UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

GRAPH THEORY IN LAMBDA CALCULUS

Absolvent

Adela-Nicoleta Corbeanu

Coordonator științific

Conf. dr. Traian Șerbănuță

București, iunie 2024

Abstract

During the past few years, science has witnessed the discovery of surprising common traits between two seemingly independent fields, namely lambda calculus and graph theory. Topics such as (but not limited to) the typing of lambda terms and the four color theorem have been found to be closely related, both on an intuitive and a rigorous mathematical level. The current thesis aims to perform a well structured condensation of the most relevant research in this area of study, as well as provide any necessary background. To achieve this goal, we begin by discussing topics such as topology, graph theory, category theory, combinatorics and lambda calculus, and culminate with transitions between graphs and lambda terms and vice-versa. Little to no prior knowledge in mathematics and computer science is assumed. Lastly, we explore potential future progress and impact of this novel subject.

Rezumat

În ultimii ani, știința a fost martoră la descoperirea unor surprinzătoare trăsături comune între două domenii aparent independente, și anume lambda calcul și teoria grafurilor. Aspecte precum determinarea tipurilor lambda-termenilor și teorema de 4-colorare

(și nu numai) s-au dovedit a fi strâns conectate, atât la nivel intuitiv, cât și riguros matematic. Lucrarea curentă își propune să creeze o centralizare bine structurată a celei mai relevante cercetări din această arie de studiu, precum și să pună la dispoziție cunoștințele preliminarii necesare. Pentru a îndeplini acest scop, începem prin a discuta subiecte precum topologie, teoria grafurilor, teoria categoriilor, combinatorică și lambda calcul, și culminăm cu tranziții între grafuri și lambda-termeni și vice-versa. Foarte puține cunoștințe anterioare de matematică și informatică sunt necesare. În ultimul rând, vom explora potențial viitor progres și impact al acestui subiect inedit.

Contents

Chapter 1

Introduction

1.1 Motivation & How the topic chose me

In the following paragraphs, I will detail the factors that determined me to choose this topic for my thesis. However, note that I strongly believe that the topic chose me rather than the other way around.

Lambda calculus and the notorious four color theorem from graph theory are things that have seemingly very little in common and one would not naturally associate them. This is the reason why my surprise was not small when I stumbled across a presentation that supposedly explored precisely this association.

Having recently been introduced to lambda calculus, I was quite fascinated with it and this led to the night when I randomly had a curiosity about lambda calculus that I was not able to figure out on my own. Browsing the web in search for answers, I stumbled upon a presentation whose title I had to read twice because, at least to me (a lambda calculus novice), it seemed too shocking and unexpected to be real. The title read ”From Lambda Calculus to the Four Color Theorem” and it made me so confused that I ended my browsing session and even forgot what I was searching for in the first place.

I realized that I was on the obscure side of the internet, far from the information I was looking for. In the heat of the moment, I even told some friends about the absurdity of my finding and we had a good laugh about how I reached the ”bottom” of the internet.

Over the following month, my curiosity only grew stronger. Eventually, I revisited the presentation. Although it initially made little to no sense, my understanding improved bit by bit with each visit. I also noticed that the presentation was quite recent, and I even found a recording of the author holding the presentation in question during a seminar. Watching it multiple times only deepened my fascination as well as raised more questions, particularly due to my lack of necessary background at that moment.

In the end, I connected the dots and I realized: this topic is innovative, I have an unexplainable continuously growing interest in it, it is challenging, and I would enjoy sharing it with other people. What other motivation could one need for his thesis?

1.2 Purpose

The current thesis primarily aims to centralize the most relevant research and results concerning the connections between lambda calculus and graph theory. On a secondary level, the purpose is to achieve the previous goal in an accessible manner, as well as provide a comprehensive overview of the background necessary for understanding the concepts presented. We can expect to contour a general idea of the topic along with view it through multiple perspectives. We will not delve too deeply into complex formalities when they are not vital to the understanding of the subject, but I would confidently say that this thesis provides more than just an introduction, offering instead a foundation to further study any of the mentioned concepts or even track future published progress.

We begin with the topic's motivation in the first chapter. We continue with the rather ample second chapter, where we explore various preliminary subjects, ranging from topology to category theory and lambda calculus. Next, in the third chapter we dive straight into the topic and present important connections between lambda calculus and graph theory. In the fourth chapter we discuss additional perspectives that the research in this area brought upon lambda calculus, graph theory and even other related fields. In the same chapter we also detail the factors that triggered these discoveries. The last chapter consists of conclusions, emphasizing the importance of the topic, its short history, the current state of research, future expectations, and the potential impact of the results.

Chapter 2

Prerequisites

2.1 Maps

Maps are a fundamental element in rather mathematical fields such as topology and graph theory. The terms “map”, “map graph” and “planar graph” are all closely related and are often used interchangeably. However, the definitions of these terms tend to slightly vary among different scientific articles. We will define all of them in this chapter, but we will start with some introductory concepts.

2.1.1 Topology Concepts

As an introduction, we will begin by defining a metric space.

By “metric space”, we mean the association between a set and a relation (a function) between its elements considered the “distance” between them. The function that gives the distance between two elements of the set is called a metric (therefore the name “metric space") or, more popularly, a distance function. The elements of such set are usually called points. [7]

Formally, a metric space is defined as a pair consisting of a set S and a distance function $d : S \times S \rightarrow \mathbb{R}$, such that the distance from a point to itself is 0, the distance between any two distinct elements is positive, d is symmetric and d satisfies the triangle inequality.

A very common metric space is the association between the real numbers and the absolute difference between them $d(x, y) = |x - y|$, where $x, y \in \mathbb{R}$. Another common example would be the one involving the Euclidean plane as the set, which has had many metrics associated with it, such as the Euclidean distance

$$d(x, y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where $x_1, y_1, x_2, y_2 \in \mathbb{R}$ or the Manhattan distance

$$d(x, y) = |x_2 - x_1| + |y_2 - y_1|$$

where $x_1, y_1, x_2, y_2 \in \mathbb{R}$.

At a high level, geometry is about measuring quantities that can be used to formulate properties about the space being studied. Besides geometry, there is also a science that studies spaces in a rather qualitative manner (for example, answering whether a space is connected or not). This science is called topology.[2]

Intuitively, a topological space is a space together with a notion (function) of nearness. In the case of metric spaces, the nearness is given by the distance function. That is, all metric spaces are topological spaces.

In mathematics, an *isomorphism* is a function F that maps two entities in a way that preserves their structure and the inverse function F^{-1} is the reversed mapping.

If we think about a geometric object, we can notice that we are able to deform it by stretching, twisting, crumpling and bending. We are interested in such deformations that do not create new holes in the object, do not fill current holes of the object, do not tear/glue the object and do not make the object pass through itself.

A property of a geometric object (or even a space) that still holds after continuous deformations such as the ones mentioned above is called a *topological property*. Common examples are the cardinality of the space and the connectedness (for example, differentiating between a single object and two separate objects that do not intersect).

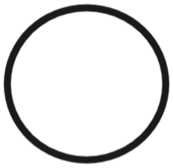


Figure 2.1: A connected object

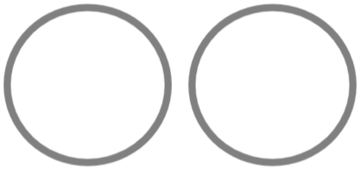


Figure 2.2: A disconnected object

A deformation that preserves all of a space's topological properties is called a homeomorphism. Two spaces that have a homeomorphism are said to be homeomorphic, with the meaning that they are “the same” from a topological perspective. Homeomorphisms are the isomorphisms in the category of topological spaces. For example, a cube and a sphere are homeomorphic, as they can be deformed into each other while respecting the rules above. However, a sphere and a torus (“donut”) are **not** homeomorphic, because we cannot deform a sphere into a torus without piercing a hole into it, nor can we deform a torus into a sphere without filling its hole. By hole we mean a hole through which something can pass, not just a dent (for example, we consider that a cup with no handles does not have any holes).

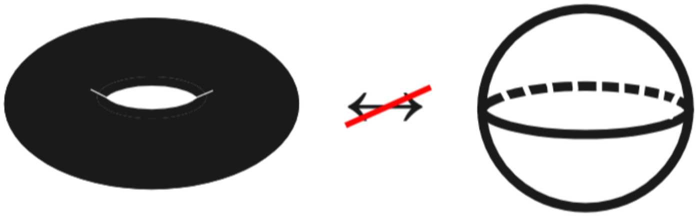


Figure 2.3: A torus (one hole) is **not** homeomorphic to a sphere (no holes).

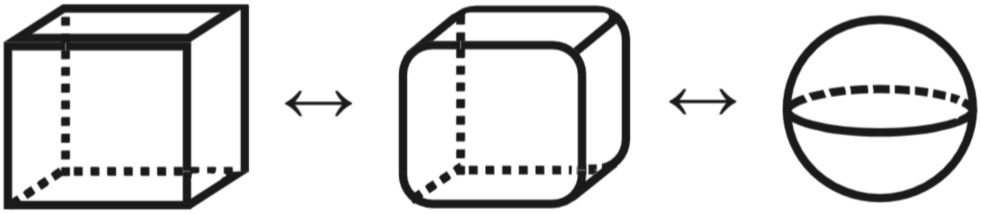


Figure 2.4: A cube (no holes) is homeomorphic to a sphere (no holes).

We call the number of holes in an object the *genus* of the object. For example, a sphere has a genus $g = 0$, while a torus ("donut") has a genus $g = 1$.

2.1.2 Maps Definitions and Classification

First, we want to discuss about graphs (as in graph theory). The most common definition for a graph is the pair $G = (V, E)$, where V is the set of vertices (also called *nodes*) and E is the set of edges, each edge being an unordered pair (x, y) with the meaning that the edge links vertex x to vertex y and vice-versa.

The above defines an *undirected* graph. Graphs can also be *directed*, where an edge is an **ordered** pair *(x, y)* with the meaning that the edge only links vertex x to vertex y.

A very popular technique is representing graphs visually. However, if we look at the above definition, we notice that there is no information about the way the graph's elements are arranged. This tells us that there is no unique way of drawing a given graph. In Figure 2.5 we can see the same graph drawn in two different ways:

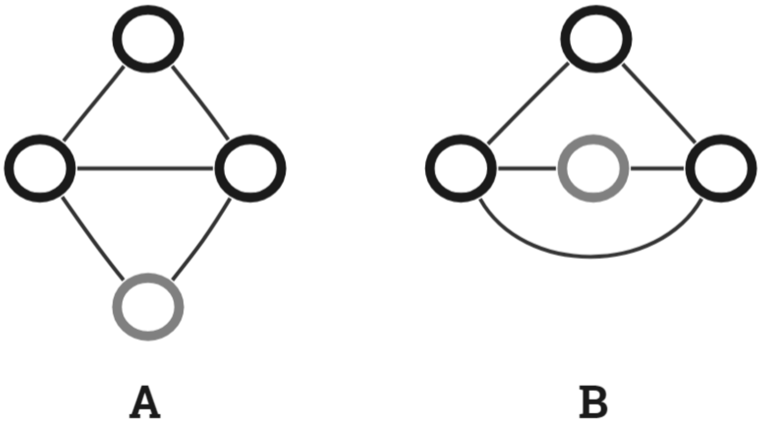


Figure 2.5: Two ways of drawing the same graph

Given two graphs, no matter their visual representation, if there exists a mapping (a bijection that acts as an isomorphism) between them, we say they are *isomorphic* ("equivalent"). The graphs in Figure 2.6 are isomorphic.

The graph isomorphism problem (deciding whether two graphs are isomorphic) is thought to be in the (rather interesting) **NP-intermediate** computational complexity class, i.e. neither polynomial nor NP-Complete, but it has not been proven yet. Whether the problem can be solved in polynomial time or not is an open question in computer science.

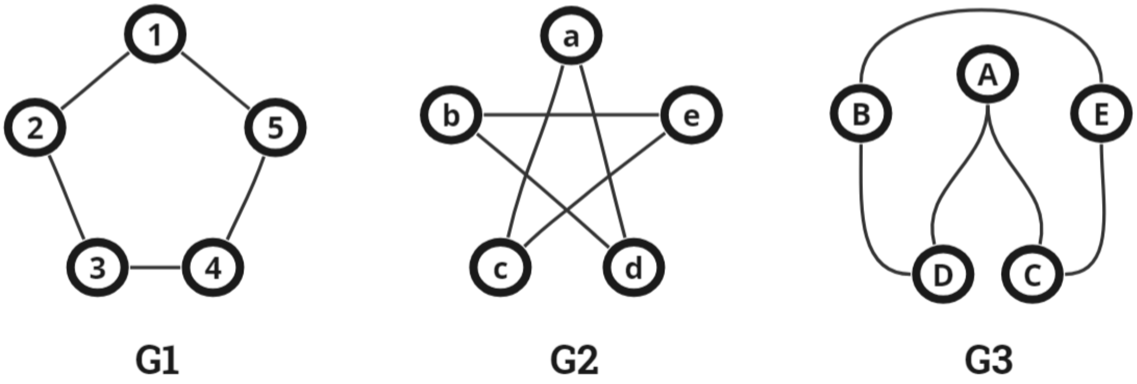


Figure 2.6: Three isomorphic graphs $G1 \cong G2 \cong G3$

If we think about the multitude of ways in which a graph can be represented visually, we may notice that some graphs can be represented in a way that no two edges cross each other, while other graphs cannot. The graphs that can be drawn in that way are called *planar graphs*.

From a topological point of view, a planar graph can be embedded in the plane without its edges crossing. Similarly, it can also be embedded onto a sphere. Both the plane and the sphere have a genus $g = 0$, and this is an alternative definition of a planar graph: a graph that can be embedded into an object of genus $g = 0$.

Graphs also have a genus, and it is given by the minimum number g such that the graph can be embedded into an object of genus g, without its edges crossing. For example, the graph K_5 (the complete graph with 5 vertices) cannot be drawn on the sphere, but it can be drawn on the torus, therefore it has a genus $g = 1$.

A planar graph drawn in this way (with no intersecting edges) is called a *plane graph*. The regions delimited by the edges of a plane graph are called *faces* and their number can

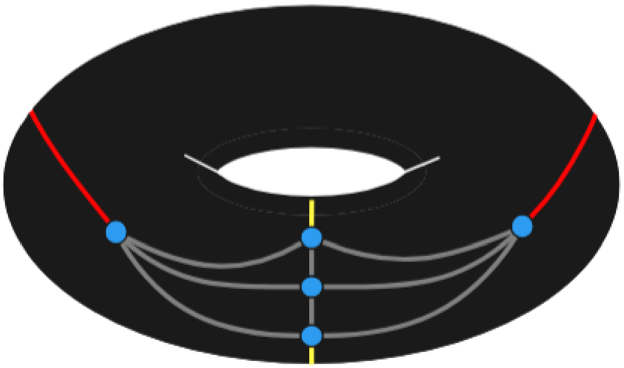


Figure 2.7: The K_5 graph embedded into the torus

be computed by the Euler characteristic (which equals 2 for planar graphs): $F = 2+E-V$, where F is the number of faces, E is the number of edges and V is the number of vertices.

A disk is the set of all points inside a circle. If the disk includes the points on the circle (the boundary), it is called a *closed disk*, otherwise it is called an *open disk*. A 2-cell embedding is an embedding in which every face of the graph is homeomorphic to an open disk. Such embedding is more commonly known as a ***map***. [11]

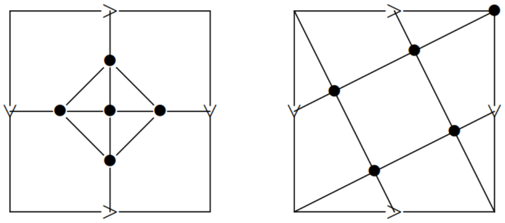


Figure 2.8: Two different embeddings of the K_5 graph into a torus. The first one is a 2-cell embedding, while the second one is not. Source: [11]

For genus $g = 0$, a *map* is a graph embedded into the plane in a way such that its edges do not intersect each other. That is, the *plane graph* is a particularization of a map, more precisely, it is the map of genus $g = 0$. Initially, a *map* was called a *planar map*, but over time the term has been more popularized as simply *map*. [21]

Considering the definitions above, we can see in Figure 2.9 how the first two graphs are equivalent both as maps and as graphs, but the last two graphs are only equivalent as graphs. The reason for which they are equivalent as graphs is that they have the same number of vertices and any edge connects the same two vertices in all the graphs (their set of vertices and their set of edges are the same among all three of them). However, map equivalence further requires *faces* equivalence, meaning that an edge should belong to the same face among all three maps. For example, the edge connecting vertices 1 and 3 belongs to the interior face in the first two graphs, but it belongs to the exterior face in the third one.

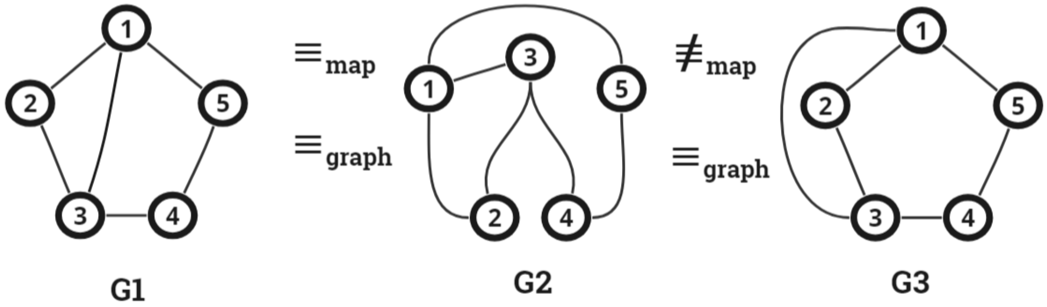


Figure 2.9: Three equivalent *graphs*, but **not** three equivalent *maps*

If we think about the faces of a planar map, we notice that any two faces always have one or more of the following: a common edge, a common vertex, or nothing in common at all. We can construct a related graph by turning each face into a vertex and drawing edges between vertices if their corresponding faces have at least one common *edge*.

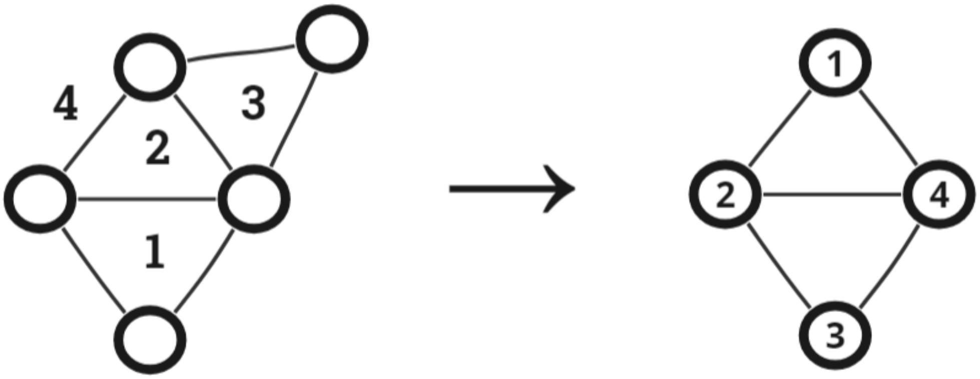


Figure 2.10: A graph and its dual

This newly-constructed graph is called the *dual graph* of the initial one. It may sometimes even be called a *map graph*, because if we think of a real-life geographical map, this is most likely how we would represent it as a graph.

If a plane graph and its dual are isomorphic, we say that the graph is *self-dual*.[20]

2.2 Intermediary Concepts

2.2.1 Introduction to Category Theory

In the field of mathematics, people often seek abstraction and generalization. This is where category theory comes in, a relatively young branch of mathematics, sometimes considered the most abstract and general one. It aims to generalize objects (any mathematical structures) by placing them in groups/classes called categories. Like lambda calculus, it stays behind numerous concepts in functional programming.[9]

Because category theory generalizes mathematical concepts, the definition of a category will most likely not look unfamiliar to someone with a little experience in mathematics. Very broadly, a category consists of two types of entities: objects ("object" is an extensive term, which works perfectly with the purpose of categories) and relations between the objects. The said relations are called *morphisms*, as well as *arrows* (this is the rather popular term) or *maps* (because they map one object to the other).

More formally, a category C consists of two collections:

1. Objects – $Ob(C)$
2. Arrows – $Ar(C)$

- To each arrow we associate the pair $(source, target)$ with the notation $f : A \to B$ and the meaning that f is a morphism (mapping) from $source$ (also called *domain*) object A to $target$ (also called *codomain*) object B.
- Given two arrows $f : A \to B$ and $g : B \to C$, there is a third one, $g \circ f : A \to C$ called the *composite arrow* of g and f.
- For each object A, there is an arrow $\mathrm{id}_A : A \to A$ (also written as 1_A or just 1) called the identity of object A.

Arrows are governed by the following two axioms: [3]

1. The identity morphism axiom:

$$1_B \circ f = f = f \circ 1_A \quad \forall f : A \to B$$

2. The morphism associativity axiom:

$$h \circ (g \circ f) = (h \circ g) \circ f \quad \forall\, f : A \to B,\, g : B \to C,\, h : C \to D$$

Below, we have the category containing objects A, B, C, morphisms f, g, $f \circ g$, identity morphisms 1_A, 1_B, 1_C. The figure does not come as a surprise, as we have often seen it in the context of function composition!

$$\begin{tikzcd}
\boldsymbol{A} \arrow[r, "\boldsymbol{f}"] \arrow[loop above, "\mathbf{1}_{\boldsymbol{A}}"] \arrow[dr, "\boldsymbol{g \circ f}"'] & \boldsymbol{B} \arrow[d, "\boldsymbol{g}"] \arrow[loop right, "\mathbf{1}_{\boldsymbol{B}}"] \\
& \boldsymbol{C} \arrow[loop right, "\mathbf{1}_{\boldsymbol{C}}"]
\end{tikzcd}$$

Oftentimes (although not always true!), given two particular objects A and B from the same category, it is assumed that the totality of morphisms with source A and target B form a set, called a *homset* and written $Hom(A, B)$.

Categories can also have a notion of *product* between elements, whose goal is to emphasize the essence of how some objects are actually constructed. Formally defining this said product is outside of our scope, but we will provide some intuition in the upcoming examples.

Some of the most common categories include, but are not limited to:

- **Set** – the category of sets and set functions, where the composition of morphisms is the basic, well known, composition of functions (careful, it is not mentioned anywhere that the morphisms in category theory are necessarily functions!); every requirement for forming a category is checked: the composition of functions is associative, each set has an identity function. Moreover, the product of two sets is none other than the *Cartesian product*!

- **Grp** – the category of groups and homomorphisms. Its product is the group-theoretical analogue of the Cartesian product.

To name few of the most popular morphisms, we can think of:

- **Isomorphism** – mapping that can be reversed through an inverse mapping
- **Epimorphism** – a morphism f for which $g1 \circ f = g2 \circ f$ implies $g1 = g2$ for all morphisms $g1, g2 : A \rightarrow B$
- **Monomorphism** – a morphism f for which $f \circ g1 = f \circ g2$ implies $g1 = g2$ for all morphisms $g1, g2 : A \rightarrow B$
- **Bimorphism** – a morphism that is both an *epimorphism* and a *monomorphism*
- **Endomorphism** – a morphism from an object to itself
- **Automorphism** – a morphism that is both an *endomorphism* and an *isomorphism*

Now, after discussing all these kinds of mappings between objects of the same category, one may wonder whether categories can be mapped to other categories. And they can! A mapping between two categories A and B is called a *functor* and it associates an object X from A to an object $F(X)$ from B, as well as associates each morphism $f : X \to Y$ from A to a morphism $F(f) : F(X) \to F(Y)$ from B, while preserving the following properties:

- Identity morphism:

$$F(\mathrm{id}_X) = \mathrm{id}_{F(X)} \quad \forall X \in A$$

- Composition of morphisms:

$$F(g \circ f) = F(g) \circ F(f) \quad \forall\, f : X \to Y,\, g : Y \to Z,\, f, g \in A$$

2.2.2 Combinatorial Classes

In combinatorial mathematics, enumeration and counting of various objects are subjects of great interest. There are some formal structures that allow these objects to be handled in a systematic and organized manner. The structures we are referring to are called *combinatorial classes*.

By *combinatorial class* we denote the association of a set of objects and a function known as the weight (or size) function posessing some particular properties. Let the set of objects be C. The weight function is a function $f : C \to \mathbb{N}$ such that for any weight (any value of f) there are a finite number of objects of that weight in C. Formally, the set

$$C_n = \{x \in C \mid f(x) = n\}$$

that contains all objects from C of weight n has to be finite for all values $n \in \mathbb{N}$.[17]

Each combinatorial class has a *counting sequence* associated with it. This counting sequence a_n denotes the number of objects in C of weight n:

$$a_n = |C_n|$$

2.3 Lambda Calculus

2.3.1 General Concepts

Lambda calculus is a formal system that, to some extent, acts as a notation for functions and application. [1] From a computational point of view, lambda calculus (Church, 1936 [1]) notoriously stands on the same level as Turing machines (Turing, 1936-1937) and recursive functions (Gödel, 1931) [13], as these are the three classical models of computation that were used to define the notion of computability, all equivalent and developed during the same period of time. [15]

First we will discuss the untyped lambda calculus and we will be referring to it simply as lambda calculus. Consider the lambda-term $\lambda x.x^2$. This is the representation of the function that takes an input x and returns the input squared. Not to be confused with an instruction, as in imperative programming. Now, $(\lambda x.x^2)(3)$ is the application of $\lambda x.x^2$ to the argument 3, precisely the result returned by the function when it is run with the input $x = 3$.

In lambda calculus there are two main basic operations:

[1] Church did publish (versions of) lambda calculus earlier, but in 1936 he published solely the untyped lambda calculus, that is, the part of lambda calculus which is relevant to computation. [6]

1. **Application**

   MN – the term M is applied to the term N

$$(\lambda x.x^2)(3) \longrightarrow M = \lambda x.x^2, N = 3$$

2. **Abstraction**

   $\lambda x.M$ – function that takes x as an argument and has the body M

$$\lambda x.x^2 \longrightarrow M = x^2$$

As seen above, we have used the term *lambda term*. A lambda term is a lambda calculus expression that is valid from a syntactic point of view. A lambda term can be any of the following:

- Variable
- Application
- Abstraction

From now on, we will denote variables by lowercase letters (x, y, etc.) and lambda terms by uppercase letters (M, N, etc).

In the term $\lambda x.x^2$, x is a variable. The term is equivalent to $\lambda y.y^2$, $\lambda z.z^2$, etc. That happens because x is what we call a *bound* variable (the equivalent of a *local* variable in programming). Lambda terms can also have *free* variables, for example$\lambda x.y^2$has the free variable y, because it is not bound to any lambda. In the term$\lambda x.yx$, x is bound and y is free. However, the expression is not equivalent to $\lambda y.yy$, because this equivalence would bind the variable y that was initially free.

Such equivalence between lambda terms is called *alpha equivalence* (also spelled *α-equivalence*) and it aims to emphasize that the choice of bound variables is not significant as long as it preserves the initial bindings of all variables and it does not shadow other namings (i.e.: does not change the initial sense of the term). The action of renaming variables in order to obtain an α-equivalent term is called *α-conversion* (also spelled *alpha-conversion*).

When a variable x is free, we call that occurrence of x a *free occurrence*.

Formally, to denote α-equivalence of two terms, we write $\lambda x.M =_\alpha \lambda y.(M \langle y/x \rangle)$ (if y is not in M) and we read the term on the right as *"the function that takes y as input and has the body M in which x is replaced by y"*. What we did here by taking the term M and replacing the occurrences of the bound variable x inside it with the new bound variable y can also be done with lambda terms instead of variables, except that in this case we only replace the free occurrences of variables.. This action is known as substitution. Same as α-conversion, substitution also must not change the initial bindings of variables. To denote substitution, we write $\lambda y.(M[N/x])$ and we read *"the function that takes y as input and has the body M in which the free occurrences of the variable x are replaced by the term N"*.

Examples:

- $(\lambda y.x)[y/x] \equiv \lambda y'.y \equiv \lambda z.y \not\equiv \lambda y.y$
- $(\lambda y.x)[(\lambda z.zw)/x] \equiv \lambda y.\lambda z.zw$

When there are chained abstractions such as $\lambda x.\lambda y.\lambda z.M$, they can be written simply as $\lambda xyz.M$.

Operators precedence:

In lambda calculus, application is left associative. This means that MNP is the same as $((MN)P)$, where M, N, P are lambda terms.

Abstraction extends to the right as far as possible: $\lambda x.MN$ is the same as $\lambda x.(MN)$, but not the same as $(\lambda x.M)N$.

These rules blend together perfectly in defining the lambda terms. Moreover, application has higher precedence than abstraction (as previously stated, $\lambda x.MN$ is the same as $\lambda x.(MN)$)!

Combinatory logic in computer science:

In computer science, a *higher order function* is a function that either takes functions (one or more) as arguments or returns a function as the result.

In mathematics, a *fixed point* of a function is a point that is mapped to itself by the function. Formally, given a function f and a point x, x is a fixed point of f iff $f(x) = x$. A function may have multiple fixed points. Moreover, if a topological space has a fixed point it is said to have the fixed point property (FPP), which is a topological invariant!

In lambda calculus, a lambda term with no free variables is called a *combinator* (or a closed lambda term). A *fixed-point combinator* is a higher order function that takes a function as the input and returns a fixed point (if it exists) of that given function. [12]

If we take a fixed-point combinator fpc and a function f, then $fpc(f) = f(fpc(f))$.

Fixed-point combinators can be conveniently expressed using lambda terms, which is what makes recursion possible in lambda calculus.

2.3.2 β-reduction

We can notice that some lambda terms allow for function parameters to be 'consumed', for example in $(\lambda x.x\ x)\ y$ we can pass y to the function and obtain $y\ y$, which is entirely equivalent to the initial term. What we just did is called a β-*reduction* step. We reduced until it was no longer possible to do it anymore. This is not always achieved in one step. Let's look at another example, that reduces in two steps:

$$(\lambda x.x)\ (\lambda x.x\ x)\ y \twoheadrightarrow_\beta (\lambda x.x\ x)\ y \twoheadrightarrow_\beta y\ y$$

This process can be generalized at any number of steps and is called β-*reduction*. We call a term $(\lambda x.M)\ N$ (where M and N are lambda terms) a β-*redex*, and its result is $M[N/x]$ (M, where each occurrence of x is replaced by N). When a lambda term does not have any β-redexes (i.e. it cannot be reduced) we say that the term is in its *normal form*.

2.3.3 η-conversion

η-conversion is the less popular sister of β-reduction. What is special about it is that it can not only reduce lambda terms, but also expand them. By η-reduction we mean the following:

$$\lambda x.M\ x \equiv_\eta M$$

η-expansion, the opposite operation of η-reduction, is based on the following:

$$M \equiv_{\eta} \lambda x.M\ x$$

The term η-conversion can be used to denote any of the above.

Chapter 3

From Lambda Calculus to Graphs

3.1 Brief History of Lambda Terms as Graphs

It has always been convenient to represent lambda terms as graphs. There are even some popular methods of using graphical representations to perform β-reduction. Those are called reduction graphs and have gained popularity during the 1980s.

Although D. Knuth is not confirmed to be the earliest, he is the most easily trackable reference of having used graphs to represent lambda terms decades ago. He did so in his 1970s paper ”Examples of Formal Semantics”[14], but some ideas are so intuitive that they are very likely to have been used much earlier in the past and we cannot necessarily give credit to a single person.

Nowadays the idea of representing lambda terms as various types of graphs is rather common and is generally thought to originate from the ”folklore”.

3.2 Visual Representation of Lambda Terms

There is an intuitive and traditional way of representing lambda terms as graphs, more precisely, as trees. It is based on the definition of lambda terms, which states that they consist of *lambda abstractions*, *applications* and *variables*. In this section, we will be working solely with closed lambda terms, in order to provide an overview and a general intuition without getting lost in the details. A **closed lambda term** is a term that has no free variables (i.e. all its variables are bound).

The visual representation consists of nodes of three types, each corresponding to one element from the definition of lambda terms. That is, when we encounter a lambda abstraction, we will have a node of type lambda, when we encounter an application, we will have a node of type application, and the same for nodes of type variable. The child(ren) of each node depend(s) on the content of each node type. Lambda nodes will have one child, either another lambda node or an application node, based on its content. Application nodes will have two children, the left child is the term we need to apply to the term inside the right child. Variable nodes have no children because they do not possess any further content (they will be the leaves of the tree). These rules can be observed in Figure 3.1.

In Figure 3.2 we can take a look at an example displayed horizontally from left to right for space purposes.

The ”issue” with this representation is that, in some way, it has redundant information or, more precisely, redunant nodes. We know that each variable is bound to one lambda abstraction that also corresponds to a node in the graph. As a matter of fact, it comes intuitively to say that a lambda node will always appear in the graph *before* (on a lower level than) the variable nodes that correspond to its variable. By using these observations

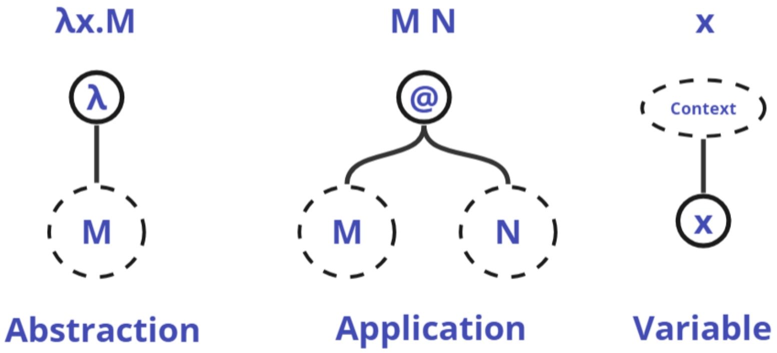


Figure 3.1: The three rules of visually representing lambda terms as graphs in the traditional and intuitive way.

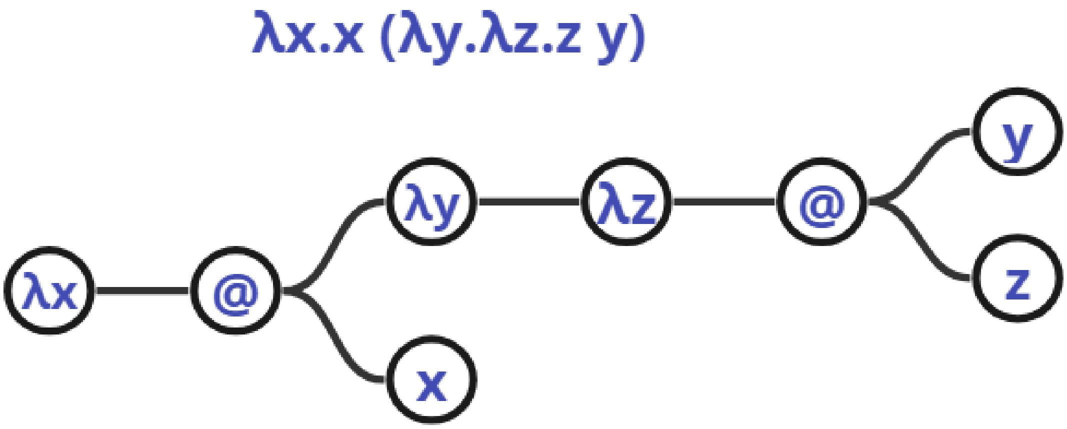


Figure 3.2: The representation of the term $\lambda x.x\ (\lambda y.\lambda z.z\ y)$ using the traditional way.

we can safely remove all variable nodes, and replace them with a *”link”* (an edge) to the lambda node that binds the variable. This process can be seen in Figure 3.3.

One detail that might appear confusing is the fact that this new representation results into a *multigraph*, which means that any two vertices can be connected by multiple edges.

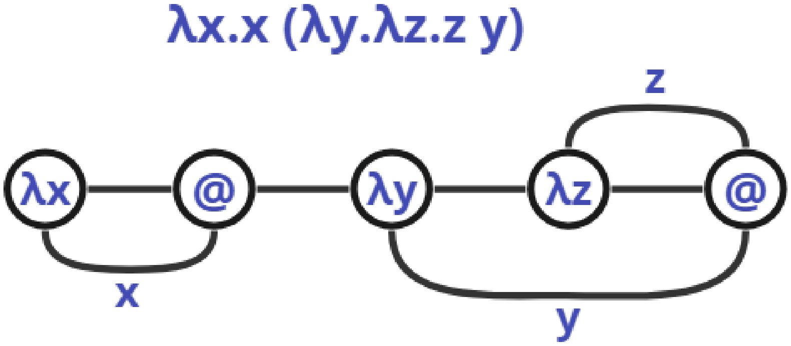


Figure 3.3: The representation of the term $\lambda x.x\ (\lambda y.\lambda z.z\ y)$ without variable nodes.

This will not be an issue. We may further notice that the obtained graphs are, in fact, maps. We will later see that this is no coincidence.

3.2.1 Exchange of Adjacent Abstractions

Any number n of abstractions are said to be adjacent if they are chained and they are not separated by anything. In the following

$$\underbrace{\lambda x_1.\lambda x_2.\ \ldots\ \lambda x_n}_{n \text{ abstractions}}.M, \text{ where } M \text{ is a lambda term}$$

abstractions $\lambda x_1, \lambda x_2, \ldots, \lambda x_n$ are adjacent.

An important thing to note in these graphical representations of lambda terms is that we are considering the terms modulo adjacency. For example, we will think of $\lambda x.\lambda y.x\ y$ as being the same as $\lambda y.\lambda x.x\ y$, therefore also the same as $\lambda x.\lambda y.y\ x$. It is also important to note that in most cases there are ways to work around this aspect and avoid this exchange of adjacent abstractions, but they will not be our focus.

The Representation of Free Variables

If we did want to represent free variables using the discussed method, they would be respresented using *free edges*. A *free edge* is an edge that, instead of connecting two nodes, one of its ends is connected to a node, and its other end is not connected to anything. Simply put, it is an edge that is ”hanging”. A very simple example can be seen in Figure 3.4.

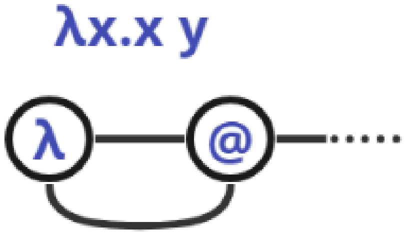


Figure 3.4: The visual representation of the term $\lambda x.x\ y$. The application node uses the bound variable x, which comes from the lambda node, and the free variable y, that appears to be coming out of nowhere.

The set of free edges in a map form what is called the *boundary* of the map. We say that a map with k free edges has a boundary of degree k. For example, the map in Figure 3.4 has a boundary of degree 1.

Note that, following the same mindset established by adjacent abstractions exchange, we are not interested in the particular name of each free variable, but rather in knowing about their existence and their location. For example, we do not differentiate between $\lambda x.y\ x$ and $\lambda x.z\ x$, the way we are viewing it is $\lambda x. < free\_variable >\ x$

3.3 Connections between Lambda Terms and Maps

Several connections between lambda terms and maps have been identified in the recent years. Intuitively, we might think that a connection relies solely on representing a lambda term as a map, but this is far from right. In reality, the connection does not only go one-way: not only does it allow us to obtain a unique map from a lambda term, but also to obtain a unique lambda term from a given map. That is, multiple bijections between lambda terms and maps have been developed.

This is not as simple as it initially sounds, as the bijections do not work with lambda terms or maps in general, but instead with certain categories of each. And they are not straightforward either.

In order to obtain general results, we need to define a notion of size for both lambda terms and maps, so that we can find mappings between, for example, ”lambda terms of size x” and ”maps of size y” for some integers x, y. The most basic notions of size we might consider for lambda terms are the number of variables (free, bound, or altogether), the number of abstractions, the number of applications, etc. Some possible ones for maps are the number of edges, the number of vertices, the number of faces, the degree of the boundary, etc.

Unfortunately, the sizes used in describing the bijections between lambda terms and maps are far more complex in most cases, with some of them involving, for example, custom-made colorings of lambda terms. However, this is not the case for all of them. In this section, we will look at various traits of lambda terms and their correspondence in maps, without diving too deep into the formalities behind them.

Most of the time, we will not be referring to individual lambda terms, but rather to alpha-equivalence classes. The alpha-equivalence class of a lambda term M consists of all the terms that are alpha-equivalent to M. For example, $\lambda x.x$, $\lambda y.y$, $\lambda w.w$ and $\lambda z.z$ are all in the same alpha-equivalence class. Same thing goes for maps, where we will be referring to the set of all maps that are isomorphic to a particular map. This kind of set is called an *isomorphism class*.

3.3.1 Linear Lambda Terms & 3-valent Maps

Let us begin by defining a linear lambda term. A lambda term is said to be *linear* if every single variable inside it appears exactly once. For example, $\lambda x.\lambda y.x\ y$ is linear, while $\lambda x.x\ x$ and $\lambda x.\lambda y.x\ y\ x$ are not. They work in the same way for free variables: the term $\lambda x.y\ x\ z$ is linear, while $\lambda z.y\ z\ y$ is not.

In Figure 3.5 we can look at what some linear terms look like as maps. The term on the left is fairly basic, and one thing that can be observed by looking at it is the fact that, besides the root, all nodes are touching exactly *three* edges. This is a key property of linear lambda terms in general that we will discuss further. However, the term on the right contains a bizarre detail: the identity $\lambda z.z$, that has one loop and is only connected by two edges. The identity term is somewhat of an exception. The case of the root is not really an exception and is rather an advantage that allows us to easily identify where the term starts, even if we remove the nodes' annotations.

Property: If a lambda term is linear, its underlying map will consist only of vertices that touch exactly three edges, except for the root (that will touch exactly two edges) and the identity subterm $\lambda x.x$ (that will touch exactly one basic edge and one loop edge).

The explanation lies in the following:

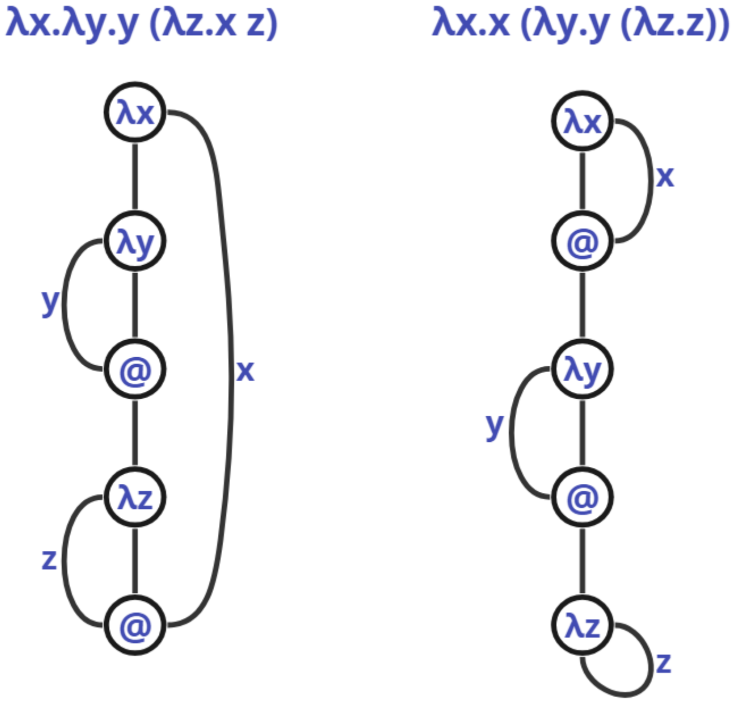


Figure 3.5: Two linear lambda terms and their corresponding maps.

- **Application nodes** - they apply one thing to another, so by default they will be connected to at least two vertices (therefore at least two edges); furthermore, they are also connected to their *context*, for example (but not limited to) the lambda abstraction that they belong to.

- **Abstraction nodes** - they are connected to their *body* (their content), so this gives us one edge; they are also connected to each occurrence of their bound variable, and in linear terms each variable appears exactly once, so this gives us one more edge; the third and last edge belongs to the context that they appear in.

A map in which each vertex touches exactly three edges is called a *3-valent* (trivalent)

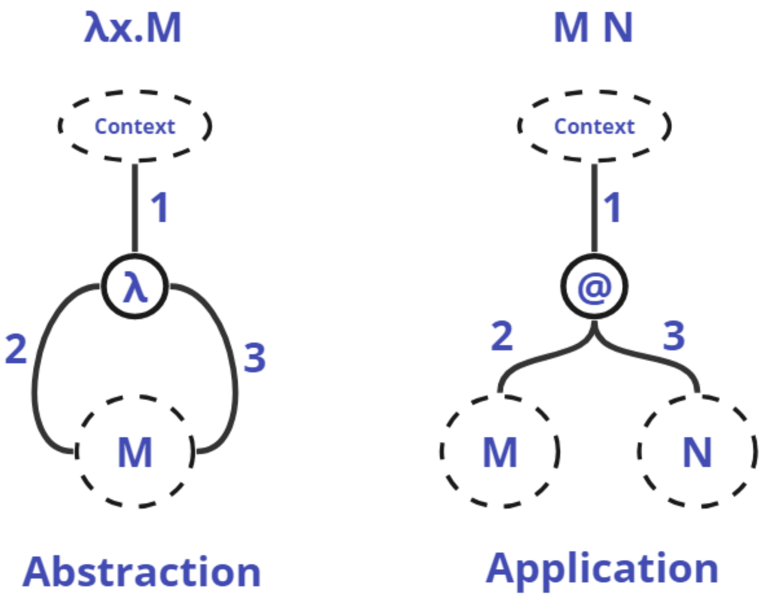


Figure 3.6: The representation of abstraction and application in linear terms. In both cases, edge 1 connects the node to its context. **Left:** M is a term where x appears exactly once (connected by edge 2). M is also the body of the term (M is connected to its context by edge 3). **Right:** The node applies the term M (connected by edge 2) to the term N (connected by edge 3).

map. Let us now think about how many vertices are in the representation of a lambda term. We know that each lambda abstraction has one corresponding vertex. If the lambda term has p abstractions, then the underlying map will have p lambda nodes. Next, we have the application nodes. For this case, it may be useful to look at some examples along with their number of applications:

1. $\lambda x.x\ (\lambda y.y) \longrightarrow 1$
2. $\lambda x.\lambda y.x\ y \longrightarrow 1$
3. $\lambda x.\lambda y.\lambda z.x\ y\ z \longrightarrow 2$
4. $\lambda x.x\ \lambda y.\lambda z.\lambda w.y\ w\ z \longrightarrow 3$
5. $\lambda x.y\ x\ z \longrightarrow 2$
6. $(\lambda x.x)\ z\ (\lambda y.y)\ z \longrightarrow 3$

The first four examples are closed linear terms, and we can notice that the pattern seems to be related to the number of variables. Precisely, the number of applications is the number of variables from which we subtract one. The fifth example is a linear term (that is not closed), and the rule still holds. The last example (number 6) is just a term that is neither linear nor closed. Here, the number of variables is 3, and the number of applications is also 3. While this seems to break the rule, the key detail is that the number of variables occurrences is 4 (x and y appear once each, and z appears twice), and $4-1 = 3$ verifies the rule.

The intuition is also that every variable must, in some way, be part of an application: either directly (in Example number 1, x is directly applied to something), or indirectly (in Example number 6, the term that contains x is applied to z). All signs lead to the result that the number of applications in a lambda term (so also the number of application nodes in its corresponding map), let it be called q, is equal to the number of occurrences of variables from which we subtract 1. Since in linear terms each variable appears exactly once, in their case $q = \#\text{variables} - 1$.

For obvious reasons, the total number of nodes is $p+q$ (the total number of abstractions and applications). If the term has k free variables, then its map will have k free edges, which gives us a boundary of degree k.

Property: A linear lambda term with p abstractions, q applications and k free variables can be mapped to a unique 3-valent map with $p+q$ vertices and a boundary of degree k.[23]

In Figure 3.7 there is an example for visualizing the above property. It shows both the annotated version of the map and the simple, ”empty” version. Although we might think that by doing this we are losing information, the surprising result about the correspondence

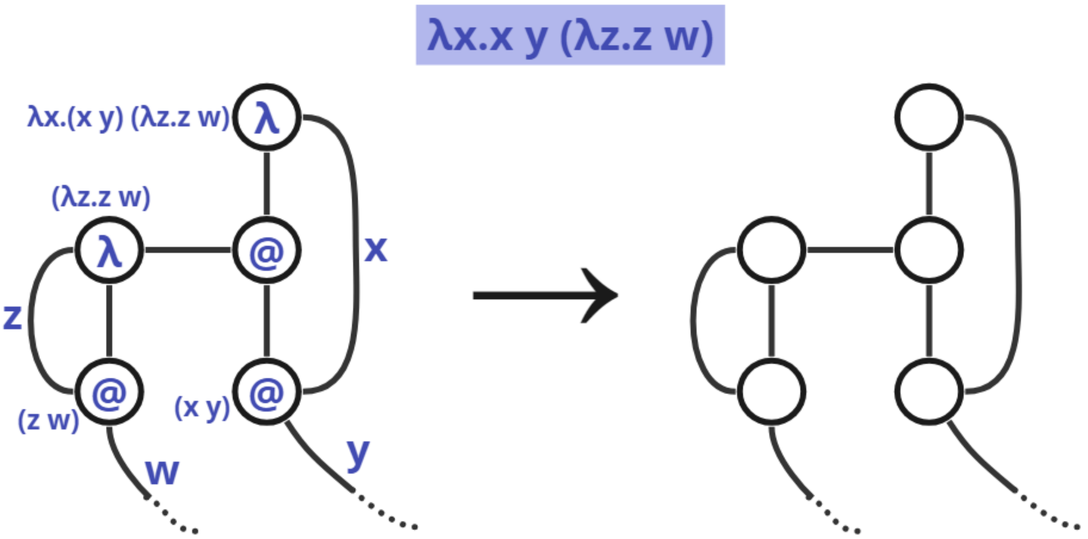


Figure 3.7: The representation of a linear lambda term with $p = 2$, $q = 3$, and $k = 2$, where some vertices and edges have been annotated for easier understanding. The obtained map has $p + q = 5$ vertices and $k = 2$ free edges.

between linear lambda terms and 3-valent maps is that it is reversible, meaning that from a blank map we can obtain a unique linear lambda term.

When it comes to reversing the transformation, more precisely, obtaining a lambda term from a given map, we need to identify two types of nodes. We call them *connecting* nodes and *disconnecting* nodes. We say that a node is connecting if, by removing it, the map remains connected. That is, a node is said to be connecting if it is part of a cycle. Connecting nodes correspond to lambda abstractions.

A disconnecting node is the opposite of a connecting node. A node is disconnecting if it is not connecting, meaning that, by removing it from the map, the map is not connected anymore. Disconnecting nodes correspond to applications.

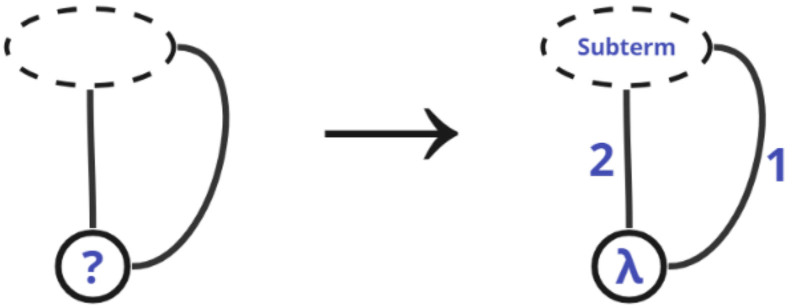


Figure 3.8: A connecting node. Edge 1 connects the node to the occurrence of its bound variable, that appears somewhere in the subterm. The node in question is the context of the subterm, so edge 2 connects the subterm to its context. The node in question is inside a cycle, meaning that, by starting there, we can get back to it without using the same edge twice.

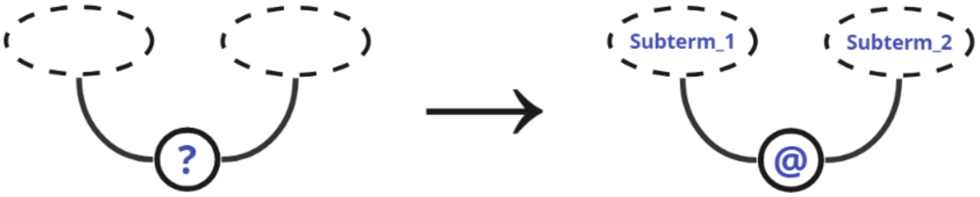


Figure 3.9: A disconnecting node. It connects two subterms, with the meaning that **Subterm_1** is applied to **Subterm_2**. If we were to remove the node in question, the map would become disconnected (we would not be able to start in **Subterm_1** and reach **Subterm_2** or vice-versa.

The trivalent map to lambda term transformation process always begins with the node that is not trivalent (it is only connected to two edges, as previously mentioned). After we identify the type of a node, we repeat the process with the rest of the map. The key detail is that the identification needs to be done on the *remaining map*, instead of the whole map. By *remaining map* we mean the map that is left after the removal of the nodes identified so far.

The process is exemplified in Figure 3.10. We start from the map in the left upper corner. We easily identify the root because it is the only non-trivalent node, and this is where we start. The first node is part of a cycle, so it is an abstraction node. When an abstraction node is found, the edges that form its corresponding cycle are marked in blue. The remaining map in each step is surrounded by dotted line. As previously mentioned, it is important to only look at the remaining map and to remember to take out a node after its identification. The next node to look at is among the neighbours of the lastly identified node, and is, again, the one that is not trivalent.

Of course, this process is only guaranteed to work on trivalent maps, as discussed. The mapping between trivalent maps and linear lambda terms is 1:1, meaning that any linear lambda term M has a unique corresponding map G, and M is the only linear lambda term that corresponds to G.

3.3.2 Unitless Lambda Terms & Bridgeless Maps

We have already discussed closed lambda terms. Similar (or rather opposite) to the closeness of terms, there is also a concept describing something called a *unitless* lambda term. A lambda term is said to be *unitless* iff it does not contain any closed subterms, besides the term itself. We have seen that lambda terms have a recursive definition, and this is perfect for checking properties on subterms.

In Figure 3.12 we can see all the subterms of three different lambda terms. The term on the left is unitless because none of its subterms is closed. The term in the middle is a closed term that, besides itself, does not have any closed subterms. In order to be unitless,

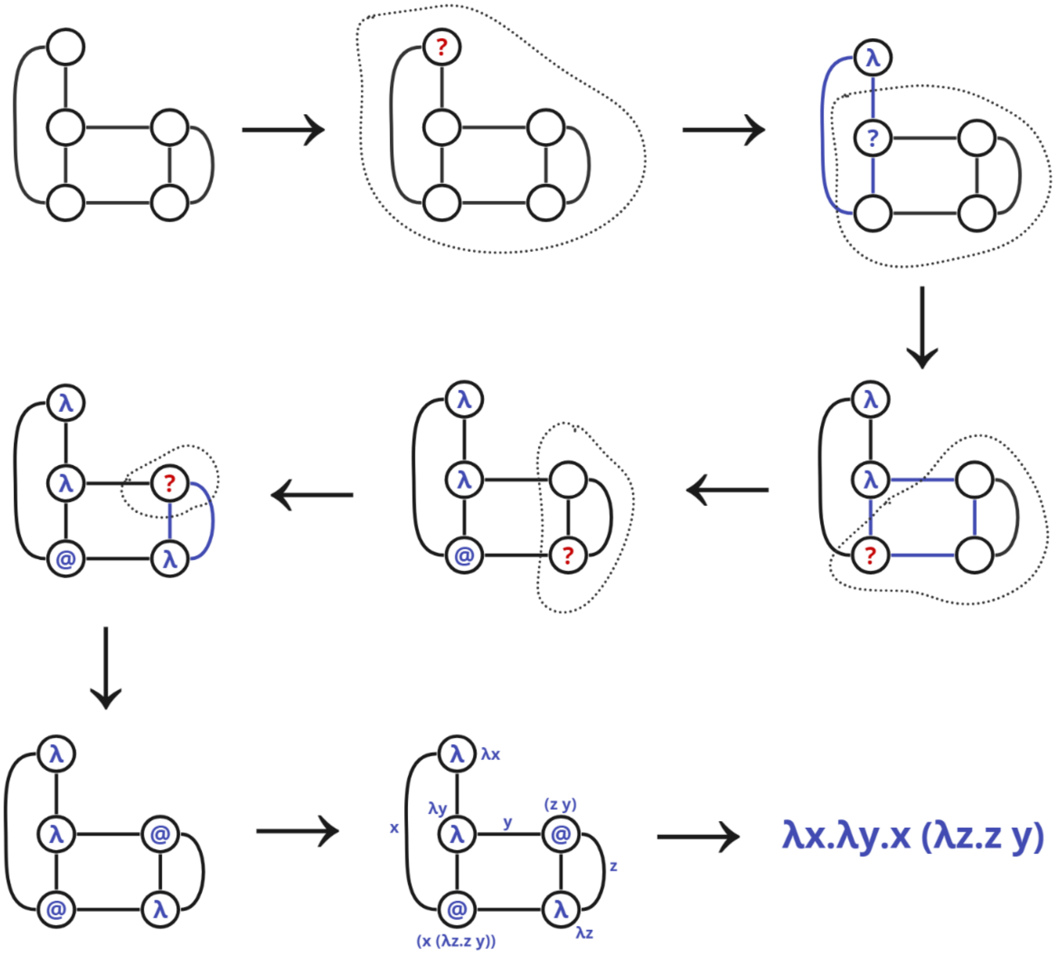


Figure 3.10: Step-by-step tranformation into the term $\lambda x.\lambda y.x\ (\lambda z.z\ y)$, starting from its corresponding map. For simplicity, a closed term has been chosen (i.e.: a map without free edges).

the term itself is allowed to be closed, but not to have any other closed subterms, so the term in the middle is also unitless (as well as closed). The term on the right is not closed because of the free variable y, but it does contain a closed subterm $\lambda z.z$, also called a *unit*, making it non-unitless. The *unit* is marked with red.

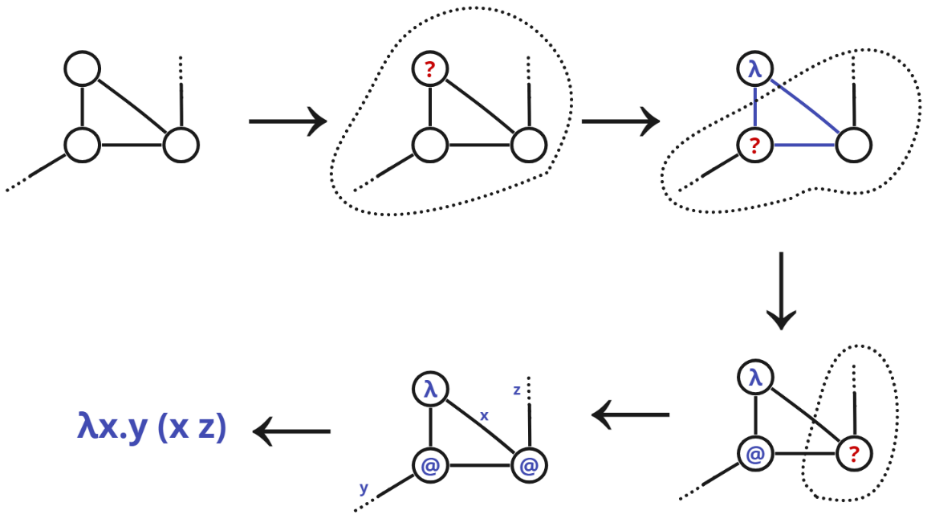


Figure 3.11: The trivalent map to linear lambda term transformation performed on a map with free edges (i.e.: a term with free variables).

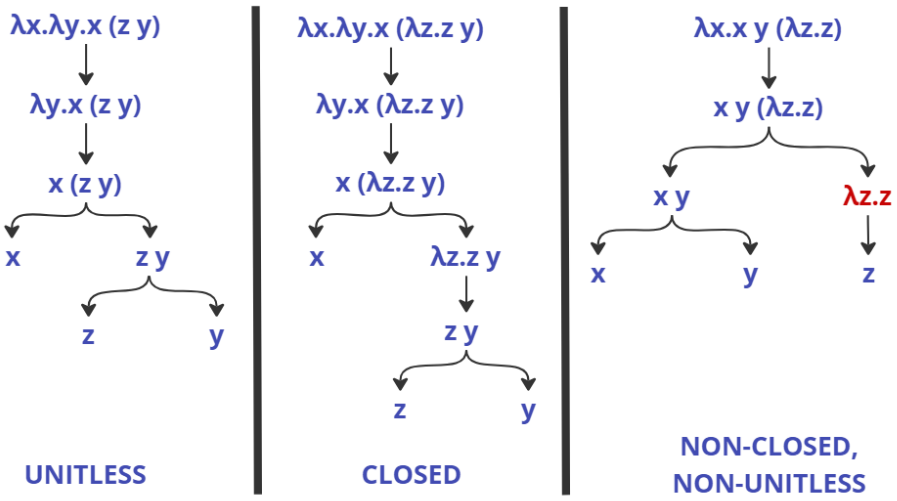


Figure 3.12: Three lambda terms and their subterms. Unitless **VS** Closed **VS** Neither closed, nor unitless

Now, switching to maps, we want to discuss what is known as a *bridgeless* map. In a map (or a graph in general), a bridge is an edge whose potential removal would cause the map to become disconnected. A map is said to be *bridgeless* iff it does not contain any bridge. The concept is exemplified in Figure 3.13.

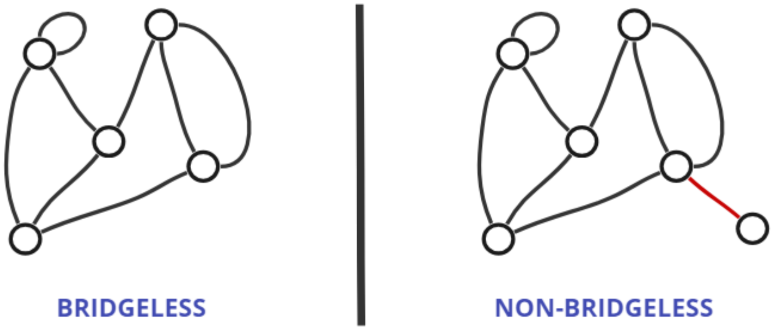


Figure 3.13: Example of a bridgeless map and a non-bridgeless map. In the non-bridgeless map, the bridge is marked with red.

It turns out that the lambda terms' property of being unitless translates into maps' property of being bridgeless![8] The both-ways transformation process works similarly to the one from the previous section, so we will not be focusing on it. Instead, let us try to understand why this property translation makes sense even at an intuitive level.

By looking at Figure 3.14 we can get an idea about what closed terms look like. They look ”tight” and very connected, like a closed circuit. When we apply a closed term to another, bridges are inevitably formed. The application node's purpose is only to connect the two formerly independent terms, and it cannot do so without creating bridges. The same goes for when we ”wrap” a closed term inside an abstraction whose bound variable does not appear in the said closed term. This will also inevitably create a bridge, because there will only be one edge (the bridge) connecting the closed term to its context (the abstraction that acts as a wrapper).

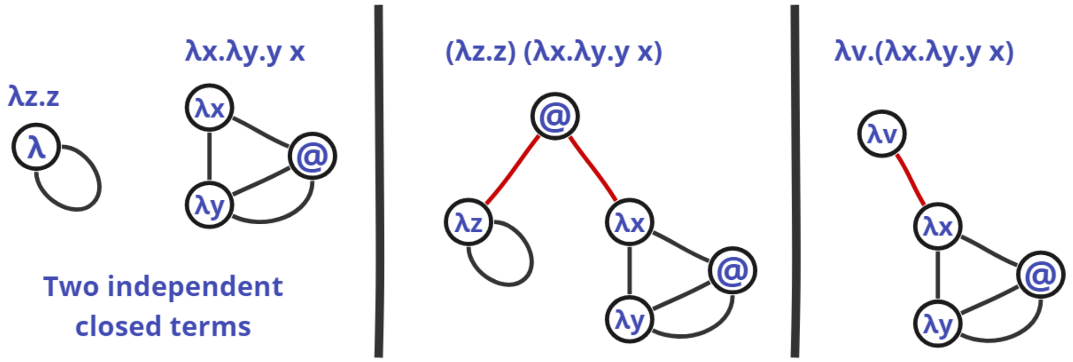


Figure 3.14: Examples how non-unitless terms create bridges. Bridges are marked with red. **Left:** Two closed lambda terms. **Middle:** The previous two terms, applied to each other. **Right:** One of the closed terms, inside an abstraction.

Chapter 4

Additional Relevant Topics

4.1 Enumerative Aspects

The idea of counting lambda terms is not necessarily something that one naturally thinks about. However, this has proved to be the first trigger to questioning potential connections between lambda calculus and graph theory. Invited to OASIS (The Oxford Advanced Seminar on Informatic Structures)[22], Noam Zeilberger, the ”father” of lambda calculus and graph theory connections, explained how he happened to count some lambda terms while researching an unrelated subject.

The lambda terms he counted were β-normal ordered linear terms. A linear lambda term is said to be ordered iff all variables are used in the order they are bound. While it may seem a little confusing at first, examples will help clarify:

- $\lambda x.\lambda y.x\ \ y$ and $\lambda x.x\ \ (\lambda y.\lambda z.y\ \ z)$ are ordered (λx appears first and x is the first variable used, etc.)

- $\lambda x.\lambda y.y\ x$ is not ordered (the order of the binding is x, y and the order of the use is y, x)

One consequence of a term being ordered is that the term is also closed, since the definition of the order property does not account for free variables.

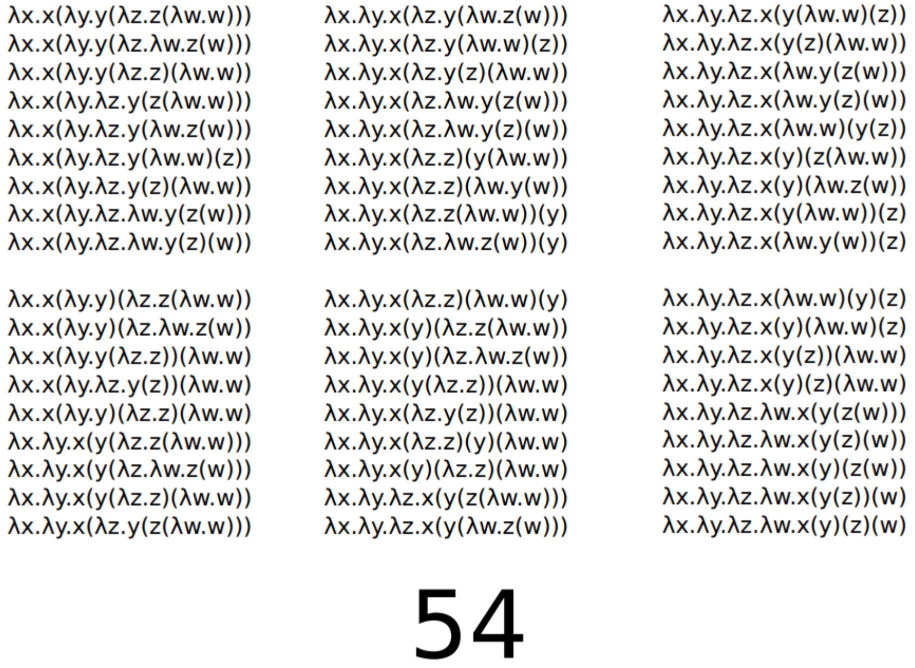
Let us try to recreate the original experiment. The criteria by which we are counting β-normal ordered linear terms is the number of variables. We will try to count all β-normal ordered linear terms with one variable, then with two variables, and so on.

The only β-normal linear lambda term with one variable is $\lambda x.x$. With two variables, we have $\lambda x.x\ (\lambda y.y)$ and $\lambda x.\lambda y.x\ y$. With three variables we have the following:

1. $\lambda x.x\ (\lambda y.y\ (\lambda z.z))$
2. $\lambda x.x\ (\lambda y.\lambda z.y\ z)$
3. $\lambda x.x\ (\lambda y.y)\ (\lambda z.z)$
4. $\lambda x.\lambda y.x\ (y\ (\lambda z.z))$
5. $\lambda x.\lambda y.x\ (\lambda z.y\ z)$
6. $\lambda x.\lambda y.x\ (\lambda z.z)\ y$
7. $\lambda x.\lambda y.x\ y\ (\lambda z.z)$
8. $\lambda x.\lambda y.\lambda z.x\ (y\ z)$
9. $\lambda x.\lambda y.\lambda z.x\ y\ z$

For the terms with four variable, we may consult Figure 4.1. Just like the original experiment, we obtained a *sequence*: 1, 2, 9, 54, $\ldots$. So far, we know that the sequence corresponds to the number of *β-normal ordered linear lambda terms with n variables*. The author searched this sequence in OEIS (Online Encyclopedia of Integer Sequences) and was more than surprised to see it already existed. The sequence ID is A000168 and it also enumerates a category of maps called *rooted planar maps with n edges*. This connection is sometimes called a *combinatorial interaction* between lambda terms and maps.

As for the enumeration of maps, this has been a subject of interest ever since decades ago, and a significant part of it has been settled over the years. W.T. Tutte has published



```
λx.x(λy.y(λz.z(λw.w)))
λx.x(λy.y(λz.λw.z(w)))
λx.x(λy.y(λz.z)(λw.w))
λx.x(λy.λz.y(z(λw.w)))
λx.x(λy.λz.y(λw.z(w)))
λx.x(λy.λz.y(λw.w)(z))
λx.x(λy.λz.y(z)(λw.w))
λx.x(λy.λz.λw.y(z(w)))
λx.x(λy.λz.λw.y(z)(w))

λx.x(λy.y)(λz.z(λw.w))
λx.x(λy.y)(λz.λw.z(w))
λx.x(λy.y(λz.z))(λw.w)
λx.x(λy.λz.y(z))(λw.w)
λx.x(λy.y)(λz.z)(λw.w)
λx.λy.x(y(λz.z(λw.w)))
λx.λy.x(y(λz.λw.z(w)))
λx.λy.x(y(λz.z)(λw.w))
λx.λy.x(λz.y(z(λw.w)))
```

```
λx.λy.x(λz.y(λw.z(w)))
λx.λy.x(λz.y(λw.w)(z))
λx.λy.x(λz.y(z)(λw.w))
λx.λy.x(λz.λw.y(z(w)))
λx.λy.x(λz.λw.y(z)(w))
λx.λy.x(λz.z)(y(λw.w))
λx.λy.x(λz.z)(λw.y(w))
λx.λy.x(λz.z(λw.w))(y)
λx.λy.x(λz.λw.z(w))(y)

λx.λy.x(λz.z)(λw.w)(y)
λx.λy.x(y)(λz.z(λw.w))
λx.λy.x(y)(λz.λw.z(w))
λx.λy.x(y(λz.z))(λw.w)
λx.λy.x(λz.y(z))(λw.w)
λx.λy.x(λz.z)(y)(λw.w)
λx.λy.x(y)(λz.z)(λw.w)
λx.λy.λz.x(y(z(λw.w)))
λx.λy.λz.x(y(λw.z(w)))
```

```
λx.λy.λz.x(y(λw.w)(z))
λx.λy.λz.x(y(z)(λw.w))
λx.λy.λz.x(λw.y(z(w)))
λx.λy.λz.x(λw.y(z)(w))
λx.λy.λz.x(λw.w)(y(z))
λx.λy.λz.x(y)(z(λw.w))
λx.λy.λz.x(y)(λw.z(w))
λx.λy.λz.x(y(λw.w))(z)
λx.λy.λz.x(λw.y(w))(z)

λx.λy.λz.x(λw.w)(y)(z)
λx.λy.λz.x(y)(λw.w)(z)
λx.λy.λz.x(y(z))(λw.w)
λx.λy.λz.x(y)(z)(λw.w)
λx.λy.λz.λw.x(y(z(w)))
λx.λy.λz.λw.x(y(z)(w))
λx.λy.λz.λw.x(y)(z(w))
λx.λy.λz.λw.x(y(z))(w)
λx.λy.λz.λw.x(y)(z)(w)
```

54

Figure 4.1: All 54 β-normal ordered linear lambda terms with four variables. Source:[22]

a remarkable series of papers in the field of map enumeration that were initially intended to support him in proving the four color theorem, but they were also very useful in understanding the connections between lambda calculus and graph theory (the methods used are known as *Tutte decomposition*). The result itself was not only useful, but also the underlying ideas behind obtaining sequences of maps proved valuable in discovering bijections between lambda terms and maps. Many categories of maps can be enumerated with rather elegant formulas thanks to these papers. For example, the family of rooted maps with n edges is given by the following formula:

$$\frac{2 \cdot (2 \cdot n)! \cdot 3^n}{(n!) \cdot (n+2)!}$$

A *rooted* map is simply a map in which an edge, called *the root* of the map, is marked in a way that allows us to differentiate it from the others. Rooted maps are usually used in enumerative problems because they are somewhat easier to count. This is due to the absence of non-trivial automorphisms.

In category theory, there is the concept of a bijection being *natural*. What it means for a bijection to be natural at a very high and intuitive level is that it is a mapping between two entities (potentially sets, categories, etc.), that preserves their structure in a consistent and universal way. So far, only a few natural bijections between the combinatorial classes of lambda terms and those of maps have been found (with great effort I may add).

An important result is, for example, that closed rooted trivalent maps are denoted by the same combinatorial class as linear lambda terms[4]. In addition to that, for some families of lambda terms and maps, bijections that are not natural have been found, which are not particularly satisfying if we want to transfer results from the field of lambda calculus to the field of graph theory or vice-versa. The discovery of a natural bijection between many of these families is still an open problem.

The enumerative aspect of these connections has been the key in identifying them, because it has hinted which family of lambda terms is related to which family of maps. This is what prompted people to search for bijections in the first place.

4.2 String Diagrams in relation to Lambda Calculus

This topic is too complex to be discussed at its full depth, but it is a good proof of the generality of the connections between lambda calculus and graph theory. So far it might have seemed that the results are very specific, with little to no generality whatsoever, but this is not true.

In early 1980s, Dana Scott has documented multiple advanced mathematical properties of lambda calculus, among which he formulated the interpretation of lambda terms as endomorphisms of a reflexive object (at a very high level, a reflexive object is an object that has a meaningful and structured mapping to itself) in a category that posseses some specific properties[19]. This is what allowed for lambda terms to be represented via string diagrams[22] in the 2010s.

String diagrams are a graphical language in applied category theory whose purpose is to facilitate the understanding of the more complex categorical structures. In relation to lambda calculus, they are making use of its well-known rules, β-reduction and η-expansion. Just as a curiosity, we may see these rules represented via string diagrams in Figure 4.2.

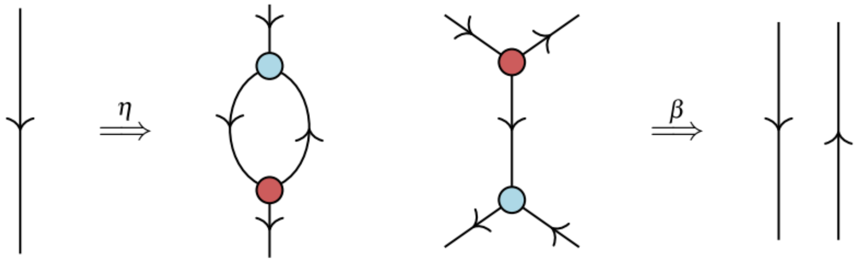


Figure 4.2: η-expansion and β-reduction as string diagrams. Blue nodes denote applications, while red nodes denote abstractions. Source:[22]

4.3 The Four Color Theorem & Typing of Lambda Terms

This section discusses potentially the most important result transferable between graph theory and lambda calculus, as we will see in a moment.

4.3.1 The Four Color Theorem

The four color theorem has been a notorious theorem in mathematics and particularly in graph theory for a long time, mainly because great efforts over many decades have been made in order to prove it. In the end, the theorem was ”defeated” by a computer-assisted proof that only partially satisfied the scientific community.

Its statement refers to proper colorings of graphs. We may differentiate three types of proper colorings:

- vertex-coloring: coloring all vertices such that no two adjacent vertices have the same color

- face-coloring: coloring all faces such that no two adjacent faces have the same color (reminder that two faces are adjacent if they share at least an edge)

- edge-coloring: coloring all edges such that no two adjacent edges have the same color

A c-coloring (where $c \in \mathbb{N}$) refers to a proper coloring using at most c distinct colors. A graph is said to be n-edge-colorable if its edges admit a proper n-coloring (similarly for vertices and faces). The four color theorem traditionally states:

Every planar graph is 4-vertex-colorable.

In 1880, mathematician Peter Tait proved that the above statement can be reduced to the following:

Every planar bridgeless trivalent graph is 3-edge-colorable.[10]

So, put together, we obtain: *every planar graph is 4-colorable iff every planar bridgeless trivalent graph is 3-edge-colorable.*[10]

Starting with a planar graph, the theorem states that it is 4-vertex-colorable. Let us introduce the notion of *maximal planar* graph. A graph is said to be maximal planar if it is planar and the potential addition of a new edge between two vertices would break the planarity of the graph.

Let us briefly introduce the process of going from one statement of the theorem to the other. Adding more edges to a planar graph without breaking its planarity would only make the problem of 4-vertex-coloring harder. This means that we could add edges until the graph became maximal planar. If the maximal planar version of the original graph is 4-vertex-colorable, then it is guaranteed that the original graph is also 4-vertex-colorable. An example of computing the maximal planar version of a graph can be seen in Figure 4.3.

It has also been shown (and it is also somewhat intuitive) that the vertex-coloring of a graph is equivalent to the face-coloring of its dual. This does make sense because each vertex in the original graph corresponds to a face in the dual graph. Computing the dual graph for our previous example is done in Figure 4.4. The coloring of the original graph as well as the coloring of its dual can be seen in Figure 4.5

Now, if we take a maximal planar graph G and we compute its dual G', one can

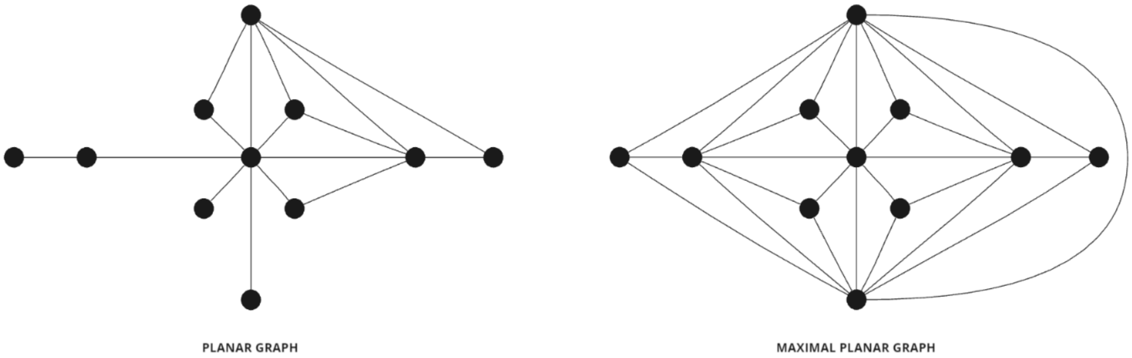


Figure 4.3: A planar graph and its corresponding maximal planar graph. The graph on the right is known as the Goldner–Harary graph.

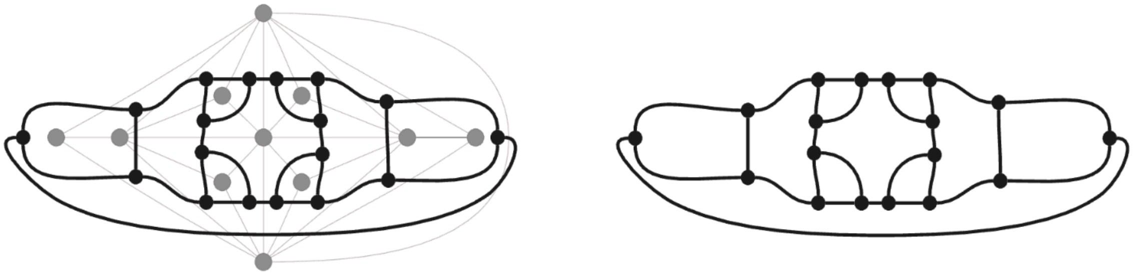


Figure 4.4: The Goldner-Harary graph (left, colored in grey) and its dual graph (right).

prove by double implication that the 4-face-coloring of G' is also equivalent to the 3-edge-coloring of G', although this is not very intuitive at first[18]. Moreover, the dual in question is also a bridgeless trivalent graph! The trivalent property comes from the fact that in a maximal planar graph, each face is surrounded by exactly three edges. The 3-edge-coloring of our example can be seen in Figure 4.6.

The key takeaway that will serve us in this section is that the four color theorem can refer to 3-edge-coloring of bridgeless trivalent maps.

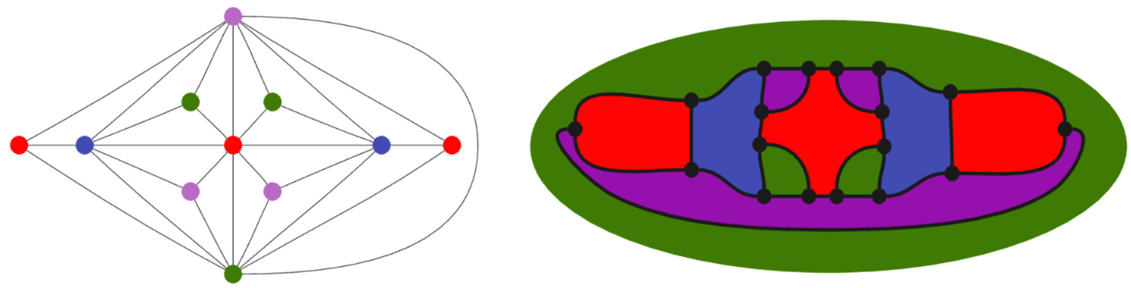


Figure 4.5: The 4-vertex-coloring of the Goldner-Harary graph (left) and the 4-face-coloring of its dual (right).

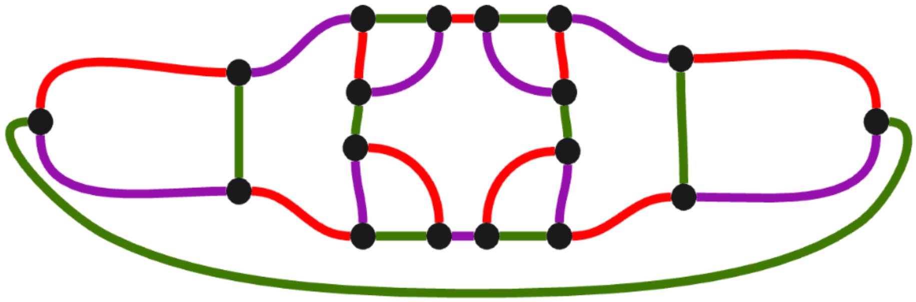


Figure 4.6: The 3-edge-coloring of the dual of the Goldner-Harary graph.

4.3.2 Simply Typed Lambda Calculus

So far we have only handled untyped lambda calculus. Now we are going to dig a little into simply typed lambda calculus and the typing of lambda terms.

Simply typed lambda calculus is a type system developed by Alonzo Church, the ”father” of lambda calculus. While we are not going to dive into its formal definitions and properties, it will be useful to understand how it can be used. At a high level, a type is something that a group of objects have in common. For example, $3, 25, -9$ are of type integer. Since we are discussing lambda calculus, we care about the types of abstractions and applications. If a term t has type $\sigma \to \tau$, then t is a term that takes an argument of type σ and returns a result of type τ.

If a variable x has a type $\sigma \to \tau$ and a variable y has a type σ, then their application $(x\ y)$ has type τ. Similarly, if y has a type σ, then, for the application $x\ y$ to be possible, x needs to have a type of the form $\sigma \to \tau$.

As for abstraction, if $\lambda x.M$ (where M is a lambda term) has a type $\sigma \to \tau$, then M has the type τ. Types work very similarly to how the definitions of functions (their domain and codomain) work in mathematics.

Typed lambda calculus is important for ensuring that functions are applied to compatible arguments. The typing of a term, sometimes called type inference and used in programming languages, is the action of finding a proper (compatible) type for a term.

The basic rules of typing can be seen below, where $x : \sigma$ means that x has type σ, and Γ is a context:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (\text{Var})$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.t) : \sigma \to \tau}\ (\text{Abs})$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (t_1\ t_2) : \tau}\ (\text{App})$$

Using these rules we are able to find and check types of lambda terms. Let us, for example, perform this on the term $\lambda f.\lambda x.f\ x$:

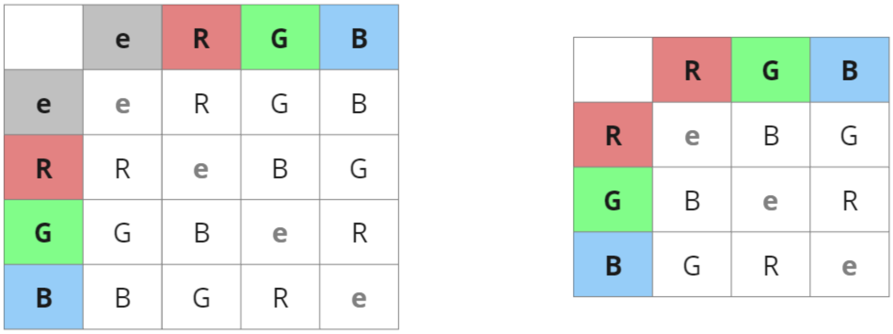
$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \sigma \to \tau, x : \sigma \vdash f : \sigma \to \tau}\ (\text{Var}) \qquad \dfrac{}{f : \sigma \to \tau, x : \sigma \vdash x : \sigma}\ (\text{Var})}{f : \sigma \to \tau, x : \sigma \vdash (f\ x) : \tau}\ (\text{App})}{f : \sigma \to \tau \vdash (\lambda x : \sigma.f\ x) : \sigma \to \tau}\ (\text{Abs})}{\vdash (\lambda f : \sigma \to \tau.\lambda x : \sigma.f\ x) : (\sigma \to \tau) \to (\sigma \to \tau)}\ (\text{Abs})$$

These typing rules also happen to follow the pattern of natural deduction from logic, building a notorious connection between proofs and computer programs known as the Curry-Howard correspondence.

4.3.3 Typing via Coloring

In this subsection we will put everything together in order to obtain a reformulation of the four color theorem, in relation to lambda calculus.

We need to make use of a mathematical structure known as the Klein four-group. This is a commutative group consisting of four elements, one of which is the identity element noted e. Let the other three elements be called R, G and B so that they resemble the primary colors red, green and blue (it is purely optional). We can see this group's multiplication table in Figure 4.7. For our purpose, we will use the one on the right.



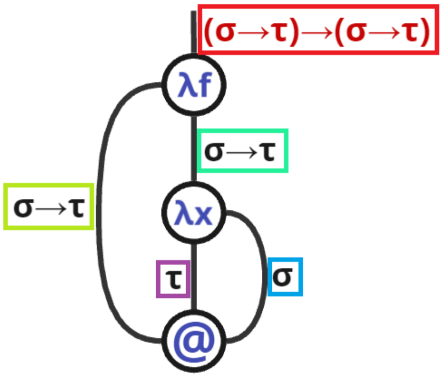
	e	**R**	**G**	**B**
e	e	R	G	B
R	R	e	B	G
G	G	B	e	R
B	B	G	R	e

	R	**G**	**B**
R	e	B	G
G	B	e	R
B	G	R	e

Figure 4.7: The Klein four-group multiplication table (left) and the same table but without the identity element (right).

In the previous section we have derived the type of the term $\lambda f.\lambda x.f\ x$ and we obtained the type $(\sigma \to \tau) \to (\sigma \to \tau)$. To do this, we used the basic typing rules 5 times. This is exactly the number of edges of the term's graphical representation if we add an edge that acts as the root. This prompts us to think that edges might have types themselves.

In Figure 4.8 we can see what the typing of the edges (note that we are considering a root edge) will look like. For the moment, colors do not have anything to do with edge-coloring, they have only been used to point where each type occurs. The type on the root edge is the type of the entire term.



$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\sigma\to\tau, x:\sigma \vdash f:\boxed{\sigma\to\tau}}\,(\text{Var}) \qquad \dfrac{}{f:\sigma\to\tau, x:\sigma \vdash x:\boxed{\sigma}}\,(\text{Var})}{f:\sigma\to\tau, x:\sigma \vdash (f\ x):\boxed{\tau}}\,(\text{App})}{f:\sigma\to\tau \vdash (\lambda x:\sigma.f\ x):\boxed{\sigma\to\tau}}\,(\text{Abs})}{\vdash (\lambda f:\sigma\to\tau.\lambda x:\sigma.f\ x):\boxed{(\sigma\to\tau)\to(\sigma\to\tau)}}\,(\text{Abs})$$

Figure 4.8: The edge-typing of the term $\lambda f.\lambda x.f\ x$ in relation to the term's type derivation.

Now, let us go back to the Klein four-group. The atomic types we used in the example term are σ and τ. Let $\sigma = R$ and $\tau = G$. To "compute" $\sigma \to \tau$, we will instead compute $R \cdot G$. We can see this transition in Figure 4.9. The root of this typing will always have type e.

As you may have noticed, it is not a proper 3-edge-coloring in the traditional sense. This is because of an important property that has been discovered: the 3-edge-coloring using the Klein four-group elements is a proper coloring iff no edge is assigned the type e, except for the root. Instead of coloring, this process is rather called *3-typing*.

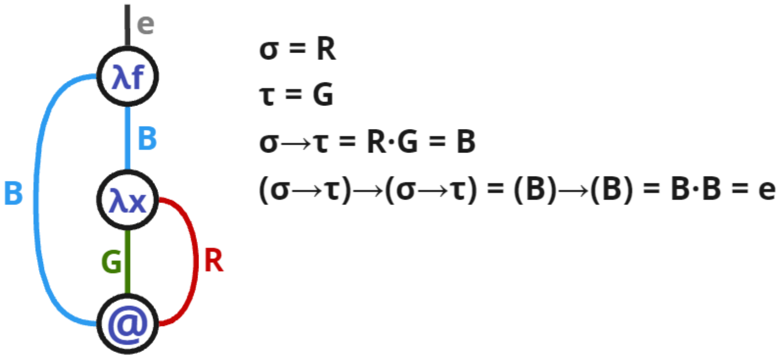


Figure 4.9: The term $\lambda f.\lambda x.f\ x$ edge-colored using the elements of the Klein four-group.

In the end, the four color theorem can be reformulated as[23]:

Every planar indecomposable linear lambda term has a proper 3-typing.

where planar indecomposable lambda tearms are a certain (rather broad) family of lambda terms.

Chapter 5

Conclusions

We find ourselves at the end of this journey of untangling unexpected connections between lambda calculus and graph theory. The area of the topics addressed in this thesis is far from having been thoroughly studied yet, with its underlying complexity and implications only recently beginning to gradually unfold before our eyes. The primary interest is held by a group of French researchers, who have been actively searching for collaborators, but making progress at a rather slow pace. The first time these connections initially occurred to them was around 2013, but it was not before 2018 that they started giving few international presentations about the topic. In 2022, they put together and received funding for a 4-year research project (based solely on this topic) called LambdaComb, whose plan and contact details can be found online. What they mainly hope for is a complete bilingual dictionary between lambda calculus and graph theory (some connections remain open problems to this day), as well as bringing an enumerative perspective into lambda calculus, among other (more complex and highly mathematic) aims. For the moment, the transferability between lambda calculus results and graph theory results has not necessarily been the subject of much attention, because of the complex and relatively recent nature of the connections between them, which have yet to be fully explored in the upcoming future and integrated into existing knowledge.

In a 2022 article, the original authors leveraged maps and their connection to lambda calculus for sampling β-normal closed linear lambda terms. The authors believe that this sampling is especially relevant in testing theorem provers, by using a correspondence between these lambda terms and proofs of tautologies in implicational linear logic[5].

As for the implications, I believe they are both theoretical and practical. Personally, I am a strong believer of the idea that discovering useless mathematics is a challenging task. Although some of them may have been ahead of their time, mathematical correspondences have a long history of revolutionizing science, one of the most famous being Boole's combination of logic and algebra.

Although originating from very theoretical backgrounds, both lambda calculus and graph theory already have strong roots in practical fields, with lambda calculus serving as the base for functional programming and even becoming more present in general-purpose programming languages, and graph theory being found (to some extent) in most practical areas, with route planning being one used by the entire world in the daily life. The result I believe to have the most potential is the lambda calculus' relation to the four color theorem and it is also the result I found most surprising. It is well-known that deciding whether a graph is 4-colorable is an NP-complete problem, therefore it can use all the help it can get. As far as the typing of simply typed lambda terms goes, there are many articles exploring and proving that the typing of some broad categories of lambda terms is PTIMEcomplete[16]. Furthermore, a shifting of perspective of such great amplitude may at some point even contribute to a more pleasing, formal proof of the four color theorem, instead of the existing proof by exhaustion done by a computer.

Even if all potential paths reach a dead end, the correspondence in itself is profoundly fascinating. In the end, a proverb from the folklore goes:

“It’s not that the bear dances so well, it’s that he dances at all.”

Bibliography

[1] Jesse Alama and Johannes Korbmacher. “The Lambda Calculus”. In: *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2023.

[2] Denis Auroux. *Introduction to Topology*. 2019. URL: `https://people.math.harvard.edu/~auroux/131f19/Math_131_Notes_Beckham_Myers.pdf` (visited on 02/03/2024).

[3] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. New York: Springer-Verlag, 1985.

[4] Olivier Bodini, Alexandros Singh, and Noam Zeilberger. “Asymptotic Distribution of Parameters in Trivalent Maps and Linear Lambda Terms”. In: *arXiv:2106.08291* (2021).

[5] Olivier Bodini, Alexandros Singh, and Noam Zeilberger. “Sampling β-normal linear λ-terms”. In: *Pure Mathematics and Applications* 30 (2022).

[6] Felice Cardone and J. Hindley. *History of lambda-calculus and combinatory logic*. 2006.

[7] Eduard Čech. *Point Sets*. Academic Press, 1969. ISBN: 0121648508.

[8] Wenjie Fang. “Bijections between planar maps and planar linear normal lambda terms with connectivity condition”. In: *Advances in Applied Mathematics* 148 (2023).

[9] M.M. Fokkinga. “A Gentle Introduction to Category Theory - the calculational approach”. In: *STOP 1992 Summerschool on Constructive Algorithmics*. 1992.

[10] Frank Harary. *Graph Theory*. Addison Wesley Publishing Company, 1969.

[11] Rani Hod et al. “Strong embeddings and 2-isomorphism”. In: *Notices of the International Congress of Chinese Mathematicians* 4 (2016).

[12] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International (UK) Ltd., 1987.

[13] Stephen C. Kleene. “Origins of Recursive Function Theory”. In: *Annals of the History of Computing* 3.1 (1981).

[14] Donald Ervin Knuth. “Examples of formal semantics”. In: *Symposium on Semantics of Algorithmic Languages*. 1971.

[15] Ciro Lopez. “Three computational models and its equivalence”. In: *arXiv:2010.15600* (2020).

[16] Harry G. Mairson. “Linear lambda calculus and PTIME-completeness”. In: *Journal of Functional Programming* 14.6 (2004).

[17] Stephen Melczer. *An Invitation to Enumeration*. URL: `https://enumeration.ca/basics/classes/` (visited on 06/05/2024).

[18] Philip Rideout. *Tait Coloring*. 2020. URL: `https://prideout.net/blog/tait_coloring/` (visited on 06/05/2024).

[19] Dana S. Scott. “Relating theories of the lambda calculus”. In: *To HB Curry: Essays on combinatory logic, lambda calculus and formalism* (1980).

[20] Brigitte Servatius and Peter R. Christopher. “Construction of Self-Dual Graphs”. In: *The American Mathematical Monthly* 99.2 (1992).

[21] W. T. Tutte. “A Census of Planar Maps”. In: *Canadian Journal of Mathematics* 15 (1963).

[22] Noam Zeilberger. “Lambda calculus and the Four Colour Theorem”. In: *The Oxford Advanced Seminar on Informatic Structures*. 2018.

[23] Noam Zeilberger. “Linear lambda terms as invariants of rooted trivalent maps”. In: *Journal of Functional Programming* 26 (2016).