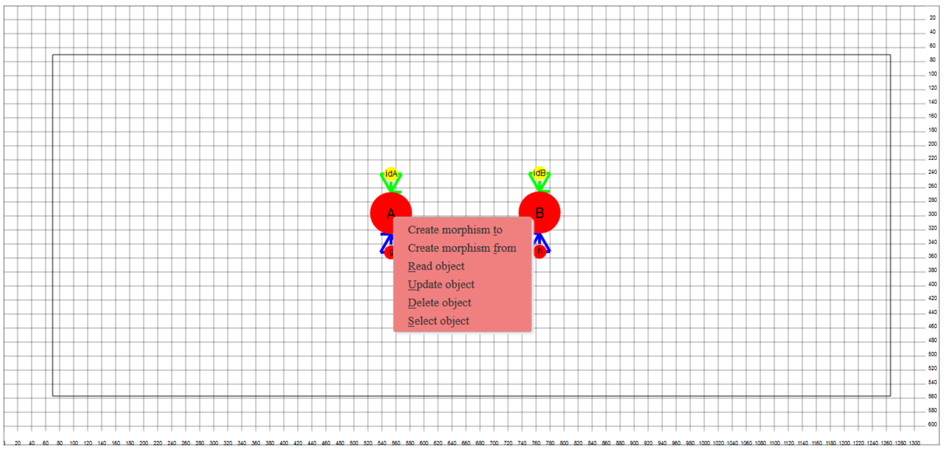
Source: The Author

Figure 5.10: CatViz right mouse button menu on canvas



Source: The Author

Figure 5.11: CatViz right mouse button menu on object

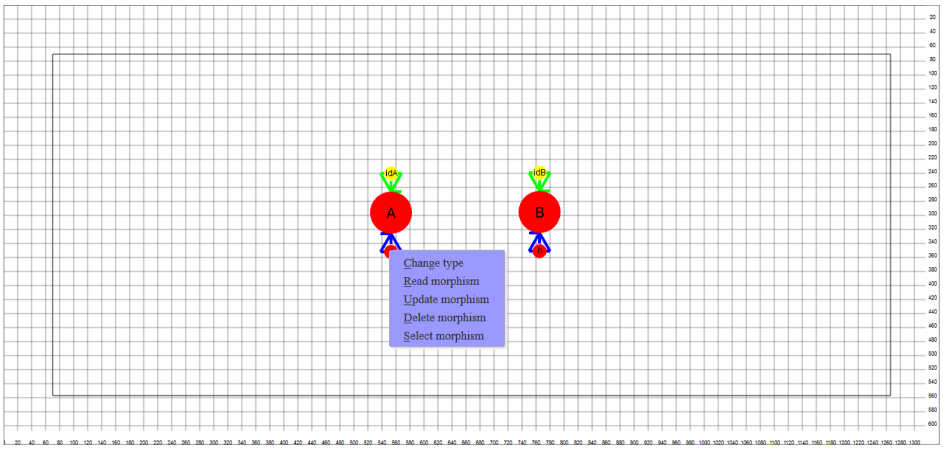


Source: The Author

more control over attributes of the data structure of the elements by configuring them through the dialog that pops up replacing the menu. The name *dialog* is inherited from a third-party jQuery library called jQuery UI, which failed to continue functioning after a major refactoring of the code. It was replaced by a copy of the right button menu. Right after the user chooses an action available in the menu (as we shall call the right button menu from this moment on), it is replaced by the dialog. The trick is to set the menu invisible and setting the dialog visible after its position is defined to be that of the menu. Figures 5.10, 5.11 and 5.12 show the menu open after the user clicked on the canvas, object and morphism. Its background color intentionally differs according to that, so the user knows if he clicked correctly.

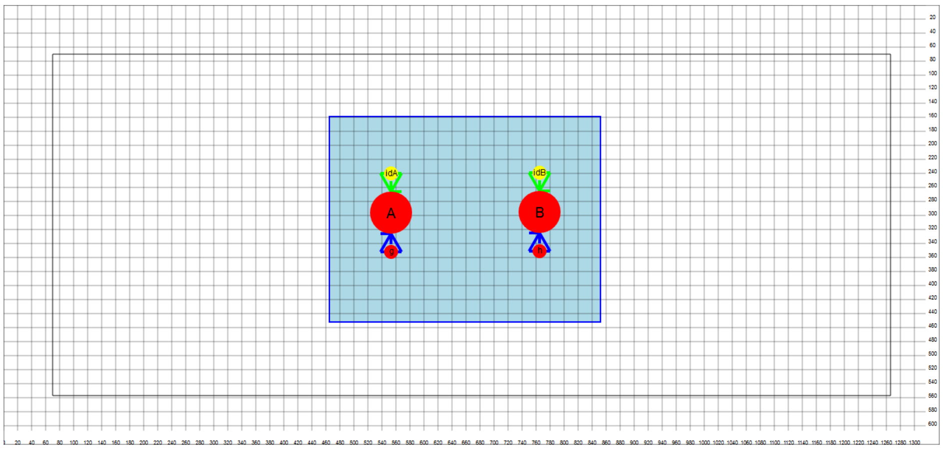
To create elements, the act of pressing the button down and back up has to be done quickly and the mouse must remain in position, otherwise the application considers

Figure 5.12: CatViz right mouse button menu on morphism



Source: The Author

Figure 5.13: CatViz selection rectangle



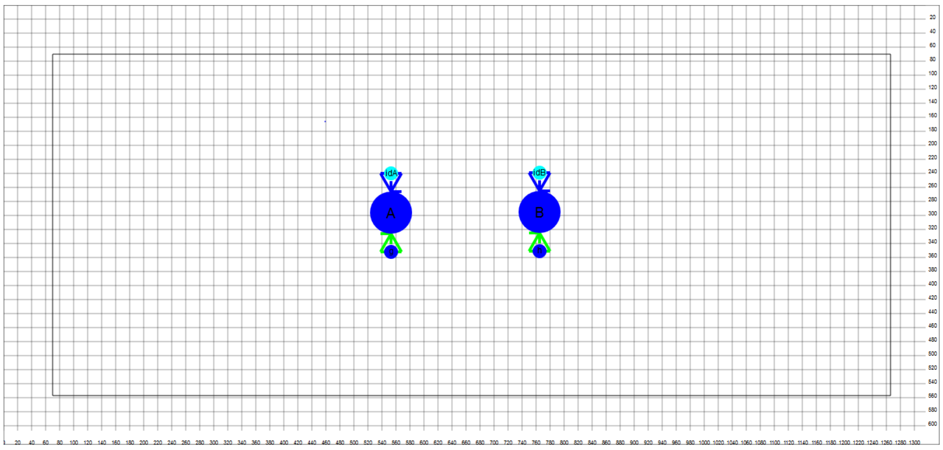
Source: The Author

that the user wants to create a selection rectangle. This rectangle lets him select elements faster by dragging the rectangle along the area of the desired elements and then releasing the button. It is useful for the deletion of multiple elements. Figure 5.13 shows the selection rectangle being dragged, figure 5.14 shows multiple elements selected with that and figure 5.15 shows that they were deleted.

If CatViz automatically created the compositions, at least three problems would occur:

- diagrams would be quickly polluted;
- the student that was doing an assignment would not be responsible for everything that is shown in them;
- he would not be able to create multiple compositions, which are necessary in some theorem proving assignments.

Figure 5.14: CatViz multiple elements selected

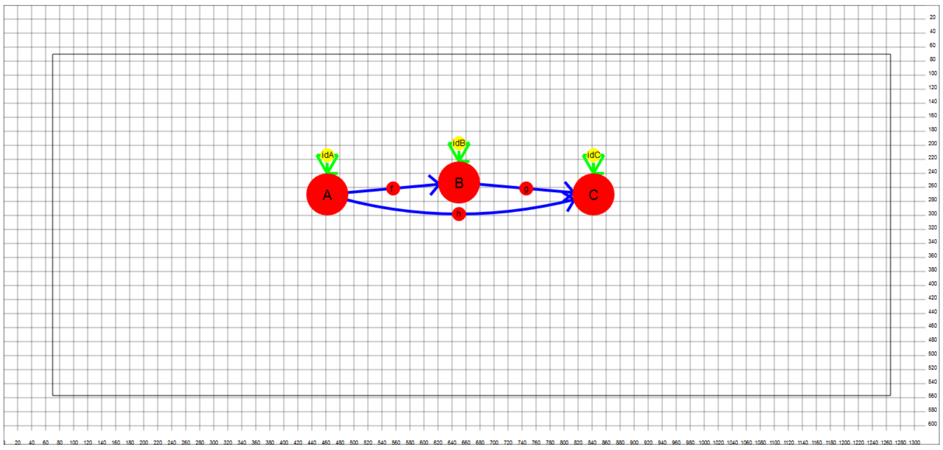


Source: The Author

Figure 5.15: CatViz selected elements deleted

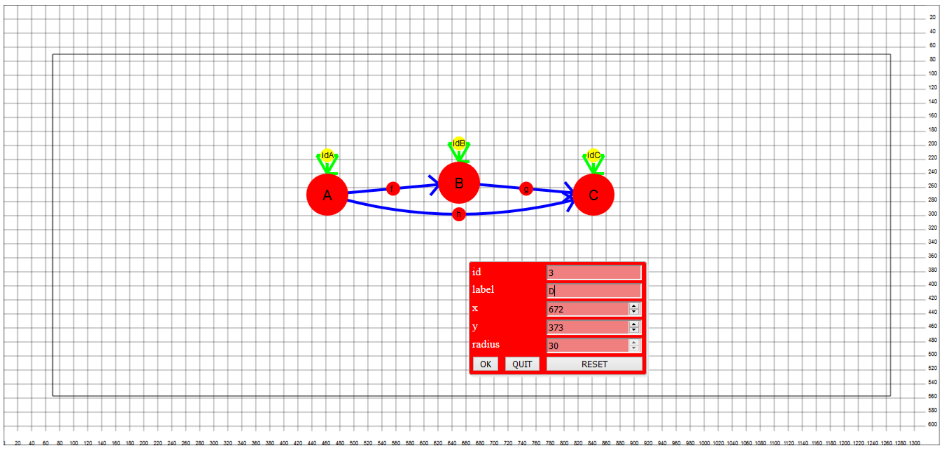
Source: The Author

Figure 5.16: CatViz composition represented in the diagram



Source: The Author

Figure 5.17: CatViz composition update



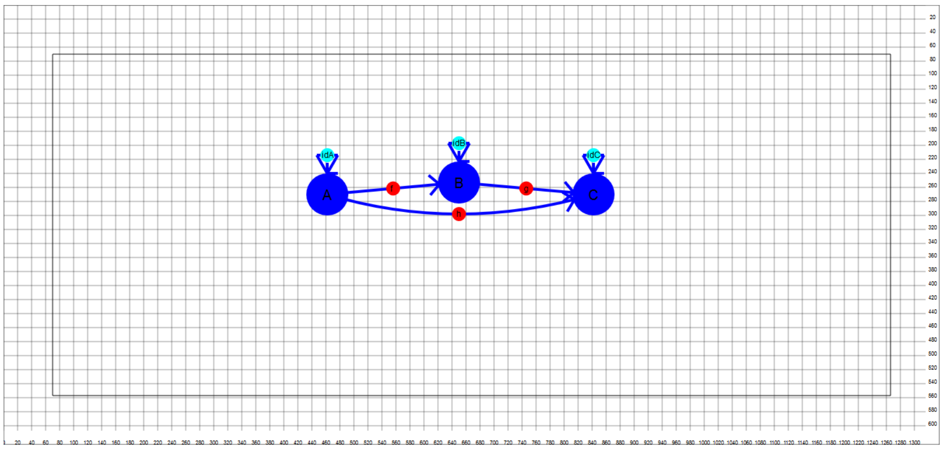
id	3
label	D
x	672
y	373
radius	30

Source: The Author

All of these issues would not help the process of learning Category Theory. That is why the user can choose to represent compositions in a diagram. As CatViz lacks information regarding composition, one problem that arises is that of the distinguishment between a common morphism and a composition. Future work will fix that issue by creating a table of morphisms. It will contain a list of morphisms that are actually compositions of others and will be useful for theorem proofs. Figure 5.16 shows a composition and figure 5.17 shows a right mouse menu action being set up to be performed, namely, updating the attributes of that morphism.

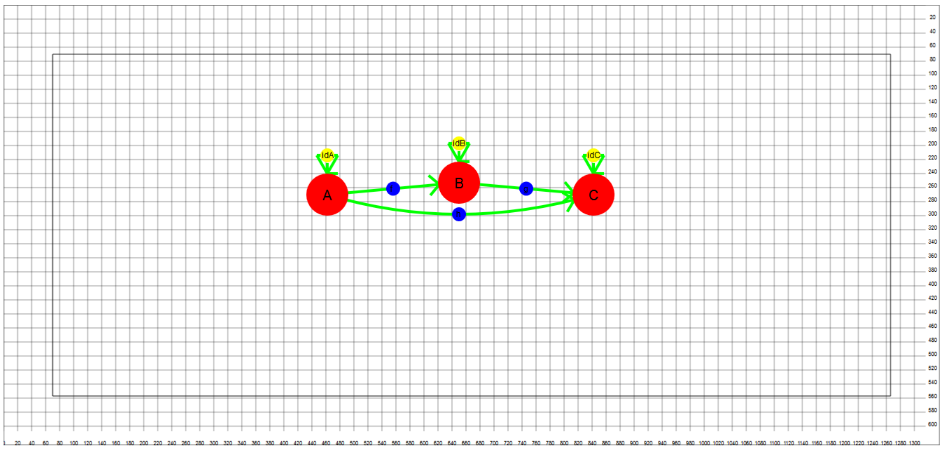
Either by using the selection rectangle throughout the extension of the canvas, by pressing the right combination of keys or through the menu, the user can select all visible elements. Also, either by selecting individually or through the menu, he can select all elements of same type separately (all objects, all morphisms or all endomorphisms).

Figure 5.18: CatViz all elements selected



Source: The Author

Figure 5.19: CatViz all morphisms selected



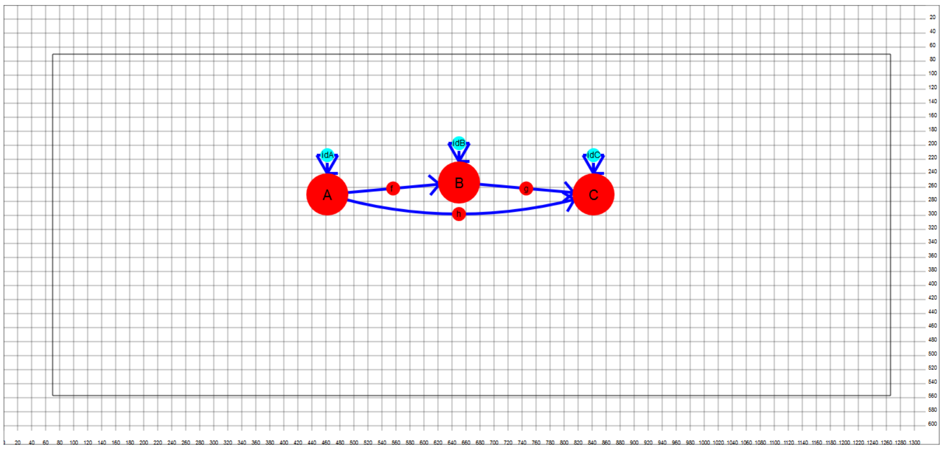
Source: The Author

When an element is selected, its colors change, so the user knows that this action really occurred. Figures 5.18, 5.19 and 5.20 show all elements, all morphisms and all endomorphisms selected, respectively.

The menu displays actions that resemble CRUD operations. In fact, the strings that explain them were carefully chosen with that in mind. One advantage of that is that every string has one and only one character underlined, preferably those of the acronym CRUD, that indicate to the user that he can press the equivalent key on the keyboard to perform that action, and having the underlined letters matching those of the acronym is more organized in my point of view. Figures 5.21, 5.22, 5.23, 5.24 and 5.27 show the operations available for an object. Figures 5.25 and 5.26 show that these operations can be undone and redone.

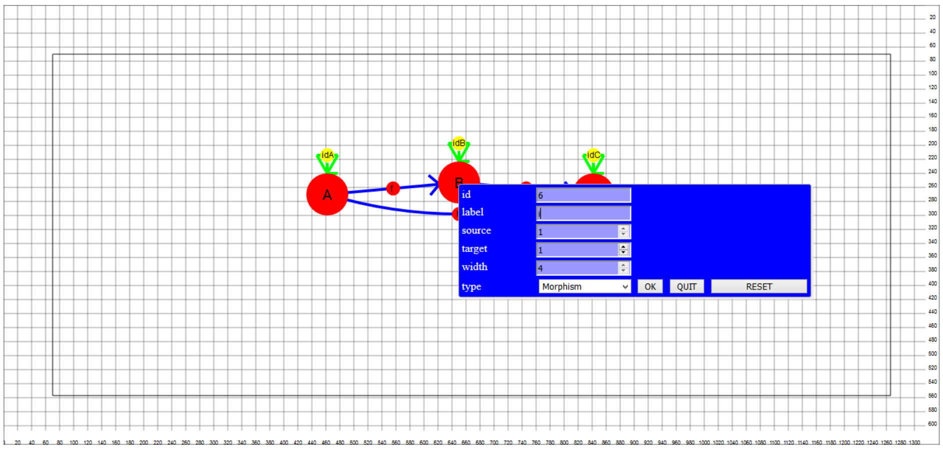
The strings that explain them follow the pattern of starting with a verb and the

Figure 5.20: CatViz all endomorphisms selected



Source: The Author

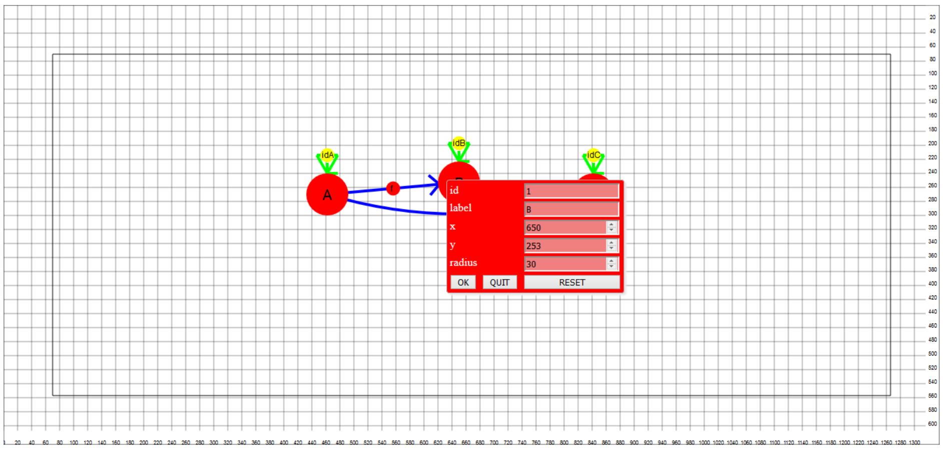
Figure 5.21: CatViz morphism creation from object



id 6
label i
source 1
target 1
width 4
type Morphism

Source: The Author

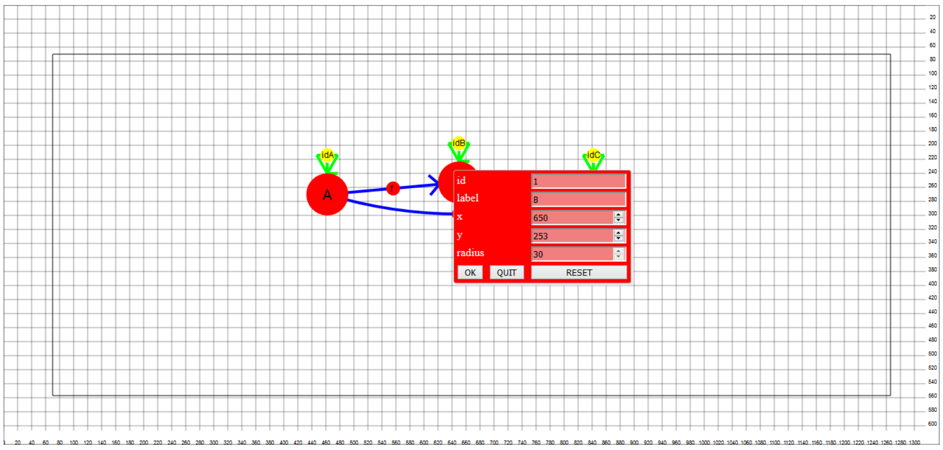
Figure 5.22: CatViz object reading



id 1
label B
x 650
y 253
radius 30

Source: The Author

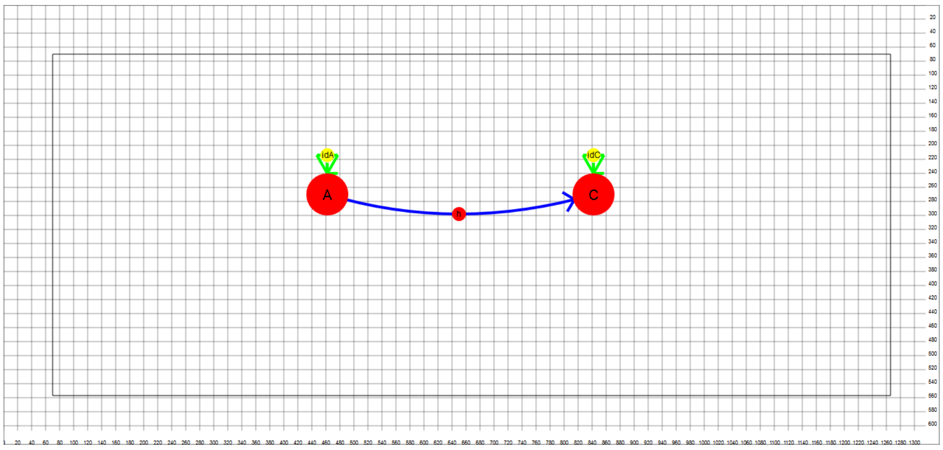
Figure 5.23: CatViz object update



id 1
label B
x 650
y 253
radius 30

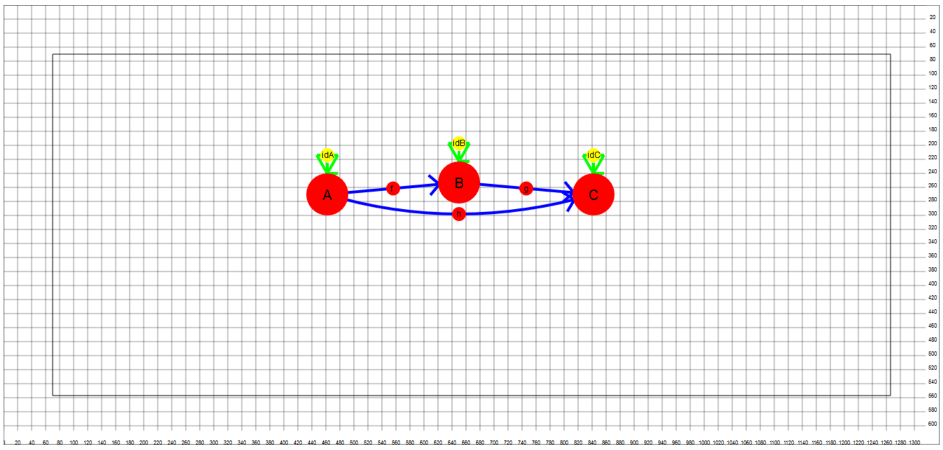
Source: The Author

Figure 5.24: CatViz object deleted



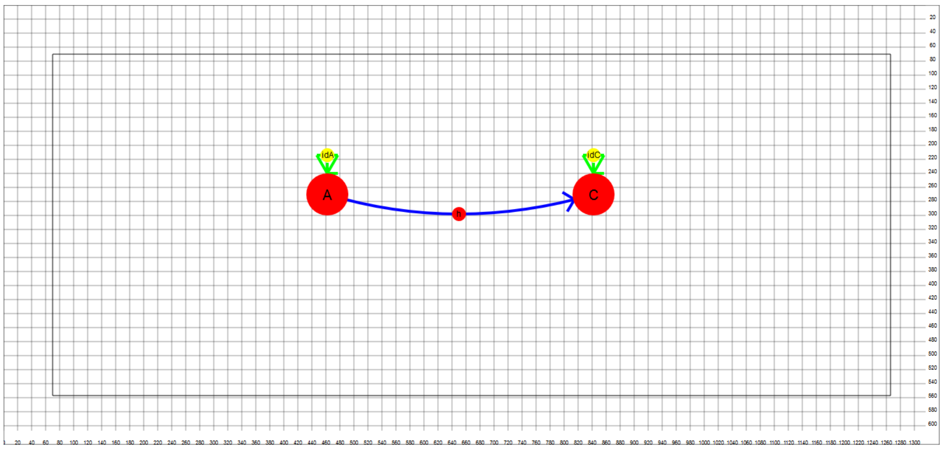
Source: The Author

Figure 5.25: CatViz object deletion undone



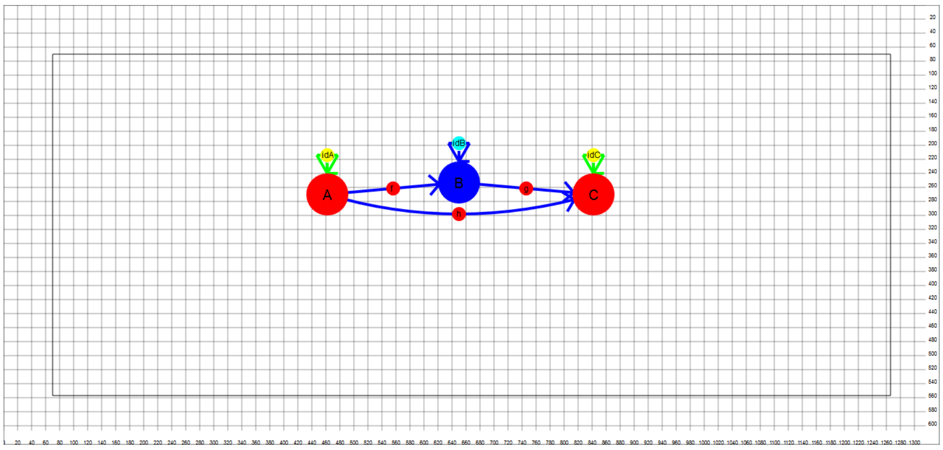
Source: The Author

Figure 5.26: CatViz object deletion redone



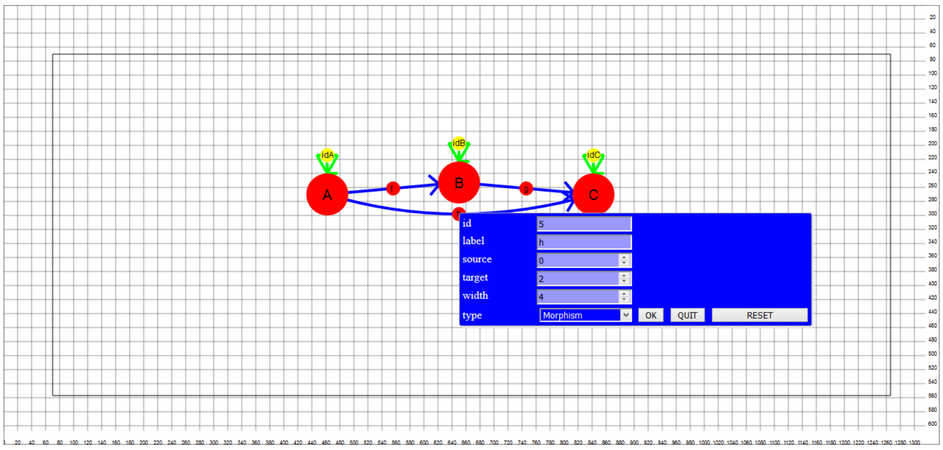
Source: The Author

Figure 5.27: CatViz object selected



Source: The Author

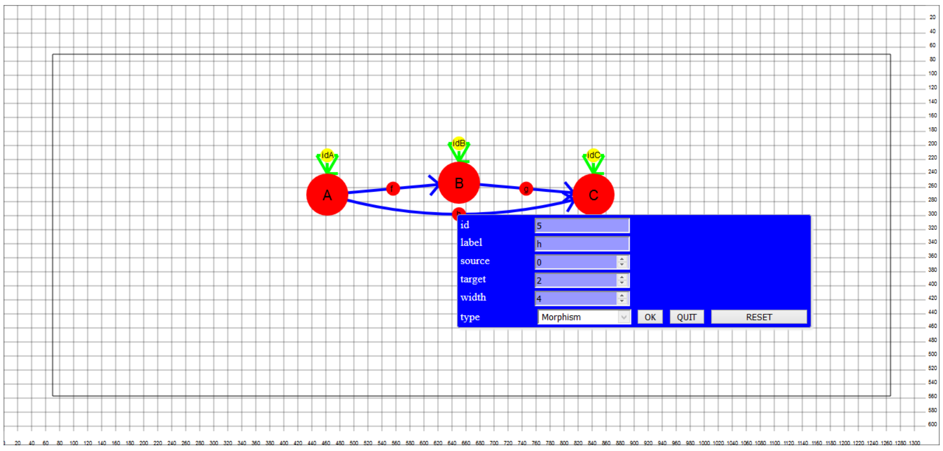
Figure 5.28: CatViz morphism type change



id 5
label h
source 0
target 2
width 4
type Morphism

Source: The Author

Figure 5.29: CatViz morphism reading



id 5
label h
source 0
target 2
width 4
type Morphism

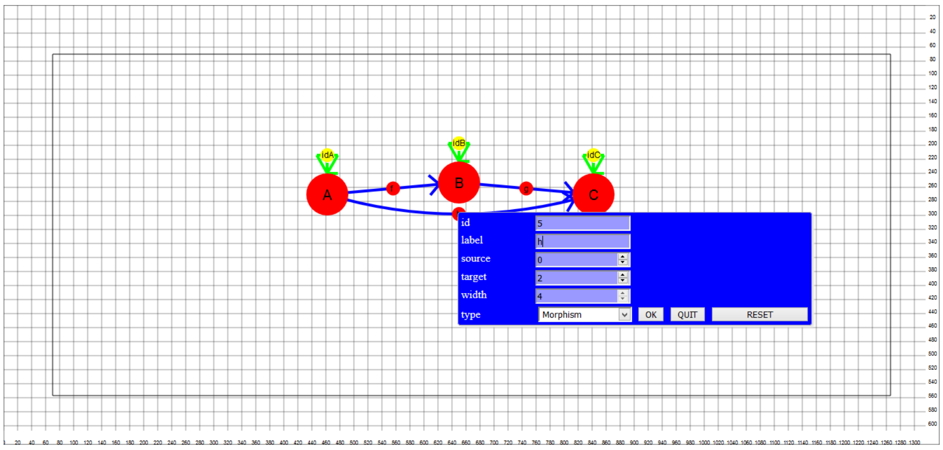
Source: The Author

underlined letter is preferably the first of the verb, so the attempt of setting the underlined letters according to the acronym is not always successful. For different elements, different operations are possible. Figures 5.28, 5.29, 5.30, 5.31 and 5.34 show the operations available for a morphism. Figures 5.32 and 5.33 show that these operations can also be undone and redone.

The operations on endomorphisms can be similar to those of regular morphisms but they require some alterations in CatViz's source code to correctly model the rules of Category Theory. Figures 5.35, 5.36, 5.37 and 5.39 show some of the operations available for an endomorphism, which can also be undone and redone. Figure 5.38 shows that the deletion of an identity endomorphism is not allowed.

Elements of different types can be collectively selected and the operations available for them are those that be performed in more than one element, even of same type.

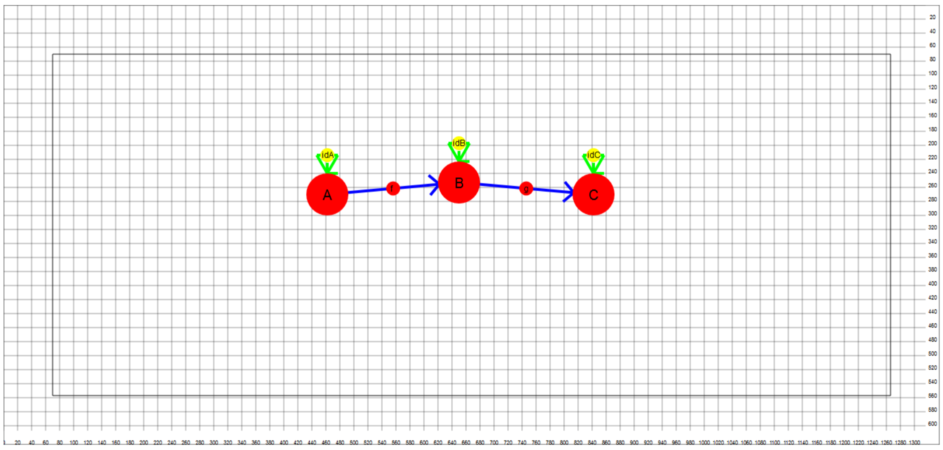
Figure 5.30: CatViz morphism update



id 5
label h
source 0
target 2
width 4
type Morphism

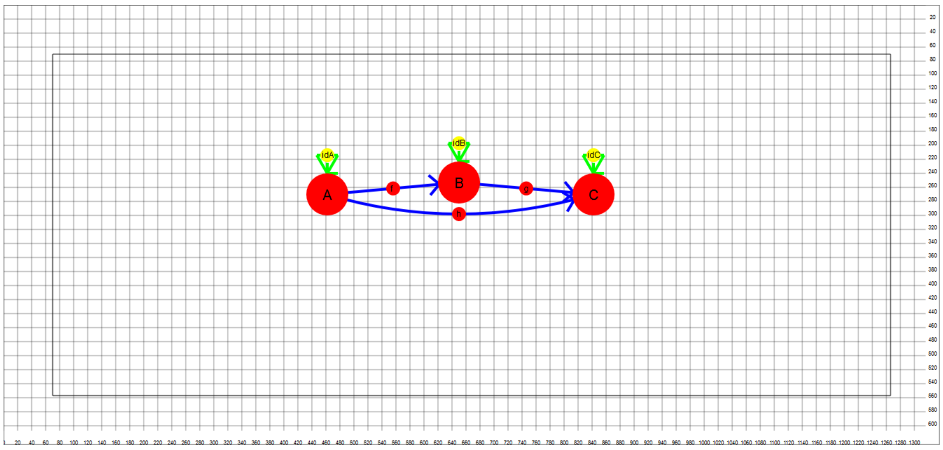
Source: The Author

Figure 5.31: CatViz morphism deletion



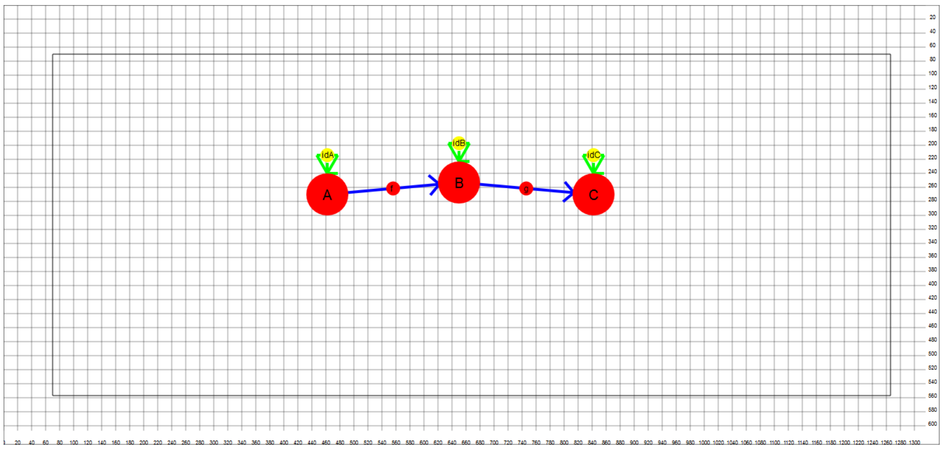
Source: The Author

Figure 5.32: CatViz morphism deletion undone



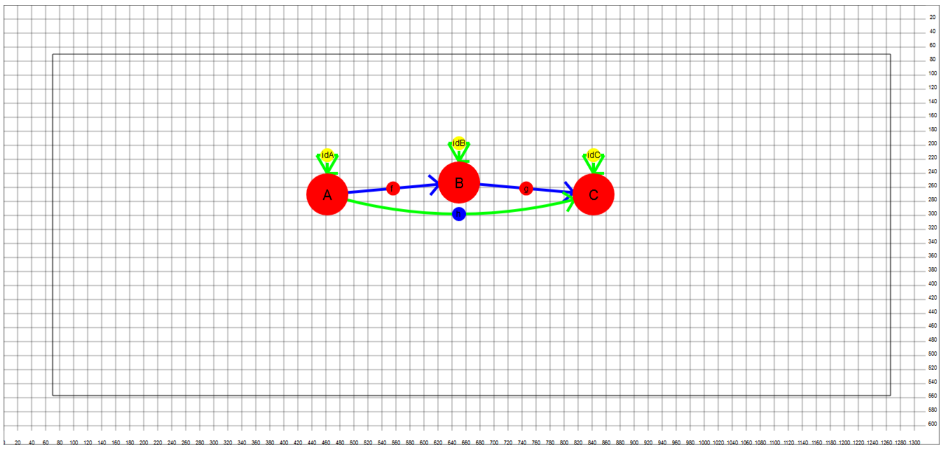
Source: The Author

Figure 5.33: CatViz morphism deletion redone



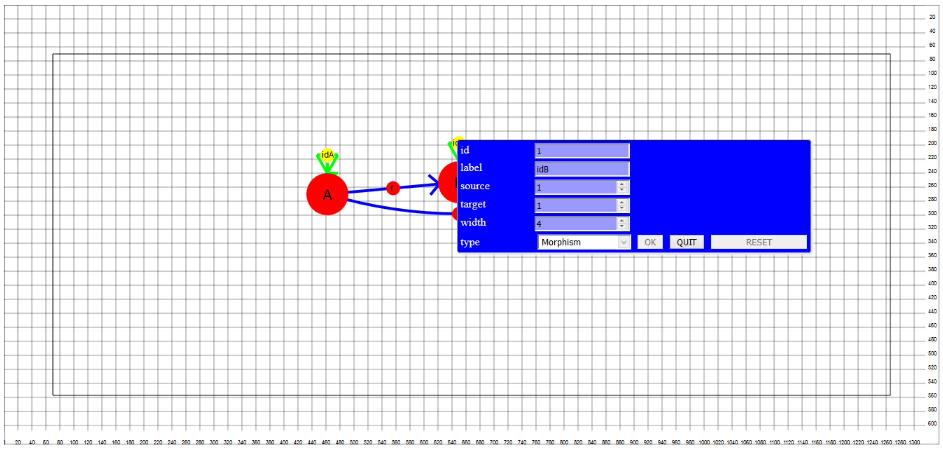
Source: The Author

Figure 5.34: CatViz morphism selected



Source: The Author

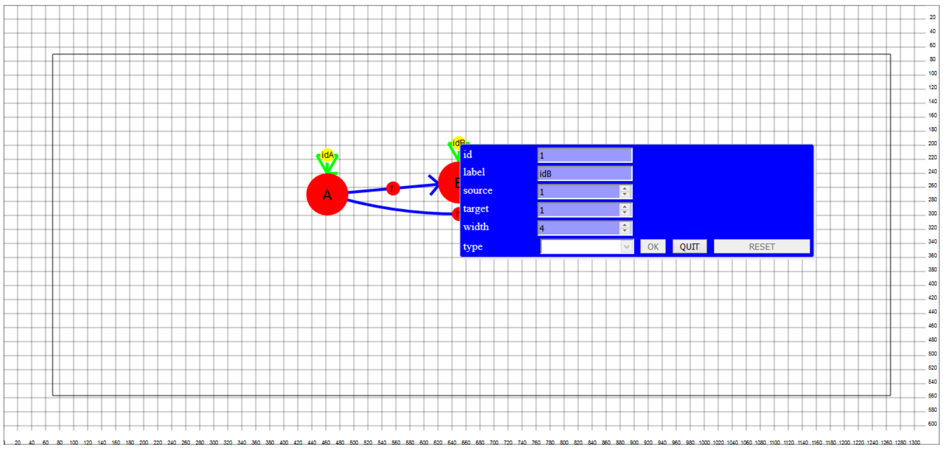
Figure 5.35: CatViz endomorphism type change



id	1
label	idB
source	1
target	1
width	4
type	Morphism

Source: The Author

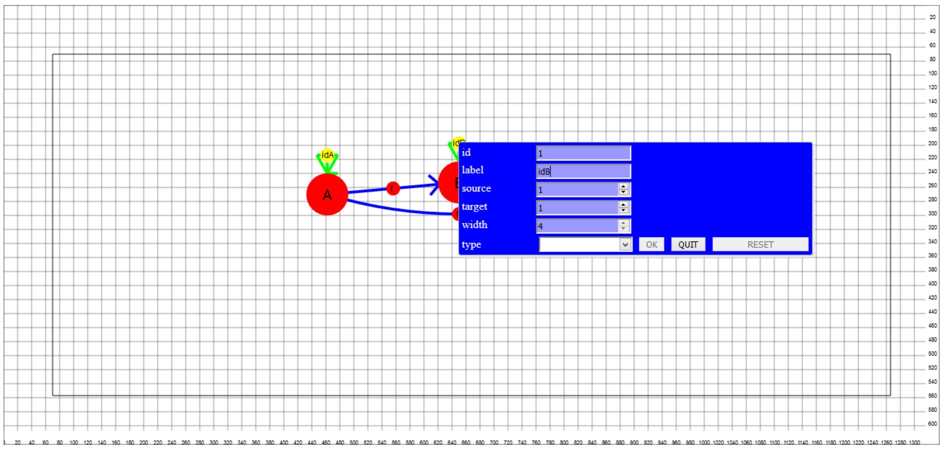
Figure 5.36: CatViz endomorphism reading



id 1
label idB
source 1
target 1
width 4
type

Source: The Author

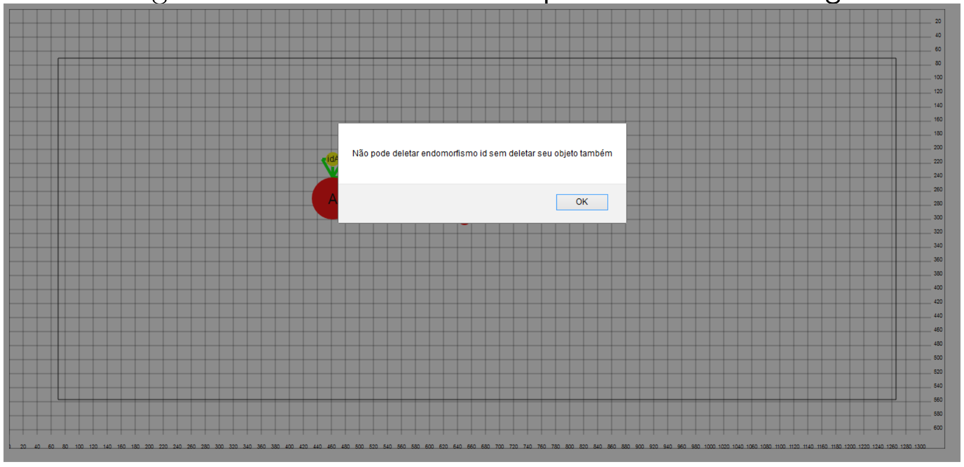
Figure 5.37: CatViz endomorphism update



id 1
label idB
source 1
target 1
width 4
type

Source: The Author

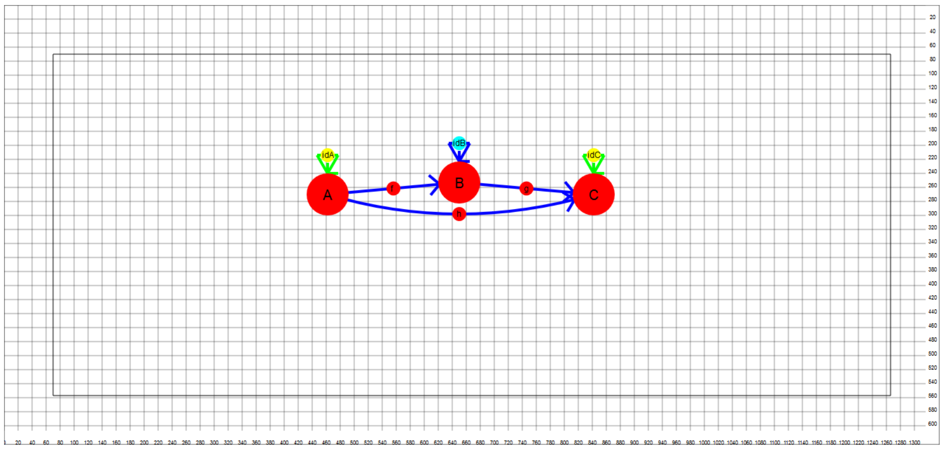
Figure 5.38: CatViz id endomorphism deletion warning



Não pode deletar endomorfismo id sem deletar seu objeto também

Source: The Author

Figure 5.39: CatViz id endomorphism selection



Source: The Author

For now, CatViz lets users cut and copy multiple selected elements. By doing that, the system internally creates a copy of them so the user can paste them if he wants. Cutting differs from copying by deleting the selected elements.

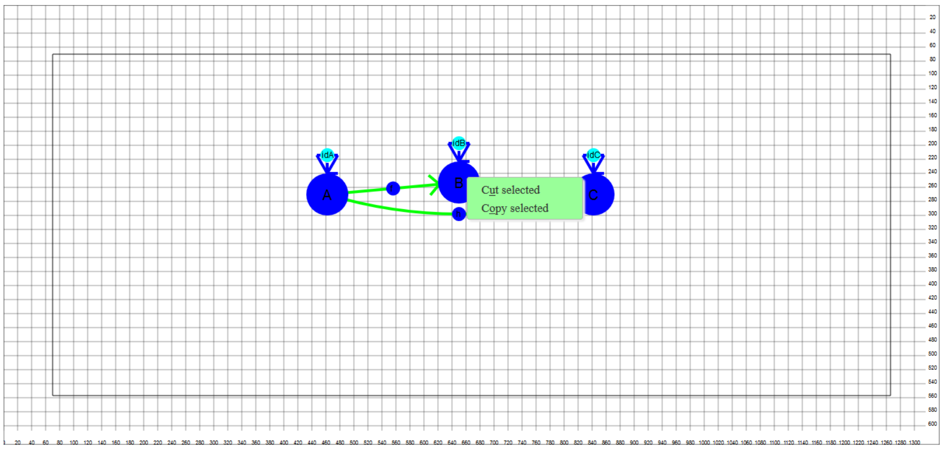
Deleting elements, at first, just turns them invisible to the user but they are deleted for good if the user performs another operation. It has to do with the stack of states of the application. Initially it was thought to be a good idea because it would be simple to write code to turn the visibility on and off and not recreate elements, which is based on counters that would need to be decremented and later incremented, but it turned out to be as complex. However, it is better this way for the case of cutting elements, because they are temporarily invisible and undoing the cutting is quick. Also, despite it is not what happens, it would be trivial to paste the elements at the position of those cut.

When the user presses the mouse down, the position of the mouse is used in a collision function against all elements and the first collided element is considered as clicked. For that reason, when the user pastes selected elements, these are offset diagonally to the original ones.

Figure 5.40 shows the available operations on the menu for multiple elements, figure 5.41 shows the result of cutting selected elements and figure 5.44 shows the result of copying and pasting them. Figures 5.42 and 5.43 show that these operations can be undone and redone.

Users are capable of dragging elements individually and collectively, while that happens, the involved elements are considered by the system to be selected, and so, their colors change. Figures 5.45, 5.46 and 5.47 demonstrate the dragging of individual

Figure 5.40: CatViz right mouse button on multiple elements

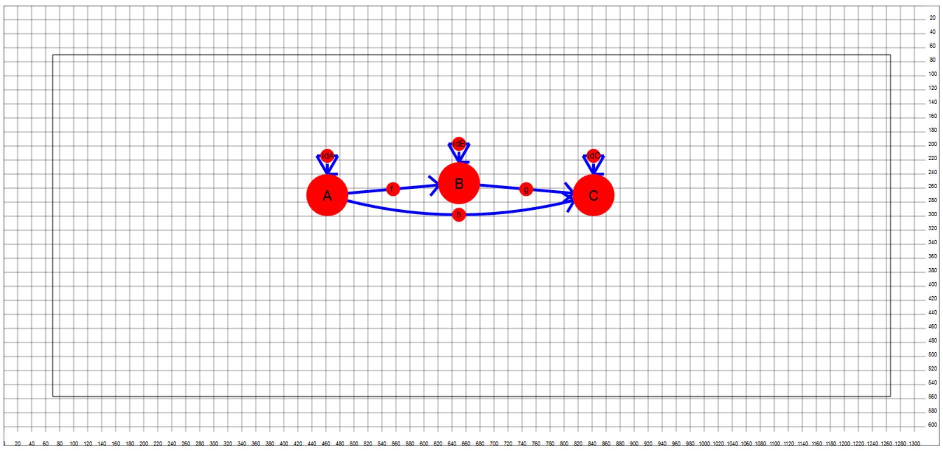


Source: The Author

Figure 5.41: CatViz multiple elements deleted

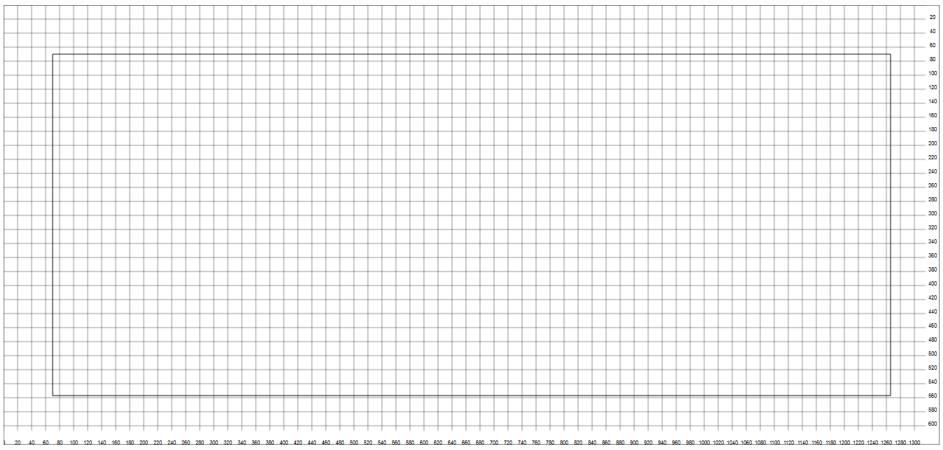
Source: The Author

Figure 5.42: CatViz multiple elements deletion undone



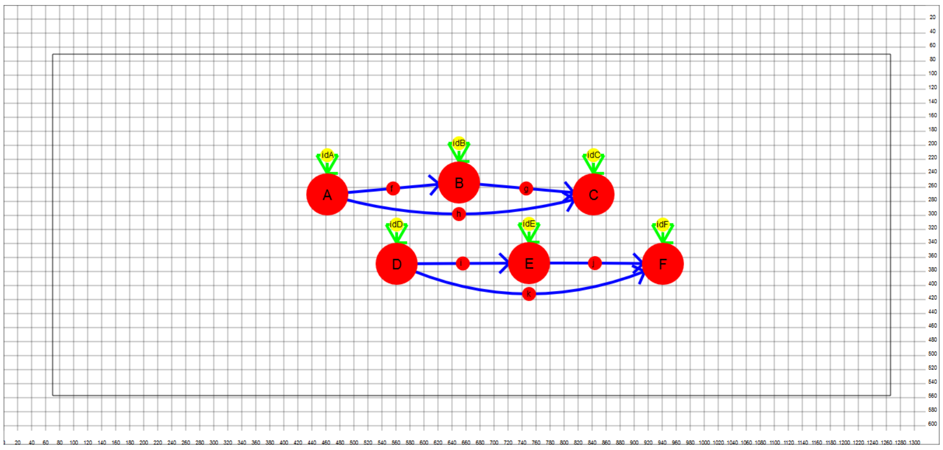
Source: The Author

Figure 5.43: CatViz multiple elements deletion redone



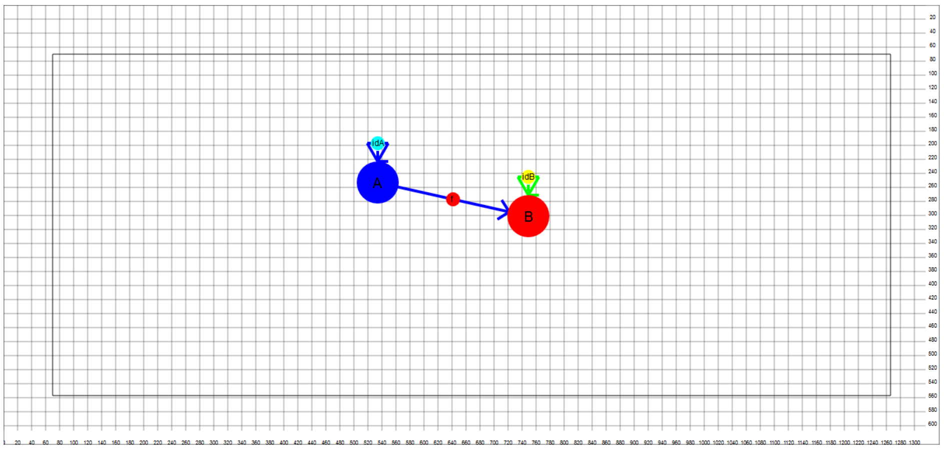
Source: The Author

Figure 5.44: CatViz selected elements paste



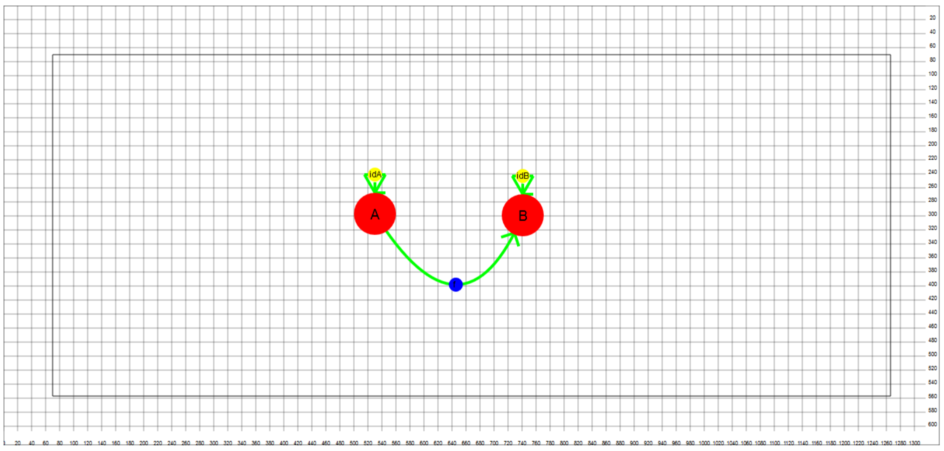
Source: The Author

Figure 5.45: CatViz dragging object



Source: The Author

Figure 5.46: CatViz dragging morphism



Source: The Author

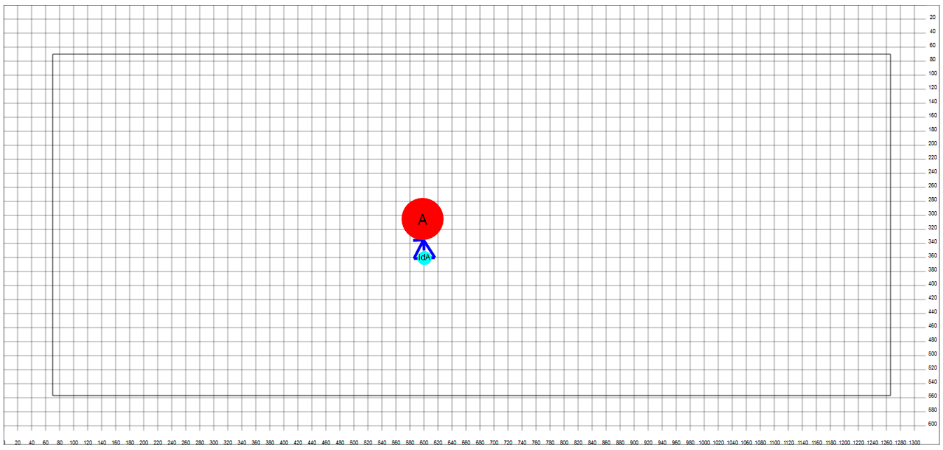
elements.

For didactic purposes, CatViz offers a drawing brush with 10 tones of color and adjustable size. It can be used to assist the teacher in class and perhaps students to explain something done in an assignment. Figure 5.48 shows the available colors.

The canvas is divided in layers for things to come on top of others on rendering. When CatViz is not focused, a blur rectangle indicates that. It is the first thing to be rendered, so everything else is visible when it is enabled. When the focus is lost in CatViz, elements that were being selected while the mouse is pressed down are deselected while menu and dialog are invisibilized and the state of the application is set to *idle*. If elements were selected by other means, the state is set to *idle with selected element(s)* Figure 5.49 shows the blur rectangle with some elements selected.

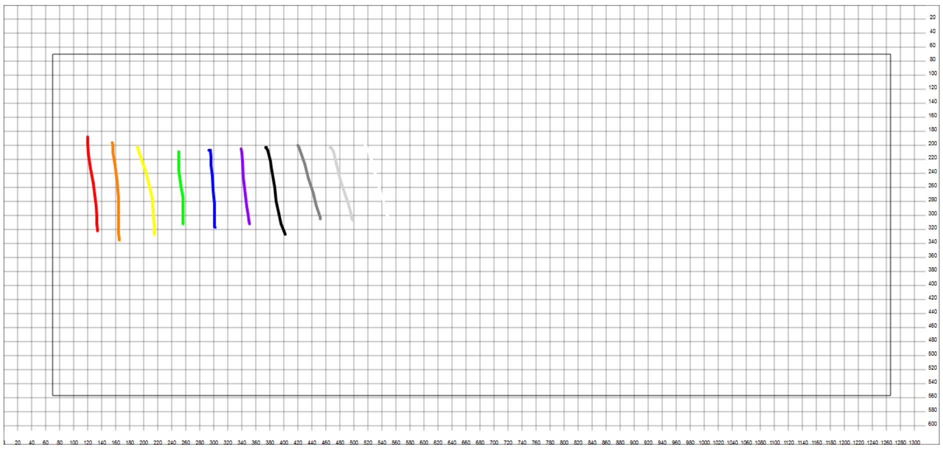
Another very important requirement defined in the beginning of the development

Figure 5.47: CatViz dragging endomorphism



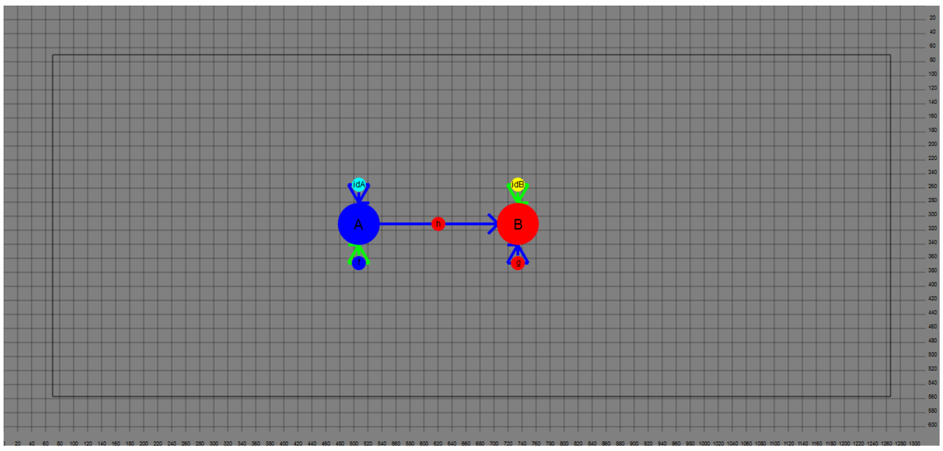
Source: The Author

Figure 5.48: CatViz drawing



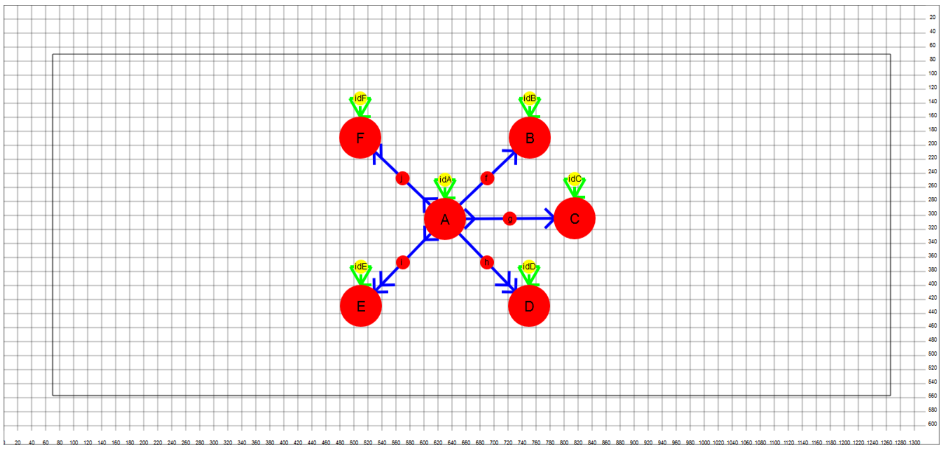
Source: The Author

Figure 5.49: CatViz lost focus



Source: The Author

Figure 5.50: CatViz different arrows



Source: The Author

was to have a visual representation for each type of arrow. Since morphisms, monomorphisms and epimorphisms are denoted by $A \rightarrow B$, $A \rightarrowtail B$ and $A \twoheadrightarrow B$, respectively, it was reasonable to represent them like that. However, morphisms that are both mono and epi do not have a specific denotation, so it seemed reasonable to represent those as a combination of both mono and epi. Isomorphisms, denoted by $A \leftrightarrow B$, would cause a problem if they were represented by that. If an object is the source of an isomorphism and the target of many morphisms, for example, it would easily be difficult to distinguish the pointy ends of those morphisms. The alternative representation tries to mitigate this problem. Figure 5.50 shows the representation of morphisms f, g, h, i and j of kind, respectively, morphism, mono, epi, monoepi and iso.

5.2 Architecture

As stated previously, for Web-based technology, the equivalent to a main file found in most compiled languages is the index.html file. From that, one can include CSS and JS files.

CatViz uses CSS files for:

1. disabling mouse dragging, which usually is represented by a blue shade over the area that was dragged. That would cause awful effects but mostly, due to the purpose of the software, not allow an important part of the user interaction with the elements;
2. the menu which lets the user perform various operations;

3. the dialog that pops up when the user chooses one of the operations available in the menu.

Also, the software uses JS files to include third-party libraries, jQuery and D3.js, and to modularize the different parts of the application. JS files in CatViz's context are called modules. They are:

1. collider;
2. config;
3. drawing;
4. files;
5. focus;
6. globals;
7. keyboard;
8. locale;
9. menu;
10. morphism;
11. mouse;
12. object;
13. state;
14. utils;
15. view.

Some of those are actually classes, namely Collider, Drawing, Files, Menu, State and View, so after including the modules, these are instantiated in the main file. New instances of classes Object and Morphism are created for each new element in the canvas.

5.3 Class Diagram of Modules

When constructing the class diagram for CatViz there were two criteria considered:

1. attributes and methods would not be displayed, otherwise the diagram would be too large;
2. D3.js and jQuery usage is ignored, because they are expected to be free of errors;

3. indirect inheritance would not be considered a real inheritance, otherwise the diagram would be even more polluted than it already is.

By indirect inheritance, I mean the usage of attributes or methods of an instance where this instance is a variable of a second class but those attributes or methods are used by a third class. This often occurs with the Morphism and Object modules, for instance.

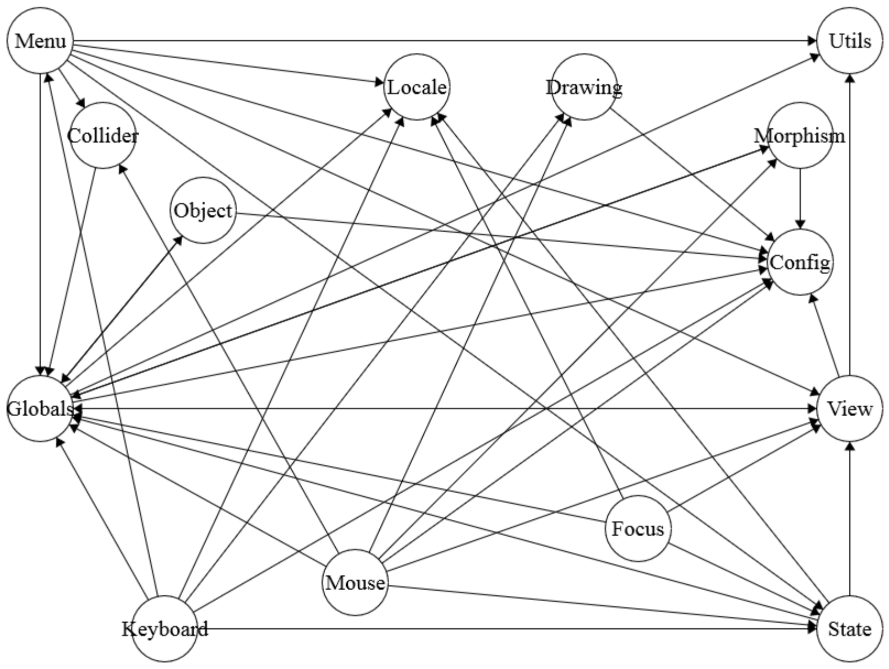
When index.html imports all modules, it makes the whole system know about the inner content of classes (as long as these are public), that is another reason why indirect inheritance can be disconsidered.

Example 33. Indirect Inheritance

```
//in morphism.js
function Morphism(...) {
...
}

//in object.js
function Object(...) {
...
}

//in globals.js
...
var objects = [];
...
var morphisms = [];
...

//in keyboard.js
...
var id = objects[i].getId();
...
var id = morphisms[i].getId();
...
```

In the example above, Globals module contains arrays of instances of Object and Morphism, but these classes' methods are used by the Keyboard module. Figure 5.51 shows a class diagram for CatViz, where nodes represent modules and arrows represent dependencies between them.

Figure 5.51: CatViz class diagram



Source: The Author

Considering the dependencies between modules, they can be classified into three groups. The main reason for doing that is that when we need to fix bugs or add features, error/effect propagation can be better understood by following their flow and if the module does not propagate them, there is no need to verify the others, supposedly.

No External Usage. Three modules use no external variables (from other modules). At most, they use their own (auxiliary) variables or functions/methods. They are: Config, Locale and Utils.

Minimal External Usage. Seven modules use external variables, from modules in this group and others. They are: Collider, Drawing, Focus, Morphism, Object, State and View.

Large External Usage. Four modules use many external variables, so it is better to make sure that the original modules from where these variables come from are correct. They are: Globals, Keyboard, Menu and Mouse.

By looking at the class diagram, we also see that:

1. 3 classes are not used others;

2. 7 classes are used by 1 to 4 others;
3. 4 classes are used by 5 or more others.

But unfortunately, in CatViz, a class not using others does not imply that it is used by many others and vice-versa. It would be great to have the code well partitioned, but that could not be achieved.

We also see that the diagram is a little polluted. This is caused by the choice of modularizing the code in files that are responsible for their own specific function in the software.

However, this effort on modularizing functionality shows an interesting result: if we consider a metric where we give 1 point to a module for each time another module uses it and take away 1 point of a module for each time it uses another one, the final value is 1, which means that, considering the relationship between modules as it is now, they almost use others as much as they are used.

5.4 Modules

It is possible to separate CatViz's modules in four different groups, according to their characteristics and usage. Organizing parts of the software in different files speeds up correction and reduces errors.

5.4.1 Definition Modules

These modules have constants, variables, functions and methods that are used throughout the application and can be easily fixed if a bug appears.

Config Module. Config consists of various constants used throughout the software. Each one can be regarded as similar to a #DEFINE in C language. That way, a constant keyword in config has a value that is not changed on runtime and is directly read from config.js. The user can change their values to his will and then refresh the application for the change to take effect. This is quite safe as long as the type remains the same, although some may require attention as they can cause major alterations in functionality. This module is used by all but three others.

Example 34. Config Module

```
//MODULE_NAME configuration variables
const KEYWORD = value;
...
```

Keywords in config.js are sorted by the module in which they are used. A keyword's value can be anything allowed in JS: numbers, booleans, chars, strings, arrays, math operations, method calls, etc.

Globals Module. Globals stores all variables (different from Config, that keeps constants) and functions that are used throughout the software. Their value can be changed anytime by the system from any other file. The most important variables are those that are arrays to store instances and those used in event-driven listeners.

Example 35. Globals Module

```
//MODULE_NAME stuff
var variable = value;
function foo(params) {...}
...
```

Some variables and functions used in specific parts are sorted by that criterion, but most of the file's variables and functions are accessed from multiple other files, so they do not have a sorting criterion.

Locale Module. Locale contains variables that are used for software localization and this module can be extended to new languages by anyone willing to do so.

Example 36. Locale Module

```
var STR_NAME = "value";
...

function updateLocalization() {...}
```

Due to their usage as localized text, all these variables' values are strings that differ in language, the user chooses the language of alerts and console logs by setting the value of a constant in config.js. Current languages available are German, English, Spanish, French, Italian and Portuguese. The texts are translated using a metodology to minimize mistakes as much as possible.

Utils Module. Utils has several methods that extend the functionality of objects that already exist in JS: Array, Math and String. These methods are used by some of the other modules.

Example 37. Utils Module

```
/*
Description of what it does
*/
if(Array.prototype.foo) console.warn("Overriding existing
    Array.prototype.foo at utils.js");
Array.prototype.foo = function() {...}

/*
Description of what it does
*/
if(Math.foo) console.warn("Overriding existing Math.foo at utils.js
    ");
Math.prototype.foo = function() {...}

/*
Description of what it does
*/
if(String.prototype.foo) console.warn("Overriding existing
    String.prototype.foo at utils.js");
String.prototype.foo = function() {...}
```

Since the future is uncertain, it is not possible to predict if the language will evolve to have those methods, so before each method there is a verification to check if there is already one with that name, in which case a console log will be emitted and the method will be overridden anyway, so the expected behavior continues until the code is modified.

5.4.2 Event-Driven Modules

Such modules listen constantly for events emitted by the user as he interacts with the application using keybord and mouse. For that reason, they are not used by any of the other modules, but use several methods that these provide.

Focus Module. This module is responsible for checking if the user lost or regained focus of the application. In web jargon, something is focused if the user clicked a field. In CatViz, the only thing that is checked for focus is the page. If the focus is lost, a gray rectangle covers the grid.

Example 38. Focus Module

```
/*
Listener that runs when focus is lost
*/
window.addEventListener("blur", function(e) {...});

/*
Listener that runs when focus is regained
*/
window.addEventListener("focus", function(e) {...});
```

The focus is lost when opening the console, clicking and/or interacting with the console, clicking on the browser address and search bars, when an alert is being shown, etc.

Keyboard Module. The second most import module listens for keyboard events. One of the listeners has hundreds of lines of codes.

Example 39. Keyboard Module

```
window.addEventListener("keypress", function(e) {...});

window.addEventListener("keydown", function(e) {...});

window.addEventListener("keyup", function(e) {...});
```

This module checks for keyboard input. On a keyboard, a key is either pressed, kept pressed or released (press, down and up, respectively, in JS).

Mouse Module. The most important module, responsible for managing events produced by the primary input source.

Example 40. Mouse Module

```
document.addEventListener("dragstart", function(e) {
 e.preventDefault();
});

$(document).bind("contextmenu", function(e) {
 e.preventDefault();
});

window.addEventListener("mousedown", function(e) {...});

window.addEventListener("mousemove", function(e) {...});

window.addEventListener("mouseup", function(e) {...});
```

The first two listeners prevent the default steps a web-based application would take when the events they listen to happen. Instead, they allow for CatViz to run what other scripts in other files are instructed to do: perform mouse drags and open a menu with the right mouse button.

Similarly to the keyboard events, two of the last three events for which the mouse module checks are the pressing of a mouse button and its release. The other one is for when the mouse is moved, depending on the context, something may need to happen.

5.4.3 Data Structure Modules

These modules define classes that represent objects and morphism of Category Theory. Each new object or morphism is inserted in the array of objects or morphisms.

Morphism Module. A module responsible for defining a class representing a morphism, it stores information the user sees, and so, may resemble what a model in a MVC pattern is.

Example 41. Morphism Module

```
/*
id: natural number
label: string
source: natural number
target: natural number
```

```
width: natural number
type: string in { "endomorphism", "morphism", "monomorphism", "
   epimorphism", "monoepimorphism", "isomorphism" }
isIdEndomorphism: boolean
selected: boolean
visible: boolean
p0, p1, p2: 2D coordinates
curve: pointer to the curve's path
handle: pointer to the curve's handle
*/
function Morphism(source, target, type, isIdEndomorphism, ptr =
   null) {...}
```

Object Module. Similarly, this module defines a modelling class representing objects. It was a good idea to comment the type of each attribute before the functions that define them as classes because the variable typing is very weak.

Example 42. Object Module

```
/*
id: natural number
label: string
x: natural number
y: natural number
radius: natural number
selected: boolean
visible: boolean
endomorphisms: list of morphisms
*/
function Object(p) {...}
```

5.4.4 Object-Oriented Modules

Like the last two modules, these are classes. They are responsible for handling most of what would be controller and view in MVC pattern: collision detection, free hand drawing, operations called by the menu, undoing and redoing operations and rendering. Each one can be seen as a singleton class, although nothing forbids a seconds instance to be created, only one exists along the application's life cycle. The variable that stores the instance is created and set as null in globals.js, but initiated in index.html.

Collider Module. This class contains methods responsible for returning to the system information regarding collision. They do not consider objects and morphisms to be a physical thing. All they see are geometric shapes, so object collision is treated as collision between two shapes. That way, Category Theory elements could be represented by any shape on which this module can operate. The simplest collision testing is between two circles, that is why objects and morphisms' handles are represented by circles.

Example 43. Collider Module

```
/*
Get a list of all circles with point (x, y) inside
*/
this.getAllCirclesWithPointInside = function(p) {...}

/*
Check if a circle with center (x1, y1) and radius r1 and a circle
   with center (x2, y2) and radius r2 intersect
*/
this.circlesIntersect = function(x1, y1, r1, x2, y2, r2) {...}

/*
Get a list of all circles inside area
*/
this.getAllCirclesInsideArea = function(p1, p2) {...}

/*
Get a list of all curve handles with point (x, y) inside
*/
this.getAllHandlesWithPointInside = function(p) {...}

/*
Get a list of all curve handles inside area
*/
this.getAllHandlesInsideArea = function(p1, p2) {...}

/*
Check if the mouse coordinates are inside any circle
*/
this.mousePositionCollidesWithSomeCircle = function(x, y) {...}
```

These methods use private auxiliary methods and return booleans or lists. With them, the software can figure out if the user clicked something, selected several elements, etc.

Drawing Module. The drawing class exclusively handles the drawing brush. Its methods are actually completely separated from the view module because that makes the code more modularized, since they operate solely over an exclusive canvas layer for drawing with their own variables.

Example 44. Drawing Module

```
var drawingsCounter = 0;

/*
Erase all drawings
*/
this.eraseAllDrawings = function() {...}

/*
Erase last drawing (which begins on mousedown and ends on mouseup)
*/
this.eraseLastDrawing = function() {...}

/*
Change brush color
*/
this.changeBrushColor = function(key) {...}

/*
Decrease brush size
*/
this.decreaseBrushSize = function() {...}

/*
Increase brush size
*/
this.increaseBrushSize = function() {...}

/*
Start drawing
*/
```

```
this.startDrawing = function(p) {...}

/*
Update draw
*/
this.updateDrawing = function(p) {...}

/*
Stop drawing
*/
this.stopDrawing = function() {...}
```

The brush size and color can be changed and drawings can be erased if necessary. While holding the mouse pressed, small lines with the same HTML class are added to the canvas. When the mouse is released, the class counter is incremented and if a drawing is erased, it is decremented. All drawings can be erased and the counter goes to back to zero.

Menu Module. This class prepends invisible divs to the body of the HTML file. They become visible when the user presses the right mouse button to choose an operation to be performed. They are also used for the dialog that is actually an HTML form that does not really execute anything, but instead, its fields are interpreted by the application to perform operations.

Example 45. Menu Module

```
$("body").prepend('\
 <div id="canvasMenu">
     <ul id="canvasItems"></ul>
    </div>
 ...
');

$("#canvasItems").append('<li onclick="menu.openInputDialog(
   CREATE_OBJECT)"><u>C</u>reate object</li>');
...

$("body").prepend('\
<div id="objectInput">\
    <form>\
        ...
    </form>\
</div>\
...
');

/*
Open input dialog
*/
this.openInputDialog = function (opt) {...}

/*
Execute input dialog
*/
this.executeInputDialog = function() {...}

/*
Close object input dialog
*/
this.closeObjectInputDialog = function () {...}

/*
Close morphism input dialog
*/
this.closeMorphismInputDialog = function () {...}

/*
Reset input dialog
*/
this.resetInputDialog = function() {...}

/*
Close input dialog
*/
this.closeInputDialog = function() {...}
```

Which operations are to be shown to the user depends on where he clicked. When the canvas is clicked, the user can:

1. create an object;
2. select all elements;

3. select all objects only;
4. select all morphisms only;
5. select all endomorphisms only.

When an object is clicked, the user can:

1. create a morphism to another object;
2. create a morphism from another object;
3. read information of the object;
4. update the object;
5. delete the object;
6. select the object.

When a morphism is clicked, the user can:

1. change the type of the morphism;
2. read information of the morphism;
3. update the morphism;
4. delete the morphism;
5. select the morphism.

The background color of the menu is shaded differently, depending on where the user clicked, so he can know if he clicked in the correct place. That is because there is a separate div for each type of element.

State Module. This class stores private arrays that are used to maintain previous moments of the application. It keeps a stack of the operations performed, called actions in the context of this module. Almost every operation done with either keyboard, mouse or menu/menu+dialog produces a new action. An action contains a description, which in turn contains a JS object with elements (objects and/or morphisms) and, depending on the operation, the object may also have some other auxiliary field.

Example 46. State Module

```
/*
doneActions/undoneActions: string in {
 "changeType",
 "createEndomorphism", "createMorphism", "createMorphismFrom", "
    createMorphismTo", "createObject",
 "cutSelected",
 "deleteEndomorphism", "deleteMorphism", "deleteObject", "
    deleteSelected",
 "dragEndomorphism", "dragSelected",
 "pasteSelected",
 "updateEndomorphism", "moveMorphism", "updateMorphism", "
    moveObject", "updateObject"
}
states: list of descriptions
currentStateIndex: natural number or -1, indicating empty list
*/
function State() {
    ...

    /*
 Remove undone actions when new action is done
 */
 function removeUndoneActions() {...}

 /*
 Create state with new action
 */
 this.createState = function(action, elements) {...}

 /*
 Go to next state on the stack of states
 */
 this.gotoNextState = function() {...}

 /*
 Go to previous state on the stack of states
 */
 this.gotoPrevState = function() {...}

 /*
 Save current state
 */
 this.saveCurrentState = function() {...}

 /*
 Delete previous state
```

```
  */
  this.deletePreviousState = function() {...}

  this.getDoneActions = function() {return doneActions;};
  this.getUndoneActions = function() {return undoneActions;};
  this.getStates = function() {return states;};
}
```

Thanks to this module, the user is able to undo and redo operations. For safety reasons, all of its variables (4 arrays and 1 counter) are private but can be read if desired (for debugging reasons, mostly).

View Module. By far the longest file in the application, this class would be the view in the MVC pattern, along with the drawing module. The file is long due mostly to CRUD operations on Category Theory elements shown in the Canvas. With them, the user and the system can:

- create the canvas;
- create the square grid;
- create the limiting area in which object can be dragged;
- draw the blur rectangle;
- draw the selection rectangle;
- create new objects;
- create new morphisms with different types;
- create new endomorphisms;
- move these three elements along with their labels;
- select these elements;
- hide deleted elements.

Two things should be pointed out. First, that the canvas is limited to 99% of the page's height and 96% of its width. This was done to prevent the browser from adding a scrollbar. Second, the canvas contains various layers to modularize graphical elements that are being drawn and to give a better visual effect: the backmost layer renders the grid, other layers are for free hand drawings, morphism handles, objects, morphisms, selection rectangle and both the grid border and blur rectangles. By doing that, a selection rectangle will not be rendered on top of objects and morphisms, for example, which would harden the selection of elements.

Looking at the four individual groups of modules using that metric of adding and subtracting a point, we see that the sum of points in the definition group is 19, in the event-driven group, the value is -17, data structure classes add up to 0 and the score for object-oriented modules is -1. This can indicate that the modules are well grouped.

5.5 User Input State Machine

Behind the curtains, the state of the system is controlled by a state machine (which should not be confused with the state module). Without it, it would be much harder to properly handle user interaction. The variable used for that is *keyboardMouseStatus*. Its value is used and modified by all event-driven modules, the menu and the dialog.

Focus, the smallest module to work with the variable, for instance, sets it to one of two possible values when the page focus is lost.

Example 47. User Input State Machine - Focus Module

```
...
if (selectedElements.length === 0) {
 keyboardMouseStatus = "idle";
} else if (selectedElements.length > 0) {
 keyboardMouseStatus = "idle with element(s) selected";
}
...
```

The variable keyboardMouseStatus simply stores a string. A transition in the state machine is just a change in the value of the string.

Example 48. User Input State Machine - Keyboard Module

```
...
if ((evtobj.keyCode === 110 || evtobj.keyCode === 188) &&
    keyboardMouseStatus !== "tsv open") {//, (comma)
 alert(STR_CURRENT_STATE + ": " + keyboardMouseStatus);
}
...
if (ctrlPressed) {
 keyboardMouseStatus = "ctrl";
} else if (evtobj.keyCode === 8) {//backspace
 keyboardMouseStatus = "backspace";
 drawing.eraseLastDrawing();
 keyboardMouseStatus = "idle";
}
...
```

Every valid action with the mouse causes a transition on the state machine (that is when the state machine brings out the benefit of its existence the most, because the system was prone to bugs when it was not implemented). Some of these states are persistent, meaning that that state will not change until something else happens, and some are temporary before quickly transitioning to a persistent state. The same can occur with the keyboard, like in the example above. It was designed this way to imitate the way debugging is done in JS: printing gibberish to the console to find the location where bugs happen. In fact, when fixing the state machine, printing its variable's string constantly can be very helpful for that same reason.

Example 49. User Input State Machine - Mouse Module

```
...
if (leftMousedownOnCanvas) {
 keyboardMouseStatus = "rect start";
 ...
} else if (leftMousedownOnMorphism) {
 keyboardMouseStatus = "deselect all";
 deselectAll();
 keyboardMouseStatus = "morphism update";
 ...
}
...
```

It should be pointed out that there are other auxiliary variables used by the state machine, like those that keep track of the ctrl, shift and tab keys.

Example 50. User Input State Machine - Auxiliary Variables

```
var ctrlPressed = false;
var ctrlReleased = false;
...
```

The other moment in which the state machine stars is when the system has to know if the user is entering values on the dialog or is pressing a shortcut. That is noticeable when the user types something in the dialog and the system does not pop up some system information that would be brought up by that same key if it was not by the state machine.

Figures 5.52 and 5.53 show an early sketch of the state machine for CatViz that is pretty close to how it is now.

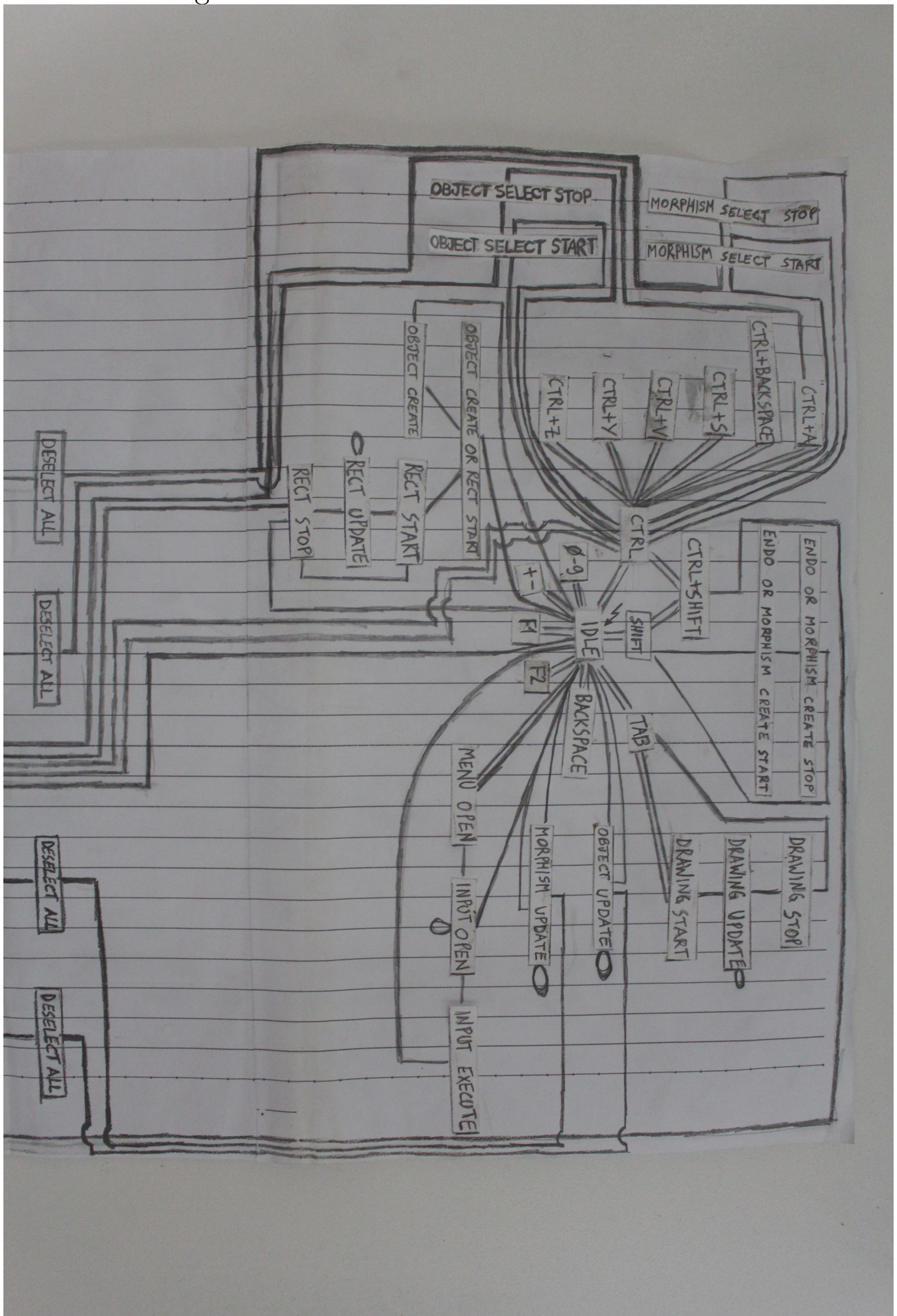
In the images, there are two almost identical sets of states. In the first ones, there are no elements selected. In the second set, there is at least one element selected. States of the second set have the letters WES appended to their names, that mean with element(s) selected. If the user selects elements, the machine transitions to a state in the second set. If there were elements selected but not anymore, the machine transitions to the first set. The state in the opposite set is determined by the state of the machine before it transitions to the other set. This is represented by two rules:

1. Rule 1 to 2: X $\rightarrow$ Y $\rightarrow$ X WES.
2. Rule 2 to 1: X WES $\rightarrow$ Y WES $\rightarrow$ X.

For example, if the user was holding the *ctrl* key down and then pressed the *A* key, the machine goes to *ctrl WES* because the user is still holding the *ctrl* key, but elements are selected (if there are no elements to be selected, the machine would go back to *ctrl*, which is not represented in the image, as it is an early design).

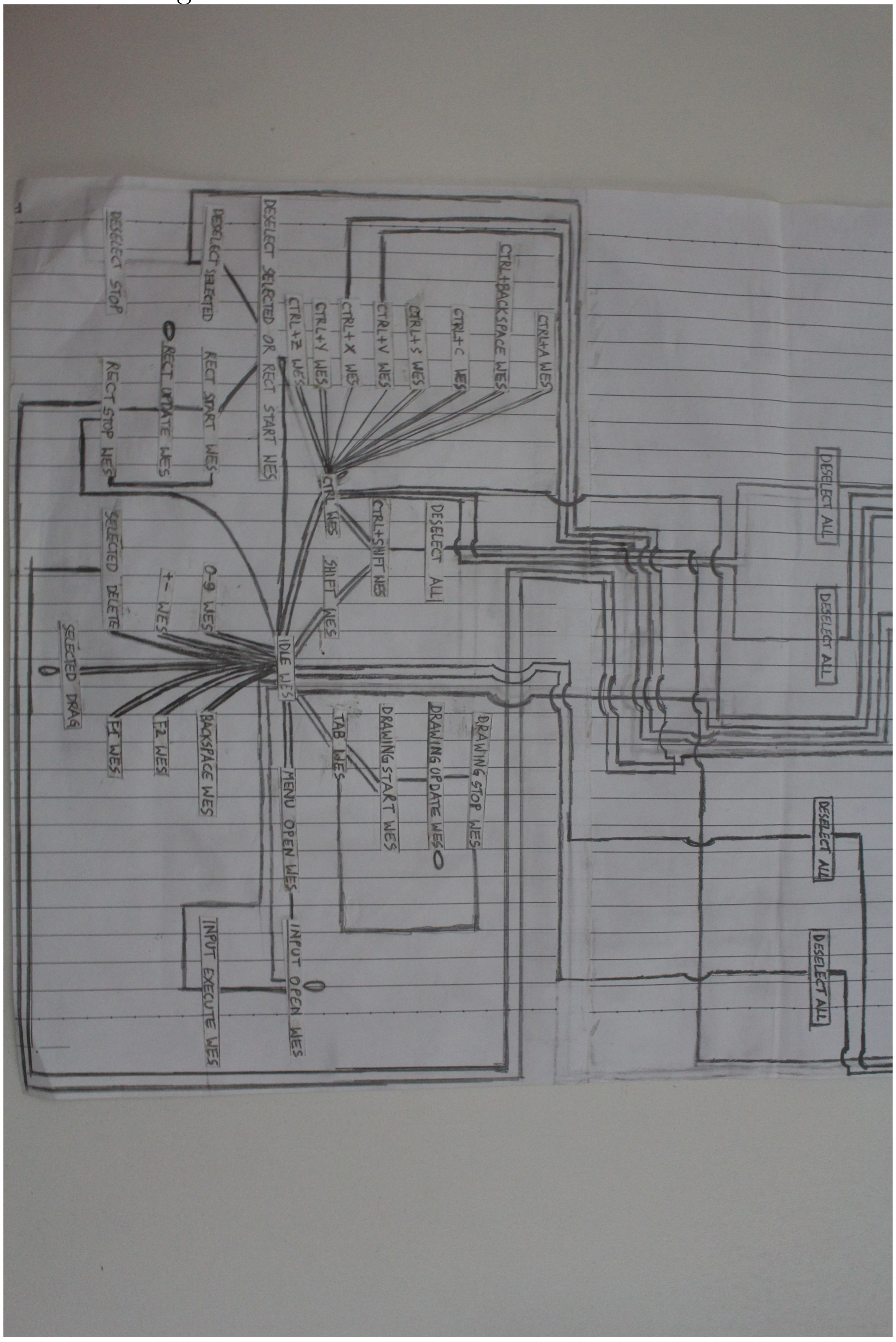
The system starts in state *idle*, keys *0 to 9*, *+*, -, *F1*, *F2* and *backspace* transition quickly to their states and back to *idle*. *Ctrl* and *shift* transition to their own states and their combination transitions to *ctrl+shift*, which is used in combination to a third key to do the opposite of the combination of that key with ctrl (ctrl+a selects all while ctrl+shift+a deselects all). Some keys combined with *ctrl* execute operations and so, cause transitions. With *ctrl* key held down, the user can individually select elements, which causes a transition. With *shift* key held, the user creates morphisms and endomorphisms. With *tab* key held, the user draws. In the sketch, when he presses the left mouse button down on the canvas, the system cannot yet distinguish if he is going to create a new object or use the selection rectangle, if he releases the button, a new object is created and a quick transition occurs, otherwise, if the user moves the mouse even a pixel, the system considers that he is using the rectangle, and the machine keeps looping while the mouse is moved and not released. Currently, the system can

Figure 5.52: CatViz state machine - first set



Source: The Author

Figure 5.53: CatViz state machine - second set



Source: The Author

distinguish those two sequences by not allowing the selection rectangle to be created with the *shift* key being held, while on the other hand, objects are created with that key pressed. If the user clicks on an element, the system already considers it to be an element update. The right mouse button opens the menu, the correct sequence of steps flows through specific states, otherwise, the machine goes back to idle. There are multiple states called *deselect all* that transition to different states, that is why they are homonymous but not completely equivalent.

When the machine is idle but elements are selected, the state is actually *idle with element(s) selected*. There are many states that do the same: an operation like those previously explained but with elements that were selected beforehand. The operation may not be applied to these elements, so sometimes the system has to deselect them all and the machine has to transition to a state of the first set, according to rule 2 to 1. There are two states that are opposite to those without elements selected: *deselect selected* and *deselect all*. They transition to the first set according to the same rule. With selected elements, the user can also delete them with that key or drag them with the mouse.

As mentioned before, all this states are represented by the value of the variable keyboardMouseStatus, which is a string that contains the strings in the images.

5.6 State Stack

Even though they share the name, these states are not those of the state machine of the system. These are called states because they are to be used to save the state of the diagrams with the right combination of keys so the student can show the sequence of his demonstration of a theorem, for instance. It was designed to be used later with an interactive theorem prover but now it can also be used to undo and redo operations, in case the student thinks he made a mistake in the assignment. Perhaps a better name to this module would be Step.

There are three rules that govern the stack:

1. a new state must be pushed to the top of the stack when the user performs a relevant action;
2. the last action pushed in has to be popped out when he wishes to undo the action;
3. it has to be reinserted when he wishes to redo the action.

Obviously, these operations must be done strictly in this order, but the second and the third are optional. The relevant actions are:

1. changing the type of a morphism;
2. creating a new endomorphism;
3. creating a new morphism;
4. creating a new morphism from a specific object;
5. creating a new morphism to a specific object;
6. creating a new object;
7. cutting selected elements;
8. deleting an endomorphism;
9. deleting a morphism;
10. deleting an object;
11. deleting selected elements;
12. dragging endomorphisms;
13. dragging selected elements;
14. moving a morphism;
15. moving an object;
16. pasting selected elements after copying or cutting them;
17. updating an endomorphism;
18. updating a morphism;
19. updating an object.

Operations 8, 12 and 17, involving endomorphism manipulation require it to be previously selected, which means that the operations are actually based on a selected element.

5.7 Current Status and Future Work

Currently, CatViz has more than 8000 lines of code, as many as each third-party library, jQuery and D3.js. These libraries are mostly used to replace pointer methods in a more readable way. This benefit is visible in the model and view part of the application, that need to point to the elements of the DOM and modify them. CatViz contains 14 modularized script files with mostly authoral code. Few code snippets were modified versions of existing code: in these cases, authorship is properly referenced in comments in the source code. CatViz source code and main page are available from the following Github project repository:

<http://github.com/gmramella/CatViz>.

The help page details how to use the tool and defines the format of an e-mail for bug reporting. The currently implemented features of CatViz include:

- manipulation (CRUD) of objects;
- manipulation (CRUD) of morphisms, including morphism types (mono, epi, monoepi, iso);
- text localization;
- brush for drawings and annotations on top of diagrams;
- undo/redo stack.

CatViz, however, is not complete. Some important operations from Category Theory which are not currently implemented include:

- a table of current morphism compositions;
- a morphism composition operation;
- open and save diagrams as files;
- record of a sequence of diagram snapshots (to record proofs).

Finally, there are still some bugs that remain to be fixed, which are well described in the project's repository.

5.8 Related Work

Two previous works developed similar editors, CaTLeT (PFEIFF, 2002) and CaTReS (VIEIRA, 2006). Although the set of features is similar, both are built upon Java Technology as desktop applications, requiring the Java Virtual Machine environment previously set up in the user device. This may cause some problems regarding compatibility with newer platforms and, in some educational settings, the Java Runtime Environment may be blocked due to security reasons. Web-based technology is more pervasive and easier to integrate with platforms such as Moodle. On the other hand, the mentioned tools are in a more mature state of development, allowing the user to create, edit and save diagrams.

6 CONCLUSION

Category Theory is an interesting area of mathematics with important applications in Computer Science. For being very abstract, however, it is considered challenging for students at first contact. The fact that its definitions and proofs rely mostly in diagram manipulation could be explored to assist the study of categorical concepts by means of suitable learning tools.

This text presented two contributions: a survey on the basic categorical constructions, and an web-based visual editor for categorical diagrams, named CatViz. The initial chapters established many mathematical definitions on set theory and category theory. Later chapters focused on the technologies and the development of CatViz.

Although CatViz is not yet complete or free of bugs, it is usable to present basic concepts and could be utilized to demonstrate categorical constructions by means of diagrams at the classroom. The next planned features, in particular morphism composition, would allow CatViz to be used to demonstrate categorical proofs by means of diagram manipulation. Another interesting feature would be registering diagram manipulation steps. These could then be animated or even validated by generating a constructive proof in an interactive theorem prover like Coq, for instance.

I hope CatViz can be improved by the community and that it helps many people to learn and enjoy Category Theory.

REFERENCES

ANTON, H.; BIVENS, I.; DAVIS, S. **Cálculo Volume I**. 8. ed. Porto Alegre: Artmed Editora SA, 2005.

BOSTOCK, M. **D3: data-driven documents**. 2018. Available from Internet: <https://d3js.org/>.

BRONSHTEIN, I. N. et al. **Handbook of Mathematics**. 5. ed. Berlin: Springer-Verlag, 2007.

BURRIS, S. N.; SANKAPPANAVAR, H. P. **A Course in Universal Algebra**. 1. ed. New York: Springer-Verlag, 1981.

BUSCHMANN, F. et al. **Pattern-oriented Software Architecture: A System of Patterns**. Chichester: John Wiley & Sons Ltd., 2001.

CONSORTIUM, W. W. W. **W3C**. 2018. Available from Internet: <https://www.w3.org/>.

DATA, R. **w3schools.com: the world's largest web developer site**. 2018. Available from Internet: <https://www.w3schools.com/>.

FOUNDATION, T. J. **jQuery user interface**. 2018. Available from Internet: <https://jqueryui.com>.

FOUNDATION, T. J. **jQuery: write less, do more.** 2018. Available from Internet: <https://jquery.com/>.

LANE, S. M. **Categories for the Working Mathematician**. 2. ed. New York: Springer-Verlag, 1998.

MENEZES, P. B. **Matemática Discreta para Computação e Informática**. 2. ed. Porto Alegre: Artmed Editora SA, 2005.

MENEZES, P. B. **Linguagens formais e autômatos**. 6. ed. Porto Alegre: Artmed Editora SA, 2011.

MENEZES, P. B.; HAEUSLER, E. H. **Teoria das Categorias para Ciência da Computação**. 2. ed. Porto Alegre: Artmed Editora SA, 2008.

MENEZES, P. B.; TOSCANI, L. V.; LóPEZ, J. G. **Aprendendo matemática discreta com exercícios**. 1. ed. Porto Alegre: Artmed Editora SA, 2009.

NETWORK, M. D. **Object-oriented JavaScript for beginners**. 2018. Available from Internet: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS>.

NLAB. **All pages**. 2018. Available from Internet: <https://ncatlab.org/nlab/all_pages>.

PFEIFF, F. V. **CaTLeT : ferramenta computacional de apoio ao ensino/aprendizado de teoria das categorias**. Dissertation (Master) — UFRGS, 2002.

PIERCE, B. C. **Basic Category Theory for Computer Scientists**. 1. ed. Cambridge: MIT Press, 1991.

ROSEN, K. H. **Matemática Discreta e Suas Aplicações**. 6. ed. New York: McGraw-Hill, 2009.

RYAN, B.; RONAN, C. **Head First jQuery**. 1. ed. [S.l.]: O'Reilly Media, Inc., 2011.

SEBESTA, R. W. **Programming the World Wide Web**. 6. ed. Boston, MA, USA: Addison-Wesley Publishing Company, 2010.

SOMMERVILLE, I. **Software Engineering**. 9. ed. USA: Addison-Wesley Publishing Company, 2010.

VIEIRA, R. B. **CaTReS : ferramenta de apoio à pesquisa e ensino em teoria das categorias**. Dissertation (Master) — UFRGS, 2006.

WILLARD, R. An overview of modern universal algebra. In: **Logic Colloquium**. [S.l.: s.n.], 2004. p. 197–220.