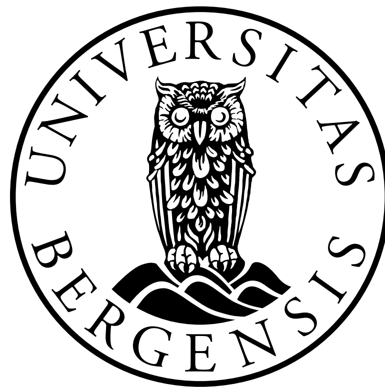
Investigations in Graph-theoretical Constructions in Homotopy Type Theory

Jonathan Prieto-Cubides



Thesis for the Degree of Philosophiae Doctor (PhD)

University of Bergen

2024

© Copyright Jonathan Prieto-Cubides
The material in this publication is covered by the provisions of the Copyright Act.

Year:	2024
Title:	Investigations in Graph-theoretical Constructions in Homotopy Type Theory
Author:	Jonathan Prieto-Cubides
Print:	Skipnes Kommunikasjon / University of Bergen

Scientific Environment

This thesis, a product of the Department of Informatics at the University of Bergen within the ICT Research School, was supervised by Håkon Gylterud and co-supervised by Marc Bezem.

Abstract

This thesis presents a constructive and proof-relevant development of graph theory concepts within Homotopy Type Theory (HoTT). HoTT, an extension of Martin-Löf's intuitionistic type theory, incorporates novel features like Voevodsky's Univalence principle and higher inductive types. Its structuralist perspective aligns with standard mathematical practise by promoting isomorphisms to equalities - inhabitants of the corresponding Martin-identity Löf's type. This thesis primarily delves into the foundational mathematics of graphs, with less emphasis on their practical aspects. The core contributions are definitions, lemmas, and proofs, which often arise from a synthesis of informal presentation and formalisation within a proof assistant. We specifically work within the category of directed multigraphs in HoTT.

Inspired by the topological and combinatorial facets of graphs on surfaces, we formulate an elementary characterisation of planar graphs. This is done without defining a surface or directly working with real numbers, as found in some literature. Our approach hinges on graph maps and faces for locally directed and connected multigraphs. A graph qualifies as planar if it features a graph map and an outer face, allowing any walk in the embedded graph to be merely walk-homotopic to another.

Among our key discoveries, we ascertain that this kind of planar maps constitutes a homotopy set, which is finite when the graph is finite. Additionally, we introduce extensions of planar maps to inductively generate examples of planar graphs. We also delve into further concepts such as spanning trees and the representation of graphs as spaces through graph maps.

2020 Mathematics Subject Classification: 03B38, 03B70, 68V20, 68V35, 03F65, 05C10

Keywords: constructive mathematics, type theory, mathematical formalisation, univalent foundations, graphs, surfaces, planar graphs, trees

Abstrakt

Denne avhandlingen presenterer en konstruktiv og bevisrelevant utvikling av grafteorikonsepter innen Homotopi-typeteori (HoTT). HoTT er en utvidelse av Martin-Löfs intuisjonistiske typeteori og inkorporerer nyskapende elementer som Voevodskys univalensprinsipp og høyere induktive typer. Dette strukturalistiske perspektivet er mer i tråd med standard matematisk praksis enn tidligere formelle systemer fordi i HoTT forfremmes isomorfier til likheter — elementer i den korresponderende Martin-Löf identitetstypen. Denne avhandlingen går hovedsakelig inn i de grunnleggende aspektene av grafteori, med mindre vekt på de praktiske sidene. Hovedbidragene er definisjoner, lemmaer og bevis, som oppstår fra en syntese av uformell presentasjon og formalisering ved hjelp av en bevisassistent. Vi arbeider spesifikt i kategorien av rettede multigrafer i HoTT.

Inspirert av topologiske og kombinatoriske aspekter av grafer på overflater, formulerer vi en grunnleggende karakterisering av planare grafer. Dette gjøres uten å definere en overflate eller direkte arbeide med reelle tall, som man finner i annen litteratur. Vår tilnærming er basert på grafkart og flater for lokalt rettede og sammenhengende multigrafer. En graf kvalifiserer som plan hvis den inneholder et grafkart og en ytre flate, slik at enhver sti i den underliggende grafen bare er turhomotopisk til en annen.

Blant våre viktigste funn, fastslår vi at denne typen planarkart utgjør en homotopisk mengde, som er endelig når grafen er endelig. I tillegg introduserer vi utvidelser av planarkart for å induktivt generere eksempler på planære grafer. Vi ser også på konsepter som spenntrær og representasjon av grafer som rom gjennom grafkart.

▷ *“What matters in life is not what happens to you but what you remember and how you remember it.”*

Gabriel García Márquez

Acknowledgements

For those who inspire me and do not even know it.

I would like to express my gratitude to UiB and, in particular, to my mentors Håkon Gylterud and Marc Bezem for providing me the opportunity to pursue my research without any restrictions. I extend special thanks to Håkon for his assistance, motivation, intriguing questions, countless hours spent at the whiteboard, calls, and much more. Also, I am deeply thankful to Marc for his guidance, encouragement to engage in conferences and summer schools, personal advice, and the time he devoted to reviewing and providing feedback on my manuscripts. My sincere gratitude to both of you.

I was privileged to be part of the Programming Theory Group (PUT) at UiB. My gratitude extends to my friends and office mates in PUT: Benjamin Chetioui, Tam Thanh Truong, Elisabeth Bonnevier, and Knut Anders Stokke. Stimulating discussions with Mikhail Barash, Magne Haveraaen, Erlend Raa Vågset, Uwe Wolter, Daniel Hernandez, and Michal Walicki were greatly appreciated.

I extend special thanks to UiB’s Department of Informatics’ administrative staff, particularly Ingrid Kyllingmark, for their consistent support. In Bergen, my gratitude goes to Camila Pachecho and Diana Piedra for numerous cooking and chatting sessions, and the Graham Linge family for making the lockdown more bearable during the pandemic.

This thesis discusses mathematical constructions in a recent and exciting research field in type theory where many branches of mathematics and computer science intersect. Throughout the process of working on this document, I had the privilege of attending conferences and summer schools, meeting many people and gaining valuable knowledge. Also, this document has been crafted using several software tools: Agda, Emacs, Ipe, XeLaTeX, Mathematica, VsCode, q.uiver.app, and the Pragmata and Libertinus fonts. Thanks to their developers, whose dedication has been instrumental in my work.

I express my gratitude to the following organisations for their support:

- COST organisation, for funding my attendance at the EU Types Summer School 2018 (Action CA15123),
- Department of Informatics, University of Bergen, for covering all expenses related

to my participation in the 2019 Midlands Graduate School in Birmingham and CMU HoTT Summer School in Pittsburgh, and

- Agda Dev Team, for inviting me to join the Agda meetings in 2017 and 2019.

I am deeply grateful to numerous talented members of the community who provided direct and indirect inspiration and assistance. My thanks extend to Andreas Abel, Andrej Bauer, Ulrik Buchholtz, Pierre Cagne, Jesper Cockx, Bjørn Dundas, Martin Escardo, Favonia, Nicolai Kraus, Stefano Piceghello, Egbert Rijke, Jakob von Raumer, and Noam Zeilberger, among many others. For those not mentioned here, please see the reference section. Your dissemination of high-quality research and generous sharing of time and knowledge are greatly appreciated.

I am immensely grateful to my family and friends around the world for their unwavering encouragement and support throughout my journey. I want to express my deepest love and appreciation to Polis and Agdis. I also want to extend my unconditional gratitude to my parents, Luz Mila Cubides and Rafael Prieto, as well as my sister, Lis, for their support in pursuing my goals, even when it meant being away from them.

Gracias a todos.

To my beloved Polis and Agdis.

Contents

Scientific Environment i

Abstract ii

Acknowledgements iv

1 Introduction 1

1.1 Foundations of mathematics . . . 1

1.1.1 Set theories . . . 2

1.1.2 Constructive formal systems . . . 3

1.1.3 Type theories . . . 4

1.1.4 Martin-Löf type theories . . . 5

1.1.5 Typing rules . . . 5

1.1.6 Types, terms, and logic . . . 8

1.1.7 Formulas as types . . . 9

1.1.8 Dependent types . . . 11

1.1.9 Identity types . . . 14

1.1.10 Extensional and intensional type theories . . . 16

1.1.11 The groupoid model and the homotopy interpretation . . . 17

1.2 Exploring graph theory in univalent mathematics . . . 19

1.2.1 Structure identity principle . . . 20

1.2.2 The type of graphs and their symmetries . . . 21

1.2.3 Drawing graphs on surfaces . . . 23

1.2.4 The notion of graph maps and faces . . . 24

1.2.5 Planar drawings . . . 26

1.3 Formalisation of mathematics . . . 29

1.4 Formalisation of graph-theoretical concepts . . . 30

1.5 Short outline of this thesis . . . 31

2 Mathematical Foundations **33**
2.1 Notation . . . 34
2.2 Homotopy levels . . . 36
2.3 Handy equivalences . . . 38
2.4 Finite types . . . 39
2.5 Cyclic types . . . 42

3 Graphs in Univalent Mathematics **46**
3.1 The type of graphs . . . 46
3.2 The category of graphs . . . 48
3.3 Subtypes and structures on graphs . . . 50
3.4 Finite graphs . . . 51
3.5 Walks and strongly connected graphs . . . 51
3.6 Graph families . . . 52
3.7 Cyclic graphs . . . 53
3.8 The identity type on graphs . . . 54

4 Graph Maps **56**
4.1 Symmetrisation of graphs . . . 56
4.2 Stars and locally finite graphs . . . 58
4.3 The type of combinatorial maps . . . 59
4.4 The type of faces . . . 61
4.4.1 The finiteness property . . . 67
4.4.2 The boundary of a face . . . 71
4.5 Examples of graph maps . . . 73
4.5.1 Generating graph maps . . . 73

5 Walks and Spherical Maps **78**
5.1 The type of walks . . . 78
5.1.1 Structural induction for walks . . . 79
5.1.2 A well-founded order for walks . . . 80
5.1.3 Walk splitting . . . 81
5.2 The type of quasi-simple walks . . . 83
5.2.1 The finiteness property . . . 85
5.3 Normal forms for walks . . . 89
5.4 The notion of walk homotopy . . . 92
5.5 The type of spherical maps . . . 94
5.6 Discussion . . . 101

6 Planar Maps **104**
6.1 Planarity in graph theory 104
6.2 A type of planar maps for a graph 105
6.3 Planar extensions 108
6.3.1 Path additions 108
6.3.2 Planar synthesis of graphs 116
6.3.3 Biconnected planar graphs 118

7 Concluding Remarks and Future Work **121**
7.1 Directions of further developments 123
7.2 Formalisation 126

Epilogue **127**

A Computer Formalisation in Agda **130**
A.1 Proof assistants 130
A.2 Agda notation 131
A.3 Library 133
A.4 Short excerpts from the library 135

Appendices

B On Trees and Their Topological Realisation **148**
B.1 Introduction 148
B.2 Computer formalisation in Cubical Agda 149
B.3 Basic concepts 149
B.3.1 The type of graphs 149
B.3.2 The type of walks 150
B.3.3 Rooted trees and subgraphs 151
B.4 Enlarging rooted subtrees 153
B.4.1 Oriented spanning trees 160
B.5 Topological realisation of graphs 162
B.6 Discussion 166

C Yet Another HIT for Graphs **169**
C.1 The 2-cell topological realisation of graphs 171
C.1.1 Promoting walks to equalities 171
C.1.2 Recursion principle 172
C.1.3 Induction principle 173
C.1.4 Eliminating into propositions 176

C.1.5 Eliminating into sets 177
C.2 Promoting walk homotopies to 2-paths 178

D Other Constructions 181

1

Introduction

This introduction succinctly presents the mathematical foundation pertinent to this thesis, supplemented with relevant historical context and examples. It concludes with an overview of the thesis structure.

1.1 Foundations of mathematics

Mathematics can be seen as the general study of structures such as the *symmetry* of objects in geometry, algebra, and category theory (Reck and Schiemer 2020). These objects may include numbers, sets, groups, graphs, topological spaces, and more.

The study of all mathematical objects is fundamentally built upon certain entities, often overlooked —the *primitive concepts*. These are not defined by other mathematical objects, but rather provided by the *mathematical foundation* in use. Examples include the concept of a *set* in set theories and the concept of a *type* in type theories.

The choice of a suitable foundation depends on the nature of the objects studied and their research goals. Over the past century, a variety of formalism proposals have emerged (Troelstra 2011). Set theories, having attracted more attention, contrast with type theories. The choice of Homotopy type theory (HoTT) for this thesis is motivated by its unique features not present in set theoretic foundations, particularly, due to its inherently structuralist foundational language for mathematics, and the way it integrates logic internally.

1.1.1 Set theories

Set theories, initially proposed by Cantor and Dedekind in the late 19th century and later reformulated by mathematicians such as Zermelo and Fraenkel in the Zermelo–Fraenkel set theory (ZF), are often considered the standard approaches for conducting mathematics in conjunction with classical logic. In set theories, the fundamental concept is that of a *set*. Each mathematical object is built upon this idea. A distinguishing feature of these theories is the external logic, governed by a first-order theory. This system forms the basis for creating *propositions* about sets using an equipped binary membership relation, denoted ($\in$).

A proposition in this context is a statement with a truth value —either true or false. The existence of objects within set theory and the validity of certain propositions via proofs may hinge on axioms such as the Axiom of Choice (AC). Furthermore, principles like the Law of Excluded Middle (LEM) and Reductio ad Absurdum (RAA) can also play a role in the reasoning process of these proofs.

Therefore, in the foundations of set theory, we observe a clear distinction between the deductive system and the objects under consideration.

The emergence of *paradoxes* in set theory, primarily stemming from the acceptance of *impredicative* statements like Richard's and Russell's paradoxes (Bagaria 2021), questioned the *consistency* of mathematics. This period, bridging the late 19th and early 20th century, was dubbed by mathematicians and philosophers as *The Foundational Crisis* and marked a pivotal moment in the history of mathematics.

In the midst of these developments, Hilbert proposed an agenda in the early 20th century, aiming to establish a solid foundation for mathematics through formalisation and consistency. The endeavor, termed as Hilbert's Program, aimed to axiomatize mathematics, assert its completeness and consistency, and outline the scope of mathematical knowledge. At the core of Hilbert's Program lay his belief in the *Entscheidungsproblem*, which assumes a universal procedure for determining whether a mathematical statement is provable or not based on a given set of axioms.

Note 1.1. Hilbert's Program fostered the development of *Mathematical Logic*, particularly proof theory, and yielded significant results that contradicted his initial intuition about the nature of mathematics. These encompass Gödel's incompleteness theorems, which not only refuted the assumption of the existence of a complete and consistent set of axioms for all mathematics but also underscored the inherent limitations of formal systems, including, notably, their incapacity to affirm their own consistency. Furthermore, these theorems indirectly imply the non-existence of a universal algorithm that could conclusively determine the truth or falsehood of all mathematical statements, thereby refuting the feasibility of a comprehensive solution to the Entscheidungsproblem.

1.1.2 Constructive formal systems

Apart from classical treatments of set theory, such as ZF and von Neumann-Bernays-Gödel, we encounter *constructive formal systems* such as the Constructive Zermelo–Fraenkel set theory (CZF), and several *type theories*. These alternative systems not only present distinct philosophical perspectives and mathematical constructs, such as the primary notion of *type* in type theories, but they also adopt different reasoning methodologies compared to traditional mathematics.

Constructive formal systems involve moving away from the reliance on AC and its variations like Zorn's Lemma. Instead of merely positing the existence of an object or employing principles such as LEM and RAA to same effect, within a constructive system, we seek for an actual, tangible method for constructing the object in question. This aligns with the *intuitionism* philosophical perspective, where only objects that one can construct in time are considered to exist.

Note 1.2. Brouwer's philosophy of mathematics, known as *intuitionism*, is the reasoning framework of *intuisionistic mathematics*, one kind of *constructive mathematics*, such as Kleene's, Markov's and Bishop's approach. In Brouwer's perspective, *intuition* is the *creative* device to do mathematics. Thus, his mathematics can not conform to classical principles, such as the proof of the existence of an *infinite* object, or the use of oracles such as AC. Constructive mathematics, on the other hand, is one branch of mathematics where the idea of *proofs as objects*, and the idea of *constructive proofs* as algorithms are taken seriously for the way one does mathematics.

In the context of our study, the role of intuitionistic logic is noteworthy. Contrary to sets, proofs are not perceived as mathematical objects. Moreover, logic is often viewed not as a mathematical object itself, but as a meta-mathematical tool —the reasoning system upon which mathematics relies, as observed in set theoretic foundations. Brouwer's perspective diverges here as hinted in Note 1.2, considering, among other aspects, logic as an integral part of mathematics. This stands in contrast to the *formalist approach* promoted by Hilbert and his collaborators.

However, it was not until the 1960s that intuitionistic mathematics began to gain traction, largely due to Bishop's work. Bishop adopted a pragmatic approach, using constructive methods and reevaluating Brouwer's and Heyting's ideas, as documented in his seminal book *Foundations of Constructive Analysis* (Bishop and Bridges 1985). Bishop's work on constructive mathematics later inspired the development of Martin-Löf's type theories (Petrakis 2019), subject of the following section.

Bishop's mathematics is primarily guided by three fundamental principles, which we strive to adhere to in this thesis (Troelstra 2011).

- ▹ Avoid concepts defined negatively; also avoid negative results.

▷ Avoid defining irrelevant concepts in favour of more relevant ones.

▷ Avoid pseudo-generality. Introduce any assumption if it facilitates the theory and the examples one is interested in satisfying the assumption.

1.1.3 Type theories

While no single definition unifies all type theories, a type theory can typically be characterised as a finite collection of rules relating *types* and *terms*, expressed using a formal *language* with semantics that justify the constructions and prevent inconsistencies (Barendregt, Dekkers, and Statman 2013; Nordström, Petersson, and Smith 1990).

In type theories, *type* is a primitive concept and hence, is not explicitly defined. However, to provide some intuition, a type can be viewed as a collection of values sharing a particular property, with terms being the elements that inhabit their type. This view allows us to consider other concepts such as the *empty* type as a type devoid of any terms and the *unit* type as the type with a single term. Other examples of types include the type of naturals, booleans, and lists, as seen in programming languages, which contain terms such as 123, true, and nil, respectively.

Type theories are not exclusively for non-classical reasoning (A. Bauer 2017). Classical mathematics can be incorporated simply by adding necessary axioms such as AC into the context. This approach ensures users have explicit knowledge of the axioms in use, thereby avoiding the implicit and occasionally arbitrary application common in non-constructive set theories.

As hinted earlier, one attractive feature of type theories, is the internalisation of logic within the theory. This stands in contrast to set theories, where we encounter two layers: the deduction system and the objects —sets being the domains of discourse. In type theories, we have a unified layer. This is achieved by the Curry–Howard correspondence, discussed in Section 1.1.7.

Prominent type theories include Simple Typed Lambda Calculus (STLC) , Martin-Löf's Intuitionistic Type Theory (MLTT) (Martin-Löf 1975), Coquand's and Huet's Calculus of Constructions (CoC) (Coquand and Huet 1988), and several others present in modern programming languages and notably in proof assistants like Agda (The Agda Development Team 2023), Coq (The Coq Development Team 2021), and Lean (Moura, Kong, Avigad, et al. 2015).

Here we focus on the application of HoTT for conducting constructive mathematics. HoTT extends MLTT with several additional features, some of which will discuss in Chapter 2.

Note 1.3. Russell's *Ramified Type Theory* (RTT) is a historical precursor to the type theories discussed above. While different in nature, it has influenced subsequent theories like the Simply Theory of Types (Church 1940), see also Note 1.8. RTT was employed in *Principia Mathematica* (PM), marking the first attempt at formalising mathematics within a set-theoretic context using type theory and symbolic logic principles to avoid inconsistencies. In RTT, mathematical objects are classified into types such as individuals, propositions, or n-ary relations (Linsky and Irvine 2022). For a comprehensive historical account on constructive mathematics, Troelstra's investigation serves as an excellent reference (Troelstra 2011), and for a detailed discussion on history of type theories prior HoTT, see (Kamareddine, Laan, and Nederpelt 2005).

1.1.4 Martin-Löf type theories

Martin-Löf Type Theories (MLTTs) is a family of formal systems stemming from Martin-Löf's seminal work on intuitionistic type theories in the 1970s (Martin-Löf 1975), serving as a foundation for constructive mathematics. MLTTs interweave terms and types via *dependent types*, allowing types to depend on terms, thus forming type families. Moreover, these types can be terms of other types, known as *universes*, denoted later by $\mathcal{U}$. These constructs, along with other features like inductive types, not only endow MLTTs with the expressiveness power to encode mathematical structures and other concepts, but also facilitate the creation of new constructions, all while maintaining constructive reasoning capabilities.

Modern constructive type theories have originated from MLTTs, adopting or reformulating the language and analysing concepts such as proposition, judgement, and equality. MLTTs are formulated using rules that infer valid judgements, that take one of the following primitive forms.

- ▷ A is a *type* denoted by $A : \mathcal{U}$.
- ▷ A and B are *equal* types denoted by $A \equiv B$.
- ▷ a is of type A denoted by $a : A$.
- ▷ a and b are equal terms of type A denoted by $a \equiv b$.

1.1.5 Typing rules

Typing rules in MLTTs are formulated to infer valid judgements, often presented in natural deduction style, as introduced by Gentzen (Gentzen 1964).

A *typing context*, denoted by a finite sequence of term-type pairs, is employed when considering judgements, although sometimes omitted for brevity. Contexts can range

from empty to containing multiple assumptions such as $x : A$ and $y : B$, symbolised by $x : A, y : B$.

For instance, given an arbitrary context Γ, one could represent the four primitive judgements introduced earlier as follows. The turnstile symbol ($\vdash$) is used to indicate that the judgement is established within the context Γ.

- ▷ $\Gamma \vdash a : A$, read as a is a well-formed term of type A in Γ.
- ▷ $\Gamma \vdash a \equiv b : A$ read as a and b are equal terms of type A in Γ.

Now to determine whether an arbitrary judgement is well-formed, we need to consider the rules of the type theory in use. These rules are presented in natural deduction style and share a common structure. Each rule consists of premises and conclusions, accompanied by a context. Premises are placed above the line, indicating implication, while the conclusion is below it.

Consider, for example, the rule to state that if a is of type A in Γ, then a remains of type A in the extended context $\Gamma, y : B$ for any type B; a rule called *weakening*. This rule is presented as follows.

$$\frac{\Gamma \vdash a : A}{\Gamma, y : B \vdash a : A}$$

Consider another example, the *assume* rule, also known as the declare rule. Given a context containing a variable x of type A, one can obtain $x : A$. The rule is represented in the following way.

$$\frac{}{(\ldots, x : A, \cdots) \vdash x : A}$$

The nature of rules varies based on the specific type theory and its purpose. Some rules guide the construction of new types and terms. Others, such as the weakening rule or the declare rule above, are fundamental to the foundational principles, logic, and operational semantics of the type theory.

However, in a typical presentation as for MLTT, we mostly encounter *formation*, *introduction*, *elimination*, and *computation rules*. To illustrate how these rules work, we use product type, denoted by $A \times B$. Here, the product type of A and B for pairs (a, b) where $a : A$ and $b : B$. Assuming an ambient context for these rules, the context Γ and the turnstile symbol are henceforth omitted.

Formation rules

Formation rules guide the construction of new types. These are formed from primitive types, like the unit type, via type formers such as (co)products, Σ- and Π-types. The formation rule for the product is given by:

$$\frac{A : \mathcal{U} \qquad B : \mathcal{U}}{A \times B : \mathcal{U}}$$

Introduction rules

Introduction rules specify how to construct terms of a given type. With the product type, we have the following introduction rule:

$$\frac{a : A \qquad b : B}{(a, b) : A \times B}$$

Elimination and computational rules

Elimination rules dictate the use of types and terms, specifying functions on the type. For the product type, there are two elimination rules. These rules allow us to take a pair of elements with the goal of extracting either the first or the second element. We achieve this introducing two functions, namely π_1 and π_2 that intend to project the parts of the product.

$$\frac{p : A \times B}{\pi_1(p) : A} \qquad \frac{p : A \times B}{\pi_2(p) : B}$$

The functions π_1 and π_2, previously introduced, are yet to be specified. We need to define their behaviour under the elimination rules during term computation, which involves introducing equalities that dictate how these functions reduce.

Consider a pair $(a, b) : A \times B$. Computation rules simplify term expressions such as $\pi_1((a, b))$ and $\pi_2((a, b))$ to a and b, the first and second elements of the pair, respectively.

$$\frac{a : A \qquad b : B}{\pi_1(a, b) \equiv a} \qquad \frac{a : A \qquad b : B}{\pi_2(a, b) \equiv b}$$

Remark 1.4. Types may possess multiple introduction rules, a single one, or none at all. For instance, the empty type ($\mathbb{0}$) lacks an introduction rule, while the unit type ($\mathbb{1}$) has a single introduction rule with one inhabitant denoted by $\star$.

$$\frac{}{\star : \mathbb{1}}$$

The natural numbers type, denoted by $\mathbb{N}$, is an example of a type with multiple introduction rules. A natural number is either the term zero or a term derived from the function suc, which maps a natural number to its successor. These cases form the introduction rules for natural numbers.

$$\frac{}{\mathsf{zero} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{suc}(n) : \mathbb{N}}$$

Hence, natural numbers like one and two are represented as suc(zero) and suc(suc(zero)), respectively. The same type is often presented inductively in Agda-like notation, as shown below.

$$\begin{aligned} &\textbf{data}\ \mathbb{N} : \mathcal{U} \\ &\quad \mathsf{zero} : \mathbb{N} \\ &\quad \mathsf{suc} : \mathbb{N} \to \mathbb{N} \end{aligned}$$

The non-dependent elimination rule for natural numbers is known as the *recursion principle*, while the dependent elimination rule is the *induction principle*. Since the former can be seen as a particular case of the latter, let only present the induction principle here.

$$\frac{P : \mathbb{N} \to \mathcal{U} \qquad b : P(\mathsf{zero}) \qquad f : \prod_{(n:\mathbb{N})} (P(n) \to P(\mathsf{suc}(n)))}{\mathsf{ind}_{\mathbb{N}}(P, b, f) : \prod_{(n:\mathbb{N})} P(n)}$$

The term computation for the induction principle is given by the following two rules.

$$\frac{P : \mathbb{N} \to \mathcal{U} \qquad b : P(\mathsf{zero}) \qquad f : \prod_{(n:\mathbb{N})}(P(n) \to P(\mathsf{suc}(n)))}{\mathsf{ind}_{\mathbb{N}}(P, b, f)(\mathsf{zero}) \equiv b : P(\mathsf{zero})}$$

$$\frac{P : \mathbb{N} \to \mathcal{U} \qquad b : P(\mathsf{zero}) \qquad f : \prod_{(n:\mathbb{N})} (P(n) \to P(\mathsf{suc}(n)))}{\mathsf{ind}_{\mathbb{N}}(P, b, f)(\mathsf{suc}(n)) \equiv f(n)(\mathsf{ind}_{\mathbb{N}}(P, b, f, n)) : P(\mathsf{suc}(n))}$$

Example 1.5. The addition function on natural numbers, denoted by add, can be defined using the induction principle above. This function is defined by induction on the first argument. If the first argument is zero, the result is the identity function on the second argument. Conversely, if it is not zero, the result is the successor of the result of the function add applied to the predecessor of the first argument and the second argument.

$$\begin{aligned}
&\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.\\
&\mathsf{add} :\equiv \mathsf{ind}_{\mathbb{N}}(\underbrace{\lambda n.\, \mathbb{N} \to \mathbb{N}}_{P}, \underbrace{\lambda m.\, m}_{b}, \underbrace{\lambda n.\, \lambda g.\, \lambda m.\, \mathsf{suc}(g(m))}_{f}).
\end{aligned}$$

1.1.6 Types, terms, and logic

The close relationship between types, terms, and logic emerged during the development of Lambda Calculus. This connection traces back to work in the 1920s by L.E.J. Brouwer, Heyting, and Kolmogorov, who focused on a computational view of intuitionistic logic known as *the BHK interpretation*.

The BHK interpretation of formulas suggests that we consider proofs as *algorithms*. Let us consider this interpretation on the structure of a formula, avoiding extra notation for the sake of simplicity.

- ▷ A proof of $A \wedge B$ is a pair of a proof of A and a proof of B.
- ▷ A proof of $A \vee B$ is either a proof of A or a proof of B.
- ▷ A proof of $A \to B$ is a function f that transforms a proof of A into a proof of B.
- ▷ A proof of $\exists x{\in}D(A(x))$ is a pair of an element x and a proof that the $A(a)$ is true, in some domain D.

- ▹ A proof of $\forall x{\in}D(A(x))$ is a function f that converts an element x into a proof of Ax, in some domain D.
- ▹ The formula $\neg A$ is defined as $A \to \bot$, so a proof of it is a function f that converts a proof of A into a proof of $\bot$. There is no proof of $\bot$.

However, at the time of the formulations the BHK interpretation left imprecise fundamental concepts: What constitutes a proof? How do we define a function or an algorithm? Are these primary notions, or do they stem from a more fundamental concept?

Note 1.6. The term *effectively calculable*, associated with *computable functions* and *algorithms*, was coined by Hilbert and Ackermann in the 1920s. It emerged from their quest to identify a mechanism determining the provability of a mathematical statement, as referenced in Note 1.1. In pursuit of a formal definition for *effectively calculable*, several computational models were proposed. These encompass Gödel's *general recursive functions* (1934), Turing's *Turing machines* (1936), and Church's *lambda definability* in both Lambda Calculus (1936).

Church first introduced the Lambda Calculus in 1932 (Church 1932), featuring the now ubiquitous λ-symbol to denote anonymous functions (Barendregt 1997). This innovation inspired numerous formal systems and notations, including those used in HoTT. In particular, Lambda Calculus was conceived as an improved method for encoding mathematics and logic, enabling every object to be represented as a higher-order function with a single argument. For instance, a natural number n could be represented as a function mapping any other function to its n-fold composition. In other words, if we would like to represent 3 in Lambda Calculus, we can do so by defining the term $\lambda f.\lambda x.f(f(f(x)))$. True and false values are defined as combinators $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$, respectively, and used to represent logical connectives [a]. However, Rosser and Kleene identified inconsistencies in the original untyped lambda calculus, specifically the existence of a fixed point for negation and the presence of nonterminating lambda expressions. This led to the development of STT, mentioned in Note 1.8.

[a] https://plato.stanford.edu/entries/lambda-calculus/#LogLaL

1.1.7 Formulas as types

The Curry–Howard Isomorphism, initially proposed by Curry and later supplemented by Howard, offers a formal context to address the latter questions by establishing a formal relationship between logical connectives (including quantifiers) and the unspecified meanings for concepts suggested by the BHK interpretation, such as proof and function (Howard 1980).

Oversimplifying, Curry and Howard's contribution connects two seemingly unrelated domains[1], namely, Intuitionistic Logic and models of computations via Lambda Calculus (see Note 1.6). Curry and Howard's idea involves identifying formulas with types and

[1] https://www.cs.ru.nl/~herman/slidesMLNL2009.pdf

proofs of those formulas as terms of their corresponding type. This relationship, is in fact, a bijection/isomorphism, referred to as *proposition-as-types*, *formulas-as-types*, or *The Curry–Howard–de Bruijn's* correspondence.

Via this correspondence, we are able to unify the objects and the deductive system in one internal framework, this goes in contrast with naive set theories where there are separated in two layers, a first-order logic layer governing our reasoning and propositions, and the other layer comprising the objects, sets in this case. In type theory, sets and propositions, all live inside the theory. Let us consider some examples to illustrate this correspondence.

Via propositions-as-types view, the logical implication $A \to B$ for propositions A and B aligns to the function type $A \to B$ in type type theory. The formation of the logical implication $A \implies B$ from propositions A and B parallels the construction of a function that transforms a proof of A into a proof of B.

In the following, on the left, we have the implication introduction rule in propositional logic, and on the right the introduction rule for the function type in type theory.

$$\dfrac{\begin{matrix}[A\ \mathbf{true}]\\ \vdots \\ B\ \mathbf{true}\end{matrix}}{A \implies B\ \mathbf{true}} \qquad\qquad \dfrac{x : A \vdash e : B}{\vdash \lambda x.e : A \to B}$$

This interpretation casts the *modus ponens* rule in propositional logic as function application. Specifically, given a function f that transforms a proof of A into a proof of B, and a proof p of A, $f(p)$ is thereby a proof of B.

$$\dfrac{A \implies B\ \mathbf{true} \qquad A\ \mathbf{true}}{B\ \mathbf{true}} \qquad\qquad \dfrac{f : A \to B \qquad p : A}{f(p) : B}$$

Also, via propositions-as-types, the logical conjunction $A \wedge B$ aligns to the product type $A \times B$ in type theory.

$$\dfrac{A\ \mathbf{Prop} \qquad B\ \mathbf{Prop}}{A \wedge B\ \mathbf{Prop}} \qquad\qquad \dfrac{A : \mathcal{U} \qquad B : \mathcal{U}}{A \times B : \mathcal{U}}$$

Now, the introduction rule for conjunction, stating that if p is a proof of A and q is a proof of B, then, together, p and q form a proof of $A \wedge B$, which parallels the introduction rule for the product type, as shown below. Specifically, if p is a term of type A and q is a term of type B, then (p, q) is a term of type $A \times B$. Such parallelism extends to other logical rules, see Section 1.1.8.

$$\dfrac{\dfrac{\vdots}{A\ \mathbf{true}}\,p \qquad \dfrac{\vdots}{B\ \mathbf{true}}\,q}{A \wedge B\ \mathbf{true}}\ \wedge\text{-intro}(p, q) \qquad\qquad \dfrac{p : A \qquad q : B}{(p, q) : A \times B}$$

To address the correspondence with universal and existential quantifiers, we first discuss dependent types.

1.1.8 Dependent types

Bishop's constructive approach and the Curry–Howard correspondence jointly inspired the development of type theories featuring *dependent types*, which are types parametrised by terms of other types. This feature enables a richer language than existing systems, such as STLC, permitting the formulation of complex concepts and relations between the mathematical objects in use in a coherent and intuitive manner.

In dependent type theories, the concept of a *type family* is particularly relevant. A type family can be regarded as a collection of types indexed by another type. For instance, a type family B indexed by A means that for every $x : A$, there is a type $B(x)$ that may depend on x. Additional type formers for dependent types include Σ-types and Π-types.

- ▷ The dependent sum $\Sigma_{x:A}B(x)$ represents the type of pairs (a, b) where $a : A$ and $b : B(x)$ for types A and B of x. Therefore, the type of products is a particular case of dependent sums, where B is a constant type family. That is, $A \times B$ is equivalent to $\Sigma_{x:A}B$. We, therefore, use the same notation π_1 and π_2 for the projections of dependent sums and products.

- ▷ The dependent product $\Pi_{x:A}B(x)$ denotes the type of functions f such that for $x : A$, one has $f(x) : B(x)$. The type of functions can be also seen as a particular case of dependent products, where B is a constant type family. That is, $A \to B$ is equivalent to $\Pi_{x:A}B$.

For conciseness, we exclude further details such as inductive types and pattern-matching discussions. Nordström et al's book elaborates on these topics along with the rules and semantics of MLTT (Martin-Löf 1975; Nordström, Petersson, and Smith 1990).

We can now present the correspondence of formulas-as-types, specifically the connection between logical quantifiers and dependent types.

Table 1.1: Formula-as-types correspondence.

Formula	Type
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \implies B$	$A \to B$
$\bot$	$\mathbb{0}$
$\top$	$\mathbb{1}$
$\forall x \in D(A(x))$	$\Pi_{x:D} A(x)$
$\exists x \in D(A(x))$	$\Sigma_{x:D} A(x)$

- The formula $\forall x \in D(A(x))$ represents a predicate logic statement: "for all x in D, $A(x)$ is true". Here, D is a domain of discourse (a set of elements under consideration), x is an element in D, and A is a predicate on D. Thus, $A(x)$ is a truth-assignable statement about x. This formula corresponds, via Curry–Howard, to the type $\Pi_{x:D} A(x)$, where D is a type and A is a type family indexed by D such that $A(x)$ is a proposition for all x.

- Similarly, $\exists x \in D(A(x))$ represents a predicate logic statement: "there exists an x in D for which $A(x)$ is true". This formula corresponds to the type $\Sigma_{x:D} A(x)$, where again D is a type and A is a type family indexed by D. Proofs of $\exists x \in D(A(x))$, corresponds to pairs (a, p) presenting the witness a that makes the proposition holds.

Remark 1.7. A *choice* function f, as in set theory, is defined on a collection X of nonempty sets. For each set A in X, $f(A)$ is an element of A (Suppes 1972, pp. 240.). The axiom of choice can be alternatively formulated as the existence of a choice function for every set or the ability to interchange quantifiers freely, as follows, where A and B are sets, and C is a relation on A and B.

$$\forall x \in A(\exists y \in B(C(x, y))) \implies \exists f \in B^A(\forall x \in A(C(x, f(x)))).$$

Using propositions-as-types and dependent types in MLTT, we can both express and derive the logical axiom of choice (ACL[2]). This derivation is referred to as the *type-theoretic choice principle*, permitting us to alternate between Πs and Σs. Specifically, consider $A : \mathcal{U}$, $B : A \to \mathcal{U}$, and $C : (\Sigma_{x:A} B(x)) \to \mathcal{U}$.

$$\prod_{(a:A)} \sum_{(b:B(a))} C((a, b)) \to \underbrace{\sum_{\left(f : \prod_{(x:A)} B(x)\right)}}_{\text{choice function}} \prod_{(a:A)} C((a, f(a))).$$

[2] https://plato.stanford.edu/entries/axiom-choice/choice-and-type-theory.html

The type above can be inhabited by defining the function choice as follows.

$$\begin{aligned}&\Pi\Sigma\text{-comm} : \prod_{(a:A)} \sum_{(b:B(a))} C((a,b)) \to \sum_{(f:\prod_{(x:A)} B(x))} \prod_{(a:A)} C((a,f(a))).\\&\Pi\Sigma\text{-comm}(h) :\equiv (\lambda a.\pi_1(h(a)), \lambda a.\pi_2(h(a))).\end{aligned}$$

Indeed, the converse can also be derived, demonstrating the ability to alternate Πs and Σs.

$$\begin{aligned}&\Sigma\Pi\text{-comm} : \sum_{(f:\prod_{(x:A)} B(x))} \prod_{(a:A)} C((a,f(a))) \to \prod_{(a:A)} \sum_{(b:B(a))} C((a,b)).\\&\Sigma\Pi\text{-comm}(h,k) :\equiv \lambda a.(h(a),k(a)).\end{aligned}$$

In HoTT, an extension of MLTT, we derive a formulation of the axiom of choice (Univalent Foundations Program 2013, §3.8). This requires the introduction of a new type constructor, specifically, the *higher inductive type* for the propositional truncation of a type, as detailed in Definition 2.2.

Note 1.8. Dependent types, to our knowledge, were first described in logical theory not by MLTT but by de Bruijn's Automath project. This project implemented an extended version of Lambda Calculus with dependent types such as Σ-types and Π-types (Bruijn 1983). Here are some related historical works.

Church's 1940 work, *The Simple Theory of Types* (STT), introduced a formal system for representing objects as lambda expressions annotated with types (Church 1940). This provided a more general and consistent presentation of the type theory initially introduced in Principia Mathematica, albeit with limited expressive power.

This laid the groundwork for advancements starting in the 1960s, which aimed to enhance the expressiveness of current proof systems and other systems grounded on Gentzen's Natural Deduction (Prawitz 1967; Schütte 1972). These enhancements included different forms of quantifiers, such as type abstraction bound by λ, type variables, and dependent types. Notable contributions in this direction include Girard's System F and System F_ω, Andrews's exploration of *type theory with type variables*, along with the previously cited de Bruijn's Automath project.

Example 1.9. Let A be the type of vector elements, and $\mathsf{Vec}(n)$ denote vectors of length n. The dependent type Vec is a family with each member, $\mathsf{Vec}(n)$, representing vectors of length $n : \mathbb{N}$. This can be defined either inductively, as follows.

$$\begin{aligned}&\textbf{data}\ \mathsf{Vec}\ (A : \mathcal{U}) : \mathbb{N} \to \mathcal{U}\\&\quad \mathsf{nil} : \mathsf{Vec}(A, \mathsf{zero})\\&\quad \mathsf{cons} : \Pi_{n:\mathbb{N}}\ A \to \mathsf{Vec}(A,n) \to \mathsf{Vec}(A, \mathsf{suc}(n))\end{aligned}$$

Alternatively, we can define this type through the function $\mathsf{Vec} : \mathbb{N} \to \mathcal{U}$ as shown in (1.1−1). This function can be defined by case-analysis on the natural numbers, this is referred to as *pattern-matching* on $\mathbb{N}$. In this approach, we define the function Vec by a collection of *clauses*, equations that define the function for specific cases of its argument.

$$
\begin{aligned}
&\mathsf{Vec} : \mathcal{U} \to \mathbb{N} \to \mathcal{U}. \\
&\mathsf{Vec}(A, \mathsf{zero}) :\equiv \top. \qquad (1.1\text{–}1)\\
&\mathsf{Vec}(A, \mathsf{suc}(n)) :\equiv A \times \mathsf{Vec}(A, n).
\end{aligned}
$$

We could write this type even shorter. If A^n denotes the type of n-tuples of elements of A, then we could have simply written $\mathsf{Vec}(A, n) :\equiv A^n$. Similarly, the type of list of elements of A, is defined as below.

$$
\begin{aligned}
&\mathsf{List} : \mathcal{U} \to \mathcal{U}. \\
&\mathsf{List}(A) :\equiv \Sigma_{n : \mathbb{N}} \mathsf{Vec}(A, n).
\end{aligned}
$$

From now on, we denote a list of elements of A by $[a_1, \dots, a_n]$ where $a_i : A$ for $i = 1, \dots, n$. For example, $[1, 2, 3]$ is a list of natural numbers of length three.

1.1.9 Identity types

One of the main focuses in the study of MLTTs is the family of *identity* types. Specifically, these types give rise to what is known as *propositional equality*, which is an equivalence relation among terms or types. Notably, propositional equality is distinct from *definitional equality*, the latter being a built-in notion of equality within the type theory, based primarily on computational or definitional attributes and rules. This distinction between propositional and definitional equality is of considerable significance. Unlike definitional equality, propositional equality allows for more complex and intricate expressions of equivalence, thereby expanding the range of results and equality relations that can be represented within type theory.

For every type A, there is a *family* of identity types relating every pair of its terms. The identity type, $\mathsf{Id}(A, a, b)$, frequently denoted by $a =_A b$, or simply $a = b$ when an ambient type is present, is referred to as the *propositional* equality between terms a and b of type A. At its core, the type $a =_A b$ represents the type of proofs that establish the equality between a and b.

$$
\frac{A : \mathcal{U} \qquad a : A \qquad b : A}{a =_A b : \mathcal{U}}
$$

Identity types are inhabited by a *canonical* term, which asserts that every object is equal to itself in a canonical manner. We denote such a term by $\mathsf{refl}(A, a)$ for the identity type $a =_A a$ in a rule we will put it as follow.

$$
\frac{A : \mathcal{U} \qquad a : A}{\mathsf{refl}(A, a) : a =_A a}
$$

Now that we have defined the identity type, we can define the *path induction* rule, which is the dependent elimination rule for the identity type. Given a family C of types indexed by the identity type $a =_A b$, we can define a function f that takes in any $p : a =_A b$ and spits out a term $C(p)$, as a rule we will put it as follow.

$$\frac{A : \mathcal{U} \qquad C : \prod_{(x,y:A)} (x =_A y) \to \mathcal{U} \qquad c : \prod_{(x:A)} C(x, x, \mathsf{refl}(A, x))}{\mathsf{ind}_=(A, C, c) : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)}$$

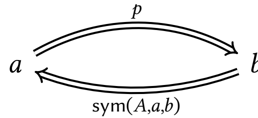
The function $\mathsf{ind}_=$ introduced above is also known as the J-rule. The term computation for the path induction rule is given by the following rule.

$$\frac{A : \mathcal{U} \qquad C : \prod_{(x,y:A)} (x =_A y) \to \mathcal{U} \qquad c : \prod_{(x:A)} C(x, x, \mathsf{refl}(A, x))}{\mathsf{ind}_=(A, C, c)(x, x, \mathsf{refl}(A, x)) \equiv c(x) : C(x, x, \mathsf{refl}(A, x))}$$

Example 1.10. The identity type forms an equivalence relation. It is reflexive by definition.

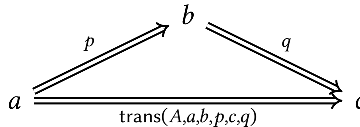
The symmetry can be shown by proving that for any $p : a =_A b$, there exists a term $\mathsf{sym}(A, a, b, p) : b =_A a$ by path induction.

$$\mathsf{sym} : \prod_{(A:\mathcal{U})} \prod_{(x,y:A)} \prod_{(p:x=_A y)} (y =_A x).$$
$$\mathsf{sym}(A, a, b, p) :\equiv \mathsf{ind}_=(A, \lambda x.\lambda y.\lambda p.\,(y =_A x), \lambda x.\mathsf{refl}(A, x))(a, b, p).$$



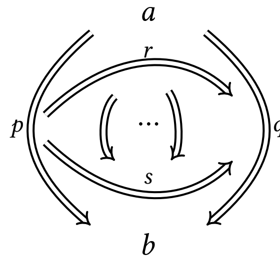
The transitivity of the identity type follows by defining the following function trans. Then, given $p : a =_A b$ and $q : b =_A c$, $\mathsf{trans}(A, a, b, c, p, q)$ is of type $a =_A c$

$$\mathsf{trans} : \prod_{(X:\mathcal{U})} \prod_{(x,y:X)} \prod_{(p:x=_X y)} \prod_{(z:X)} (y =_X z) \to (x =_X z).$$
$$\mathsf{trans}(A, a, b, c, p, q) :\equiv \mathsf{ind}_=(A, \lambda x.\lambda y.\lambda p.\prod_{(z:A)} (y =_A z) \to (x =_A z), \lambda x.\lambda y.\lambda r.\,r)(a, b, p, c, q).$$



As we will briefly discuss, the identity type is a common topic in the modern study of type theories. It introduces numerous complexities and poses intriguing questions. For example, consider two elements a, b of a type A. Suppose we establish two proofs of equality, p and q, both of type $a =_A b$. Why stop here? we can consider again the

equalities between p and q, that is, the identity type $p =_{a=_A b} q$. So, this iterative process could continue indefinitely, resulting in a tower of identity types.



Then, at what point do we reach a state/level in this recursion where we cannot longer distinguish between proofs of equality? The answer lies within the structure of the identity type. For certain types, like the type of natural numbers, we already know when this iteration halts, while for others, further investigation is required. In fact, in this document, we propose one type for a particular mathematical object, and one significant result is precisely said when this iteration halts for this particular type. We study the structure of the identity type to understand the structure of the mathematical object we are studying.

1.1.10 Extensional and intensional type theories

The structure of type of proofs establishing the equality between two terms is an essential distinction between dependent type theories. On one hand, we have *extensional* type theories where one assumes or derives the *reflection rule*, also called *Axiom K*. This rule identifies the notion of propositional equality and definitional equality (Nordström, Petersson, and Smith 1990, §3.9), which implies that any identity type is a proposition. In other words, in extensional type theories, any identity type has at most one inhabitant, and consequently, the type $a =_A a$ is only inhabited by $\mathsf{refl}(A, a)$, as in the following rule. Any other proof of $a =_A a$ is treated, by definition, as alias of $\mathsf{refl}(A, a)$.

$$\frac{a : A \qquad p : a =_A a}{p \equiv \mathsf{refl}(A, a)}$$

On the other hand, a more liberal approach is the *intensional* version of the identity type, in which such identity types may consist of more than one element. Understanding the structural intricacies of the identity type plays a fundamental role in the development of type theories and the way one conducts mathematics within these theories[3].

Consider the type $p =_{a=_A b} q$, where p and q are two proofs of the equality between terms a and b in type A. In certain type theories, the *Uniqueness of Identity Proofs* (UIP) principle holds, stating that for any such identity type, there is essentially only one proof of equality —that is, the identity type is a proposition, and therefore, it is referred to as

[3]http://www.cs.nott.ac.uk/~psztxa/martin-19.pdf

proof-irrelevant. If UIP holds, then it does not matter which proof we have: all proofs of a proposition are indistinguishable from each other. This stands in contrast with *proof-relevant* type theories, such as HoTT, where different proofs of an identity may carry different information and hence are not necessarily interchangeable. Whether UIP holds or not is a crucial aspect that differentiates various type theories.

1.1.11 The groupoid model and the homotopy interpretation

In 1998, Hofmann and Streicher laid the groundwork for a fundamental shift in perspective on the structure of identity types (Hofmann and Streicher 1998). They proposed a model for type theory wherein types were not mere sets, but *groupoids*. In the groupoid model, the UIP principle does not hold. The key idea is to treat equality as something that can have structure, possibly something more complex that just being a proposition. In other words, it embraced a framework where different proofs of equality, or paths, between the same pair of points were allowed to carry unique information and were thus not inherently indistinguishable.

Building on this radical departure, a significant evolution occurred about a decade later, instigated by the work of Awodey, Warren, and Voevodsky. In this approach, types are viewed as *homotopy types* —objects possessing the higher-dimensional structure of an ∞-*groupoid* (Awodey and Warren 2009). This so-called *homotopy interpretation* of type theory positioned types as analogous to spaces, terms as points within these spaces, and equalities as *homotopies* between points, mirroring path spaces in homotopy theory. This brought novel theoretical frameworks like HoTT, with their core principles frequently grounded in the notion of homotopy, as explained by Schulman in his view of a *theory of homotopy types*[4].

Initially, the apparent connection between homotopy theory and type theory may strike one as surprising. However, this connection may not have been so unexpected for Voevodsky. His interest in type theory could be considered a fortunate coincidence, likely sparked by concerns about the practice and foundations of mathematics. Voevodsky became deeply engaged with type theory as a promising approach to improve the way mathematics is conducted, aimed at producing better and more reliable proofs at a high level of abstraction, thereby preventing errors in human reasoning.

Voevodsky's substantial contributions to HoTT include key concepts like homotopy equivalence (Definition 2.4) and the Univalence axiom. This axiom is particularly noteworthy as it harmonises two fundamental ideas in type theory: equivalence and identity (Ahrens, North, Shulman, et al. 2021; Awodey 2018; Voevodsky 2010).

The Univalence axiom comes in the form of a term ua, acting as the inverse of the function idToEquiv, which maps the identity type $A =_{\mathcal{U}} B$ to an equivalence between A

[4] https://golem.ph.utexas.edu/category/2011/03/homotopy_type_theory_i.html.

and B for any types A and B. Therefore, the Univalence axiom extends the principle of extensionality to the universe of types, affirming that equivalent types are equal. This implies such types inherently share identical structures and properties.

This principle not only facilitates the transfer of constructions among equivalent types but also enhances the creation of new constructions and proofs. Moreover, it optimises the research development process by eliminating the need to prove identical results for different equivalent formulations.

$$A =_{\mathcal{U}} B \underset{\mathsf{ua}}{\overset{\mathsf{idToEquiv}}{\rightleftarrows}} A \simeq B \qquad (\simeq)$$

Remark 1.11. In HoTT, the assumption of Univalence implies function extensionality[5]. This extensionality principle, not provable in MLTT (but introduced as an axiom), states that two functions equal pointwise are indeed equal. This establishes the following equivalence that we employ, often without explicit mention, in the forthcoming chapters.

$$f =_{\prod_{x:A} B(x)} g \underset{\mathsf{funext}}{\overset{\mathsf{happly}}{\rightleftarrows}} \prod_{x:A} f(x) =_{B(x)} g(x) \qquad (\simeq)$$

However, utilising the Univalence axiom does present unique challenges. One such challenge, which also drives ongoing research, is the apparent lack of computational content, as in the extraction of algorithms from proofs and statements involving Univalence. Noteworthy efforts to tackle this issue include the development of Cubical type theories (Bezem, Coquand, and Huber 2017; Cohen, Coquand, Huber, et al. 2017; Coquand, Huber, and Mörtberg 2018). Almost in parallel, efforts have also been made to enhance proof assistants such as Agda, Lean, and Coq. These enhancements address the need for better tools to interact with Univalence and explore its computational implications (Vezzosi, Mörtberg, and Abel 2021).

We must conclude our brief journey here from the foundations of mathematics to the specific type theory utilised in this document — a foundation that encompasses concepts from both homotopy theory and type theory, as well as the various interpretations of type as propositions, sets, and groupoids. We will now transition to more focused discussions pertinent to this thesis. However, for readers intrigued by the intricate details of (homotopy) type theory, we highly recommend reviewing at least one of the following works.

- ▷ *Homotopy Type Theory: Univalent Foundations of Mathematics*, referred here as *The HoTT book* (Univalent Foundations Program 2013),

[5]See a proof in Agda that Univalence implies function extensionality: `https://gist.github.com/jonaprieto/bf9c151d4d7ea4f30fcd598366802e8e`.

- ▷ *An Introduction to Univalent Foundations For Mathematicians* (Grayson 2018),
- ▷ *Introduction to Univalent Foundations of Mathematics with Agda* (Escardó 2019),
- ▷ *Introduction to Homotopy Type Theory*[6] (Rijke 2021), and
- ▷ *The Symmetry Book* (Bezem, Buchholtz, Cagne, et al. 2022).

Additionally, as the interest in HoTT continues to grow, we are fortunate to find a lot of media resources and notes these days. Some notable examples include:

- ▷ Video lectures on the EPIT Summer School 2020[7],
- ▷ the HoTTEST Summer School 2022[8], and
- ▷ various Schools and Workshops on Univalent Mathematics[9].

Additional material can be found at `https://homotopytypetheory.org/`.

1.2 Exploring graph theory in univalent mathematics

As HoTT emerges as a foundational framework for mathematics, numerous domains and innovative constructions have been investigated. This includes examining algebraic structures as in Universal Algebra, delving into (synthetic) homotopy theory, topology, (higher) category theory, and more. However, there is still much to be explored, and the potential for new discoveries is vast, and combinatorics is one of these domains. Thus, in this work, we integrate graph theory concepts to enrich examples in the combinatorics expressed in HoTT.

In this section, we aim to provide intuition for a few related constructions to a proof-relevant notion of planar graphs in HoTT. These constructions are derived from transforming abstract topological concepts into their concrete (combinatorial) counterparts, potentially illuminating subtle nuances of this process.

Our study of graphs adopts a distinctive approach where, for example, graph isomorphisms are directly promoted as equalities, compared to traditional graph theory formalisation, as seen in the literature (Noschinski 2015). This uniqueness primarily stems from the adoption of Voevodsky's Univalence Axiom and the use of HoTT constructions, like propositional truncation, for expressing the existence of mathematical objects.

The forthcoming subsections are outlined as follows: We first discuss the structure identity principle and its relevance to our study in Section 1.2.1. This is followed by

[6] `https://ncatlab.org/homotopytypetheory/files/hott-intro.pdf`

[7] `https://github.com/HoTT/EPIT-2020`.

[8] `https://uwo.ca/math/faculty/kapulkin/seminars/hottest_summer_school_2022.html`.

[9] `https://hott-uf.github.io/2022/`.

an exploration of the type of graphs, their symmetries, and the appropriate notion of graph equivalence in Section 1.2.2. The topological concept of graph embeddings in surfaces is then examined to define the combinatorial notion of graph maps and faces in Sections 1.2.3 and 1.2.4, respectively. Finally, a characterisation of planar drawings of graphs is hinted at Section 1.2.5.

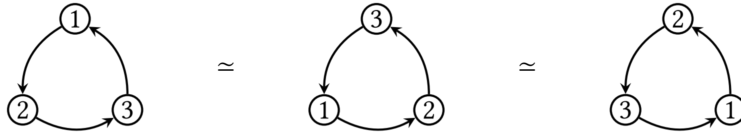
1.2.1 Structure identity principle

Finding equalities in type theory as in mathematical practice is a one common theme. Actually, one fundamental step in conducting mathematics in HoTT involves characterising identity types for a structure, such as groups, rings, and other algebraic structures. This goal relates to the question: "what does it mean for two objects to be equal?" Specifically, in HoTT, we require our identity types to adhere to the structure identity principle (SIP). This principle states that equivalent structures are considered equal. As a result, a unique canonical method exists to convert an equivalence between two structures into an equality, which consequently implies that their structural properties remain invariant under equivalence. In essence, SIP is a generalisation of two principles: the *identity of indiscernibles* and the *equivalence principle*. The first suggests that objects with identical properties are equal. The second, in broad terms, proposes that if two objects within a domain are equivalent, they share all properties reciprocally (Ahrens and North 2019).

However, defining the most suitable notion of equivalence for a specific structure in HoTT is not always straightforward and may require careful consideration. In fact, this task often presents itself as a common theme and yet rewarding challenge. So if a structure does not adhere to SIP, we likely have to reassess the definition of the structure or redefine its concept of equivalence.

Choosing the right definition of equivalence strongly depends on both the context and the symmetries that interest us, that is, how we perceive their sameness or similarity depends on the characteristics we aim to preserve. These may include, for example, some specific order or quantity of elements, or other attributes. In type theory, this translates to the type in which we identify-of. Consequently, as we introduce more distinctions to differentiate our objects, the definition of equivalence becomes more complex, breaking more symmetries.

Example 1.12. Consider the type of cyclic orders, lists up to rotations —a concept in combinatorics used later to define graph maps (Section 2.5 and Chapter 4). In a cyclic order, the elements can be arranged in a circle with a defined notion of *next* and *previous*. For example, in the cyclic order $[1, 2, 3]$, the next of 1 is 2, and the previous of 1 is 3, and so on.



In determining the appropriate equivalence for cyclic orders, we consider several potential options. Consider the following alternative possible definitions of equivalence of two cyclic orders.

1. The cyclic orders yield identical sets of elements.
2. The lists that underpin these cyclic orders exhibit pointwise equality.
3. The list elements are merely cyclic permutations of a common list.

Unsurprisingly, the initial two options violate the Structure Identity Principle. The first option neglects to recognise the cyclic order, focusing solely on elements without considering their interrelations and possibly repetetions of elements in the list. On the other hand, the second option is overly restrictive, expecting a static list of elements and ignoring any cyclic permutation.

The critical data in a cyclic order is precisely the circular arrangement of elements, a feature only captured by the last option. Therefore, despite presentation differences, two cyclic orders are considered equal if they correspond to identical lists up to a specific cyclic permutation of their elements. For instance, $[1, 2, 3]$ and $[2, 3, 1]$ are equivalent cyclic orders, whereas $[1, 2, 3]$ and $[1, 3, 2]$ are not.

1.2.2 The type of graphs and their symmetries

Say a graph is a term of type Graph, consisting of a set of elements termed nodes. Each node pair a and b is associated with a set whose elements, referred to as edges connect a and b. So, the term *graph* here refers to directed multigraphs. One can define such a type[10] in type theory as follows; N is a type of nodes, and a type family twice indexed by N, represents the edges between two nodes.

$$\mathsf{Graph} :\equiv \sum_{(\mathsf{N}\,:\,\mathcal{U})} (\mathsf{N} \to \mathsf{N} \to \mathcal{U}).$$

What does it mean for two graphs to be equal? The notion of the equivalence between graphs is embodied by what it is called graph isomorphism, a well-established concept defined as such: two graphs are considered equivalent if and only if there is a bijection between their node sets that preserves adjacency. Denoting graph isomorphism[11] as $(\cong_{\mathsf{Graph}})$, we show it adheres to SIP (see Theorem 3.7). This enables to convert any graph isomorphism into an equality in Graph. Given graphs G and H, we can define establish a canonical function idtoiso from $G =_{\mathsf{Graph}} H$ to $G \cong_{\mathsf{Graph}} H$ and prove that it is

[10]The type of graphs in the rest of the document is the set-level version as defined in Definition 3.1.
[11]See Equation (3.2–1).

an equivalence.

$$G = H \underset{\text{isotoeq}}{\overset{\text{idtoiso}}{\rightleftarrows}} G \cong H \quad (\simeq)$$

Remark 1.13. Under graph isomorphism, we can notice that all graphs in Figure 1.1 are equivalent. This follows because our Graph type only encapsulates node connections, disregarding node labels. If we aim to capture additional information like these labels or shapes on nodes and edges, a more complex identity type needs to be considered.

We could, for instance, define as in (1.2–2), L-Graph as the type of labelled graphs where each node is associated with a label of type L, and then, consider the labels part of the equality. This would allow us to distinguish between graph (III) and the rest of the graphs in Figure 1.1.

$$L\text{-Graph} :\equiv \sum_{(\mathsf{N}\,:\,\mathcal{U})} (\mathsf{N} \to \mathsf{N} \to \mathcal{U}) \times (\mathsf{N} \to L). \tag{1.2–2}$$

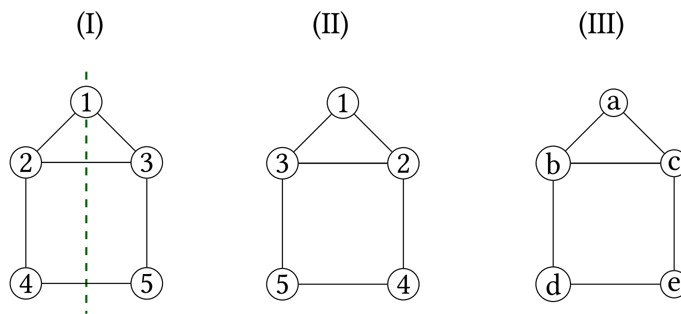


Figure 1.1: Drawing (I) represents *the house graph*, featuring its only symmetry line depicted in dark green. This graph consists of five nodes and six edges: $(1, 2)$, $(1, 3)$, $(2, 3)$, $(2, 4)$, $(3, 5)$, and $(4, 5)$. Drawing (II) demonstrates the reflection of the graph (I) along its symmetry, the vertical axis. And, in (III) we showcase a node relabelling of drawing (I).

The symmetries of a graph

The symmetries of a graph correspond to its identity type. Defined precisely by the ways it mirrors itself, these exact symmetries are the graph's isomorphisms to itself, also known as *automorphisms*. While the identity symmetry is always present, the challenge lies in discovering additional symmetries. The symmetries of an object forms precisely a group, called the *symmetry group* of an object. If we continue this road we can define the symmetries of a graph as a group, where the identity automorphism serves as the group's identity element and the composition of isomorphisms acts as the group operation. Frequently, visual representation of graphs facilitates the identification of these symmetries. For example, consider the symmetries of regular polygon as shown in Figure 1.2. These correspond to the dihedral group D_n of order $2n$, where n represents the number of poly-

gon sides. We list other similar examples in Section 3.8. Returning to our house graph example, we identify two symmetries depicted in Figure 1.1. The identity symmetry being the "do-nothing" action, while the other is a reflection along the vertical axis. As we look for symmetries in these visual representations, akin to geometry, we search for transformations that leave the object —in this case, the graph structure— invariant. Such operations include reflections and rotations.

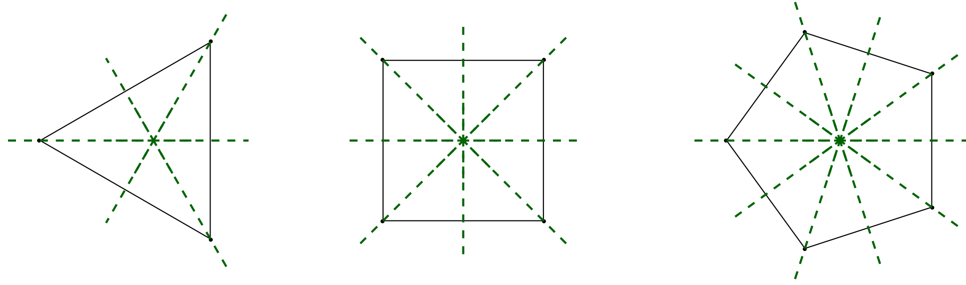


Figure 1.2: For regular polygons, symmetry lines identify the automorphisms of their underlying graphs. Each line corresponds to a reflection, rotation, or both, forming the dihedral group D_n of order $2n$. Here, n denotes the number of polygon sides. These transformations can be represented as permutations of the nodes. Take a square, a 4-sided polygon, as an example. It has 8 symmetries: 4 rotations and 4 reflections, which constitute the elements of D_4, a group of order $2 * 4 = 8$. A counterclockwise rotation of, say, 90 degrees, is represented by the permutation $(1\,2\,3\,4)$, while a vertical axis reflection is represented by $(1\,4)(2\,3)$. The numbers 1, 2, 3, and 4 are the square's nodes listed in counterclockwise order.

1.2.3 Drawing graphs on surfaces

Graphs can be depicted on various surfaces, including a two-dimensional plane, the 2-sphere, and the torus. Despite their visual appeal and widespread use in science, our focus is not on aesthetic elements such as edge lengths, angles, curvatures, or node placements. Our primary interest lies in graph drawings on closed, orientable surfaces where edges do not intersect and are equivalent under *isotopy*, that is, continuous deformation without crossing edges. While several surfaces could serve this purpose, we will mainly focus on the 2-sphere, which subsequently allows us to address the two-dimensional plane.

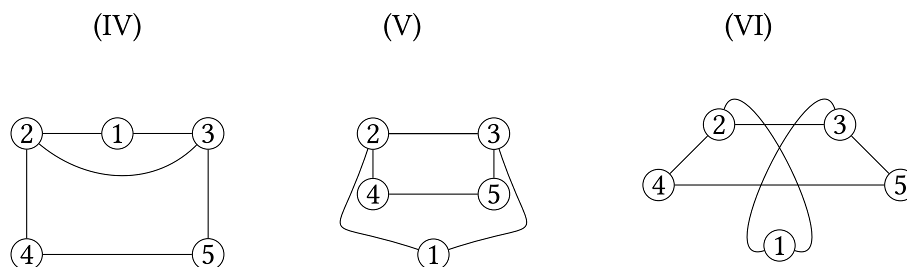


Figure 1.3: Different visual representations for the same graph map of the house graph given in Example 1.14. Note how the cyclic order of edges around each node is preserved consistently across all representations. The first two representations correspond to *drawings* —the result of planar maps for the house graph, while the last representation does not, as it features an edge crossing, so it is not an embedding.

In our exploration of graph drawings on surfaces, we must concentrate on the crucial data conveyed by their visual representations. Note that each node in a drawing, as illustrated in Example 1.14, displays a specific cyclic order of its connected edges. This order for each node is encapsulated within a combinatorial data structure known as a *rotation system* or *graph map*, widely employed in topological graph theory (Gross and Tucker 1987). These graph maps abstract the unnecessary visual aspects of drawings, focusing solely on the combinatorial structure. Consequently, we may depict the same graph in various ways, each giving rise to a potentially non-unique graph map.

1.2.4 The notion of graph maps and faces

A graph map $\mathscr{M}$ for a given graph G is a mapping from the nodes of G to their adjacent edges, arranged in a specific cyclic order. These are lists up to rotation, each associated with a node in G. Each order connects a node to its edges, as they appear on the surface. This notion is defined in type theory as in Definition 4.8. It is important to note that a graph map inherently defines a set of regions. As illustrated in Figure 1.4 for the case of the 2-sphere, these regions, when glued together, reconstruct the original drawing surface. Indeed, the surface's nature is implicitly encoded within the graph map, from which one can derive a property known as the surface's genus, which is the number of "holes" in the surface. For instance, the 2-sphere has a genus of 0, while the torus has a genus of 1.

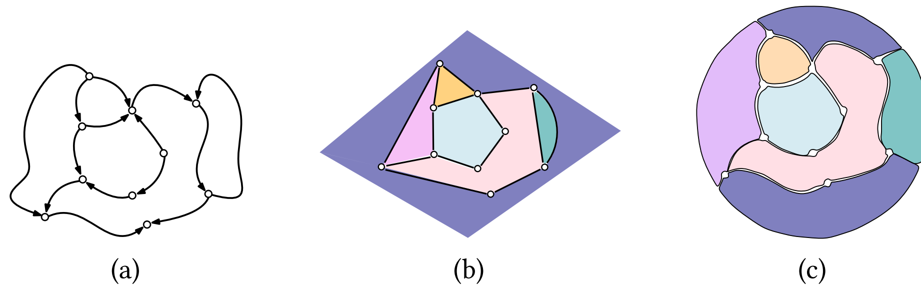
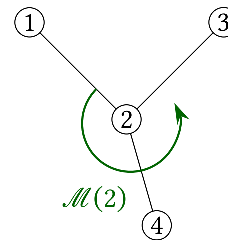


Figure 1.4: Figure (b) illustrates the plane-embedded graph from Figure (a), complete with color-coded faces derived from its representation. In Figure (c), these same faces are disassembled and used to construct the sphere. The part of the sphere not visible in (c) is the back of the sphere with purple color.

Let us now consider two examples: a graph map yielding the surface of a plane in Example 1.14, and another producing the surface of a torus in Example 1.15.

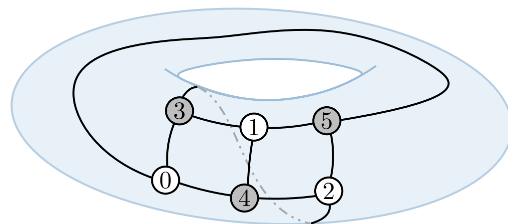
Example 1.14. Consider the house graph in Figure 1.1. The graph map assigns each node a counterclockwise cycle of adjacent nodes. For instance, the graph map $\mathscr{M}$ at node 2, i.e., $\mathscr{M}(2) :\equiv [1, 4, 3]$, signifies that edge $(2, 1)$ is succeeded by edge $(2, 4)$, and then $(2, 3)$.

Node	Adjacent nodes	Graph map
1	2, 3	[2, 3]
2	1, 3, 4	[1, 4, 3]
3	1, 2, 5	[1, 2, 5]
4	2, 5	[2, 5]
5	3, 4	[3, 4]



Example 1.15. Consider the complete bipartite graph $K_{3,3}$, a *forbidden minor* used in Kuratowski's planar graphs characterisation —embeddings of graphs in the plane. This graph comprises two sets of three nodes each: $0, 1, 2$ and $3, 4, 5$. Each node in one set connects to every other node in the opposing set. Drawing this graph on a plane inevitably leads to edge crossings. However, these issues disappear when drawn on a torus. We provide a graph map for $K_{3,3}$ and its torus embedding below.

Node	Adjacent nodes	Graph map
0	3, 4, 5	[3, 5, 4]
1	3, 4, 5	[3, 4, 5]
2	3, 4, 5	[4, 3, 5]
3	0, 1, 2	[2, 0, 1]
4	0, 1, 2	[1, 0, 2]
5	0, 1, 2	[1, 2, 0]



1.2.5 Planar drawings

A planar graph map, in essence, guides us how to draw a graph on a two-dimensional plane such that there are no edge crossings. Graphs that allow for these drawings are known as *planar graphs*. However, in our study of planar graphs, we circumvent any explicit reference of the two-dimensional plane (i.e., $\mathbb{R}^2$), thus avoiding the complexities[12] of working with real numbers and defining the edge-crossing property in HoTT. Instead, we simply concentrate on characterising the type of drawing that embeds graphs in the 2-sphere. Once the graph is embedded in the 2-sphere, the edge-crossing property comes for free.

We use the idea of a 2-sphere here as it serves as a model for the plane. Recall that the plane can be obtained by puncturing the 2-sphere at a point representing infinity. Therefore, drawing a graph in the plane is equivalent to embedding this into the 2-sphere, differentiated by the puncture point. Essential details, such as where to puncture the 2-sphere or the position of the infinity point, determine the distinguished face of the graph map, called the *outer face*.

Remark 1.16. The term 2-sphere in this context does not allude to the (higher inductive type), or the type of the 2-sphere in HoTT, it is consistently discussed within the framework of a graph map, serving as a plane model and the graph map's target. We do, however, mention the possible future work using the 2-sphere in HoTT in Section 7.1.

We introduce spherical maps, characterising graph maps that embed graphs in the 2-sphere, identified as the unique closed orientable simply connected surface. This requires us to introduce the notion of face of a map in HoTT and a property of simple connectedness for graph maps via walk homotopy. The intuition behind type of faces is discussed next, while detailed description of the latter topic is found in Chapter 5.

In the context of topology, a face is recognised as a region homeomorphic to the disk. The faces of graph map are obtained by extracting the embedded graph from the surface via the graph map. When all of its regions qualify as faces, the graph map is termed as *cellular*. The notion of spherical maps can be seen then as cellular maps subject to additional conditions.

Combinatorially, in HoTT, we characterise a face for a graph G based on a graph map $\mathscr{M}$. These faces are typically enclosed by edges, except by those representing unbounded regions, such as the outer face in planar maps. This definition involves a cyclic graph A, and a graph homomorphism f from A to an undirected variant of G. The morphism f cannot map distinct edges in A to the same edge in G. Furthermore, two consecutive edges in A must map to two consecutive edges in G, adhering to the order stipulated by the graph map $\mathscr{M}$. The exact definition of a face is provided in Definition 4.14.

[12] A similar discussion can be found in the formalisation of 2-connected planar graphs in HOL (Yamamoto, Nishizaki, Hagiya, et al. 1995).

We must pay attention to which type we characterise the identity type of a planar map because we might get a different notion of equivalence of graph maps that do not correspond to isotopy.

The data of a planar drawing of a fixed graph G consists of the choosing of a map and the outer face. Their respective type families are given by:

▷ $\mathsf{Map} : \mathsf{Graph} \to \mathcal{U}$ and

▷ $\mathsf{Face} : \prod_{G : \mathsf{Graph}} (\mathsf{Map}(G) \to \mathcal{U})$.

Then, a planar drawing, denoted as the pair (m, o), consists of a graph map m for a graph G, possessing certain additional properties, and a distinguished face o.

Remark 1.17. In the context of a fixed graph G, we consider two planar drawings, (m_1, o_1) and (m_2, o_2). These can be compared using one of the following types.

1. The identity type $(m_1, o_1) = (m_2, o_2)$ in the type $\sum_{x : \mathsf{Map}(G)} \mathsf{Face}(G, x)$, which corresponds to having labeled the graph.
2. The identity type $(G, m_1, o_1) = (G, m_2, o_2)$ in the type $\sum_{H : \mathsf{Graph}} \sum_{x : \mathsf{Map}(H)} \mathsf{Face}(H, x)$, allowing us to disregard the labeling on G for broader identification but fewer drawings.

Example 1.18. Consider planar drawings (IV) and (V) of the house graph, denoted in Figure 1.3 as $p_1 :\equiv (m, o_2)$ and $p_2 :\equiv (m, o_2)$. Despite sharing the same underlying graph map as shown in Example 1.14, the outer faces differ; o_1 is defined by edges 2-1, 1-3, 3-5, 5-4 and 4-2, whereas o_2 is determined by edges 1-2, 2-3, and 3-5. Thus, we deduce $p_1 \neq p_2$ in $\sum_{x : \mathsf{Map}(G)} \mathsf{Face}(G, x)$.

Example 1.19. More subtle is the case of planar drawings (I) and (III) in Figure 1.1 given by graph maps m_1 and m_2. Initial observation may suggest they are equal, however, as per Remark 1.17, our equivalence definition for planar drawings fixes the graph for the graph map (Item 1), and thus differentiates outer faces; $o_1 :\equiv$ 1-2-4-5-3 for (I), and $o_2 :\equiv$ *a*-*b*-*d*-*e*-*c* for (III).

If there were grounds to establish equivalence for these drawings, that is admitting node relabeling, we would need to consider identity type in $\sum_{H : \mathsf{Graph}} \sum_{x : \mathsf{Map}(H)} \mathsf{Face}(H, x)$ as per Item 2 in Remark 1.17. This would allow us to show that $(G, m_1, o_1) = (H, m_2, o_2)$ under the mapping $1 \mapsto a$, $2 \mapsto b$, $3 \mapsto c$, $4 \mapsto d$, and $5 \mapsto e$.

N_G	m_1
1	$[2, 3]$
2	$[1, 4, 3]$
3	$[1, 2, 5]$
4	$[2, 5]$
5	$[3, 4]$

N_H	m_2
a	$[b, c]$
b	$[a, c, d]$
c	$[a, e, b]$
d	$[b, e]$
e	$[c, d]$

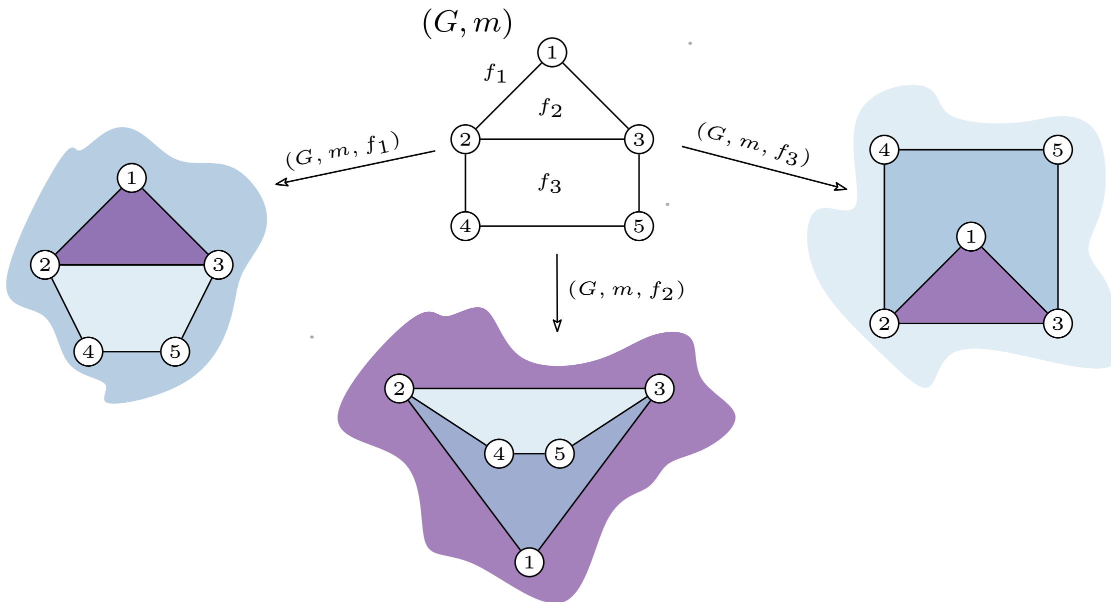


Figure 1.5: The house graph G features six unique planar drawings, split evenly between the two graph maps (I) and (II) as shown in Figure 1.1. We illustrate one of these graph maps, m of G, as in (I), also cited in Example 1.14. The three distinct planar drawings (G, m, f_i) for m are presented. Each drawing corresponds to an individually selected outer face: f_1, f_2, and f_3. These faces, enclosed by a pentagon, triangle, and rectangle respectively, are differentiated by distinct shading. The unbounded region of the plane, represented as a splashed area, denotes the outer face in each planar drawing.

To summarise, our characterisation of planarity in HoTT is founded on several key insights. Although planarity of graphs is typically defined as an inherent property of graphs, we consider it as a structure within the category of graphs, providing a notion of planar direct multigraph with an identity principle. This perspective stems from the intuitive idea that a proof establishing a graph's planarity should correspond to a witness graph map that embeds the graph into the plane, including a designated outer face as previously mentioned and illustrated in Section 1.2.5. Additionally, the equivalence of planar graph maps utilised in this context builds upon the concept of isotopy for graph maps.

Nevertheless, in order to formulate a type of planar graph maps where the identity type coincides with isotopy, careful consideration must be given to the definitions of graph maps and planarity. The essential information required to differentiate between drawings is found to be:

- ▷ The graph consisting of nodes and edges,
- ▷ the graph map (combinatorial map of the graph) into the sphere, and
- ▷ the outer face of the graph map.

This thesis comprehensively explores the construction of types for graphs, graph maps, faces, and ultimately, planar maps. Before giving an overview of the thesis, we discuss another aspect of this work, the formalisation of mathematics aid by computer proof assistants.

1.3 Formalisation of mathematics

Dependently typed theories, such as HoTT, allow for the expression of mathematical concepts with an appropriate level of abstraction. By utilising type theory in the formalisation of mathematics, concepts can be articulated using a precise and detailed language as in programming. In fact, MLTT is one foundation of many modern high-level general purpose programming languages such as Agda (The Agda Development Team 2023), Coq (The Coq Development Team 2021), Idris (Brady 2013), and the recent versions of Lean (Moura, Kong, Avigad, et al. 2015).

This means, theories such as dependently type languages and higher other logics, among other formalisms, can enable computers to mechanically verify the correctness of the mathematical proofs more efficiently than traditional methods, where the correctness of the proofs is validated by a group of experts. Same as on paper, these formal developments are susceptible of errors in their theoretical formulation, and possible in computer implementation.

A proof assistant can be employed to take advantage of the rigour provided by this mathematical approach. In essence, a proof assistant is a computer system that offers multiple modes of operation. These modes can be used to create programs that meet specific requirements, streamline the proof-writing process within a formal type system, and ensure the correctness of mathematical proofs.

The formalisation of mathematics, especially through computer proof assistants supporting dependent types, presents numerous advantages compared to the traditional method of writing proofs in natural language. Let us list some advantages of formalising mathematics from our perspective without adhering to a specific order.

First, *reliability*. By formalising mathematics as discussed in Appendix A, we obtain machine-checked proofs. The mathematical correctness of these proofs is guaranteed by the formal system and the correctness of the proof-checker implementation. Traditional written proofs are prone to errors, including unintentional mistakes such as typos and omissions.

Second, *accountability/reusability*. Formal developments provide independently verified and highly accessible mathematical resources compared to documents in prose, making it easier to share, modify, and extend the mathematical content.

Third, *reproducibility*. Formal developments as discussed here possess transparency

and reproducibility. To replicate the reasoning steps and the outcome, one requires access to both the formalisation and the proof assistant's software specifications.

Finally, employing proof assistants in mathematical writing not only provides a high level of rigour and reliability but also fosters the discovery of new objects, including enhanced proofs and theorems. The ability to mechanically verify the correctness of these constructs increases the likelihood of revealing previously undiscovered relationships and insights within the mathematical domain of interest (Avigad and Harrison 2014).

1.4 Formalisation of graph-theoretical concepts

In the context of formalising mathematics, there are numerous developments of graph theory available in different type systems presented among the most popular proof assistants are Agda, Coq, Isabelle/HOL, and Lean (Moura, Kong, Avigad, et al. 2015). The type system of Agda is an extension of Martin-Löf's intuitionistic type theory. On the other hand, both Coq and Lean utilise the Calculus of Inductive Constructions for their type systems. Isabelle/HOL, however, employs higher-order logic as its basis.

Notably, related to graph theory, significant projects and extensive libraries have been developed in the proof assistants Coq (Doczkal and Pous 2020) and Isabelle/HOL (Noschinski 2015). Among these, prominent projects are Gonthier's well-known formal proof of the Four-Colour theorem (FCT) in Coq (Gonthier 2008), Dufourd's proof of the discrete form of the Jordan Curve theorem also in Coq (J.-F. Dufourd and Puitg 2000), and the proof of Kepler's conjecture in HOL by Bauer et al. (Hales, Adams, G. Bauer, et al. 2017). More recently, several other libraries have emerged, including the Coq Graph Library[13], the Isabelle Graph Theory Library (Noschinski 2014), the Lean Mathlib library, the combinatorics section[14], and most recently, the Agda-UniMath library (Rijke et al. 2023).

Specifically to the formalisation of planarity of graphs different methods have been proposed, each founded on mathematical foundations that differ from HoTT, leading to the use of distinct mathematical objects than those discussed in this document. In this work, we employ graph maps to define the concept of planarity. Alternative approaches involve related constructions, including root maps defined in terms of permutations by Dubois et al. (Dubois, Giorgetti, and Genestier 2016), and the notion of hypermaps used by Dufourd et al., and Gonthier (J. F. Dufourd 2009; J.-F. Dufourd and Puitg 2000; Gonthier 2008), among others. The notion of hypermaps, an ad-hoc generalisation of combinatorial maps tailored for undirected finite graphs, serves as a vital component in formalising graph embedding mathematics within theorem provers. This concept has been effectively employed in computer-checked proofs of FCT (Doczkal 2021). Additionally, Du-

[13] https://github.com/coq-community/graph-theory.
[14] https://leanprover-community.github.io/mathlib-overview.html#combinatorics

fourd states and proves Euler's polyhedral formula and the Jordan Curve theorem using an inductive characterisation of hypermaps (J. F. Dufourd 2009; J.-F. Dufourd and Puitg 2000). Doczkal, using a more conventional representation of finite graphs, demonstrates that every $K_{3,3}$-free graph and K_5-free graph without isolating nodes is planar. This is in accordance with his concept of a plane map, which is founded on hypermaps. Doczkal's result corresponds to one direction in the statement of Wagner's theorem (Doczkal 2021).

An alternative approach to address planarity in a type-theoretical way without combinatorial maps is through iterative procedures. For instance, Yamamoto et al. (Yamamoto, Nishizaki, Hagiya, et al. 1995) demonstrated that every finite and biconnected planar graph can be decomposed into a finite collection of cycle graphs, with each face being the region enclosed by a closed walk, also referred to as a *circuit* (Gross and Anderson 2018, §5.2, §7.3). This construction defines an inductive data type that begins with a cycle graph C_n serving as the base case, and by repeatedly merging new instances of cycle graphs, one gets the final planar graph. Bauer formalises a similar construction of planar graphs from a set of faces in Isabelle/HOL (G. Bauer and Nipkow 2002; G. J. Bauer 2005). The approach described in Section 6.3 for handling planar extension is related to this iterative procedure.

1.5 Short outline of this thesis

In this work, we propose a new approach to graph planarity in HoTT. Our method aligns with abstract mathematical intuition, unlike traditional analytic or geometric methods that use the two-dimensional plane (i.e., $\mathbb{R}^2$) to describe this concept.

In our quest to address this topic, we begin with Chapter 2, laying out the mathematical foundation, terminology, notation, and basic constructions. Those familiar with HoTT might opt to skip this chapter, except for Section 2.5, which explains cycle types.

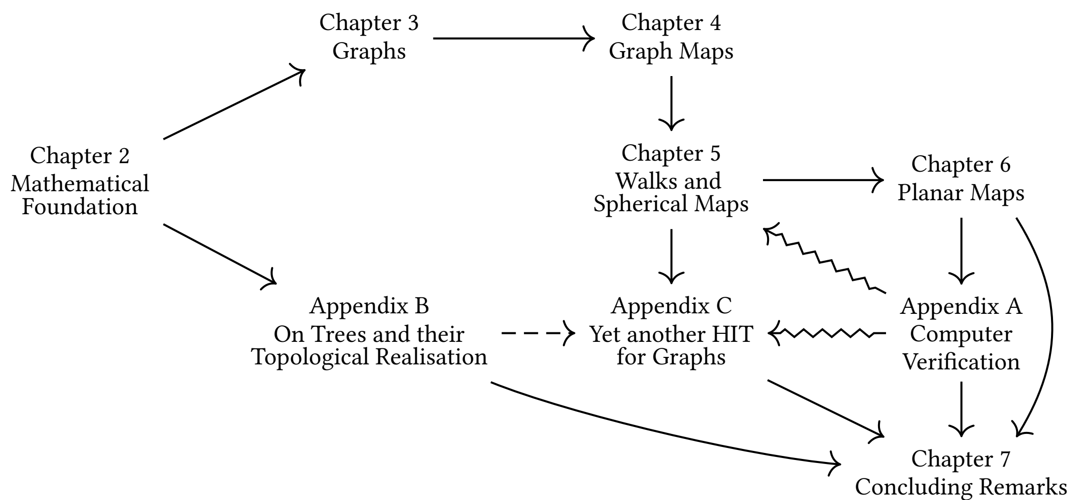
Next, we delve into the univalent category of directed multigraphs in Chapter 3, exploring graph homomorphisms, properties, structures, and specific examples and families of graphs. Later, we focus on graph maps and the notion of faces of a map in Chapter 4.

In Section 5.4, we introduce the types of walks and quasi-simple walks, presenting a normal form for walks, a normalisation procedure, and the notion of walk homotopy. The content here is significant because it serves as the foundation for our characterisation of planarity. This characterisation employs spherical maps, and its definition can be refined by using the normal form of walks in graphs with discrete node sets.

Finally, our characterisation of graph planarity in HoTT is presented in Chapter 6 built on top of the aforementioned concepts. For constructing examples of planar graphs, we present an inductive method for extending planar graphs in Section 6.3.

We summarise our findings and outline future work in Chapter 7, supplemented by

Figure 1.6: The solid arrows indicate that the starting point is a prerequisite for the ending point. The squiggly arrows indicate that the starting point influences the ending point. Finally, the dashed arrows establish a relationship not formally established in the text.



the constructions in Appendices B and C.

Suggested reading order for this document can be found in Figure 1.6.

Related publications

The work presented in this thesis is based on the following manuscripts and the Agda formalisation presented in Appendix A.

Prieto-Cubides, Jonathan (2022). On Homotopy of Walks and Spherical Maps in Homotopy Type Theory. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 338–351. URL: https://doi.org/10.1145/3497775.3503671.

Prieto-Cubides, Jonathan and Håkon Robbestad Gylterud (2022). On Planarity of Graphs in Homotopy Type Theory. Submitted, *Mathematical Structures in Computer Science*. URL: https://arxiv.org/abs/2112.06633.

"All animals are equal, but some animals are more equal than others."

George Orwell, *Animal Farm.*

2
Mathematical Foundations

In this thesis, we work with homotopy type theory, a Martin-Löf intensional intuitionistic type theory extended with the Univalence Axiom (Awodey 2018; Escardó 2018; Univalent Foundations Program 2013), proposed originally by Voevodsky (Voevodsky 2010), and some higher inductive types (HITs), such as propositional truncation. The presentation of our constructions is informal, in a similar style as in the HoTT Book (Univalent Foundations Program 2013).

HoTT emphasises the role of the identity type as a path-type. The intended interpretation is that elements, $a, a' : A$, are *points* and that a witness of an equality $p : a = a'$ is a *path* from a to a' in A, as illustrated in Figure 2.1. Since the identity type is again a type, we can iterate the process, which gives each type the structure of an ∞-groupoid (Awodey 2012).

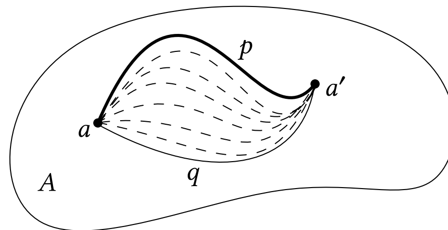


Figure 2.1: This figure shows a representation of the homotopy between two paths p, q of the identity type $a = a'$ in a type A.

This may at first seem of little relevance when working with finite combinatorics, as

one would expect only types with trivial path-types (sets) to show up in combinatorics. However, we will see that types with nontrivial path types do arise naturally in combinatorics, which should come as no surprise to anyone familiar with the role of groups and groupoids in this field, such as Joyal's work on combinatorial species (Baez, Hoffnung, and Walker 2009-08; Yorgey 2014) —and that the paths in these types are often various forms of permutations.

2.1 Notation

An informal type theoretical notation derived from the HoTT book (Univalent Foundations Program 2013) and the formal system Agda (Norrell 2007) is used throughout this paper. The following list summarises the most important conventions and notations used in this paper.

- ▷ Definitions are introduced by $(:\equiv)$, while judgemental equalities use $(\equiv)$.
- ▷ The type $\mathcal{U}$ is a *univalent* universe.
- ▷ The notation $A : \mathcal{U}$ indicates that A is a type. A term a of type A is denoted by $a : A$ and A is referred to as a type *inhabited*.
- ▷ The equality sign of the identity type of A is denoted by $(=_A)$. The constructor of the identity type $x =_A x$ is denoted by $\mathsf{relf}(x)$ for $x : A$. If the type A can be inferred from the context, we simply write $(=)$. The equalities between $x, y : A$ are of type $x = y$.
- ▷ The type of non-dependent functions between A and B is denoted by $A \to B$.
- ▷ Type equivalences are denoted by $(\simeq)$. The canonical map for types is the function $\mathsf{idToEquiv}$ of type $A = B \to A \simeq B$ and its inverse function is called ua. Given the equivalence $e : A \simeq B$, the application, $\mathsf{ua}(e)$ is denoted by $\overline{e}$, while the underlying function of the equivalence e of type $A \to B$ can be also denoted by e. Moreover, the coercion along a path $p : A = B$ is the function denoted by $\mathsf{coe}(p)$ of type $A \to B$.
- ▷ The point-wise equality for functions (also known as *homotopy*) is denoted by $(\sim)$. The function happly is of type $f = g \to f \sim g$ and its inverse function is called funext.
- ▷ The co-product of two types A and B is denoted by $A + B$. The corresponding data constructors are the functions $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$.
- ▷ Dependent product types (Π-types) are denoted by $\Pi_{x:A}B(x)$ for a type A and a type family $B : A \to \mathcal{U}$, while dependent sum types (Σ-types) are denoted by $\Sigma_{x:A}B(x)$.

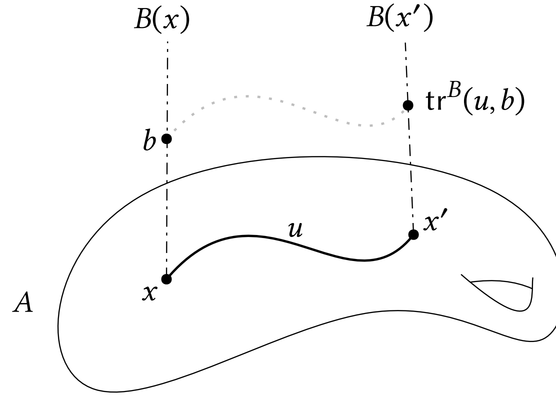


Figure 2.2: The figure shows the representation of two points, b, and $\mathrm{tr}^B(p, b)$, in the fibres of a type family B over the points x, x' in $A : \mathcal{U}$, respectively, where $\mathrm{tr}^B(p, b)$ denotes the transport of b along the path $p : x = x'$.

If $x : A$ and $y : B(x)$, then the pair (x, y) is of type $\Sigma_{x:A}B(x)$. The corresponding projection functions for a pair are denoted by π_1 and π_2, so that $\pi_1(x, y) :\equiv x$ and $\pi_2(x, y) :\equiv y$. If the type family B over A is constant, then we may denote the type $\Sigma_{x:A}B(x)$ by $A \times B$, and the $\Pi_{x:A}B(x)$ by $A \to B$.

- The empty type and the unit type are denoted by $\mathbb{0}$ and $\mathbb{1}$, respectively.
- The type $x \neq y$ denotes the function type $(x = y) \to \mathbb{0}$.
- Natural numbers are of type $\mathbb{N}$. $0 : \mathbb{N}$. The successor of $n : \mathbb{N}$ is denoted by $S(n)$ or $n + 1$. The variable n is of type $\mathbb{N}$, unless stated otherwise.
- Given $n : \mathbb{N}$, the standard type with n elements is denoted by $[\![n]\!]$.
- The universe $\mathcal{U}$ closed under the type formers considered above.
- The function transport/substitution is denoted by tr of type $\Pi_{u:x=x'}B(x) \to B(x')$, where $x, x' : A$ and $B : A \to \mathcal{U}$. Furthermore, we denote by tr_2 the function of type $\Pi_{p:a_1=a_2} \mathrm{tr}^B(p, b_1) = b_2 \to C(a_1, b_1) \to C(a_2, b_2)$, where the type family B is indexed by the type A, $a_1, a_2 : A$, $b_1 : B(a_1)$, $b_2 : B(a_2)$, and the type C is of type $\Pi_{x:A}\,(B(x) \to \mathcal{U})$.

In the next sections, we will use variables A, B and X to denote types, unless stated otherwise. To define some inductive types, we adopt a similar notation as in Agda, including the keyword data and the curly braces for implicit arguments, e.g., $\{a : A\}$ denotes a is of type A, and it is an implicit variable. The type may be omitted in the former notation, as they can usually be inferred from the context.

2.2 Homotopy levels

The following establishes a level hierarchy for types with respect to the nontrivial homotopy structure of the identity type.

Definition 2.1. Let n be an integer such that $n \geq -2$. One states that a type A is an *n-type* and that it has homotopy level n if the type is-level(n, A) is inhabited.

$$\text{is-level}(-2, A) :\equiv \sum_{(c\,:\,A)} \prod_{(x\,:\,A)} (c = x),$$
$$\text{is-level}(n+1, A) :\equiv \prod_{(x,y\,:\,A)} \text{is-level}(n, x = y).$$

For this document, the first four homotopy levels are enough to express the mathematical objects we want to construct. They are referred to in order, starting from -2, as contractible types, propositions, sets, and groupoids. For convenience, we use the following predicates:

- ▷ $\text{isContr}(A) :\equiv \text{is-level}(-2, A)$,
- ▷ $\text{isProp}(A) :\equiv \text{is-level}(-1, A)$,
- ▷ $\text{isSet}(A) :\equiv \text{is-level}(0, A)$, and
- ▷ $\text{isGroupoid}(A) :\equiv \text{is-level}(1, A)$.

Types that are propositions are of type hProp and similarly with the other levels. If A is an inhabited proposition, then we say that A *holds.* Additionally, it is possible to have an n-type out of any type A for $n \geq -2$. This can be done using the construction of a higher inductive type called n-truncation (Univalent Foundations Program 2013, §7.3) denoted by $\|A\|_n$. The case for (-1)-truncation is called *propositional truncation* (or *reflection*), and is often simply denoted by $\|A\|$.

Definition 2.2. *Propositional truncation* of a type A denoted by $\|A\|_{-1}$ is the *universal solution* to the problem of mapping A to a proposition P. The elimination principle of this construction gives rise to a map of type $\|A\| \to P$, which requires a map $f : A \to P$ and a proof that P is a proposition.

Propositional truncation allows us to model the *mere* existence of inhabitants of type A. We state that x is *merely* equal to y when $\|x = y\|$ for $x, y : A$. Then, we can express in HoTT by means of propositional truncation:

- ▷ logical conjunction $(P \vee Q) :\equiv \|P + Q\|$,

- ▷ logical disjunction $(P \wedge Q) :\equiv \|P \times Q\|$,
- ▷ logical quantification $(\forall(x : A)P(x)) :\equiv \|\Pi_{x:A} Px\|$,
- ▷ logical existential $(\exists(x : A)P(x)) :\equiv \|\Sigma_{x:A} Px\|$, and
- ▷ unique existence $(\exists!(x : A)P(x)) :\equiv \mathsf{isContr}(\Sigma_{x:A} P(x))$.

For clarity, let us define some conventions and constructions that will be useful in subsequent discussions. Let A and B be types. A type A is referred to as *inhabited* if we have a term a of type A. If $\|A\|$ is inhabited, then we say that type A is *nonempty*.

Lemma 2.3. If a type A is a proposition and A is inhabited, then A is a contractible type. In this case, we say that A *holds*.

Definition 2.4. A function $f : A \to B$ is called an *equivalence* if all its fibers are contractible, i.e.,

$$\mathsf{isEquiv}(f) :\equiv \prod_{(x:b)} \mathsf{isContr}(\mathsf{fib}_f(b))$$

where

$$\mathsf{fib}_f(b) :\equiv \sum_{(x:A)} \|f(x) = b\|.$$

Definition 2.5. A function $f : A \to B$ is called an *embedding* if the type $\mathsf{isEmbedding}(f)$ is inhabited,

$$\mathsf{isEmbedding}(f) :\equiv \prod_{(x,y:A)} \mathsf{isEquiv}(\mathsf{ap}_f(x, y))$$

where $\mathsf{ap}_f(x, y)$ (sometimes refers to as cong) is the function that maps $x = y$ to $f(x) = f(y)$ for $x, y : A$.

Definition 2.6. Given $x : A$, the *connected component* of x in A is the type $\Sigma_{y:A}\|y = x\|$.

Definition 2.7. The type A is called *connected* if $\|A\|$ holds and each $x : A$ belongs to the same connected component.

Lemma 2.8. Let $P : A \to \mathsf{hProp}$ and $x, y : A$. If $\|y = x\|$, then $P(x) \simeq P(y)$. Thus, terms in the same connected component share the same propositional properties.

2.3 Handy equivalences

In the following chapters, a few calculations/chain of equivalences are presented, in which the following equivalences are used. We present them here for the sake of completeness.

Let $A : \mathcal{U}$, $B : \mathbb{1} \to \mathcal{U}$.

$$\sum_{(a:\mathbb{1})} B(a) \simeq \prod_{(a:\mathbb{1})} B(a) \simeq B(*). \tag{2.3–1}$$

$$\sum_{(a:A)} (a = x) \simeq \sum_{(a:A)} (x = a) \simeq \mathbb{1}. \tag{2.3–2a}$$

$$\sum_{(a:A)} \mathbb{1} \simeq (A \times \mathbb{1}) \simeq A. \tag{2.3–2b}$$

$$\sum_{(x:\mathbb{0})} A \simeq (A \times \mathbb{0}) \simeq \mathbb{0}. \tag{2.3–2c}$$

$$\prod_{(a:A)} \mathbb{1} \simeq \prod_{(x:\mathbb{0})} A \simeq \mathbb{1}. \tag{2.3–2d}$$

Let $A : \mathcal{U}$, $B, C : A \to \mathcal{U}$.

$$\left(\prod_{(x:A)} B(x) \simeq C(x)\right) \to \left(\prod_{(x:A)} B(x) \simeq \prod_{(y:A)} C(y)\right). \tag{2.3–3}$$

$$\left(\prod_{(x:A)} B(x) \simeq C(x)\right) \to \left(\sum_{(x:A)} B(x) \simeq \sum_{(y:A)} C(y)\right). \tag{2.3–4}$$

Let $A, B : \mathcal{U}$, $C : A \to \mathcal{U}$, $D : B \to \mathcal{U}$.

$$\prod_{(e:B\simeq A)} \left(\sum_{(x:A)} C(x) \simeq \sum_{(y:B)} C(e(y))\right). \tag{2.3–5}$$

$$\left(\prod_{(e:A\simeq B)} \prod_{(x:A)} C(x) \simeq D(e(x))\right) \to \left(\sum_{(x:A)} C(x) \simeq \sum_{(y:B)} D(y)\right). \tag{2.3–6}$$

Let $A : \mathcal{U}$, $B, C : A \to \mathcal{U}$, $D : \Sigma_{(a,b):\Sigma_{x:A} B(x)} C(a) \to \mathcal{U}$.

$$\sum_{((a,b):\sum_{(x:A)} B(x))} \sum_{(c:C(a))} D((a,b),c) \simeq \sum_{((a,c):\sum_{(x:A)} C(x))} \sum_{(b:B(a))} D(((a,b),c)). \tag{2.3–7}$$

$$(A \to \mathcal{U}) \simeq \sum_{(P:\mathcal{U})} (P \to A). \tag{2.3–8}$$

Let $A : \mathcal{U}$, $B : A \to \mathcal{U}$, and $C : (\Sigma_{x:A} B(x)) \to \mathcal{U}$.

$$\sum_{(a:A)} \sum_{(b:B(a))} C((a,b)) \simeq \sum_{\left((a,b):\sum_{(x:A)} B(x)\right)} C((a,b)). \tag{2.3–9}$$

$$\prod_{(a:A)} \prod_{(b:B(a))} C((a,b)) \simeq \prod_{\left((a,b):\sum_{(x:A)} B(x)\right)} C((a,b)). \tag{2.3–10}$$

$$\prod_{(a:A)} \sum_{(b:B(a))} C((a,b)) \simeq \sum_{\left(f:\prod_{(x:A)} B(x)\right)} \prod_{(a:A)} C((a,f(a))). \tag{2.3–11}$$

Let $P : \mathbb{N} \to \mathcal{U}$.

$$\sum_{(n:\mathbb{N})} P(n) \simeq P(0) + \sum_{(n:\mathbb{N})} P(n+1). \tag{2.3–12}$$

$$\prod_{(n:\mathbb{N})} P(n) \simeq P(0) \times \prod_{(n:\mathbb{N})} P(n+1). \tag{2.3–13}$$

$$[\![n+1]\!] \simeq [\![n]\!] + 1. \tag{2.3–14}$$

Let $A : \mathcal{U}$, $B, C : A \to \mathcal{U}$.

$$\sum_{(x:A)} B(x) + C(x) \simeq \sum_{(x:A)} B(x) + \sum_{(x:A)} C(x). \tag{2.3–15}$$

Let $A, B : \mathcal{U}$, $C : A + B \to \mathcal{U}$.

$$\sum_{(x:A+B)} C(x) \simeq \left(\sum_{(x:A)} C(\mathsf{inl}(x))\right) + \left(\sum_{(x:B)} C(\mathsf{inr}(x))\right). \tag{2.3–16}$$

$$\prod_{(x:A+B)} C(x) \simeq \left(\prod_{(x:A)} C(\mathsf{inl}(x))\right) \times \left(\prod_{(x:B)} C(\mathsf{inr}(x))\right). \tag{2.3–17}$$

2.4 Finite types

In the following we make precise the intuition that a type is finite when it is equivalent to $[\![n]\!]$ for some $n : \mathbb{N}$. The type $[\![n]\!]$ is the standard type with n elements, which can be defined as the following Σ-type.

$$[\![n]\!] \; :\equiv \sum_{(m\,:\,\mathbb{N})} m < n, \tag{2.4–18}$$

where the binary relation $(<)$ can be defined by cases, that is, $0 < m + 1$ for all m and for all n if $m < n$ then $m + 1 < n + 1$.

Definition 2.9. A type X is *finite* if the type $\mathsf{isFinite}(X)$ in (2.4–19) is inhabited.

$$\mathsf{isFinite}(X) \; :\equiv \sum_{(n\,:\,\mathbb{N})} \| X \simeq [\![n]\!] \| . \tag{2.4–19}$$

The finiteness of a type A is the existence of a bijection between A and the type $[\![n]\!]$ for some $n : \mathbb{N}$. However, this description is not a structure on A, which provides it with a specific equivalence $A \simeq [\![n]\!]$, but rather a property, a mere proposition. This ensures that the identity type on the total type of finite types is free to permute the elements, without having to respect a chosen equivalence.

Lemma 2.10. The type $\mathsf{isFinite}(X)$ is a proposition.

Proof. Let $(n, p), (m, q) : \mathsf{isFinite}(X)$, which we want to prove equal. Since p and q are elements of a family of propositions, it is sufficient to show that $n = m$. This equation is a proposition, so we can apply the truncation-elimination principle to get $X \simeq [\![n]\!]$ and $X \simeq [\![m]\!]$. Thus, from $[\![n]\!] \simeq [\![m]\!]$ follows that $n = m$ by a well-known result on finite sets. □

A type X is considered to be *finite* if the proposition $\mathsf{isFinite}(X)$ holds. The natural number n is referred to as the cardinal number of X, which is also denoted by $\#X$. If X and Y are finite and the identity type $X = Y$ is inhabited, then both types have the same cardinal number and Y is a permutation of X. Furthermore, Definition 2.9 is equivalent to the type $\exists_{n\,:\,\mathbb{N}}(X = [\![n]\!])$. However, the former definition makes it easier to obtain the cardinal number n by projecting on the first coordinate. This is more practical for certain proofs, such as Lemma 2.25. Additionally, any property of $[\![n]\!]$, like "being a set" and "being discrete" can be transferred to any finite type.

Theorem 2.11 (Hedberg's theorem). Any type A with decidable equality, i.e., $x = y + x \neq y$ for all $x, y : A$, is a set. Types like A are below refer to as *discrete* sets.

The following lemma found in Rijke's book, Theorem 16.3.6 item (iii) has proven to be useful for the finiteness property of types appearing, e.g., in Section 4.4.1. A proof of this result can be found in the Agda-UniMath library.

Lemma 2.12. Let B be a family of a type A. Consider the following propositions.

(a) A is finite.

(b) $B(x)$ is finite for each $x : A$.

(c) $\Sigma_{x:A}B(x)$ is finite.

The following holds:

1. If (a) holds, then (b) holds if and only if (c) holds.
2. If (b) and (c), then (a) holds if and only if A is a set and $\Sigma_{x:A}\neg B(x)$ is finite.
3. If (b) and (c) hold and there is a function of type $\Pi_{x:A}B(x)$, then (a) holds.
4. If (a) and (b) hold, then the type $\Pi_{x:A}\, B(x)$ is finite.

Lemma 2.13. Finite sets are closed under (co) products, type equivalences, Σ-types, and Π-types.

The formal proof of Lemma 2.13 and other related lemmas can be found in the Coq-HoTT library (A. Bauer, J. Gross, Lumsdaine, et al. 2017). For example, one of these lemmas, used to demonstrate Lemma 5.24, states that the cardinality of X is less than or equal to the cardinality of Y if there exists an embedding from X to Y.

Lemma 2.14. If A is finite, then $\|A\|$ is finite.

Proof. We start by assuming that A is finite and determine if its cardinality n equals zero. Regardless of the outcome, we can obtain an equivalence, $A \simeq [\![n]\!]$, from applying the propositional truncation elimination to the proposition $\|A\|$ is finite. We therefore can establish an equivalence between $\|A\|$ and either $\mathbb{0}$ or $\mathbb{1}$, depending on whether n equals zero. In both cases, these are finite types. Therefore, applying Lemma 2.13, we conclude that $\|A\|$ is finite. □

As the very first examples of finite sets, we have the empty type, unit type, decidable propositions, and the family of standard finite types $[\![n]\!]$. To prove the finiteness of other types, as in Theorem 5.26, we use Corollary 2.15, a direct consequence of Hedberg's theorem and the finiteness of the empty and unit type.

Corollary 2.15. If A is a discrete set, then the identity type $x = y$ is a finite set for all $x, y : A$.

Lemma 2.16. If A is finite, then the identity type $x = y$ is finite for all $x, y : A$.

Lemma 2.17. Let Y be a finite type, then the following type is finite.

$$\sum_{(X\,:\,\mathcal{U})}\left(\mathsf{isFinite}(X)\times\sum_{(f\,:\,X\to Y)}\mathsf{isInjective}(f)\right), \tag{2.4–20}$$

where

$$\mathsf{isInjective}(f) :\equiv \prod_{(x,y\,:\,X)} f(x)=f(y)\to x=y. \tag{2.4–21}$$

Lemma 2.18. If there is an injective function from set A to set B, and both A and B are finite, then the number of elements in A is less than or equal to the number of elements in B.

Corollary 2.19. Let $n : \mathbb{N}$. The following type is finite.

$$\sum_{(X\,:\,\mathcal{U})}\ \sum_{((\#X,!)\,:\,\mathsf{isFinite}(X))} \#X\le n. \tag{2.4–22}$$

We will now introduce cyclic types, which will be used later to characterise graphs embedded in a surface combinatorially in Definition 4.8.

2.5 Cyclic types

We want to define a notion of *cyclic type* to capture the idea of a finite type together with a premutation within orbiting freely over the whole type. To do so, we use the pred function which generates a cyclic subgroup (of order n) of the group of permutations on $[\![n]\!]$. An equivalent cyclic subgroup can be defined by means of the suc function, where the function suc is the inverse of pred.

Definition 2.20. Let pred be a function from $[\![n+1]\!]$ to itself defined by induction on n and the following equations. If $n=0$, then pred is the trivial function. If $n>0$, then,

$$\begin{aligned}
&\mathsf{pred} : [\![n+1]\!]\to[\![n+1]\!].\\
&\mathsf{pred}((0,!)) :\equiv (n,p).\\
&\mathsf{pred}((m+1,q)) :\equiv (m,r).
\end{aligned}$$

Where p is a proof that $n<n+1$ and r is a proof that $m<n+1$ using q, which is a proof that $m+1<n+1$.

Definition 2.21. Cyclic(A) defines the type of cyclic structures on type A.

$$\mathrm{Cyclic}(A) :\equiv \sum_{(\varphi\,:\,A\to A)} \sum_{(n\,:\,\mathbb{N})} \left\| \sum_{(e\,:\,A\simeq[\![n]\!])} (e\circ\varphi = \mathrm{pred}\circ e) \right\|. \tag{2.5–23}$$

Notice that the type Cyclic(A) mirrors the structure of $[\![n]\!]$ given by pred for any finite type A along with an endomap $\varphi : A \to A$. This is reflected in (2.5–23) by establishing a structure-preserving map between (A,φ) and $([\![n]\!], \mathrm{pred})$. Therefore, a type A with cyclic structure is a triple such as $\langle A, f, n\rangle$ where $(f, n, \text{-}) : \mathrm{Cyclic}(A)$. Given such a triple, we refer to A as an *n-cyclic* and f as the corresponding *cyclic function*. As a notation, if $p : \mathrm{Cyclic}(A)$ and $x : A$, then $p(x)$ is the image of x under the cyclic function f.

Lemma 2.22. Let P be a family of propositions of type $\Pi_{X:\mathcal{U}}(X \to X) \to \mathrm{hProp}$ and an n-cyclic structure $\langle A, f, n\rangle$. If $P([\![n]\!], \mathrm{pred})$, then $P(A, f)$.

Proof. It follows from Lemma 2.8. Note that being cyclic for a type is equivalent to saying (A, f) and $([\![n]\!], \mathrm{pred})$ are connected in $\Sigma_{X:\mathcal{U}}(X \to X)$. □

Lemma 2.23. Let P be a family of propositions of type $\mathcal{U} \to \mathrm{hProp}$ and an n-cyclic structure $\langle A, f, n\rangle$. If $P([\![n]\!])$, then $P(A)$.

Proof. Given an n-cyclic structure $\langle A, f, n\rangle$, we have a certain p such that

$$p : \left\| \sum_{(e\,:\,A\simeq[\![n]\!])} (e\circ f = \mathrm{pred}\circ e) \right\|.$$

Our objective is to apply propositional truncation elimination on the proposition $P([\![n]\!])$ to derive $P(A)$. To achieve this, we need to construct a term of type $P(A)$ from a pair $(e, !)$, where $e : A \simeq [\![n]\!]$ and $! : e\circ f = \mathrm{pred}\circ e$. The conclusion follows from the fact that equivalences preserve propositions (Lemma 2.13), applied to e and using the predicate P on $[\![n]\!]$. □

Lemma 2.24. Let A be a type. If Cyclic(A) is inhabited, then A is a finite set.

Proof. This follows from Lemma 2.23 and the property that the standard finite type $[\![n]\!]$ is a finite set. □

In any finite type, every element is searchable. In particular, given an n-cyclic type $\langle A, f, n\rangle$, one can search any element by iterating the function f on any other element at most n times.

Lemma 2.25. If A is an n-cyclic type, then for every a and b in A, there exists a unique number k with $k < n$ such that $\mathrm{pred}_A^k(a) = b$.

The total type, $\Sigma_{A:\mathcal{U}}\mathrm{Cyclic}(A)$, is the classifying type (Bezem, Buchholtz, Cagne, et al. 2022, §4.6-7) of finite cyclic groups. Let us now compute the identity type between two finite cyclic types that we use, for example, in Example 4.32 to enumerate the maps of the bouquet graph B_2.

Lemma 2.26. Given two cyclic types, $\mathscr{A}$ and $\mathscr{B}$, defined by $\langle A, f, n\rangle$ and $\langle B, g, m\rangle$, respectively, the identity type between them is given by the following equivalence:

$$(\mathscr{A} = \mathscr{B}) \simeq \textstyle\sum_{(\alpha\ :\ A = B)} (\mathrm{coe}\,(\alpha) \circ f = g \circ \mathrm{coe}\,(\alpha)).$$

$$\begin{array}{ccc} A & \xrightarrow{\mathrm{coe}(\alpha)} & B \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ A & \xrightarrow[\mathrm{coe}(\alpha)]{} & B \end{array}$$

Proof. We show the equivalence by Calculation (2.5–24). In Equivalence (2.5–24b), we expand the definition of the type of cycle for $\mathscr{A}$ and $\mathscr{B}$. The numbers n and m are the cardinalities of the types A and B, respectively, and p and q, are propositions of the truncation appearing in the type in (2.5–23). Equivalence (2.5–24c) follows from the characterisation of the identity type between pairs in a Σ-type (Univalent Foundations Program 2013, §3.7). In Equivalence (2.5–24c), we have the product of two propositions, the identity types, $n = m$ and $p = q$. These two types are, in fact, contractible, therefore, equivalent to the one-point type. The numbers n and m are equal because A and B are finite and equal by α, and p and q are equal because truncation of any type is also a proposition. We can then simplify the inner Σ-type to its base in Equivalence (2.5–24d) to obtain by the equivalence $\Sigma_{x:A}\mathbb{1} \simeq A$, Equivalence (2.5–24e).

$$(\mathscr{A} = \mathscr{B}) \equiv \tag{2.5–24a}$$

$$((A, (f, n, p)) = (B, (g, m, q))) \simeq \tag{2.5–24b}$$

$$\sum_{(\alpha\ :\ A = B)} \sum_{(\beta\ :\ \mathrm{tr}^{\lambda X.X\to X}(\alpha, f) = g)} (n = m) \times (p = q) \simeq \tag{2.5–24c}$$

$$\sum_{(\alpha\ :\ A = B)} \sum_{(\beta\ :\ \mathrm{tr}^{\lambda X.X\to X}(\alpha, f) = g)} \mathbb{1} \simeq \tag{2.5–24d}$$

$$\sum_{(\alpha\ :\ A = B)} \mathrm{tr}^{\lambda X.X\to X}(\alpha, f) = g \simeq \tag{2.5–24e}$$

$$\sum_{(\alpha\ :\ A = B)} \mathrm{coe}\,(\alpha) \circ f = g \circ \mathrm{coe}\,(\alpha)\,. \tag{2.5–24f}$$

Finally, Equivalence (2.5–24f) is a consequence of transporting functions along the equality α. The conclusion is that the identity type $\mathscr{A} = \mathscr{B}$ is equivalent to the type of equalities between A and B along with a proof that the structure of f is preserved in the structure of g. □

Lemma 2.27. For any finite type A, Cyclic(A) is a finite set.

Proof. We unfold the definition of Cyclic(A) to obtain the type $\Sigma_{\varphi\,:\,A\to A}\Sigma_{n\,:\,\mathbb{N}}\,\|\,P(A,n)\,\|$ where $P(A,n) :\equiv \Sigma_{e\,:\,A\simeq [\![n]\!]}(e\circ\varphi = \mathsf{pred}\circ e)$.

Given the finiteness of type A, it follows that $A \to A$ is finite. We now aim to show that $\Sigma_{n\,:\,\mathbb{N}}\|P(A,n)\|$ is finite. We can show this by establishing the equivalence

$$\sum_{(n\,:\,\mathbb{N})} \|P(A,n)\| \simeq \|P(A,\#A)\| \tag{2.5–25}$$

and demonstrating that the type $P(A,\#A)$ is finite. Once established, we can conclude that the equivalence preserves the finiteness of the type $\|P(A,\#A)\|$, by the closure property of finite types under Σ-types and propositional truncation.

To establish the equivalence in (2.5–25), as both types are propositions, we only need to construct two functions f and g as follows using the propositional truncation elimination principle,

$$\begin{aligned}
&f : \textstyle\sum_{(n\,:\,\mathbb{N})}\|P(A,n)\| \to \|P(A,\#A)\|.\\
&f((n,|p|)) :\equiv |p|.\\
&g : \|P(A,\#A)\| \to \textstyle\sum_{(n\,:\,\mathbb{N})}\|P(A,n)\|.\\
&g(|r|) :\equiv (\#A,|r|).
\end{aligned}$$

The Σ-type, $P(A,\#A)$, is finite given that the base type is an equivalence between two finite types, A and $[\![\#A]\!]$, and each fiber is an identity type over a finite type, which is finite. This leads us to conclude that the type $\Sigma_{n\,:\,\mathbb{N}}\|P(A,n)\|$ is finite, thereby implying that Cyclic(A) is finite. □

3

Graphs in Univalent Mathematics

Graphs are a fundamental mathematical concept that has found widespread applications in various fields, including mathematics and computer science. They are used to modelling relationships between objects or entities, making them a versatile tool for analysing complex systems. However, the definition of a graph can vary depending on the context in which it is used. The choice of a specific notion of a graph in a given context depends on the application, such as power graphs in computational biology, quivers in category theory, and networks in network theory. In some settings, graphs are undirected, while in others, they are directed. Additionally, the inclusion of self-edges may be allowed or prohibited. In this chapter, we define the notion of graphs in type theory that we consider in this thesis. Additionally, we briefly present concepts such as the homomorphism between graphs, finite graphs, and cyclic graphs, among others. The following chapters will use these concepts unless otherwise stated.

3.1 The type of graphs

The objective of this thesis is to present a thorough characterisation of graph planarity. In pursuit of this objective, we employ a broader set-level concept of graphs that encompasses directed multigraphs, including those with self-edges, in contrast to the conventional practise of working solely with undirected graphs. The decision to adopt a set-level structure for this type of graph is informed by the observation that the objects and re-

lations studied in the graph theory literature typically involve sets. Nevertheless, this constraint can be readily relaxed for other applications, as seen in Appendix B.

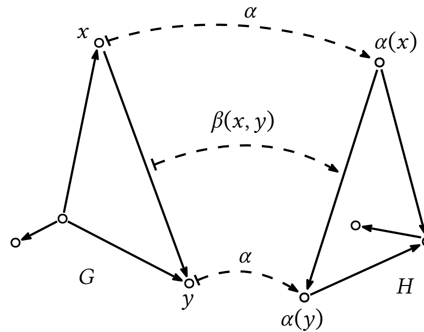
Definition 3.1. A graph is an object of type Graph. The corresponding data of a graph is a set N, elements of which we call points/vertices/nodes. Additionally, for every pair of nodes a and b, there is a family of sets E, each of which corresponds to the edges connecting a and b. The elements of these sets are referred to as edges.

$$\mathsf{Graph} :\equiv \sum_{(\mathsf{N}\,:\,\mathcal{U})} \sum_{(\mathsf{E}\,:\,\mathsf{N}\to\mathsf{N}\to\mathcal{U})} \mathsf{isSet}(\mathsf{N}) \times \prod_{(x,y\,:\,\mathsf{N})} \mathsf{isSet}(\mathsf{E}(x,y)).$$

Given a graph G, for brevity, the set of nodes and the family of edges are denoted by N_G and E_G, respectively. In this way, the graph G is defined as $(\mathsf{N}_G, \mathsf{E}_G, (p_G, q_G))$ where $p_G : \mathsf{isSet}(\mathsf{N}_G)$ and $q_G : \prod_{x,y:\mathsf{N}_G} \mathsf{isSet}(\mathsf{E}_G(x,y))$. We may refer to G only as the pair $(\mathsf{N}_G, \mathsf{E}_G)$, unless we require showing the remaining data, the propositions p_G and q_G. For example, we define the *empty* graph and the *unit* graph, respectively, as $(\mathbb{0}, \lambda\, u\, v.\mathbb{0})$ and $(\mathbb{1}, \lambda\, u\, v.\mathbb{0})$. We will use variables G and H as graphs, and variables x, y, and z as nodes in G, unless otherwise specified.

Remark 3.2. Our primary objective is to provide a comprehensive characterisation of graph planarity. To achieve this, we utilise a set-level concept of graphs, which includes directed multigraphs and those with self-edges, diverging from the traditional focus on undirected graphs. The choice of a set-level structure is based on the common use of sets in the objects and relations studied within graph theory. However, this constraint can be easily modified for different applications.

Definition 3.3. A *graph homomorphism* from G to H is a pair of functions (α, β) such that $\alpha : \mathsf{N}_G \to \mathsf{N}_H$ and $\beta : \prod_{x,y:\mathsf{N}_G} \mathsf{E}_G(x,y) \to \mathsf{E}_H(\alpha(x), \alpha(y))$. We denote by $\mathsf{Hom}(G, H)$ the type of these pairs.



We denote by id_G, for any graph G, the identity graph homomorphism where the corresponding α and $\beta(x,y)$ are the corresponding identity functions.

Lemma 3.4. The type $\mathsf{Hom}(G, H)$ forms a set.

Proof. Since sets are closed under Π- and Σ-types, and given that both $\mathsf{N}_G \to \mathsf{N}_H$ and $\prod_{x,y:\mathsf{N}_G} \mathsf{E}_G(x, y) \to \mathsf{E}_H(\alpha(x), \alpha(y))$ are function types with set codomains, it follows that $\mathsf{Hom}(G, H)$, being comprised of these types, is a set. □

3.2 The category of graphs

Graphs as objects and graph homomorphisms as the corresponding arrows form a small pre-category. In fact, the type of graphs is a small univalent category in the sense of the HoTT Book (Univalent Foundations Program 2013, §9.1). This fact follows from Theorem 3.7 and, morally, because the Graph type is a set-level structure.

In a (pre-) category, an isomorphism is a morphism which has an inverse. In the particular case of graphs, this can be formulated in terms of the underlying maps being equivalences.

Lemma 3.5. Let h be a graph homomorphism given by the pair-function (α, β). The claim h is an *isomorphism*, denoted by $\mathsf{isIso}(h)$, is a proposition equivalent to stating that the functions α and $\beta(x, y)$ for all $x, y : \mathsf{N}_G$, are all bijections.

$$\mathsf{isIso}(h) :\equiv \mathrm{isEquiv}(\alpha) \times \prod_{(x,y:\mathsf{N}_G)} \mathrm{isEquiv}(\beta(x, y)).$$

The type of all isomorphisms between G and H is denoted by $G \cong H$ and defined as

$$G \cong H :\equiv \sum_{(h:\mathsf{Hom}(G,H))} \mathsf{isIso}(h) \qquad (3.2\text{–}1)$$

or equivalently, as the following type,

$$\sum_{(\alpha:\mathsf{N}_G \simeq \mathsf{N}_H)} \prod_{(x,y:\mathsf{N}_G)} \mathsf{E}_G(x, y) \simeq \mathsf{E}_H(\alpha(x), \alpha(y)).$$

If the type $G \cong H$ is inhabited, it is said that G and H are *isomorphic*.

Lemma 3.6. The type $G \cong H$ forms a set.

Proof. Given $G \cong H$ as a subtype of $\mathsf{Hom}(G, H)$, and by Lemma 3.4 asserting that $\mathsf{Hom}(G, H)$ is a set, it immediately follows from (3.2–1) that $G \cong H$ inherits the set structure. □

We define a type to compare the sameness in graphs in Lemma 3.5; the type of graph isomorphisms. In HoTT, the identity type (=) serves the same purpose, and one expects

the two notions to coincide (Coquand and Danielsson 2013). In Theorem 3.7, we prove that they are, in fact, homotopy equivalent. The same correspondence for graphs also arises for many other structures, for example, groups and topological spaces (Ahrens and North 2019; Ahrens, North, Shulman, and Tsementzis 2020).

Theorem 3.7 (Equivalence principle). The canonical map

$$\mathsf{idtoiso} : (G = H) \to (G \cong H)$$

is an equivalence and its inverse function is denoted by isotoid.

Proof. It is sufficient to show that $(G = H) \simeq (G \cong H)$. Remember that being an equivalence for a function constitutes a proposition. We consider the following type families to shorten the presentation.

- ▷ $F_1(X) :\equiv X \to X \to \mathcal{U}$ and
- ▷ $F_2(X, R) :\equiv \Pi_{x,y:X}\mathsf{isSet}(R(x,y))$ where R is of type $F_1(X)$.

The required equivalence follows from Calculation (3.2–2).

$$(G = H) \equiv \tag{3.2–2a}$$

$$((\mathsf{N}_G, \mathsf{E}_G, (p_G, q_G)) = (\mathsf{N}_H, \mathsf{E}_H, (s_H, t_H))) \simeq \tag{3.2–2b}$$

$$\sum_{(\alpha : \mathsf{N}_G = \mathsf{N}_H)} \sum_{(\beta : \mathsf{tr}^{F_1}(\alpha, \mathsf{E}_G) = \mathsf{E}_H)} (\mathsf{tr}^{\mathsf{isSet}}(\alpha, p_G) = s_H) \times (\mathsf{tr_2}^{F_2}(\alpha, \beta, q_G) = t_H) \simeq \tag{3.2–2c}$$

$$\sum_{(\alpha : \mathsf{N}_G = \mathsf{N}_H)} \sum_{(\beta : \mathsf{tr}^{F_1}(\alpha, \mathsf{E}_G) = \mathsf{E}_H)} \mathbb{1} \times \mathbb{1} \simeq \tag{3.2–2d}$$

$$\sum_{(\alpha : \mathsf{N}_G = \mathsf{N}_H)} \mathsf{tr}^{F_1}(\alpha, \mathsf{E}_G) = \mathsf{E}_H \simeq \tag{3.2–2e}$$

$$\sum_{(\alpha : \mathsf{N}_G = \mathsf{N}_H)} \prod_{(x,y : \mathsf{N}_G)} \mathsf{E}_G(x,y) = \mathsf{E}_H(\mathsf{coe}(\alpha)(x), \mathsf{coe}(\alpha)(y)) \simeq \tag{3.2–2f}$$

$$\sum_{(\alpha : \mathsf{N}_G \simeq \mathsf{N}_H)} \prod_{(x,y : \mathsf{N}_G)} \mathsf{E}_G(x,y) \simeq \mathsf{E}_H(\alpha(x), \alpha(y)) \simeq \tag{3.2–2g}$$

$$(G \cong H). \tag{3.2–2h}$$

We first unfold definitions in (3.2–2b). The equivalence in (3.2–2c) follows from the characterisation of the identity type between pairs in a Σ-type (Lemma 3.7 in HoTT book). The equivalence in (3.2–2d) stems from the fact that being a set is a mere proposition and, thus, equations between proofs of such are contractible, similarly as in (2.26). To get (3.2–2f), we apply function extensionality twice in the inner equality in (3.2–2e). By the Univalence axiom, we replace in (3.2–2g) equalities by equivalences. Finally, (3.2–2h) follows from (3.5) completing the calculation from which the conclusion follows. □

Lemma 3.8. The type of graphs is a groupoid.

Proof. Consider graphs G and H. We want to show that the identity type $G = H$ is a set, for which we apply Theorem 3.7. This yields an equivalence between the type $G = H$ and the set of isomorphisms $G \cong H$ (refer to Lemma 3.6). Since equivalences preserve set structures, it follows that $G = H$ is indeed a set. □

3.3 Subtypes and structures on graphs

In graph theory, graphs are often classified according to their structure in different *graph classes*. This can be mirrored in type theory by considering type families over the type Graph. These type families result in a subtype of graphs if they are propositions; otherwise, they might provide a structure on graphs.

A notable example of such a structure is our characterisation of *planar* graphs. We define a type family Planar over Graph and establish that Planar(G) is a set, not a proposition, for any graph G. More details can be found in Chapter 6.

Here are some informal examples of graph subtypes that one can define in type theory.

- ▷ *Simple* graphs: The edge relation is propositional.
- ▷ *Undirected* graphs: The edge relation is symmetric.
- ▷ *Connected* graphs: A walk exists between any two nodes.
- ▷ *Complete* graphs: Each node is connected to every other node by an edge.
- ▷ *Trees*: These are connected graphs without *cycles* (refer to Appendix B).
- ▷ *Regular* graphs: Each node has the same number of connected edges.
- ▷ *Bipartite graphs*: Nodes can be split into two disjoint sets with all edges connecting a node in one set to a node in the other.

Now, since any construction in HoTT respects the structure of its constituents, graph subtypes are invariant under graph isomorphisms. Specifically, given a graph isomorphism, we can transport any property on graphs along the equality obtained by Theorem 3.7. Equivalence induction, a related principle, is discussed in (Escardó 2019, §3.15).

Lemma 3.9 (Leibniz principle). Isomorphic graphs hold the same properties.

Lemma 3.10 (Equivalence induction). Given a graph G and a family of properties P of type $\Sigma_{H:\mathsf{Graph}}(G \cong H) \to \mathsf{hProp}$, if the property $P(G, \mathsf{id}_G)$ holds then the property also holds for any isomorphic graph H to G, i.e., $P(H, \varphi)$ holds for all $\varphi : G \cong H$.

Lastly, of importance for this work is the subtype of connected finite graphs. We will assume any graph in the remaining of this document, as connected and finite, unless stated otherwise.

3.4 Finite graphs

A graph is *finite* if its node set and each edge-set are finite sets, as stated in Definition 3.11. Like finite types, a finite graph has an associated cardinal number for the count of nodes and edges. Hence, we can demonstrate that equality is decidable on both the node set and each edge set for finite graphs.

Definition 3.11. A graph G is said to be *finite* when the following proposition $\mathsf{isFiniteGraph}(G)$ holds.

$$\mathsf{isFiniteGraph}(G) :\equiv \mathsf{isFinite}(\mathsf{N}_G) \times \mathsf{isFinite}\left(\sum_{(x,y\,:\,\mathsf{N}_G)} \mathsf{E}_G(x,y)\right).$$

For a finite graph G, the cardinality of the node set and edge set are represented as $\#\mathsf{N}_G$ and $\#\mathsf{E}_G$ respectively.

3.5 Walks and strongly connected graphs

A graph G is considered to be *strongly connected* or (*connected* for short) when for any pair of nodes x and y, there is a walk from x to y in G. Intuitively, a *walk* in a graph is a sequence of edges that forms a chain; of the type stated in Definition 3.12.

Definition 3.12. A *walk* in G from x to y is a sequence of connected edges that we construct using the following inductive data type:

$$\begin{aligned}
\textbf{data}\ \mathsf{W} &: \mathsf{N}_G \to \mathsf{N}_G \to \mathcal{U} \\
\langle_\rangle &: (x : \mathsf{N}_G) \to \mathsf{W}_G(x,x) \\
(_\odot_) &: \Pi\{x\,y\,z : \mathsf{N}_G\}.(e : \mathsf{E}_G(x,y)) \\
&\quad \to (w : \mathsf{W}_G(y,z)) \\
&\quad \to \mathsf{W}_G(x,z)
\end{aligned}$$

Let w be a walk from x to y, i.e., of type $\mathsf{W}_G(x,y)$. We will denote by x the *head* of w and by y the *end* of w. If w is $\langle x\rangle$ then we refer to w as *trivial* or *one-point* walk. If w is of the form $(e \odot \langle x\rangle)$, then w is the *one-edge* walk e. Non-trivial walks are of the form, $(e \odot w)$ and a

loop is a walk with the same head and end. An equivalent notion of walk is path, which we hinted in Section 3.6.

Definition 3.13. A graph G is said to be connected when the proposition Connected(G) holds.

$$\mathsf{Connected}(G) :\equiv \prod_{(x,y\,:\,\mathsf{N}_G)} \|\mathsf{E}_{W(G)}(x,y)\|.$$

3.6 Graph families

Let us define a few graph families indexed by the type of natural numbers.

Definition 3.14. The *path graph* with n nodes is the non-connected graph P_n, defined as

$$P_n :\equiv ([\![n]\!], \lambda\, u\, v.\mathsf{toNat}(u) + 1 = \mathsf{toNat}(v)),$$

where

$$\mathsf{toNat} : [\![n]\!] \to \mathbb{N}.$$
$$\mathsf{toNat}(k, !) :\equiv k.$$

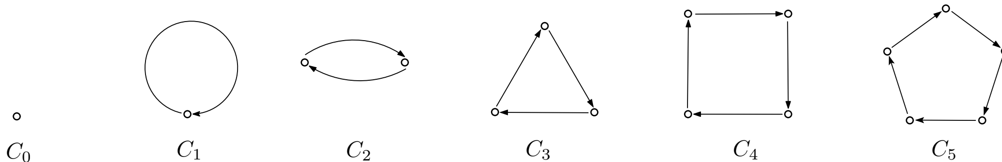
The length of path graph P_n is defined as the number of edges in P_n. Graphs P_0 and P_1 have zero length, P_2 has one edge. Hence, for $n > 0$, P_n has length $n - 1$.

Remark 3.15. The path graph definition allows us to alternatively define graph walks. Specifically, a walk in a connected graph G of length n between nodes a and b can be defined as a graph homomorphism from P_{n+1} to G for $n > 0$. This homomorphism maps node 0 to a and n to b. A trivial walk is a graph homomorphism from P_1 to G, selecting only one node a in G. If a equals b, the walk is *closed*. Closed walks, also known as cycles, are introduced using an alternative definition in Definition 3.19 that reflects cyclic types.

Definition 3.16. An n-cycle graph denoted by C_n is a graph with n edges defined as

$$C_n :\equiv ([\![n]\!], \lambda\, u\, v.u = \mathsf{pred}(v)),$$

when $n \geq 1$. Otherwise, C_0 is the one-point graph with one trivial loop. The function pred is defined in Definition 2.20. Similar to path graphs, the length of an n-cycle graph is n.

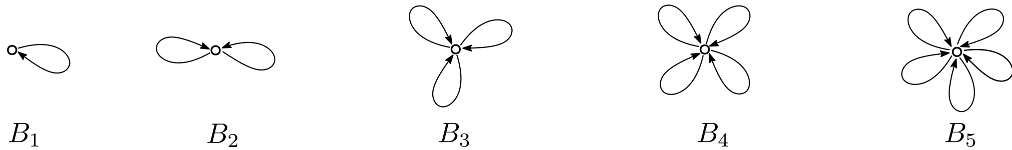


In the treatment of embeddings of graphs on surfaces, we found that bouquet graphs, besides their simple structure, have nontrivial embeddings, see Section 4.5.

Definition 3.17. The family of *bouquet* graphs B_n, given by

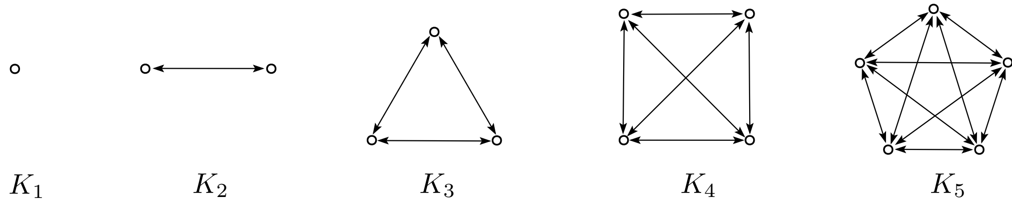
$$B_n :\equiv (\mathbb{1}, \lambda\, u\, v. [\![n]\!]),$$

consists of graphs obtained by considering a single point with n self-loops.



Definition 3.18. A graph of n nodes is called *complete* when every pair of distinct nodes is joined by an edge. The complete standard graph with node set $[\![n]\!]$ is denoted by K_n.

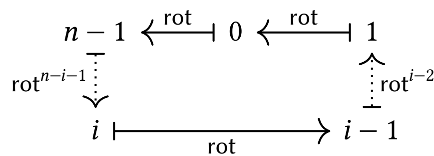
$$K_n :\equiv ([\![n]\!], \lambda\, u\, v. u \neq v).$$



For brevity, we will use a double arrow in the pictures from now on to denote a pair of edges of opposite directions.

3.7 Cyclic graphs

Similarly, as for cyclic types, we introduce a type of graphs with a cyclic structure. A graph is *cyclic* when it is in the connected component of an n-cycle graph in the Graph type.



Let us consider the homomorphism $\mathsf{rot} : \mathsf{Hom}(C_n, C_n)$ that acts similarly as the function pred in Definition 2.21. The homomorphism rot is an isomorphism on C_n, and then we can iterate it k times to obtain the isomorphism denoted by rot^k. Any of these isomorphisms can be used to define what it means for a graph to be cyclic.

In particular, the cyclic structure for graphs can be defined as the property of preserving the structure in C_n induced by the morphism rot. We will make use of the same notation as for cyclic sets to refer to cyclic graphs.

Definition 3.19. A graph G is considered to be *cyclic* if the type CyclicGraph(G) is inhabited.

$$\mathsf{CyclicGraph}(G) :\equiv \sum_{(\varphi\,:\,\mathsf{Hom}(G,G))} \sum_{(n\,:\,\mathbb{N})} \mathsf{isCyclic}(G,\varphi,n),$$

where

$$\mathsf{isCyclic}(G,\varphi,n) :\equiv \|(G,\varphi) = (C_n,\mathsf{rot})\|.$$

3.8 The identity type on graphs

For any element, x of a groupoid type, X, the type $\mathsf{Aut}_X(x) :\equiv (x = x)$ has a group structure given by reflexivity, symmetry, and path composition. Applying this definition to the groupoid of graphs, the equivalence principle of Theorem 3.7 gives that for any graph G, we identify Aut(G) with its automorphisms, $G \cong G$. This allows us to compute $\mathsf{Aut}(G) :\equiv G \cong G$ in the examples which follow.

1. Aut(B_2) is the group of two elements. With only two edges in B_2 and one node, we can only have, besides the identity function, the function that swaps the two edges. In general, the identity type $B_n = B_n$ is equivalent to the group S_n, the group which contains the permutations of n elements.

2. Any isomorphism in Aut(C_n) is completely determined by how it acts on a fixed node in C_n, stated in the following lemma.

Lemma 3.20. Let $n : \mathbb{N}$. If $n > 0$, then there exists an equivalence between the type Aut(C_n) and the type $[\![n]\!]$.

Proof. The result follows from considering the isomorphism rot as introduced in Definition 3.19 and the isomorphisms rot^k for $k < n$. The equivalence between the type $[\![n]\!]$ and the collection of isomorphisms $C_n \cong C_n$ is then given by the following function f and its inverse g.

$$\begin{aligned} &f : [\![n]\!] \to (C_n \cong C_n). && g : (C_n \cong C_n) \to [\![n]\!].\\ &f(k,!) :\equiv (\mathsf{rot}^k, p). && g(h,!) :\equiv (r,s).\end{aligned}$$

The term p used to define f is the proof that rot^k is an isomorphism. The term r is the solution to the equation $\mathsf{rot}^r = h$, and s is the proof that $r < n$. Now, since $[\![n]\!]$ is a set, we obtain a homotopy $g \circ f \sim \mathsf{id}_{[\![n]\!]}$. The other homotopy condition, i.e. $f \circ g \sim \mathsf{id}_{(C_n \cong C_n)}$, can be derived from the intermediate result stating that if $\mathsf{rot}^p = \mathsf{rot}^q$ and $p, q < n$, then $p = q$. The elaboration of this proof is given in Example A.2. □

The family of graphs C_n is presented intentionally, serving as a crucial component in defining a combinatorial map's face, referenced in Section 4.4. The previous lemma contributes to the proof that any graph's combinatorial map face type is a set, further detailed in Lemma 4.18.

4
Graph Maps

In this chapter, we explore the use of graph maps as an alternative approach to directly working with surfaces on which graphs are embedded. Our aim is to characterise graphs with no edge-crossing in the two-dimensional plane without needing to represent the surface explicitly. This is motivated by the fact that the concept of surface is not well-defined in homotopy type theory, and working with the real numbers can be laborious.

To avoid the complexities associated with the explicit notion of the surface in type theory, we focus on representing the drawings of graphs in a more abstract way, which is defining the type of graph maps, also called cellular embeddings, using their combinatorial characterisation (Stahl 1978). By leveraging the power of combinatorial representation of graph maps, we provide a more comprehensive framework for analysing graph planarity, rather than focussing exclusively on the geometric properties and how two-edges cross in the plane, which can be more challenging to study.

4.1 Symmetrisation of graphs

Here we introduce the symmetrisation construction which allows us to establish two key concepts related to graph maps, stars, and faces. The symmetrisation of a graph G, denoted by $\mathsf{Sym}(G)$, is one solution used here to encode how the edges are oriented in a graph map. This construction is similar to the concept of *half-edges* for signed rotation maps in the literature of embedded undirected graphs (Ellis-Monaghan and Moffatt 2013,

§1.1.8).

Definition 4.1. The *symmetrisation* of a graph G is the graph $\mathsf{Sym}(G)$ defined as follows.

$$\begin{aligned}&\mathsf{Sym} : \mathsf{Graph} \to \mathsf{Graph}.\\&\mathsf{Sym}\,(G) :\equiv (\mathsf{N}_G, \lambda xy.\mathsf{E}_G(x,y) + \mathsf{E}_G(y,x), p_G, r(q_G)),\end{aligned}$$

where r is a proof that the coproduct $\mathsf{E}_G(x,y) + \mathsf{E}_G(y,x)$ is a set using q_G as a proof that $\mathsf{E}_G(x,y)$ is a set for all $x, y : \mathsf{N}_G$.

Every edge $a : \mathsf{E}_G(x,y)$ in G induces two edges in $\mathsf{Sym}(G)$. The first is $\mathsf{inl}(a)$ keeping the same direction as a. This edge is denoted by $\overleftarrow{a}$ for short. The second is $\mathsf{inr}(a)$, which goes in the opposite direction of a. This edge is denoted by $\overrightarrow{a}$ for short. Since the nodes of $\mathsf{Sym}(G)$ are the same as the nodes of G, we will use the same notation for the nodes of both graphs. The following is an immediate consequence of the induced edges in $\mathsf{Sym}(G)$ by the edges in G.

Lemma 4.2. Consider a graph G. For every walk w in G, we can induce a corresponding walk in the symmetrisation $\mathsf{Sym}(G)$, denoted by $\mathsf{sym}(w)$.

Proof. The function sym generates the induced walk in $\mathsf{Sym}(G)$ from a walk w in G.

$$\begin{aligned}&\mathsf{sym} : \prod_{(x,y : \mathsf{N}_G)} \mathsf{W}_G(x,y) \to \mathsf{W}_{\mathsf{Sym}(G)}(x,y).\\&\mathsf{sym}(x, _, \langle x\rangle) :\equiv \langle x\rangle.\\&\mathsf{sym}(x, y, e \odot w) :\equiv \mathsf{inl}(e) \odot \mathsf{sym}(_, y, w).\end{aligned}$$

□

Lemma 4.3. The Sym operation on a graph G preserves the following properties:

- ▷ connectedness of G and
- ▷ finiteness of G.

Proof. Let us begin by proving the first property. Assume that G is connected, and our objective is to show that $\mathsf{Sym}(G)$ is also connected. This can be established by showing the existence of a function of type

$$\left\| \prod_{(x,y : \mathsf{N}_G)} \mathsf{W}_G(x,y) \right\| \to \left\| \prod_{(x,y : \mathsf{N}_{\mathsf{Sym}(G)})} \mathsf{W}_{\mathsf{Sym}(G)}(x,y) \right\|.$$

Since the fact that G is connected is a proposition, we can construct such a function using the elimination rule for propositional truncation and the function sym defined in

Lemma 4.2 when applied to a walk in G. In general, for A and B types, a function of type $A \to B$ can be lifted $\|A\| \to \|B\|$ by similar reasoning.

On the other hand, to prove that $\mathsf{Sym}(G)$ is finite when G is finite, we only need to consider the family of edges in $\mathsf{Sym}(G)$. This family consists of finite coproducts, as it is the coproduct of two finite sets. Furthermore, the set of nodes in $\mathsf{Sym}(G)$ is identical to the set of nodes in G, which is finite by assumption. □

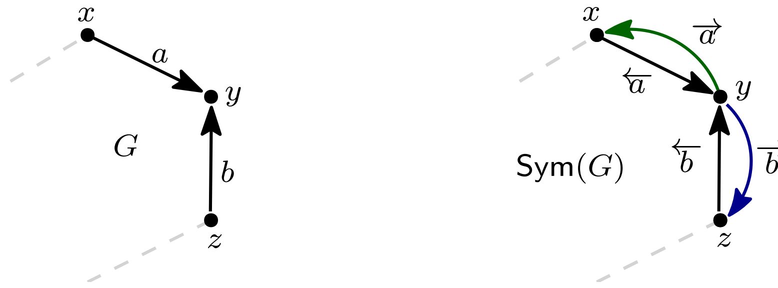


Figure 4.1: On the left we show a part of a graph G with two distinguished edges, a and b. On the right we show the corresponding symmetrisation, $\mathsf{Sym}(G)$, including the two edges, $\overleftarrow{a}$ and $\overrightarrow{a}$ induced by a, and similarly, $\overleftarrow{b}$ and $\overrightarrow{b}$ induced by b. For brevity, we will only draw a segment representing related edges in the symmetrisation, as in Figure 4.2 (b).

4.2 Stars and locally finite graphs

Definition 4.4. The *star* at a node x in a graph G is the type $\mathsf{Star}_G(x)$.

$$\mathsf{Star}_G(x) :\equiv \sum_{(y\,:\,\mathsf{N}_G)} \mathsf{E}_{\mathsf{Sym}(G)}(x, y). \tag{4.2–1}$$

Let y be a node in G. If $e : \mathsf{E}_G(x, y)$, then the pair $(y, \mathsf{inl}(e))$ is referred to as an *outgoing edge* in the start at x. Similarly, if $e : \mathsf{E}_G(y, x)$, then the pair $(y, \mathsf{inr}(e))$ is referred to as an *incoming edge* in the start at x. An *incident* edge of x is either an outgoing or an incoming edge in the star at x. The cardinality of the set of incident edges at x is known as the *valency* of x.

Example 4.5. The graph C_n is a basic example of a planar graph and a building block to construct more complex planar graphs. To enable this construction, we need to characterise the stars at any node in C_n for $n > 0$. The case when n is zero is trivial, as the star at any node in the empty graph is empty.

As C_n is a graph consisting of n nodes in $[\![n]\!]$ arranged in a polygon/cycle, one can associate the previous and the next node in the cycle, $\mathsf{pred}(x)$ and $\mathsf{suc}(x)$, for each node x in C_n, respectively. We will prove that the valency of any node in C_n is 2 by proving that there exists an equivalence f_x from $\mathsf{Star}_{C_n}(x)$ to $[\![2]\!]$ for every node x in C_n. The candidate to be the inverse of f_x is the function g_x defined below.

$$\begin{aligned} f_x &: \mathsf{Star}_{C_n}(x) \to [\![2]\!]. & g_x &: [\![2]\!] \to \mathsf{Star}_{C_n}(x). \\ f_x\,(y, \mathsf{inl}(p)) &:\equiv (0, !). & g_x\,(0, !) &:\equiv (\mathsf{suc}(x), \mathsf{inl}(a^+)). \\ f_x\,(y, \mathsf{inr}(p)) &:\equiv (1, !). & g_x\,(1, !) &:\equiv (\mathsf{pred}(x), \mathsf{inr}(a)). \end{aligned} \tag{4.2–2}$$

One can easily prove that both $\mathsf{E}_{C_n}(\mathsf{pred}(x), x)$ and $\mathsf{E}_{C_n}(x, \mathsf{suc}(x))$ are contractible types. Therefore, without loss of generality, we write a^+ to denote the edge from x to $\mathsf{suc}(x)$ and a to denote the edge from $\mathsf{pred}(x)$ to x in C_n.

To complete the proof that f_x is an equivalence, we need to show that $f_x \circ g_x \sim \mathsf{id}_{[\![2]\!]}$ and $g_x \circ f_x \sim \mathsf{id}_{\mathsf{Star}_{C_n}(x)}$. The first is immediate by case analysis. For example, $(f_x \circ g_x)((0,!)) \equiv f_x(g_x((0,!))) \equiv f_x(\mathsf{suc}(x), \mathsf{inl}(p)) \equiv (0,!)$, and one can similarly show that $f_x \circ g_x((1,!)) = (1,!)$.

To prove the second part, we show that $g_x \circ f_x \sim \mathsf{id}_{\mathsf{Star}C_n(x)}$ by performing a case analysis on the second component of a term $(y, z) : \mathsf{Star}_{C_n}(x)$. Specifically, we consider whether z is either $\mathsf{inl}(u)$ or $\mathsf{inr}(v)$. For the first case, we need to prove that $g_x(f_x((y, \mathsf{inl}(u)))) = (y, \mathsf{inl}(u))$. Evaluating the expression of the composite, we obtain an equality with the question mark below, which we need to show one can inhabit.

$$g_x(f_x((y, \mathsf{inl}(u)))) \equiv g_x((0,!)) \equiv (\mathsf{suc}(x), \mathsf{inl}(a^+)) \stackrel{?}{=} (y, \mathsf{inl}(u)).$$

However, we can establish the required equality by noting that $\mathsf{E}_{C_n}(x, \mathsf{suc}(x))$ is contractible. This implies that $\mathsf{E}_{\mathsf{Sym}(C_n)}(x, \mathsf{suc}(x))$ is a proposition, which in turn implies that $a^+ = u$ and that we have $y = \mathsf{suc}(x)$. Similarly, we can show that $g_x(f_x((y, \mathsf{inr}(v)))) = (y, \mathsf{inr}(v))$. This completes the proof that f_x is an equivalence and shows that $\mathsf{Star}_{C_n}(x)$ has only two elements.

Lemma 4.6. If G is a (finite) graph, then the type $\mathsf{Star}_G(x)$ is (finite) set.

Proof. The conclusion follows since the base type in Definition 4.4 is the set of edges in the graph, and each of the fibres of the Σ-type is a set since they are coproducts of sets. In particular, if the graph is finite, then all the types appearing in the type $\mathsf{Star}_G(x)$ are finite sets, and then our conclusion follows. □

Definition 4.7. A graph G is *locally finite* if the set of *incident* edges at the *star* at any node x in G, is a finite set.

4.3 The type of combinatorial maps

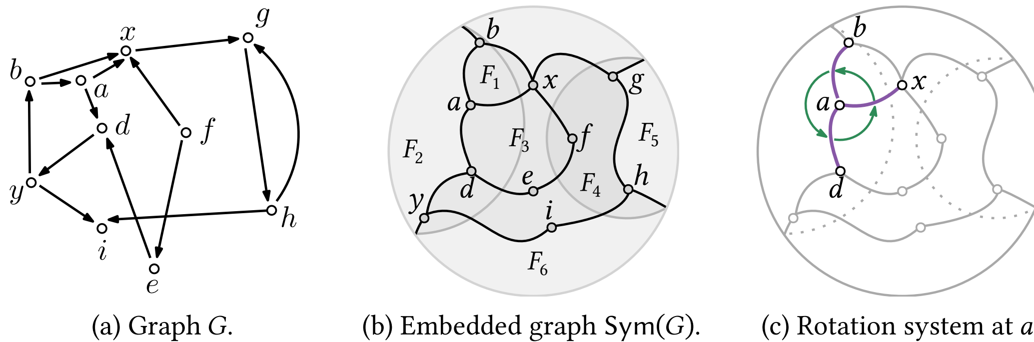
A combinatorial map is a specific type of data structure that is used to represent a graph that is embedded in a surface. This data structure offers a powerful substitute to traditional analytic/geometric techniques for representing such embeddings. Unlike geometric methods, combinatorial maps allow us to represent the combinatorial structure of the

topological embedding without the need to explicitly work with the surface in which the graph is embedded.

In this work, we focus on defining the type of combinatorial maps in type theory; see Definition 4.8. We then turn our attention to a particular kind of embedding, *cellular* embeddings. The reason for this focus is that all graph embeddings in the two-dimensional plane are cellular embeddings. Therefore, drawing graphs in the plane without edge crossings can be represented by cellular embeddings.

Cellular embeddings are particularly interesting because they can be characterised combinatorially up to isotopy by the cyclic order they induce in the set of nodes around each node in the graph (Gross and Tucker 1987), as illustrated in Figure 4.2 (b). This characterisation is minimal as no additional information is required beyond the cyclic orders.

One observation is that not all finite graphs can be drawn in the plane, but all finite graphs can be drawn on some orientable surface (Stahl 1978). The literature in graph theory has proven that a graph cannot have a cellular embedding on any surface if it has at least one node of infinite valency (Mohar 1988, Proposition §3.2). As our focus is on cellular embeddings, we will only examine locally finite graphs throughout the document.



(a) Graph G. (b) Embedded graph Sym(G). (c) Rotation system at a.

Figure 4.2: We show in (a) the drawing of a graph G with edge crossings. A representation of the graph G embedded in the sphere is shown in (b). The corresponding faces of the graph embedding shaded in (b) are named F_i for i from 1 to 6. It is shown in (c) with fuchsia colour the incident edges at the node a in Sym(G). The rotation system at a, that is, the cyclic set denoted by $(ba\, ad\, ax)$, is shown in green colour. The dashed lines represent edges not visible to the view.

Definition 4.8. Map(G) is the type of combinatorial maps (maps for short) for a graph G defined as follows:

$$\mathsf{Map}(G) :\equiv \prod_{(x\,:\,\mathsf{N}_G)} \mathsf{Cyclic}(\mathsf{Star}_G(x)).$$

Lemma 4.9. If the type $\mathsf{Map}(G)$ is inhabited, then the graph G is locally finite.

Lemma 4.10. The type of maps for a (finite) graph forms a (finite) set.

Proof. The type $\mathsf{Map}(G)$ is a set using the closure property of Π-types under (finite) sets. The type $\mathsf{Cyclic}(\mathsf{Star}_G(x))$ is a finite set by Lemma 2.27. □

For brevity, we use from now the variable $\mathcal{M}$ to denote a map of the graph G.

Example 4.11. The possible maps for the cycle C_n for $n > 0$ can be listed considering the cyclic structures of the two-point type. These correspond to the cyclic structures of the stars of C_n, see the correspondence exhibited in Example 4.5. The two maps are given by the following functions.

- ▷ $c_1 :\equiv \langle [\![2]\!], \mathsf{pred}, 2\rangle$ and
- ▷ $c_2 :\equiv \langle [\![2]\!], \mathsf{suc}, 2\rangle$.

4.4 The type of faces

In the context of cellular embeddings, faces correspond to regions homeomorphic to the open disk. Combinatorially, a face associated to a graph map consists of a cyclic walk in the embedded graph where no edges are inside the cycle, and no node occurs twice. Definition 4.14 is our attempt to make this intuition formal.

The first component of a face, as in Definition 4.14, captures the concept that its edges form a cyclic walk in the embedded graph. While working with such walks would typically necessitate a fixed starting point, as illustrated in Figure 4.3, this point does not contribute to the face's combinatorial structure. Hence, we can employ a cyclic graph to represent all such cyclic walks, thereby obviating the need for any distinguished starting point in such walks.

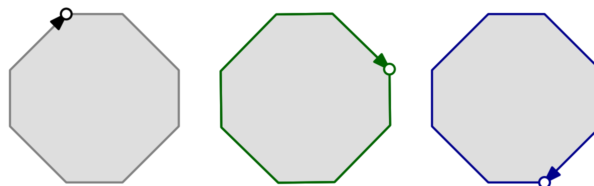


Figure 4.3: Example of cyclic walks on a face with different starting points.

The second component, the *map-compatibility* property, explicitly defines the "no edges on the inside" criterion for a face. This criterion is captured by the fact that each pair of consecutive edges on the face is a *successor-predecessor* pair in the cyclic order of the edges around their common node. In other words, when we move along the edges

of the face either clockwise or counterclockwise, we will never come across an edge that goes through the inside of the face. As our graphs are directed, we must traverse the edges in the symmetrisation of the graph rather than the graph itself.

The following two definitions are used in the definition of the type of faces.

Definition 4.12. A graph homomorphism h from G to H given by (α, β) is *edge-injective*, denoted by $\mathsf{isEdgeInj}(h)$, if the function f defined below is an embedding.

$$f : \sum_{(x,y \,:\, \mathsf{N}_G)} \mathsf{E}_G(x,y) \to \sum_{(x,y \,:\, \mathsf{N}_H)} \mathsf{E}_H(x,y).$$
$$f\,(x,y,e) :\equiv (\alpha(x), \alpha(y), \beta(x,y,e)).$$

Definition 4.13. The function flip swaps the direction of an edge in $\mathsf{Sym}(G)$.

$$\begin{aligned} \mathsf{flip} &: \prod_{(x,y\,:\mathsf{N}_G)} \mathsf{E}_{\mathsf{Sym}(G)}(x,y) \to \mathsf{E}_{\mathsf{Sym}(G)}(y,x). \\ \mathsf{flip}\,(x,y,\mathsf{inl}(e)) &:\equiv \mathsf{inr}(e). \\ \mathsf{flip}\,(x,y,\mathsf{inr}(e)) &:\equiv \mathsf{inl}(e). \end{aligned} \tag{4.4–3}$$

Since the first two arguments of the function flip are inferrable from the third argument, we will omit them below.

Definition 4.14. The type $\mathsf{Face}(G, \mathcal{M})$ is the type of faces of a combinatorial map $\mathcal{M}$ of a graph G. A face of type $\mathsf{Face}(G, \mathcal{M})$ consists of:

1. a cyclic graph A,
2. a graph homomorphism h given by (α, β) of type $\mathsf{Hom}(A, \mathsf{Sym}(G))$, such that
   (a) h is *edge-injective*,
   (b) h is *map-compatible*, denoted by $\mathsf{isMapComp}(h)$, meaning that h is star-compatible and corner-preserving, properties defined below, respectively.
      ▹ h is *star-compatible*, if the condition in (4.4–4) holds for every $x : \mathsf{N}_A$,

$$\mathsf{isStarComp}(h)(x) :\equiv \|\mathsf{Star}_G(\alpha(x))\| \to \|\mathsf{Star}_A(x)\|. \tag{4.4–4}$$

      ▹ h is *corner-compatible*, if there is evidence that h is compatible with the edge-ordering given by the map $\mathcal{M}$ at the node $\alpha(x)$ and the edge ordering coming from the star at that node x in A. To state this property, let us consider the following notation.
         – The *previous edge* at x is the edge $a : \mathsf{E}_{\mathsf{N}_A}(\mathsf{pred}(x), x)$,
         – the edge *after* a_x is the edge denoted by a_x^+ of type $\mathsf{E}_{\mathsf{N}_A}(x, \mathsf{suc}(x))$, as illustrated in Figure 4.4, and

– since $\mathscr{M}(\alpha(x))$ is a triple like $\langle f, m, !\rangle$ of type

$$\mathsf{Cyclic}(\mathsf{Star}_G(\alpha(x)))$$

for some function $f : \mathsf{Star}_G(\alpha(x)) \to \mathsf{Star}_G(\alpha(x))$ and some number m (the cardinality of the star at $\alpha(x)$), we abuse notation and use $\mathscr{M}(\alpha(x))$ to denote the function f. See more on the cyclic type in Definition 2.21.

$$\begin{aligned}\mathsf{isCornerComp}(h)(x) :\equiv & \mathscr{M}(\alpha(x))((\alpha(\mathsf{pred}(x)), \mathsf{flip}(\beta(\mathsf{pred}(x), x, a)))) \\ & =_{\mathsf{Star}_G(\alpha(x))} (\alpha(\mathsf{suc}(x)), \beta(x, \mathsf{suc}(x), a^+)).\end{aligned} \tag{4.4–5}$$

It should be noted that the truncation in (4.4–4) is intentional. By incorporating this, we aim to emphasise that if graph G has at least one edge at a given node, then a face covering that node, represented by the cyclic graph A, must have at least one edge at the corresponding node as well. Without this condition, the type of faces could be inhabited with *empty faces* using A as the cyclic graph without edges (C_0) at every node of the graph G. In Figure 4.4, we illustrate a portion of the required data to define a face F_1 for the map of graph G given in Figure 4.2 (b).

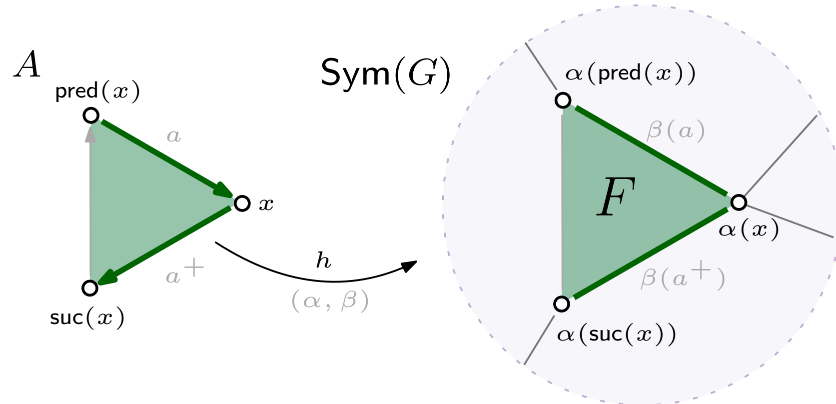


Figure 4.4: On the right side, we shade the face F of the graph G embedded in the sphere given in Figure 4.2. We have the cycle graph C_3 and $h : \mathsf{Hom}(C_3, \mathsf{Sym}(G))$ given by (α, β) on the left side. C_3 and h can be used to define the face F using C_3 as the graph A in Definition 4.14.

Lemma 4.15. For a graph homomorphism, being edge-injective is a proposition.

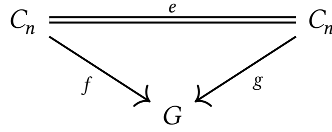
Proof. Edge-injectivity is a proposition by iteratively applying the closure of Π-types to propositions. Ultimately, we need to show that for any two terms (x, y, e_1) and (x', y', e_2) in $\Sigma_{x,y\,:\,\mathsf{N}_G} \mathsf{E}_G(x, y)$, the identity type $(x, y, e_1) = (x', y', e_2)$ is a proposition. This is true because the Σ-type in question is a set, and sets are closed under Σ-types, given that both N_G and $\mathsf{E}_G(x, y)$ are sets. □

Lemma 4.16. For a graph homomorphism, being map-compatible is a proposition.

Proof. For a graph homomorphism h, map-compatibility decomposes into star-compatibility and corner-compatibility. We must show each type in this product is a proposition. Star-compatibility is a proposition as it involves a function type with a propositional codomain —the propositional truncation of a set. Corner-compatibility is also a proposition, being a function type whose codomain is the identity type on $\mathsf{Star}_G(\alpha(x))$ at $\alpha(x)$. This identity type is a proposition since stars are sets, as established in Lemma 4.6. □

We devote the rest of this section to proving that the type of faces forms a set in Lemma 4.18. This claim rests on the fact that (i) the type of cyclic graphs forms a set, (ii) the type of graph homomorphisms forms a set, and (iii) the conditions, edge-injective and map-compatible in, Definition 4.14 are propositions. One might suspect that this type forms a groupoid from the previous facts. However, the edge-injectivity property of the underlying graph homomorphism of each face suffices to show that the type of faces is a set.

Lemma 4.17. Let f and g be edge-injective graph homomorphisms from C_n to a graph G and $n > 0$. Then the type $\Sigma_{e:C_n=C_n}\,(\mathrm{tr}^{\lambda X.\mathsf{Hom}(X,G)}(e,f) = g)$ is a proposition.



Proof. The result follows from the proof that the Σ-type in question is equivalent to a proposition. The corresponding equivalence is given by Calculation (4.4–6), in which we use some known results about Univalence and Lemma 3.20, as in the very last step.

$$\sum_{(e\,:\,C_n = C_n)} (\mathrm{tr}^{\lambda X.\mathsf{Hom}(X,G)}(e,f) = g) \simeq \sum_{(e\,:\,C_n = C_n)} (f = g \circ \mathsf{coe}\,(e)) \tag{4.4–6a}$$

$$\simeq \sum_{(e\,:\,C_n \simeq C_n)} (f = g \circ e) \tag{4.4–6b}$$

$$\simeq \sum_{(k\,:\,[\![n]\!])} (f = g \circ \mathsf{rot}^k). \tag{4.4–6c}$$

It remains to show that the last equivalent type is a proposition. Let (k_1, p_1), and (k_2, p_2) be of type $\Sigma_{k:[\![n]\!]}(f = g \circ \mathsf{rot}^k)$. We must show that (k_1, p_1) is equal to (k_2, p_2). Since $\mathsf{Hom}(C_n, G)$ is a set, we only need to prove that k_1 is equal to k_2. To show that, Lemma 3.20 is used in the proof. By computing the identity type of graph isomorphisms, we obtain that $p_1^{-1} \cdot p_2$ of type $g \circ \mathsf{rot}^{k_1} = g \circ \mathsf{rot}^{k_2}$ is equivalent to having two equalities,

▷ $p : \pi_1(g \circ \mathsf{rot}^{k_1}) = \pi_1(g \circ \mathsf{rot}^{k_2})$ and

▷ $q : \mathsf{tr}^{\lambda e. \prod_{x,y \,:\, \mathsf{N}_{C_n}} \mathsf{E}_{C_n}(x,y) \to \mathsf{E}_G(e(x),e(y))}(p, \pi_2(g \circ \mathsf{rot}^{k_1})) = \pi_2(g \circ \mathsf{rot}^{k_2})$.

By characterising the identity of the Σ-types and with the previous equalities, p and q, one can get another equality r of the type in (4.4–7) for $x, y : \mathsf{N}_{C_n}$ and $e : \mathsf{E}_{C_n}(x,y)$.

$$\begin{array}{lll} ((\pi_1(g \circ \mathsf{rot}^{k_i}))(x), & (\pi_1(g \circ \mathsf{rot}^{k_i}))(y), & (\pi_2(g \circ \mathsf{rot}^{k_i}))(x,y,e)) = \\ ((((\pi_1(g))(\pi_1(\mathsf{rot}^{k_i})))(x)), & (((\pi_1(g))(\pi_1(\mathsf{rot}^{k_i})))(y)), & (((\pi_2(g))(\pi_2(\mathsf{rot}^{k_i})))(x,y,e))). \end{array} \tag{4.4–7}$$

Now since the graph homomorphism g is edge-injective, applying Definition 4.12 to the equality r, one gets an equality r' of the type below in (4.4–8). By applying Lemma 3.20 to r', we conclude that k_1 is equal to k_2 from which the required conclusion follows.

$$\begin{array}{l} ((\pi_1(\mathsf{rot}^{k_1}))(x), (\pi_1(\mathsf{rot}^{k_1}))(y), (\pi_2(\mathsf{rot}^{k_1}))(x,y,e)) = \\ ((\pi_1(\mathsf{rot}^{k_2}))(x), (\pi_1(\mathsf{rot}^{k_2}))(y), (\pi_2(\mathsf{rot}^{k_2}))(x,y,e)). \quad \square \end{array} \tag{4.4–8}$$

Lemma 4.18. The type of faces for a graph map forms a set.

Proof. Let F_1 and F_2 be two faces of a map $\mathcal{M}$. We will show that the type $F_1 = F_2$ is a proposition in Calculation (4.4–9), with the following conventions.

▷ $\mathcal{A}$ is the cyclic graph related to the face F_1,

$$\mathcal{A} :\equiv (A, (\varphi_A, n, \mathsf{isCyclic}(A, \varphi_A, n))).$$

▷ $\mathcal{B}$ is the cyclic graph related to the face F_2,

$$\mathcal{B} :\equiv (B, (\varphi_B, m, \mathsf{isCyclic}(B, \varphi_B, m))).$$

We first unfold the definitions of F_1 and F_2 in (4.4–9a), and simplifying the propositions in Equivalence (4.4–9b), namely isEdgeInj, isMapComp, and isCyclic. Then, by expanding the definitions of $\mathcal{A}$ and $\mathcal{B}$ in (4.4–9c), and simplifying the propositions in terms such as being a cyclic graph, one gets Equivalence (4.4–9d). Next, we reorder in Equivalence (4.4–9d) the tuple equalities to create an opportunity for path induction toward the application of Lemma 4.17. Now, since we want to prove that the type of faces is a set, and that itself is a proposition, the truncation elimination principle is applied to the propositions $\mathsf{isCyclic}(A, \varphi_A, n)$ and $\mathsf{isCyclic}(A, \varphi_A, n)$. Then, the graphs A and B become, respectively, C_n and C_m in Equivalence (4.4–9e). Equivalence (4.4–9f) follows from the characterisation of the identity type between tuples in a nested Σ-type.

$$
\begin{aligned}
&(F_1 = F_2) \equiv \\
&((\mathscr{A}, f, \mathsf{isEdgeInj}(f), \mathsf{isMapComp}(f)) = (\mathscr{B}, g, \mathsf{isEdgeInj}(g), \mathsf{isMapComp}(g))) \simeq && \text{(4.4–9a)} \\
&((\mathscr{A}, f) = (\mathscr{B}, g)) \equiv && \text{(4.4–9b)} \\
&((A, (\varphi_A, n, \mathsf{isCyclic}(A, \varphi_A, n))), f) = ((B, (\varphi_B, m, \mathsf{isCyclic}(B, \varphi_B, m))), g) \simeq && \text{(4.4–9c)} \\
&((A, (\varphi_A, n)), f) = ((B, (\varphi_B, m)), g) \simeq && \text{(4.4–9d)} \\
&((n, ((C_n, f), \varphi_{C_n})) = (m, ((C_m, g), \varphi_{C_m}))) \simeq && \text{(4.4–9e)} \\
&\sum_{(p : n = m)} \sum_{\left((e', -) : \sum_{(e : C_n = C_m)} \mathsf{tr}^{\lambda X.\mathsf{Hom}(X, \mathsf{Sym}(G))}(e, f) = g\right)} \mathsf{tr}^{\lambda X.\mathsf{Hom}(X,X)}(e', \varphi_{C_n}) = \varphi_{C_m}. && \text{(4.4–9f)}
\end{aligned}
$$

It only remains to show that Equivalence (4.4–9f) is a proposition. We show this by proving that each type in Equivalence (4.4–9f) is a proposition. First, we unfold the cyclic graph definition for C_n and C_m, using Definition 3.19. Second, a case analysis on n and m is performed. This approach creates four cases where n and m can be zero or positive. However, we only keep the cases where n and m are structurally equal. One can show that the other cases are imposible with an equality between n and m.

1. If n and m are zero, then, by definition, C_n and C_m are the one-point graph. In this case, the conclusion follows easily. The base type $n = m$ of the total space in Equivalence (4.4–9f) is a proposition because $\mathbb{N}$ is a set. The type $C_0 = C_0$ is a proposition, since it is contractible. The identity graph homomorphism is the unique automorphism of C_0. Lastly, because $\mathsf{Hom}(C_n, C_n)$ is a set, the remaining type of the Σ-type is a proposition, completing the proof obligations.

2. If n and m are positive, we reason similarly. The type $n = m$ is a proposition. By path induction on $p : n = m$, the second base type of the Σ-type becomes the type in (4.4–10)

$$\sum_{(e\ :\ C_n = C_n)} (\mathsf{tr}^{\lambda X.\mathsf{Hom}(X, \mathsf{Sym}(G))}(e, f) = g), \tag{4.4–10}$$

which is a proposition by Lemma 4.17. The remaining type of the Σ-type is a proposition, because $\mathsf{Hom}(C_n, C_n)$ is a set. Therefore, the Σ-type in Equivalence (4.4–9f) is a proposition as required. □

4.4.1 The finiteness property

In the previous section, we showed that the type of faces for any graph with a map forms a set; see Lemma 4.18. The type of faces for any finite graph forms a finite set. Surprisingly, our proof for this fact is somewhat technical and involves repeatedly applying finiteness results and finding some convenient equivalences. Therefore, we have split the result into several lemmas below to ease the length of the proof. In short, the following lemmas primarily focus on massaging the inner types in the type of faces to find opportunities to apply results such as Lemmas 2.12 and 2.13.

Here and below, let G be a finite graph and $\mathcal{M}$ be a graph map for G. The number of nodes and edges of G are denoted by n and m, respectively. The type of faces of G given by $\mathcal{M}$ has been spelled out in (4.4–11). The names of the variables in the type are chosen to match the names used in Definition 4.14.

$$\mathsf{Face}(G,\mathcal{M}) :\equiv \sum_{\left((A,(p,(h,!))) : \underbrace{\sum_{(A:\mathsf{Graph})}\left(\mathsf{CyclicGraph}(A) \times \sum_{(h:\mathsf{Hom}(A,\mathsf{Sym}(G)))} \mathsf{isEdgeInj}(h)\right)}_{C}\right)} \overbrace{\prod_{(x:\mathsf{N}_A)} \mathsf{isMapComp}(h)(x)}^{B((A,(p,(h,!))))}. \tag{4.4–11}$$

The notion of faces of graph maps yields an alternate approach to proving results concerning graphs, such as the formulation of Euler's characteristic number for finite graphs, see Section 6.3. Here, we prove the finiteness of the type of faces for any finite graph, from where one, in principle, can compute its cardinality, and therefore the Euler characteristic number associated to the graph.

Lemma 4.19. Suppose A is a cyclic graph according to the definition provided in Definition 3.19. Then, A is a finite graph. Furthermore, it follows that the set of nodes N_A and the set of nodes N_B have the same cardinality.

Lemma 4.20. Let $m : \mathbb{N}$ and A be a finite graph of at most m nodes. Then, $\mathsf{CyclicGraph}(A)$ is a finite type.

Proof. Unfolding the main definition for cyclic graphs and considering that the set of nodes of A is upper bounded by m, one obtains an equivalent nested Σ-type as illustrated in (4.4–12).

$$\mathsf{CyclicGraph}(A) \simeq \sum_{((n,!)\,:\,[\![m]\!])} \sum_{(\varphi\,:\,\mathsf{Hom}(A,A))} \|\,(C_n, \mathsf{rot}) = (A, \varphi)\,\| \tag{4.4–12}$$

$$\simeq \sum_{((n,!)\,:\,[\![m]\!])} \sum_{(\varphi\,:\,\mathsf{Hom}(A,A))} \left\| \sum_{(\alpha\,:\,C_n \cong A)} \mathsf{tr}^{\lambda X.\mathsf{Hom}(X,X)}(\alpha, \mathsf{rot}) = \varphi \right\|. \tag{4.4–13}$$

The two base types in the Σ-types above are clearly finite, namely, the m point type and the set of graph endo homomorphisms on A. However, less obvious is the finiteness of the type truncated. To see that, we unfold the type inside the truncation, which is a subtype of the finite type of isomorphisms between C_n and A. There are only finitely many isomorphisms between C_n and A, one for each node in A. Therefore, the type truncated in (4.4–13) is finite. □

Lemma 4.21. Let G and A be two finite graphs, then the type of edge injective homomorphisms in (4.4–14) is finite.

$$\sum_{(h\ :\ \mathsf{Hom}(A,\ \mathsf{Sym}(G)))} \mathsf{isEdgeInj}(h). \tag{4.4–14}$$

Proof. The set of graph homomorphisms between the finite graph A and $\mathsf{Sym}(G)$ is finite by Lemma 3.4. The functor Sym preserves the finiteness of G, then $\mathsf{Sym}(G)$ is also finite. On the other hand, to see that each proposition $\mathsf{isEdgeInj}(h)$ is finite, we should unfold its definition to check every inner type is finite. Let h be the pair (α, β) of type $\mathsf{Hom}(A,\ \mathsf{Sym}(G))$.

$$\begin{aligned} \mathsf{isEdgeInj}((\alpha,\beta)) :\equiv & \prod_{(x,y,\hat{x},\hat{y}\ :N_A)} \prod_{(e\ :\ \mathsf{E}_A(x,y))} \prod_{(\hat{e}\ :\ \mathsf{E}_A(\hat{x},\hat{y}))} \\ & \Big((\alpha(x), \alpha(y), \beta(x,y,e)) = \sum_{(x,y\,:\,\mathsf{N}_G)} \mathsf{E}_{\mathsf{Sym}(G)}(x,y)(\alpha(\hat{x}), \alpha(\hat{y}), \hat{e}) \Big) \\ & \to \Big((x,y,e) = \sum_{(x,y\,:\,\mathsf{N}_G)} \mathsf{E}_A(x,y)(\hat{x},\hat{y},\hat{e}) \Big). \end{aligned} \tag{4.4–15}$$

Since G is a finite graph, the type N_G is finite, and consequently, the type $\mathsf{E}_G(x,y)$ is finite for any pair of nodes x, y. Similarly, we consider the nodes and edges of the graph A. The other types in (4.4–15) are finite since they are decidable equalities on the naturals and Π-types preserve finiteness. □

Lemma 4.22. Let A be a cyclic graph, B be a finite graph with n nodes and m edges, and h be an edge-injective graph homomorphism from A to B. Then, the following type is inhabited.

$$\sum_{(((\#\mathsf{N}_A,!),(\#\mathsf{E}_A,!)) : \mathsf{isFiniteGraph}(A))} (\#\mathsf{N}_A \leq n) \times (\#\mathsf{E}_A \leq m).$$

Consider the following types for the remainder of this section. The type C as introduced in (4.4–11) and the type family D over C are defined below in (4.4–17).

$$C :\equiv \sum_{(A : \mathsf{Graph})} \left(\mathsf{CyclicGraph}(A) \times \sum_{(h : \mathsf{Hom}(A, \mathsf{Sym}(G)))} \mathsf{isEdgeInj}(h) \right). \tag{4.4–16}$$

$$D((A, \text{-})) :\equiv \sum_{(((\#\mathsf{N}_A,!),(\#\mathsf{E}_A,!)) : \mathsf{isFiniteGraph}(A))} \#\mathsf{N}_A \leq m \times \#\mathsf{E}_A \leq m, \tag{4.4–17}$$

where $\#\mathsf{N}_A$, $\#\mathsf{E}_A$ denotes the cardinality of the sets of nodes and edges in A and m is the number of edges in G.

Lemma 4.23. The total space $\Sigma_{x:C} D(x)$ in Lemma 4.24 is finite.

Proof. The type $\Sigma_{x:C} D(x)$ is equivalent to the type given in (4.4–18), which is obtained by a convenient rearrangement of the inner types in $\Sigma_{x:C} D(x)$.

$$\sum_{((A,\text{-}) : P)} \sum_{(h : \mathsf{Hom}(A, \mathsf{Sym}(G)))} \mathsf{isEdgeInj}(h). \tag{4.4–18}$$

Where

$$P :\equiv \sum_{((A,-) : Q)} \mathsf{CyclicGraph}(A) \tag{4.4–19}$$

and

$$Q :\equiv \sum_{(A : \mathsf{Graph})} D((A, -)). \tag{4.4–20}$$

We must break down the proof into three parts and apply previous lemmas. First, we prove that the type Q is finite. Second, the type P is finite. Finally, we show that the remaining type in (4.4–18) is finite, completing the proof, which follows from the closure of Σ-types under finite types.

(i) The type Q is equivalent to the type

$$\sum_{\left(((V_A,!),(E_A,!)) : \left(\sum_{(V_A : \mathcal{U})} \sum_{((\#V_A,!) : \mathsf{isFinite}(V_A))} \#V_A \leq m\right) \times \left(\sum_{(E_A : \mathcal{U})} \sum_{((\#E_A,!) : \mathsf{isFinite}(E_A))} \#E_A \leq m\right)\right)} (E_A \to V_A) \times (E_A \to V_A), \quad (4.4\text{–}21)$$

considering the equivalence in (4.4–22) which allows us to replace the type Graph (visible as the base type in (4.4–20)) by the type of set-level displayed graphs.

$$\mathsf{Graph} \simeq \sum_{(V,E : \mathcal{U})} (E \to V) \times (E \to V) \times \mathsf{isSet}(V) \times \mathsf{isSet}(E). \quad (4.4\text{–}22)$$

The resulting Σ-type in (4.4–21) has as its basis the product of two types, which are finite types by Corollary 2.19. The remainder type, $(E_A \to V_A) \times (E_A \to V_A)$, is finite since V_A and E_A are finite sets. We conclude that the type Q is finite.

(ii) The type P can be seen as the type of all cyclic graphs with a number of edges bounded by m. Since the Σ-types are closed under finite types, the type P is finite because Q is finite and $\mathsf{CyclicGraph}(A)$ is a finite type by Lemma 4.20.

(iii) The remainder type to be shown is finite of $\Sigma_{x:C} D(x)$ is

$$\sum_{((A,\text{-}) : P)} \sum_{(h : \mathsf{Hom}(A,\mathsf{Sym}(G)))} \mathsf{isEdgeInj}(h) \quad (4.4\text{–}23)$$

which is finite by Lemma 4.21. □

Lemma 4.24. The type C in (4.4–16) is finite.

Proof. The finiteness of C can be shown by applying Lemma 2.12-(*iv*) using as the input the type C and the type family D over C defined below in (4.4–17). The type C is then finite if we show that the following three conditions are satisfied:

(i) the type family D over C is a family of finite types,

(ii) the type $\Sigma_{x:C} D(x)$ is finite, and

(iii) there exists a section of type $\Pi_{x:C} D(x)$.

To show (i), let $(A, -)$ be of type C. Then, A is a cyclic graph and so the type $\mathsf{isFiniteGraph}(A)$ is contractible by Lemma 4.19. Therefore, each type in (4.4–17) is a decidable proposition, including the two inequalities related to the cardinality of the sets N_A and E_A. By closure of the Σ-types into finite types, the type $D((A, -))$ is a finite type.

To show (*ii*) and (*iii*), refer to Lemmas 4.22 and 4.23, respectively. □

Lemma 4.25. The type family B over C is a type family of finite types.

$$B((A,(p,(h,!)))) :\equiv \prod_{(x:\mathsf{N}_A)} \mathsf{isMapComp}(h)(x).$$

Proof. Let $(A,(p,(h,!)))$ be of type C. Then, the graph A is finite because is cyclic, and so is the set of nodes N_A. It remains to show that each proposition $\mathsf{isMapComp}(h)(x)$ is finite. However, such propositions are precisely the product of the two following finite types:

(i) $\mathsf{isStarComp}(h)(x) :\equiv \| \mathsf{Star}_G(\alpha(x))\| \to \|\mathsf{Star}_A(x)\|$, and

(ii) $\mathsf{isCornerComp}(h)(x)$ as defined in (4.4–5).

Since, both graphs, G and A, are locally finite, then, each star in G and A, is finite. Then, the type (i) is finite because Π-types are closed by finite types and because $\|X\|$ is finite if X is finite. Lastly, the type (ii) is finite as it is a decidable equality between edges in a finite type —the star at x in A. □

Theorem 4.26. The type of faces of a finite graph forms a finite set.

Proof. All the components of the Σ-type in the type (4.4–11) are finite types:

1. The base type C is finite by Lemma 4.24.
2. The type family B is a family of finite types according to Lemma 4.25. □

4.4.2 The boundary of a face

Each face $\mathcal{F}$ of a map $\mathcal{M}$ consisting of a cyclic graph A, a homomorphism h and some extra data as described in Definition 4.14 induced a closed walk that follows the edges of its defining polygon, which we refer to as its *boundary*.

Definition 4.27. Let $\mathcal{F}$ be a face for a map of the graph G, the boundary of $\partial\mathcal{F}$ is the subgraph of the image of the associated function, h, given in the definition of the type of $\mathcal{F}$.

$$\partial\mathcal{F} \equiv \partial((A,(h,-))) :\equiv \mathsf{Img}(h).$$

Here, $\mathsf{Img}(h)$ is the induced subgraph of G by the image of h. More specifically, it is defined as:

$$\mathsf{Img}(h) :\equiv (\Sigma_{x:\mathsf{N}_A}, (\pi_1(h))(x), \lambda x.\lambda y.\lambda e.(\pi_2(h))(x,y,e)).$$

The *degree* of a face $\mathcal{F}$ is the length of $\partial\mathcal{F}$, which is the number of nodes in A. The boundary $\partial\mathcal{F}$ can be walked in two directions with respect to the orientation given by its map.

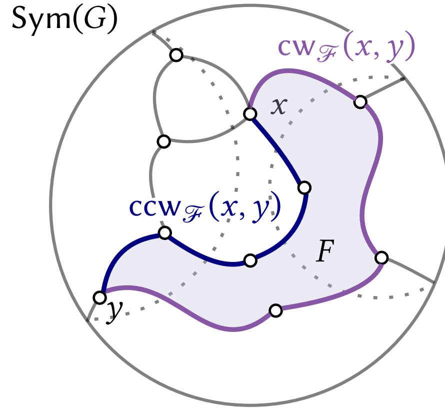


Figure 4.5: It is shown a face $\mathcal{F}$ given by $\langle A, f\rangle$ for the graph embedding $\mathsf{Sym}(G)$ given in Figure 4.2. Two quasi-simple walks exist in the underlying cyclic graph A between two different nodes x and y. Such walks are clockwise and counterclockwise closed walks in $\mathsf{Sym}(G)$, denoted by $\mathsf{cw}_{\mathcal{F}}(x, y)$ and $\mathsf{ccw}_{\mathcal{F}}(x, y)$, respectively.

As illustrated by Figure 4.5, given two different nodes x and y in $\partial\mathcal{F}$, we can connect x to y using the walk in the clockwise direction, $\mathsf{cw}_{\mathcal{F}}(x, y)$. Similarly, one can connect x to y using the walk in the counterclockwise direction, $\mathsf{ccw}_{\mathcal{F}}(x, y)$. Such walks are induced by the walks in the cyclic graph A, see Lemma 4.29.

Lemma 4.28. Supposing $x, y : \mathsf{N}_{C_n}$, the following claims hold for the cycle graph C_n.

1. The type $\mathsf{E}_{C_n}(x, y)$ is a proposition.
2. For $n > 0$, there exists an edge of type $\mathsf{E}_{C_n}(\mathsf{pred}(x), x)$ and an edge of type $\mathsf{E}_{C_n}(x, \mathsf{suc}(x))$.
3. For $n > 0$, there exists a walk going in the clockwise direction denoted by $\mathsf{cw}_{C_n}(x, y)$ from x to y.

Lemma 4.29. Supposing $x, y : \mathsf{N}_{C_n}$, the following claims hold for the graph $\mathsf{Sym}(C_n)$.

1. If $n > 1$, then the type $\mathsf{E}_{\mathsf{Sym}(C_n)}(x, y)$ is a proposition.
2. There exists an edge of type $\mathsf{E}_{\mathsf{Sym}(C_n)}(\mathsf{pred}(x), x)$ and of type $\mathsf{E}_{\mathsf{Sym}(C_n)}(x, \mathsf{suc}(x))$.
3. There exist two walks from x to y in $\mathsf{Sym}(C_n)$, denoted by $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, y)$ and $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$, respectively.
   (a) The walk $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, y)$ represents the walk in the clockwise direction from x to y.
   (b) On the other hand, the walk $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ represents the walk in the counter-

clockwise direction from x to y. In case $x = y$, the walk $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ corresponds to the trivial walk $\langle x \rangle$.

4.5 Examples of graph maps

Similarly to the discussion in Section 1.2, we will explore some examples of graph maps in this section to enhance our understanding of their structure and visual representation. Occasionally, we may deviate from type theory to provide a more comprehensible explanation of these instances. For instance, we will analyse various cases using Mathematica to enumerate different graph maps for some of these examples.

Example 4.30. For cycle graphs C_n, only one combinatorial map exists. Cyclic structures of two-point type c_1 and c_2, defined in Example 4.5, precisely induce the maps of C_n. In other words, one can obtain a map $\mathcal{M}$ can be obtained using c_1 by (4.5–24) and

$$(\mathsf{pred}, 2, |(\mathsf{ideqv}, \mathsf{refl}_{\mathsf{pred}})|) : \mathsf{Cyclic}([\![2]\!]).$$

Moreover, using function extensionality, Lemma 2.26 implies that the map induced by c_2 and the map $\mathcal{M}$ are equal.

$$\begin{aligned} \mathsf{Map}(C_n) &\equiv \prod_{(x:[\![n]\!])} \mathsf{Cyclic}(\mathsf{Star}_{C_n}(x)) \\ &\simeq \prod_{(x:[\![n]\!])} \mathsf{Cyclic}([\![2]\!]). \end{aligned} \tag{4.5–24}$$

4.5.1 Generating graph maps

In this section, the experiments were conducted using Mathematica v13 and the third-party package IGraph v0.6.5. A graph map is represented in Mathematica as *associations* (also known as dictionaries), where keys are associated with the graph's nodes and values consist of lists of nodes that form the star edges for the corresponding node (regardless of direction).

It is crucial to mention that while the illustrations display directed graphs, we have streamlined the analysis and map generation process by focussing on connected undirected graphs without loops or multiple edges in the outputs. We employ the `allMaps` function to generate graph maps, which returns a list of all possible maps for a given graph. Although the `allMaps` function can be extended to include simple directed graphs, the resulting output might be more difficult to interpret, and we put it here. Nevertheless, we provide a small example of a directed multigraph with loops in Example 4.32 to give a sense of how the `allMaps` function should be modified to include such graphs.

```
allMaps[graph_] := With[
    {emb = getInitialMap[graph]},
```

```
    Map[
    Function[comb,
      Association[
      Table[ i -> comb[[i]],
        {i, 1, Length@VertexList@graph}]]] ,
    Distribute[
      Table[
      #[[1, 1]] & /@ Union@Map[
          Cycles[{#}] &,
          Permutations@emb[v]],
        {v, VertexList@graph}], List] ]
    ];
```

Where

```
getInitialMap[graph_] := With[{
  ledges = EdgeList[graph], nodes = VertexList@graph},
    Association@Map[
      Function[node,
        node ->
        Select[
          Flatten[Select[
            Map[Union[{#[[1]], #[[2]]}] &,
            ledges], #[[1]] == node || #[[2]] == node &], 1],
          # ≠ node &]], nodes]
    ];
```

For a given graph G, it is possible to determine the number of maps for G without using the function `allMaps` on G and then calculating the length of the resulting list. Instead, we can directly compute that number by using the formula in (4.5–25), which utilises the valencies for all nodes in G and takes into account the possible cyclic orderings. The valency of a node refers to the number of edges present at its star.

$$\prod_{(x:\mathsf{N}_G)} (\text{valency}(x) - 1)!. \tag{4.5–25}$$

Example 4.31. House graph. We previously examined the house graph in Section 1.2 and made a number of observations related to its maps and drawings, as showcased in Figure 1.3. We now use Mathematica to generate all the maps.

```
HouseGraph = Graph[{
  1 -> 2,
  1 -> 3,
  2 -> 3,
  2 -> 4,
3 -> 5,
  4 -> 5
  }];
```

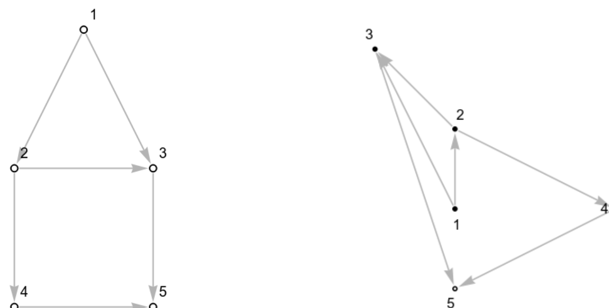


Figure 4.6: On the left, we define the house graph. On the right, there are two pictures depicting two possible drawings of this graph in the two-dimensional plane.

```
In  := allMaps[HouseGraph]
Out := {
⊲ 1 -> {2, 3}, 2 -> {1, 3, 4}, 3 -> {1, 2, 5}, 4 -> {2, 5}, 5 -> {3, 4} ⊳, (* (a) *)
⊲ 1 -> {2, 3}, 2 -> {1, 3, 4}, 3 -> {1, 5, 2}, 4 -> {2, 5}, 5 -> {3, 4} ⊳, (* (b) *)
⊲ 1 -> {2, 3}, 2 -> {1, 4, 3}, 3 -> {1, 2, 5}, 4 -> {2, 5}, 5 -> {3, 4} ⊳, (* (c) *)
⊲ 1 -> {2, 3}, 2 -> {1, 4, 3}, 3 -> {1, 5, 2}, 4 -> {2, 5}, 5 -> {3, 4} ⊳} (* (d) *)
```

The total number for the house graph is 4, calculated as $(2!/2)\cdot(3!/3)\cdot(3!/3)\cdot(2!/2)$. We find four embeddings: two non-planar and two planar, drawings (a) and (d), and (b) and (c), respectively, in Figure 4.7. The two planar maps correspond to the following combinatorial maps.

```
In  := Select[allMaps[HouseGraph], IGPlanarQ]
Out := {
⊲1 -> {2, 3}, 2 -> {1, 3, 4}, 3 -> {1, 5, 2}, 4 -> {2, 5}, 5 -> {3, 4}⊳,
⊲1 -> {2, 3}, 2 -> {1, 4, 3}, 3 -> {1, 2, 5}, 4 -> {2, 5}, 5 -> {3, 4}⊳ }
```

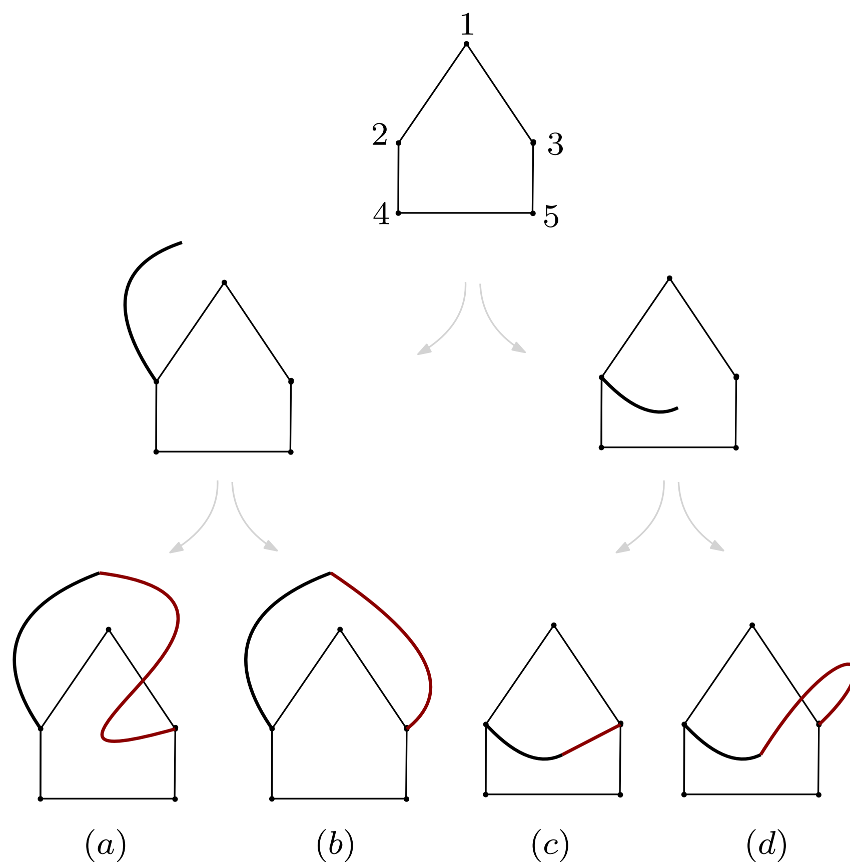


Figure 4.7: From top to bottom, we demonstrate a step-by-step approach to visualise graph maps for the house graph. We use the initial embedding at the top as a skeleton, and in each step, we consider adding new edges to complete the drawing.

Example 4.32. Bouquet B_2. A graph consisting of a single node and n loop edges is referred to as an n-bouquet, denoted by B_n. To enumerate the maps of B_2, we can label the edges of its sole star as (xin, xout, yin, yout). It is important to note that reflection is not treated as symmetry here. Consequently, we identify six distinct combinatorial maps for B_2, as depicted in Figure 4.8.

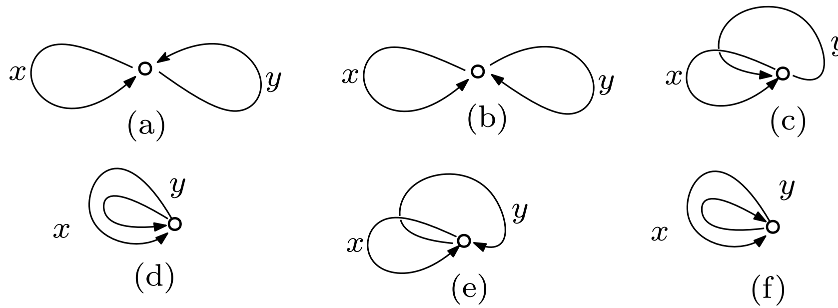


Figure 4.8: The six possible maps of the bouquet B_2.

For the 4-element set (xin, xout, yin, yout), each distinct cyclic permutation generates a map. Therefore, we should employ the `Cycles` function in Mathematica to explicitly indicate that all these permutations are cyclic.

```
CP = Union[Cycles[{#}] & /@ Permutations@Range[4]];
```

The six distinct cyclic permutations corresponding to the illustrations in Figure 4.8 are presented below.

```
{ Cycles[{{xin, xout, yin, yout}}], (* (a) *)
  Cycles[{{xin, xout, yout, yin}}], (* (b) *)
  Cycles[{{xin, yin, xout, yout}}], (* (c) *)
  Cycles[{{xin, yin, yout, xout}}], (* (d) *)
  Cycles[{{xin, yout, xout, yin}}], (* (e) *)
  Cycles[{{xin, yout, yin, xout}}]} (* (f) *)
```

Example 4.33. 2-Grid graph. The grid graph is usually presented as an undirected graph with n nodes arranged in a regular n-gon, with each node connected to its two neighbours. Here, we consider the directed grid graph with n nodes arranged in a regular n-gon, as illustrated for $n = 6$ in Figure 4.9. The following Mathematica code generates all the maps associated with this graph.

```
In[] := grid2 = Graph[{
  1 -> 2,
  2 -> 3,
  1 -> 4,
  4 -> 3,
  4 -> 5,
  5 -> 6,
  3 -> 6
  }
  ];
```

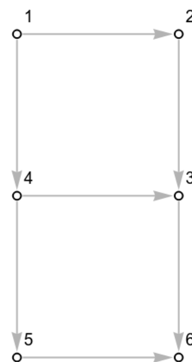


Figure 4.9: The grid graph with 6 nodes and 7 edges.

```
In[] := allMaps[grid2]
Out[] :=
{<|1 -> {2, 4}, 2 -> {1, 3}, 3 -> {2, 4, 6}, 4 -> {1, 3, 5}, 5 -> {4, 6}, 6 -> {3, 5}|>,
<|1 -> {2, 4}, 2 -> {1, 3}, 3 -> {2, 4, 6}, 4 -> {1, 5, 3}, 5 -> {4, 6}, 6 -> {3, 5}|>,
<|1 -> {2, 4}, 2 -> {1, 3}, 3 -> {2, 6, 4}, 4 -> {1, 3, 5}, 5 -> {4, 6}, 6 -> {3, 5}|>,
<|1 -> {2, 4}, 2 -> {1, 3}, 3 -> {2, 6, 4}, 4 -> {1, 5, 3}, 5 -> {4, 6}, 6 -> {3, 5}|>}
```

5

Walks and Spherical Maps

In Chapter 4, we work with combinatorial maps to represent graph embeddings in surfaces up to isotopy. The surface in which the graph is embedded remains implicit in this approach, eliminating the need for explicit specification in HoTT. This chapter presents a refinement of one characterisation of graph maps in the sphere, called spherical maps, for connected and directed multigraphs with discrete node sets. A combinatorial notion of homotopy for walks and the normal form of walks under a reduction relation are introduced. The first characterisation of spherical maps states that a graph can be embedded in the sphere if any pair of walks with the same endpoints are merely walk-homotopic. The refinement of this definition filters out any walk with inner cycles. As we prove in one of the lemmas, if a spherical map is given for a graph with a discrete node set, then any walk in the graph is merely walk homotopic to a normal form.

5.1 The type of walks

The notion of walks, as introduced in Definition 3.12, plays an essential role in graph theory. Many algorithms using graph data structures are based on this concept. This section provides the necessary tools to develop two normalisation algorithms for walks, as seen in Theorems 5.48 and 5.38. These algorithms are used, for example, in Lemma 5.49.

5.1.1 Structural induction for walks

By *structural induction* or *pattern matching* on a walk, we will refer to the elimination principle of the inductive type in Definition 3.12. An induction principle allows us to define outgoing functions from a type to a type family. For example, if we want to use the induction principle to inhabit a predicate on the type of walks, $P : \Pi\{x\ y : \mathsf{N}_G\}.\mathsf{W}_G(x,y) \to \mathcal{U}$, one can inhabit (5.1−1). Given a walk $w : \mathsf{W}_G(x,y)$, to construct a term of type $P(w)$, the base case must first be constructed, i.e., give a term of type $P(\langle x \rangle)$, for every $x : \mathsf{N}_G$. Subsequently, we must prove the case for composite walks, i.e., $P(e \odot w)$. To show this, $P(w)$ is assumed for any walk w, and we construct a term of type $P(e \odot w)$ from this assumption. Thus, one gets $P(w)$ for any walk w. Another induction principle for walks is stated in Theorem 5.4.

$$\begin{aligned} \prod_{(x:\mathsf{N}_G)} P(\langle x \rangle) \times \prod_{(x,y,z:\mathsf{N}_G)} \prod_{(e:\mathsf{E}_G(x,y))} \prod_{(w:\mathsf{W}_G(y,z))} P(w) \to P(e \odot w) \\ \to \prod_{(x,y:\mathsf{N}_G)} \prod_{(w:\mathsf{W}_G(x,y))} P(w). \end{aligned} \tag{5.1−1}$$

The *composition*, also called concatenation, of walks is an associative binary operation on walks defined by structural induction on its left argument. Given walks $p : \mathsf{W}_G(x,y)$ and $q : \mathsf{W}_G(y,z)$, we refer to their composition as the *composite* denoted by $p \cdot q$. The node y is called the *joint* of the composition. The *length* of the walk w is denoted by $\mathsf{length}(w)$ and represents the number of edges used to construct w. A trivial walk has length zero, whilst a walk $(e \odot w)$ has one more length than w. We display a point to represent trivial walks and with a normal arrow to represent positive length walks, as illustrated in Figure 5.1.

Lemma 5.1. The type of walks forms a set.

Proof. One can show that the type $\mathsf{W}(x,y)$ is equivalent to $\Sigma_{n:\mathbb{N}} \hat{W}(n,x,y)$ with $\hat{W}$ defined as follows.

$$\hat{W} : \mathbb{N} \to \mathsf{N}_G \to \mathsf{N}_G \to \mathcal{U} \tag{5.1−2a}$$

$$\hat{W}(0,x,y) :\equiv (x = y), \tag{5.1−2b}$$

$$\hat{W}(S(n),x,y) :\equiv \sum_{(k:\mathsf{N}_G)} \mathsf{E}_G(x,k) \times \hat{W}(n,k,y). \tag{5.1−2c}$$

It suffices to show that the type $\hat{W}(n,x,y)$ forms a set for $n : \mathbb{N}$, which will be proven by induction on n. If $n = 0$, one obtains the proposition $x = y$ which is a set. Consequently, we must now show that the type in (5.1−2c) is a set. By the graph definition, the

base type N_G and E_G are both sets. Thus, one only requires that $\hat{W}(n,k,y)$ forms a set, which is precisely the induction hypothesis. □

Although it is not included in the formalisation of this work, one can show that the type of walks forms a category. If Graph is the category of graphs using Definition 3.1 and $\mathscr{C}$ is the category of small categories, there is a functor $R : \mathsf{Graph} \to \mathscr{C}$ mapping every graph G to its *free* pre-category. The object set of $R(G)$ is N_G, and the morphisms correspond to the collection of all possible walks in G. By Lemma 5.1, it follows that $R(G)$ is a small category. Let L be the forgetful functor from $\mathscr{C}$ to Graph. Then, L is the left adjoint of R. The *graph of walks* of G is generated by using the endofunctor $W : \mathsf{Graph} \to \mathsf{Graph}$, the monad from the composite $L \circ R$.

5.1.2 A well-founded order for walks

Structural induction is a particular case of a more general induction principle to define recursive programs called *well-founded* or Noetherian induction. Note that, for the structural induction principle, one must always guarantee that every argument in a recursive call in the program is strictly smaller than its arguments. However, there is no reason to believe that this will always be the case.

In constructive mathematics, a binary relation R on a set A is *well-founded* if every element of A is *accessible*. An element $a : A$ is accessible by R, if $b : A$ is accessible for every bRa (Nordström 1988; Univalent Foundations Program 2013, §10.3). Then, if a has the property that there is no b such that bRa, then a is vacuously accessible. If $(\leq)$ represents the *less or equal than* relation on the natural numbers, then the number zero is vacuously accessible by $\leq$ on $\mathbb{N}$.

Let us define a well-founded order for walks in a graph by considering their lengths, from where the well-founded induction for walks follows, see Theorem 5.4.

Definition 5.2. Given $p, q : \mathsf{W}_G(x,y)$ for $x, y : \mathsf{N}_G$, the relation $(\preccurlyeq)$ states that $p \preccurlyeq q$ when $\mathsf{length}(p) \leq \mathsf{length}(q)$.

Lemma 5.3. The relation $(\preccurlyeq)$ on $\Sigma_{x,y:\mathsf{N}_G}\mathsf{W}_G(x,y)$ is well–founded.

Proof. It follows from the fact that the poset $(\mathbb{N}, \leq)$ is well-founded. □

We refer to the following lemma as the *well-founded induction principle for walks* induced by Definition 5.2.

Theorem 5.4. Suppose the following is given,

1. a predicate P of type $\Sigma_{x,y:\mathsf{N}_G}\mathsf{W}_G(x,y) \to \mathscr{U}$ such that,

2. given (a, b, q) of type $\Sigma_{x,y:\mathsf{N}_G}\mathsf{W}_G(x, y)$, if $P(p)$ for each walk $p : \mathsf{W}_G(x', y')$ with $x', y' : \mathsf{N}_G$ and $p \preccurlyeq q$, then $P(a, b, q)$.

Then, given any walk $w : \mathsf{W}_G(x, y)$ and $x, y : \mathsf{N}_G$, we have $P(x, y, w)$.

Remark 5.5. The induction principle stated in Theorem 5.4 using Lemma 5.3 is equivalent to performing induction on the length of the walk.

Theorems 5.38 and 5.48 define algorithms for which many of their recursive calls are on subwalks of the input walk. A *subwalk* of a walk w is a contiguous subsequence of edges in w. Subwalks are not structurally smaller than their corresponding walk unless one takes, for example, the subwalk w or e for the composite walk $(e \odot w)$. Excluding the previous case, to deal with other subwalk cases, we can use the *well-founded* induction principle given in Theorem 5.4.

5.1.3 Walk splitting

In this subsection, a function to split/divide a walk w from x to z into subwalks, w_1 and w_2, is given. Such a division of w, of (5.1–3), is handy e.g., for proving statements where the induction is not on the structure but on the length of the walk.

Let x, y, z be variables for nodes in G and let w be a walk from x to z, unless stated otherwise. We refer to the walk w_1 in (5.1–3) as a prefix of w and w_2 as the corresponding suffix given w_1.

$$\sum_{(y:\mathsf{N}_G)}\ \sum_{(w_1:\mathsf{W}_G(x,y))}\ \sum_{(w_2:\mathsf{W}_G(y,z))} (w = w_1 \cdot w_2). \tag{5.1–3}$$

Definition 5.6. Given two walks p and q with the same head, one says that p is a *prefix* of q if the type $\mathsf{Prefix}(p, q)$ is inhabited.

$$\begin{aligned}
\textbf{data}\ \mathsf{Prefix}\ &:\ \Pi\{x, y, z\}\,.\,\mathsf{W}_G(x, y) \to \mathsf{W}_G(x, z) \to \mathcal{U}\\
\mathsf{by\text{-}head}\ &:\ \Pi\{x\, y\}\,.\,\Pi\{w : \mathsf{W}_G(x, y)\}\,.\,\mathsf{Prefix}(\langle x\rangle, w)\\
\mathsf{by\text{-}edge}\ &:\ \Pi\{x\, y\, z\, k\}\,.\,\Pi\{e : \mathsf{E}_G(x, y)\}.\,\Pi\{p : \mathsf{W}_G(y, z)\}\,.\,\Pi\{q : \mathsf{W}_G(y, k)\}.\\
&\quad\ \mathsf{Prefix}(p, q) \to \mathsf{Prefix}(e \odot p, e \odot q)
\end{aligned}$$

Lemma 5.7. Given a prefix w_1 for a walk w, we can prove that there is a term of (5.1–4) named $\mathsf{suffix}(w_1, w, t)$, referring to as the suffix of w given w_1, where $t : w = w_1 \cdot w_2$.

$$\sum_{(w_2:\mathsf{W}_G(y,z))} (w = w_1 \cdot w_2). \tag{5.1–4}$$

Proof. For brevity, we skip the trivial cases for w_1 and w. The remaining cases are proved by induction; first, on w_1, and second, on w. The resulting non-trivial case occurs when $w_1 = e \odot p$, $w = e \odot q$ and $t : \mathsf{Prefix}(p, q)$ for two walks p and q. By the induction hypothesis applied to p, q, and t, the term $\mathsf{suffix}(p, q, t)$ is obtained, from which one gets the suffix walk w_2 along with a proof $i : q = p \cdot w_2$. Thus, the required term is the pair $(w_2, \mathsf{ap}(e \odot \text{-}, i))$. □

We now encode the case where the walk w is divided at the first occurrence of the node y, using the type family $\mathsf{SplitAt}(w, y)$ defined in Definition 5.8. The corresponding method to inhabit the type $\mathsf{SplitAt}(w, y)$ is the function given in Lemma 5.9, assuming the node set in the graph is discrete. This walk-splitting encoding is implicitly used in several parts of the proof of Theorem 5.48.

Definition 5.8. The type $\mathsf{SplitAt}(w, y)$ is the inductive type defined as:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{SplitAt}\ \{x\, z\}(w : \mathsf{W}_G(x, z))\,(y : \mathsf{N}_G) : \mathcal{U}\\
&\quad \mathsf{nothing} : \Pi\{x\, y\}.\Pi\{w : \mathsf{W}_G(x, y)\}.\,(y \notin w) \to \mathsf{SplitAt}(w, y)\\
&\quad \mathsf{just} : \Pi\{x\, y\}.\Pi\{w : \mathsf{W}_G(x, y)\}.\,(p : \mathsf{W}_G(x, y))\\
&\qquad\qquad \to \mathsf{Prefix}(p, w) \to (y \notin p) \to \mathsf{SplitAt}(w, y)
\end{aligned}
$$

Lemma 5.9. The type $\mathsf{SplitAt}(w, y)$ is inhabited if the node set of the graph is discrete.

Proof. By induction on the structure of the walk.

1. If the walk is trivial, then the required term is nothingid, as by definition, $y \notin \mathbb{0}$.

2. If the walk is the composite $(e \odot w)$ with $e : \mathsf{E}_G(x, y')$ and $w : \mathsf{W}_G(y', z)$, we ask whether y is equal to x or not.

   (a) If $y = x$, then the required term is $\mathsf{just}(\langle y \rangle, \mathsf{head}, \mathsf{id})$.

   (b) If $y \neq x$, then, by the induction hypothesis on w and y, the following cases need to be considered.

      i. If the case is nothing, then there is enough evidence that $y \notin w$ and we use for the required term the nothing constructor.

      ii. Otherwise, there is a prefix w_1 for w and a proof $r : y \notin w_1$. Using r and the fact $x \neq y$, we can construct $r' : y \notin (e \odot w_1)$. Then, the term that we are looking for is $\mathsf{just}(e \odot w_1, \mathsf{by\text{-}edge}(p), r')$ of type $\mathsf{SplitAt}(e \odot w, y)$, as required in the conclusion. □

5.2 The type of quasi-simple walks

In this subsection, we characterise walks with shapes as in Figure 5.1 and refer to such as *quasi-simple* walks in Definition 5.12.

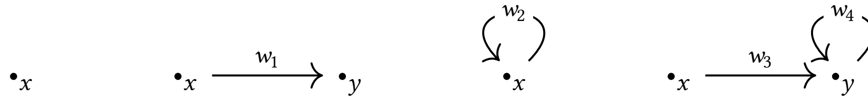


Figure 5.1: The arrows in the picture can represent edges or walks of a positive length. In the sense of Definition 5.12, a quasi-simple walk can only be one of these kinds: i) one-point walk ii) path iii) loop without inner node repetitions, or iv) composite walk between a path and a quasi-simple walk of kind iii. The walks w_3 and w_4 only share the occurrence of y that is explicitly shown.

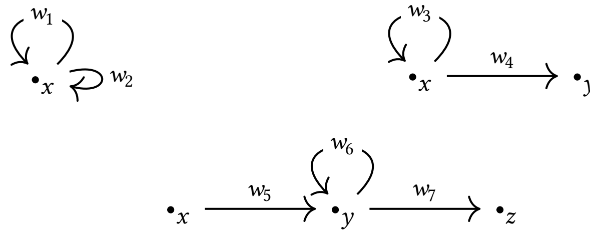


Figure 5.2: These are three examples of walks that are not quasi-simple in the sense of Definition 5.12. The walks w_1 and w_2 only share the node x, and the same happens with the walks w_3 and w_4. The walks w_5, w_6 and w_7 only share the node y. The walks w_i for i from 1 to 7 are nontrivial walks.

The notion of a quasi-simple walk will be used to introduce a reduction relation on the set of walks to remove their inner loops; see Definition 5.29. A notion related to the definition of a quasi-simple walk is that of a path (Diestel 2012). The usual graph-theoretical notion of a (simple) *path* is a walk without repeated nodes. Here, quasi-simple walks are introduced, since paths are not suitable in our description of graph maps in Section 5.5. There, the totality of walks is considered, which includes closed walks, also called loops. For graph maps in the sphere, we found out that the type of quasi-simple walks can replace the type of walks under certain conditions. Quasi-walks are conveniently defined in a way that permits their end to appear at most twice in the walk.

To define quasi-simpleness for walks, we introduce an unconventional relation, denoted by $(x \in w)$, meaning that the node x is in the walk w, and it is not the last, see Definition 5.10. $(x \in w)$ is a proposition, and decidable if the walks belong to graphs with a discrete node-set. Consequently, Lemma 5.17 shows that being quasi-simple is also a decidable proposition on such graphs. Quasi-simple walks play an important role in this work. They are required to give an alternative definition of graph maps in the sphere, as

stated in Definition 5.44.

Definition 5.10. Let $x, y, z : \mathsf{N}_G$ and $w : \mathsf{W}_G(x, z)$. The relation $(\in)$ on a walk w for a node y is defined as the node y that is not z but belongs to w, that is, whenever the type $(y \in w)$ is inhabited.

1. $y \in \langle z \rangle :\equiv \mathbb{0}$.
2. $y \in (e \odot w) :\equiv (y = \mathsf{source}(e)) + (y \in w)$.

Lemma 5.11. If the node set of the graph G is discrete, then the type $(x \in w)$ is decidable proposition for any node x and walk w in G.

Definition 5.12. Given $x, y : \mathsf{N}_G$, a walk in G from x to y is *quasi-simple* if $\mathsf{isQuasi}(w)$ holds.

$$\mathsf{isQuasi}(w) :\equiv \prod_{(z : \mathsf{N}_G)} \mathsf{isProp}(z \in w). \tag{5.2–5}$$

Lemma 5.13. Being quasi-simple is a proposition.

Proof. It follows since $\mathsf{isProp}(z \in w)$ is a proposition. □

Thus, Definition 5.12 presents a quasi-simple walk as a path where the end could only be present at most twice. Examples of walks that are not quasi-simple are illustrated in Figure 5.2.

Lemma 5.14. Given $x, y, z : \mathsf{N}_G$, $e : \mathsf{E}_G(x, y)$ and a quasi-simple walk $w : \mathsf{W}_G(y, z)$, if $x \notin w$ then the walk $(e \odot w)$ is quasi-simple.

Proof. Given a node r, we must show that $r \in (e \odot w)$ is a proposition. That is equivalent to showing that the type $(r = x) + (r \in w)$ is a proposition. The coproduct of mutually exclusive propositions is a proposition. Then, remember that $r = x$ is a given proposition and that the type $(r \in w)$ is also a proposition, since the walk w is quasi-simple by hypothesis. Thus, it remains to show that there is no term (p, q) where $p : (r = x)$ and $q : (r \in w)$. A contradiction arises, since by hypothesis $x \notin w$ but from $\mathsf{tr}^{\lambda z \to z \in w}(p)(q) : x \in w$. □

Lemma 5.15. Given $x, y, z : \mathsf{N}_G$, $e : \mathsf{E}_G(x, y)$, and a walk $w : \mathsf{W}_G(y, z)$, if the walk $(e \odot w)$ is a quasi-simple walk then w is also a quasi-simple walk.

Proof. Given any node $u : \mathsf{N}_G$ and two proofs $p, q : u \in w$, we must show that $p = q$. By definition, $\mathsf{inr}(p)$ and $\mathsf{inr}(q)$ are proofs of that $u \in (e \odot w)$. Because $(e \odot w)$ is a quasi-simple walk, the equality $\mathsf{inr}(p) = \mathsf{inr}(q)$ holds. The constructor inr is an injective function, and one therefore obtains $p = q$ as required. □

Corollary 5.16. Trivial and one-edge walks are quasi-simple walks.

Lemma 5.17. If the node set of the graph is discrete, then being quasi-simple for a walk is a decidable proposition.

Proof. Let $x, z : \mathsf{N}_G$ and $w : \mathsf{W}_G(x, z)$, we want to show that isQuasi(w) is decidable. The proof is by induction on the structure of w.

1. If w is trivial then, by Corollary 5.16, the walk w is quasi-simple.
2. If w is the composite walk $(e \odot w')$ for $e : \mathsf{E}_G(x, y)$ and $w' : \mathsf{W}_G(y, z)$, we recursively ask whether the walk w' is quasi-simple or not.
   (a) If w' is not quasi-simple, then w is not quasi-simple by the contrapositive of Lemma 5.15.
   (b) If w' is quasi-simple, then we ask if $x \in w'$. If so, then w is not quasi-simple. Otherwise, that would contradict the definition of quasi-simpleness, as the node x would appear twice in w. Now, if $x \notin w'$, one obtains that w is quasi-simple by Lemma 5.14. □

5.2.1 The finiteness property

This subsection presents a proof that the collection of quasi-simple walks in a finite graph G constitutes a finite set (Theorem 5.26). The proof hinges on demonstrating the finiteness of an equivalent type to (5.2–6).

For clarity, we define the standard type with n elements, denoted by $[\![n]\!]$, inductively as follows:

- ▷ $[\![0]\!] :\equiv \mathbb{0}$
- ▷ $[\![1]\!] :\equiv \mathbb{1}$
- ▷ $[\![n+1]\!] :\equiv [\![n]\!] + \mathbb{1}$

To establish the desired equivalence (Lemma 5.25), it is necessary first to derive some intermediate results.

$$\sum_{(w : \mathsf{W}_G(x,y))} \mathrm{isQuasi}(w). \qquad (5.2\text{–}6)$$

Lemma 5.18. Given any walk $w : \mathsf{W}_G(x,z)$ of length n, then

$$[\![n]\!] \simeq \sum_{(y:\mathsf{N}_G)} (y \in w). \tag{5.2–7}$$

Proof. By induction on the structure of w.

1. If the walk is trivial, the required equivalence follows from the type equivalence between $\mathbb{0}$ and $\Sigma_{z:\mathsf{N}_G}\mathbb{0}$.

2. If the walk is $(e \odot w)$ for $e : \mathsf{E}_G(x,y)$ and $w : \mathsf{W}_G(y,z)$, the equivalence is established by the following calculation. Let n be the length of w.

$$\sum_{(y:\mathsf{N}_G)} (y \in (e \odot w)) \equiv \sum_{(y:\mathsf{N}_G)} (y = x) + (y \in w) \tag{5.2–8a}$$

$$\simeq \sum_{(y:\mathsf{N}_G)} (y = x) + \sum_{(y:\mathsf{N}_G)} (y \in w) \tag{5.2–8b}$$

$$\simeq \mathbb{1} + \sum_{(y:\mathsf{N}_G)} (y \in w) \tag{5.2–8c}$$

$$\simeq \mathbb{1} + [\![n]\!] \tag{5.2–8d}$$

$$\equiv [\![n+1]\!]. \tag{5.2–8e}$$

Equivalence (5.2–8a) is accomplished by Definition 5.10. Σ-type distributes coproducts as in Equivalence (5.2–8b). We can simplify in Equivalence (5.2–8c) because the type $\Sigma_{y:\mathsf{N}_G}(y = x)$ is contractible. Note that the inner path is fixed, and it is then equivalent to the unit type. Equivalence (5.2–8d) is by the induction hypothesis applied to w. Equivalence (5.2–8e) is accomplished by the definition of $[\![n]\!]$ using the coproduct definition. □

Lemma 5.19. Given $x, y, z : \mathsf{N}_G$, and $w : \mathsf{W}_G(x,y)$ the type $(z \in w)$ is a finite set if the node set of G is discrete.

Proof. By induction on the structure of w: in case the walk is trivial, the type in question is finite as it is equal to the empty type by definition. In the composite walk case, $z \in (e \odot w)$, we must prove that the type $(z = x) + (z \in w)$ is finite. Note that the former is finite by Corollary 2.15. By the induction hypothesis, the type $z \in w$ is finite. The required conclusion then follows, since finite sets are closed under coproducts. □

We can now prove that for finite graphs, there exists a finiteness property for the collection of all quasi-simple walks, derived from the finiteness of the set of quasi-simple walks of a fixed length n for $n : \mathbb{N}$.

Definition 5.20. Given $x, y : \mathsf{N}_G$ and $n : \mathbb{N}$, the type qswalk collects all quasi-simple walks of a fixed length n.

$$\mathsf{qswalk}(n, x, y) :\equiv \sum_{(w : \mathsf{W}_G(x,y))} \mathsf{isQuasi}(w) \times (\mathsf{length}(w) = n).$$

Lemma 5.21. Given a graph G, $n : \mathbb{N}$, and $x, z : \mathsf{N}_G$, the following equivalence holds.

$$\mathsf{qswalk}(S(n), x, z) \simeq \sum_{(y : \mathsf{N}_G)} \sum_{(e : \mathsf{E}_G(x,y))} \sum_{(w : \mathsf{qswalk}(n,y,z))} (x \notin w). \tag{5.2–9}$$

Proof. The back-and-forth functions are extensions of the functions derived from Lemmas 5.14 and 5.15. □

Lemma 5.22. Given a finite graph G, $x, y : \mathsf{N}_G$ and $n : \mathbb{N}$, the type $\mathsf{qswalk}(n, x, y)$ in Definition 5.20 is a finite set.

Proof. It suffices to show that the type $\mathsf{qswalk}(n, x, y)$ is finite. The proof is made by induction on n.

1. If $n = 0$, the type defined by $\mathsf{qswalk}(0, x, z)$ is equivalent to the identity type $x = y$, as the only walks of length zero are the trivial walks. Given that the set of nodes is discrete, the path space $x = y$ is finite by Corollary 2.15.

2. Otherwise, given $x, z : \mathsf{N}_G$, we must prove that the type $\mathsf{qswalk}(S(n), x, z)$ is finite, for $n : \mathbb{N}$, assuming that $\mathsf{qswalk}(n, x, z)$ is finite. This is equivalent to showing that the equivalent type given by Equivalence (5.2–9) is finite. The required conclusion follows by Lemma 2.13, as each type of the Σ-type in the right-hand side of the equivalence in Equivalence (5.2–9) is finite. The set N_G and the sets by E_G are each finite, as G is a finite graph. The type $\mathsf{qswalk}(n, y, z)$ is finite by induction hypothesis. Lastly, any decidable proposition is finite i.e., $(x \notin w')$ is finite. □

Lemmas 5.24 and 5.25 prove the fact mentioned earlier on the node repetition condition in a quasi-simple walk. A node can only appear once in a quasi-simple walk, unless the node is the end of the walk. From now on, unless otherwise stated, we will refer to n as the cardinality of N_G whenever the node set of the graph G is finite. The number of nodes in any quasi-simple walk is bounded by $n + 1$.

Lemma 5.23. Let G be a finite graph. Then (5.2–10) is a finite set.

$$\sum_{(x,y:\mathsf{N}_G)} \sum_{(m:[\![n+1]\!])} \mathsf{qswalk}(m,x,y). \tag{5.2–10}$$

Proof. The conclusion follows since finite sets are closed under Σ-types. N_G is finite, since G is a finite graph. $[\![n+1]\!]$ is finite. The type $\mathsf{qswalk}(m,x,y)$ is finite by Lemma 5.22. □

Lemma 5.24. Given a graph G with finite node set of cardinality n, $x, y : \mathsf{N}_G$ and a quasi-simple walk $w : \mathsf{W}_G(x,y)$ of length m, then it holds that $m \leq n$.

Proof. It suffices to generate an embedding between the finite set $[\![m]\!]$ and the finite node set in G. Such an embedding is the projection function $\pi_1 : \Sigma_{x:\mathsf{N}_G}(x \in w) \to \mathsf{N}_G$. Note that the domain of the function π_1 is equivalent to $[\![m]\!]$ by Lemma 5.18. □

Now, even when the type of walks forms an infinite set, thanks to Lemma 5.24 and Theorem 5.26, we will be able to prove that for any nodes x and y, the collection of quasi-simple walks from x to y forms a finite set as long as the graph is finite.

Lemma 5.25. Given a graph G with a finite node set of cardinality n and $x, y : \mathsf{N}_G$, the following equivalence holds.

$$\sum_{(w:\mathsf{W}_G(x,y))} \mathsf{isQuasi}(w) \simeq \sum_{(m:[\![n+1]\!])} \mathsf{qswalk}(m,x,y). \tag{5.2–11}$$

Proof. Apply Lemma 5.24. □

It is not immediately clear that quasi-simple walks forms a finite set, even when the graph is finite. A quasi-simple walk can contain a loop at its terminal node. One might think there are infinitely many walks if each walk loops at its terminal nodes. However, by constraining walks to be quasi-simple, we obtain the finiteness property.

Theorem 5.26. The quasi-simple walks of a finite graph G forms a finite set. In other words, the following type is inhabited.

$$\mathsf{isFinite}\left(\sum_{(x,y:\mathsf{N}_G)} \sum_{(w:\mathsf{W}(x,y))} \mathsf{isQuasi}(w)\right). \tag{5.2–12}$$

Proof. The conclusion clearly follows from Lemmas 5.23 and 5.25, since finite sets are closed under type equivalences and Σ-types by Lemma 2.13. □

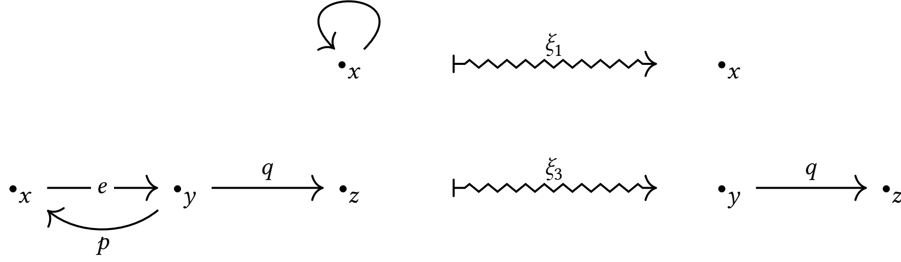


Figure 5.3: The rules ξ_1 and ξ_3 of the loop-reduction relation in (5.3–13).

5.3 Normal forms for walks

In this subsection, a reduction relation in Definition 5.29 is established on the set of walks of equal endpoints. Some cases considered by such a relation are illustrated in Figure 5.3. This relation provides a way to remove loops from walks in a graph with a discrete set of nodes. The notion of normal form for walks presented in this work is based on the loop reduction relation in Definition 5.34.

The following definitions establish a few type families to encode walks of a certain basic structure. For example, nontrivial walks and loops which are necessary for the formalisation.

Definition 5.27. Let $x, y : \mathsf{N}_G$ and $w : \mathsf{W}_G(x, y)$.

1. The walk w is a loop whenever the head is equal to the end, i.e., $\mathsf{Loop}(w)$.

$$\begin{aligned} \textbf{data}\ \mathsf{Loop} &: \Pi\{x, y\}.\, \mathsf{W}_G(x, y) \to \mathcal{U} \\ \mathsf{is\text{-}loop} &: \Pi\{x\, y\}.\, \Pi\{w : \mathsf{W}_G(x, y)\}.\, x = y \to \mathsf{Loop}(w) \end{aligned}$$

2. The walk w is trivial if its length is zero, i.e., $\mathsf{Trivial}(w)$.

$$\begin{aligned} \textbf{data}\ \mathsf{Trivial} &: \Pi\{x, y\}.\, \mathsf{W}_G(x, y) \to \mathcal{U} \\ \mathsf{is\text{-}trivial} &: \Pi\{x\, y\}.\, \Pi\{w : \mathsf{W}_G(x, y)\}.\, \mathsf{length}(w) = 0 \to \mathsf{Trivial}(w) \end{aligned}$$

3. A walk w is not trivial, if it has one edge at least, i.e., $\mathsf{NonTrivial}(w)$.

$$\begin{aligned} \textbf{data}\ \mathsf{NonTrivial} &: \Pi\{x, y\}.\, \mathsf{W}_G(x, y) \to \mathcal{U} \\ \mathsf{has\text{-}edge} &: \Pi\{x\, y\, z\}.\, \Pi\{w : \mathsf{W}_G(y, z)\}.\, (e : \mathsf{E}_G(x, y)) \\ &\quad \to \mathsf{NonTrivial}(e \odot w) \end{aligned}$$

4. A walk w does not *reduce* if $\mathsf{NoReduce}(w)$.

$$\begin{aligned} \textbf{data}\ \mathsf{NoReduce} &: \Pi\{x, y\}.\, \mathsf{W}_G(x, y) \to \mathcal{U} \\ \mathsf{is\text{-}dot} &: \Pi\{x\}.\, \mathsf{NoReduce}(\langle x \rangle) \\ \mathsf{is\text{-}edge} &: \Pi\{x\, y\}.\, \Pi\{e : \mathsf{E}_G(x, y)\}.\, (x \neq y) \to \mathsf{NoReduce}(e \odot \langle y \rangle) \end{aligned}$$

5. A walk w is not a trivial loop if $\mathsf{NonTrivialLoop}(w)$.

$$
\begin{aligned}
\textbf{data}\ \mathsf{NonTrivialLoop} \ &:\ \Pi\{x,y\}\,.\,\mathsf{W}_G(x,y) \to \mathcal{U} \\
\mathsf{is\text{-}loop} \ &:\ \Pi\{x\,y\,z\}.\{e\ :\ \mathsf{E}_G(x,y)\}\,.\,(p\ :\ x = z)\ \to\ (w\ :\ \mathsf{W}_G) \\
&\quad \to\ \mathsf{NonTrivialLoop}(e \odot w)
\end{aligned}
$$

Lemma 5.28. Given $x, y\ :\ \mathsf{N}_G$ and $u\ :\ \mathsf{W}_G(x,y)$, the following claims hold.

1. If $x \neq y$ then $\mathsf{NonTrivial}(u)$.
2. If $\mathsf{NonTrivial}(u)$ then $x \in u$.
3. Given $z\ :\ \mathsf{N}_G$, if $\mathsf{NonTrivial}(u)$ and $v\ :\ \mathsf{W}_G(y,z)$ then $\mathsf{NonTrivial}(u \cdot v)$.

Remember that a reduction relation R on a set M is an irreflexive binary relation on M. If R is a reduction relation, we use xRy to refer to the pair (x, y) in R. If xRy then one says that x *reduces* to y or simply x *reduces.*

Definition 5.29. The *loop-reduction* relation $(\rightsquigarrow)$ on walks is (5.3–13).

$$
\begin{aligned}
\textbf{data}\ (\rightsquigarrow)\ &:\ \Pi\{x, y\ :\ \mathsf{N}_G\}.\mathsf{W}_G(x,y) \to \mathsf{W}_G(x,y) \to \mathcal{U} \\
\xi_1\ &:\ \Pi\{x\ y\}\,.\,(p\ :\ \mathsf{W}_G(x,y))\,(q\ :\ \mathsf{W}_G(x,y)) \\
&\quad \to \mathsf{NonTrivialLoop}(p) \to \mathsf{Trivial}(q) \\
&\quad \to p \rightsquigarrow q \\
\xi_2\ &:\ \Pi\{x\ y\ z\}\,.\,(e\ :\ \mathsf{E}_G(x,y))\,(p, q\ :\ \mathsf{W}_G(y,z)) \\
&\quad \to \neg\,\mathsf{Loop}(e \odot p) \to x \neq y \\
&\quad \to (p \rightsquigarrow q) \to (e \odot p) \rightsquigarrow (e \odot q) \\
\xi_3\ &:\ \Pi\{x\ y\ z\}\,.\,(e\ :\ \mathsf{E}_G(x,y))\,(p\ :\ \mathsf{W}_G(y,x)) \\
&\quad \to (q\ :\ \mathsf{W}_G(x,z)) \\
&\quad \to \neg\,\mathsf{Loop}((e \odot p)\cdot q) \to \mathsf{Loop}(e \odot p) \\
&\quad \to \mathsf{NonTrivial}(q) \\
&\quad \to (w\ :\ \mathsf{W}_G(x,z)) \to w = (e \odot p)\cdot q \\
&\quad \to w \rightsquigarrow q
\end{aligned}
\tag{5.3–13}
$$

The following provides hints about the intuition behind each of the data constructors above.

1. The rule ξ_1 is "a nontrivial loop reduces to the trivial walk of its endpoint".
2. The rule ξ_2 is "the relation $(\rightsquigarrow)$ is right compatible with edge concatenation".
3. The rule ξ_3 is "the relation $(\rightsquigarrow)$ removes left attached loops".

Remark 5.30. The data constructors in (5.3–13) follow a design principle to avoid certain unification problems occurring in dependently-typed programs (Kokke, Siek, and Wadler 2020; McBride

n.d.).

Definition 5.31. The relation ($\rightsquigarrow^*$) is the reflexive and transitive closure of the relation ($\rightsquigarrow$) in Definition 5.29.

Lemma 5.32. Given $x, y : \mathsf{N}_G$ and $p, q : \mathsf{W}_G(x, y)$, the following claims hold:

1. If $x \in q$ and $p \rightsquigarrow^* q$ then $x \in p$.
2. If $p \rightsquigarrow q$ then $\mathsf{length}(q) < \mathsf{length}(p)$.

One can prove that our reduction relation in Definition 5.29 satisfies the progress property, similarly as proved for simply-typed lambda calculus in Agda (Kokke, Siek, and Wadler 2020, §2). Evidence that a walk reduces is encoded using the following predicate.

Definition 5.33. Given a walk $p : \mathsf{W}_G(x, y)$,

$$\mathsf{Reduce}(p) :\equiv \sum_{(q : \mathsf{W}_G(x,y))} (p \rightsquigarrow q).$$

The predicate Normal defined in Definition 5.34 is the evidence that a walk is a quasi-simple walk that can no longer reduce.

Definition 5.34. Given a walk p, one states that p is in *normal form* if $\mathsf{Normal}(p)$. If $p \rightsquigarrow q$ and the walk q is in normal form; we refer to q as the normal form of p.

$$\mathsf{Normal}(p) :\equiv \mathsf{isQuasi}(p) \times \neg\,\mathsf{Reduce}(p).$$

Lemma 5.35. Being in normal form for a walk is a proposition.

Proof. It follows from Lemmas 2.13 and 5.17. □

Example 5.36. The very basic normal forms for walks are the trivial ones, and the one-edge walks with different endpoints. Given a walk w and a term of $\mathsf{NoReduce}(w)$, one can easily show that the walk w is in normal form.

Definition 5.37. Given nodes x and y in a graph G, we encode the fact a walk can reduce or not by using the inductive data type Progress.

$$
\begin{aligned}
&\textbf{data}\ \ \mathsf{Progress}\,\{x\,y\}\,(p : \mathsf{W}_G(x,y)) : \mathcal{U}\\
&\quad \mathsf{step} : \mathsf{Reduce}(p) \to \mathsf{Progress}(p)\\
&\quad \mathsf{done} : \mathsf{Normal}(p) \to \mathsf{Progress}(p)
\end{aligned}
$$

Theorem 5.38. Given a graph G with a discrete node-set, there exists a reduction for each walk to one of its normal forms, i.e., (5.3–14) is inhabited for all $w : \mathsf{W}_G(x, y)$.

$$\sum_{(v : \mathsf{W}_G(x,z))} (w \rightsquigarrow^* v) \times \mathsf{Normal}(v). \qquad (5.3\text{–}14)$$

Remark 5.39. The reduction relation ($\rightsquigarrow$) has the termination property. There is no infinite sequence of walks reducing, since the length of each walk in a chain, like $w_1 \rightsquigarrow w_2 \rightsquigarrow w_3 \rightsquigarrow \cdots$, decreases at each reduction step. See also Lemma 5.3.

Corollary 5.40. Given a graph G with a discrete node-set, and a walk w of type $\mathsf{W}_G(x, y)$ for two $x, y : \mathsf{N}_G$, the following claims hold.

1. The type $\mathsf{Reduce}(w)$ is decidable.
2. The proposition $\mathsf{Normal}(w)$ is decidable.
3. The walk w progresses in the sense of Definition 5.37.

For simplicity, the proofs of Theorem 5.38 and Corollary 5.40 are omitted. Neither of them requires the law of excluded middle. However, if we want to construct the normal form for a walk, the node set of the graph has to be discrete. In the case of Theorem 5.38, its proof can use the same reasoning given for the proof of Theorem 5.48.

5.4 The notion of walk homotopy

In this subsection, we introduce a binary relation, denoted by $(\sim_{\mathcal{M}})$, on the set of walks between fixed endpoints in a graph. This relation is designed to capture the behaviour of walks in an embedded graph in a surface such as the two-dimensional plane, where all the walks can be deformed one into another along the faces of the graph map in use.

Definition 5.41. Let w_1, w_2 be two walks from x to y in $\mathsf{Sym}(G)$. The expression $w_1 \sim_{\mathcal{M}} w_2$ denotes that one can *deform* w_1 into w_2 along the faces of $\mathcal{M}$, as illustrated in Figure 5.4. We acknowledge evidence of this deformation as a walk homotopy between w_1 and w_2, of type $w_1 \sim_{\mathcal{M}} w_2$.

The relation $(\sim_{\mathcal{M}})$ has four constructors, as follows. The first three constructors are functions to indicate that homotopy for walks is an equivalence relation; they are hrefl, hsym,

and htrans. Let $x, y : \mathsf{N}_G$.

$$\begin{aligned}
\mathsf{hrefl} &: \prod_{(w_1 : \mathsf{W}_{\mathsf{Sym}(G)}(x,y))} w_1 \sim_{\mathcal{M}} w_1. \\
\mathsf{hsym} &: \prod_{(w_1, w_2 : \mathsf{W}_{\mathsf{Sym}(G)}(x,y))} w_1 \sim_{\mathcal{M}} w_2 \to w_2 \sim_{\mathcal{M}} w_1. \\
\mathsf{htrans} &: \prod_{(w_1, w_2, w_3 : \mathsf{W}_{\mathsf{Sym}(G)}(x,y))} w_1 \sim_{\mathcal{M}} w_2 \to w_2 \sim_{\mathcal{M}} w_3 \to w_1 \sim_{\mathcal{M}} w_3.
\end{aligned} \tag{5.4–15}$$

The fourth constructor, illustrated in Figure 5.5, is the hcollapse function that establishes the walk homotopy:

$$(w_1 \cdot \mathsf{ccw}_{\mathcal{F}}(a,b) \cdot w_2) \sim_{\mathcal{M}} (w_1 \cdot \mathsf{cw}_{\mathcal{F}}(a,b) \cdot w_2),$$

supposing one has the following,

(i) a face $\mathcal{F}$ given by $\langle A, f \rangle$ of the map $\mathcal{M}$,

(ii) a walk w_1 of type $\mathsf{W}_{\mathsf{Sym}(G)}(x, f(a))$ for a node x in G with a node a in A, and

(iii) a walk w_2 of type $\mathsf{W}_{\mathsf{Sym}(G)}(f(b), y)$ for a node b in A with a node y in G.

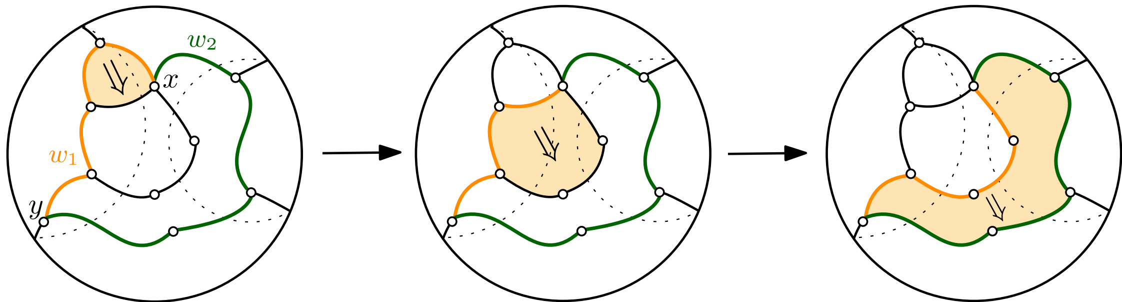


Figure 5.4: It is shown that three homotopies between two walks from x to y in an embedded graph in the sphere. In each case, the arrow ($\Downarrow$) indicates the face and the direction in which the corresponding walk deformation is performed. We obtain a homotopy between the two highlighted walks, w_1 and w_2, by composing, from left to right, the homotopies from each figure.

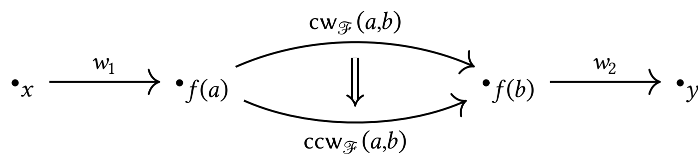


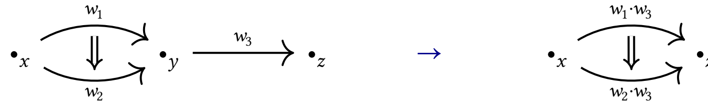
Figure 5.5: Given a face $\mathcal{F}$ of a map $\mathcal{M}$, we illustrate here hcollapse, one of the four constructors of the homotopy relation on walks in Definition 5.41. The arrow ($\Downarrow$) represents a homotopy of walks.

One consequence of Definition 5.41 is that, in each face $\mathcal{F}$, there is a walk-homotopy between $\mathsf{ccw}_{\mathcal{F}}(x, y)$ and $\mathsf{cw}_{\mathcal{F}}(x, y)$ using the constructor hcollapse.

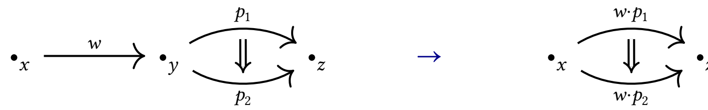
The following lemma shows how to compose walk homotopies horizontally and vertically. We consider a map $\mathcal{M}$ for a graph G and distinguishable nodes, x, y, and z where w, w_1, and w_2 are walks from x to y.

Lemma 5.42.

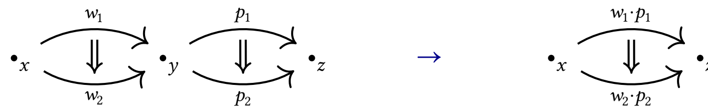
1. (Right whiskering) Let w_3 be a walk of type $\mathsf{W}_{\mathsf{Sym}(G)}(y,z)$. If $w_1 \sim_{\mathcal{M}} w_2$ then $(w_1 \cdot w_3) \sim_{\mathcal{M}} (w_2 \cdot w_3)$.



2. (Left whiskering) Let p_1, p_2 be walks of type $\mathsf{W}_{\mathsf{Sym}(G)}(y,z)$. If $p_1 \sim_{\mathcal{M}} p_2$ then $(w \cdot p_1) \sim_{\mathcal{M}} (w \cdot p_2)$.



3. (Horizontal composition) Let p_1, p_2 be walks of type $\mathsf{W}_{\mathsf{Sym}(G)}(y,z)$. If $w_1 \sim_{\mathcal{M}} w_2$ and $p_1 \sim_{\mathcal{M}} p_2$, then $(w_1 \cdot p_1) \sim_{\mathcal{M}} (w_2 \cdot p_2)$.



5.5 The type of spherical maps

In topology, the property of being simply connected to the sphere states that one can freely deform/contract any walk on the sphere into another whenever they share the same endpoints. This property of the sphere leads to the predicate in Definition 5.43, which sets the criteria for a graph to be embeddable in the 2-sphere. An alternative definition of this criteria for graphs with discrete nodes is provided in Definition 5.44. We use spherical maps to develop basic planarity criteria for graphs, which were initially described in (Prieto-Cubides and Håkon Robbestad Gylterud 2022).

Definition 5.43. Given a graph G, a map $\mathcal{M}$ for G is said to be spherical if the type in (5.5–16) is inhabited.

$$\prod_{(x,y\,:\,\mathsf{N}_{\mathsf{Sym}(G)})}\ \prod_{(w_1,w_2\,:\,\mathsf{W}_{\mathsf{Sym}(G)}(x,y))} \|\, w_1 \sim_{\mathcal{M}} w_2 \,\| \,. \tag{5.5–16}$$

To prove a given map is spherical following Definition 5.43, one must consider the set of all possible walk-pairs for each node-pair. This is not easy unless the set of walks fol-

lows a certain property, since the type of walks forms an infinite set. Therefore, it is proposed an alternative formulation for spherical maps based on Definition 5.29. Any walk is homotopic to its normal form, and only quasi-simple walks can be in normal form. By removing such a "redundancy" created by loops in the graph, a more convenient definition is obtained for spherical maps for graphs with discrete node-set, see Definition 5.44. Furthermore, using Theorem 5.48, we show that both definitions are equivalent for graphs with discrete node set in Corollary 5.49.

Definition 5.44. Given a graph G, a map $\mathcal{M}$ for G is considered to be spherical if the type in (5.5–17) is inhabited.

$$\prod_{(x,y:\mathrm{N}_G)} \prod_{(w_1,w_2:\mathrm{W}_{\mathrm{Sym}(G)}(x,y))} \mathrm{isQuasi}(w_1) \times \mathrm{isQuasi}(w_2) \to \| w_1 \sim_{\mathcal{M}} w_2 \| . \qquad (5.5\text{–}17)$$

Lemma 5.45. Being spherical for a map is a proposition. Furthermore, if the graph of the map is finite, such a proposition is decidable.

Proof. The first part follows straightforwardly. To show the second part, let $\mathcal{M}$ be a map for a finite graph. Then, the conclusion will follow if for any pair of quasi-simple walks, p and q, sharing endpoints, one can always determine whether a walk homotopy exists or not between them. To check if $p \sim_{\mathcal{M}} q$, let us assume, without loss of generality, that p and q are different walks without a prefix or suffix walk in common. Otherwise, using left/right whiskering as in Lemma 5.42, one could reconstruct a walk homotopy from a walk homotopy between p and q without prefixes and suffixes.

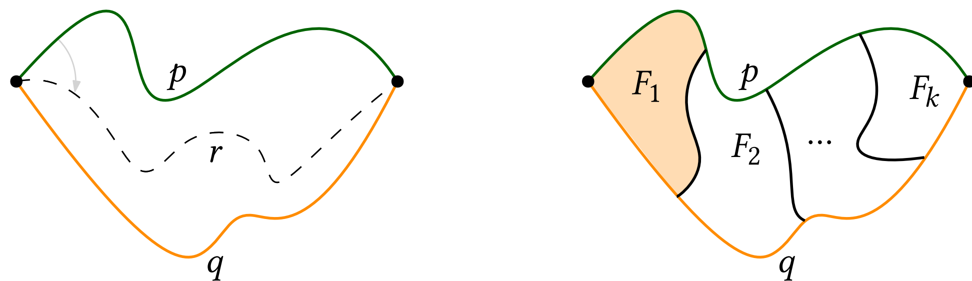


Figure 5.6: The figure shows the cases considered in the proof of Lemma 5.45 for two walks p and q with the same endpoints. In the first case, there is at least one walk between p and q, while in the second case, there are no faces but those given by a map.

Using the map, one can check whether there is a walk r *between* p and q. Note that the set of quasi-simple walks is finite, and one can then freely iterate through any subset of it; see Theorem 5.26.

Now, if there is no such walk r, then one checks if p and q *cover* one or more faces from the finite set of faces of $\mathcal{M}$. If so, one can repeatedly use data constructors like

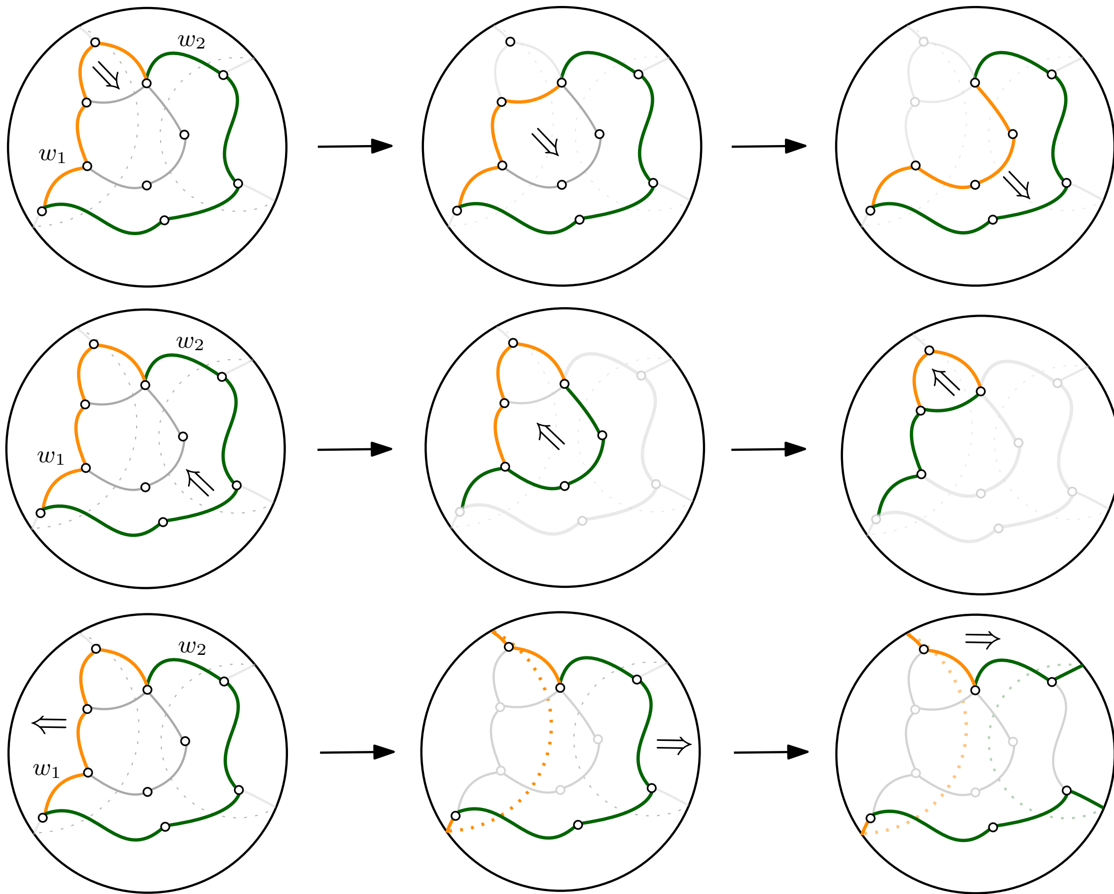


Figure 5.7: The figure shows three different walk homotopies between the walks w_1 and w_2 in a graph with a spherical map.

htrans and hcollapse to build up a walk homotopy using the faces between p and q, as illustrated in Figure 5.6. Covering a face using p and q means that the boundary of such a face is the concatenation of a subwalk of p and a subwalk of q. Otherwise, there is a *hole*, which allows concluding that such a map is not spherical.

On the other hand, if there is a walk r between p and q, then we recursively check if $p \sim_{\mathcal{M}} r$ and $r \sim_{\mathcal{M}} q$. If both walk homotopies exist, one continues with a different pair of walks. Otherwise, the map is not spherical. □

Lemma 5.46. The collection of all spherical maps for a (finite) graph is a (finite) set.

Proof. This is a subtype of the type of maps, which, when considering a finite graph, turns out to be finite. Subtypes of finite types are finite. □

We will only refer to spherical maps as maps that follow Definition 5.44, unless otherwise indicated. It is straightforward to prove that loops are homotopic to the corresponding trivial walk if a spherical map is given.

Lemma 5.47. Given a graph G, a spherical map $\mathcal{M}$ and $x : \mathsf{N}_G$, it follows that $\|(e \odot \langle x \rangle) \sim_{\mathcal{M}} \langle x \rangle\|$ for all $e : \mathsf{E}_{\mathsf{Sym}(G)}(x, x)$.

Proof. Apply $\mathcal{M}$ to the walks $(e \odot \langle x \rangle)$ and $\langle x \rangle$. □

Theorem 5.48. Given a graph G with a spherical map $\mathcal{M}$ and discrete set of nodes, for any walk $p : \mathsf{W}_{\mathsf{Sym}(G)}(x, z)$, there exists a normal form of p, denoted by $\mathsf{nf}(p)$, such that p is merely homotopic to $\mathsf{nf}(p)$, in the sense of Definition 5.41.

Proof. Given a walk p in $\mathsf{Sym}(G)$ from x to z of length n, we will construct a term of type $Q(\mathcal{M}, x, z, p)$ defined as follows.

$$Q(\mathcal{M}, x, z, w) :\equiv \sum_{(v : \mathsf{W}_{\mathsf{Sym}(G)}(x,z))} (w \rightsquigarrow^* v) \times \mathsf{Normal}(v) \times \|w \sim_{\mathcal{M}} v\|.$$

The proof is done by using strong induction on n.

- ▷ Case n equals zero. The walk p is the trivial walk $\langle x \rangle$, and it is then in normal form and also, by hrefl, homotopic to itself.
- ▷ Case n equals one. The walk p is a one-edge walk. We then ask if $x = z$.
  1. If $x = z$, the walk p reduces to the trivial walk $\langle x \rangle$ by ξ_1. Applying $\mathcal{M}$, one obtains evidence of a homotopy between p and $\langle x \rangle$, as the two walks are quasi-simples.
  2. If $x \neq z$, the one-edge walk p is its own normal form and homotopic to itself by hrefl.
- ▷ Assuming that $Q(x', z', w)$ for any walk w from x' to y' of length $k \leq n$, we must prove that $Q(x, z, p)$ when the length of p is $n + 1$.
- ▷ Therefore, let p be a walk $(e \odot w)$ where $e : \mathsf{E}_{\mathsf{Sym}(G)}(x, y)$ and the walk $w : \mathsf{W}_{\mathsf{Sym}(G)}(y, z)$ is of length n. The following cases must be considered with respect to equality $x = y$.
  1. If $x = y$ then by the induction hypothesis applied to w, one obtains the normal form $\mathsf{nf}(w)$ of the walk w, along with $r : w \rightsquigarrow \mathsf{nf}(w)$ and $h_1 : \|w \sim_{\mathcal{M}} \mathsf{nf}(w)\|$. We ask if $x = z$ to see if p is a loop.
     (a) If $x = z$ then the walk p reduces to the trivial walk $\langle x \rangle$ by ξ_1. By applying $\mathcal{M}$ to the quasi-simple walk $\mathsf{nf}(w)$ and $\langle x \rangle$, $h_2 : \| \mathsf{nf}(w) \sim_{\mathcal{M}} \langle z \rangle \|$ is obtained. It remains to show that p is homotopic to $\langle x \rangle$. Because being homotopic is a proposition, the propositional truncation in h_1 and h_2 can

be eliminated to get access to the corresponding homotopies. The required walk homotopy is as follows.

$$\begin{aligned} p &\equiv (e \odot w) \\ &\sim_{\mathcal{M}} e \odot \mathsf{nf}(w) && \text{(By Lemma 5.42 and } h_1) \\ &\sim_{\mathcal{M}} e \odot \langle z \rangle && \text{(By Lemma 5.42 and } h_2) \\ &\sim_{\mathcal{M}} \langle x \rangle && \text{(By Lemma 5.47 applied to } \mathcal{M}). \end{aligned}$$

(b) If $x \neq z$, then the walk p reduces to $\mathsf{nf}(w)$ by the following calculation using h_1.

$$\begin{aligned} p &\equiv (e \odot w) \\ &\equiv (e \odot \langle x \rangle) \cdot w && \text{(By def. of walk composition)} \\ &\rightsquigarrow^* w && \text{(By } \xi_3) \\ &\rightsquigarrow^* \mathsf{nf}(w) && \text{(By } r). \end{aligned}$$

2. If $x \neq y$, then we split w at x using Lemma 5.9. Hence, two cases have to be considered: whether x is in w or not, see Definition 5.8.

   (a) If $xis \in w$, then, for every node k in G, there are walks $w_1 : \mathsf{W}_{\mathsf{Sym}(G)}(y, k)$ and $w_2 : \mathsf{W}_{\mathsf{Sym}(G)}(k, z)$ such that $\gamma : w = w_1 \cdot w_2$, along with evidence that $x \notin w_1$ by Lemma 5.9. By the induction hypothesis applied to w_1 and to w_2, we obtain the normal forms $\mathsf{nf}(w_1)$ and $\mathsf{nf}(w_2)$, and the terms $r_i : w_i \rightsquigarrow \mathsf{nf}(w_i)$ and $h_i : \|w_i \sim_{\mathcal{M}} \mathsf{nf}(w_i)\|$ for $i = 1, 2$. The following cases refer to whether $x = z$ or not.

      i. If $x = z$, the walk p reduces to $\langle x \rangle$ by the rule ξ_1. To show that p is homotopic to $\langle x \rangle$, let s_1 and s_2 of type, respectively, $\|p \sim_{\mathcal{M}} \mathsf{nf}(w_2)\|$ and $\|\mathsf{nf}(w_2) \sim_{\mathcal{M}} \langle x \rangle\|$, as given below. Assuming one has the terms s_1 and s_2, by elimination of the propositional truncation and the transitivity property of walk homotopy with s_1 and s_2, the required conclusion follows. The walk homotopy s_1 is as follows.

$$\begin{aligned} p &\equiv (e \odot w) \\ &\sim_{\mathcal{M}} e \odot (w_1 \cdot w_2) && \text{(By the equality } \gamma) \\ &\sim_{\mathcal{M}} (e \odot w_1) \cdot w_2 && \text{(By assoc. property of } (\cdot)) \\ &\sim_{\mathcal{M}} (e \odot \mathsf{nf}(w_1)) \cdot \mathsf{nf}(w_2) && \text{(By Lemma 5.42, } h_1\text{, and } h_2) \\ &\sim_{\mathcal{M}} \langle x \rangle \cdot \mathsf{nf}(w_2) && \text{(By the homotopy from } h_4) \\ &\sim_{\mathcal{M}} \mathsf{nf}(w_2) && \text{(By definition),} \end{aligned}$$

      where $h_4 : \|(e \odot \mathsf{nf}(w_1)) \sim_{\mathcal{M}} \langle x \rangle\|$ is given by applying the map $\mathcal{M}$ to the quasi-simple walks, $(e \odot \mathsf{nf}(w_1))$ and $\langle x \rangle$. The walk $(e \odot \mathsf{nf}(w_1))$

is quasi-simple by Lemma 5.14. Also, note that $x \notin \mathsf{nf}(w_1)$ by Lemma 5.32 and the assumption $x \notin w_1$. Finally, the remaining walk homotopy s_2 is obtained by applying $\mathcal{M}$ to the quasi-simple walks, $\mathsf{nf}(w_2)$ and the trivial walk at x.

ii. If $x \neq z$, then the walk p reduces to $\mathsf{nf}(w_2)$ by the reduction reasoning in (5.5–18). As the walk $\mathsf{nf}(w_2)$ is in normal form, it remains to show that p is homotopic to $\mathsf{nf}(w_2)$. However, the reasoning is similar to Item 2(a)i.

$$
\begin{aligned}
p \equiv\ & (e \odot w) && \\
\rightsquigarrow^*\ & e \odot (w_1 \cdot w_2) && \text{(By splitting } w \text{ using Lemma 5.9)} \\
\rightsquigarrow^*\ & (e \odot w_1) \cdot w_2 && \text{(By assoc. property of } (\cdot)) \\
\rightsquigarrow^*\ & \langle x \rangle \cdot w_2 && \text{(By } \xi_2 \text{ applied to the loop } (e \odot w_1)) \\
\rightsquigarrow^*\ & w_2 && \text{(By definition of walk composition)} \\
\rightsquigarrow^*\ & \mathsf{nf}(w_2) && \text{(By the induction hypothesis).}
\end{aligned}
\tag{5.5–18}
$$

(b) Otherwise, there is evidence that $x \notin w$. By the induction hypothesis applied to w, the walk $\mathsf{nf}(w)$ is obtained, along with a reduction $r : w \rightsquigarrow \mathsf{nf}(w)$ and evidence $h : \| w \sim_{\mathcal{M}} \mathsf{nf}(w) \|$. The proof is by structural induction on the walk, $\mathsf{nf}(w)$.

i. If $\mathsf{nf}(w)$ is the trivial walk $\langle y \rangle$, then the walk p reduces either to $\langle x \rangle$, if $x = z$, or to the walk $(e \odot \langle z \rangle)$, if $x \neq z$. Either way, it is possible to construct the corresponding homotopies, similarly as for Item 1a.

ii. If the walk $\mathsf{nf}(w)$ is the composite walk $(u \odot v)$ for $u : \mathsf{E}_{\mathsf{Sym}(G)}(y, y')$, $v : \mathsf{W}_{\mathsf{Sym}(G)}(y', z)$ and nodes $y', z : \mathsf{N}_G$, then we ask if $x = z$.

- If $x = z$, then the walk p reduces to the trivial walk $\langle x \rangle$ by ξ_1. It remains to show that the walk $(e \odot \mathsf{nf}(w))$ is homotopic to $\langle x \rangle$. The spherical property of the map $\mathcal{M}$ is applied to observe this. Note that the walk $(e \odot \mathsf{nf}(w))$ is quasi-simple by Lemma 5.14, as $x \notin \mathsf{nf}(w)$ by Lemma 5.32 applied to the assumption $x \notin w$.
- If $x \neq z$, then the walk p reduces to the walk $(e \odot \mathsf{nf}(w))$ by ξ_2. By the propositional truncation elimination applied to the evidence of Lemma 5.42 and the homotopy h, one can obtain evidence that the walk $(e \odot w)$ is homotopic to $(e \odot \mathsf{nf}(w))$. It remains to show that the composite walk $(e \odot \mathsf{nf}(w))$ is in normal form. By Lemma 5.14, this walk is quasi-simple. By case analysis on the possible reductions using Definition 5.29, one proves that this walk does not reduce.

Therefore, $(e \odot \mathsf{nf}(w))$ is in normal form. □

Corollary 5.49. The two spherical map definitions, Definition 5.43 and Definition 5.44, are equivalent when considering graphs with a discrete set of nodes.

Proof. The definitions in question are propositions. Thus, it is only necessary to show that they are logically equivalent.

1. Every spherical map by Definition 5.44 is a spherical map with additional data in the sense of Definition 5.43

2. Let $\mathcal{M}$ be a spherical map by Definition 5.44. To see $\mathcal{M}$ also satisfies Definition 5.43, let w_1 and w_2 be two quasi-simple walks from x to y. We must now exhibit evidence that w_1 is homotopic to w_2. By Theorem 5.48, a walk homotopy h_1 between w_1 and the normal form $\mathsf{nf}(w_1)$ exists. Similarly, one can obtain a term h_2 of type $\|w_2 \sim_{\mathcal{M}} \mathsf{nf}(w_2)\|$.

$$\begin{aligned} w_1 &\sim_{\mathcal{M}} \mathsf{nf}(w_1) && \text{(By } h_1 \text{ from Theorem 5.48)} \\ &\sim_{\mathcal{M}} \mathsf{nf}(w_2) && \text{(By } h_3 \text{ from Definition 5.44)} \\ &\sim_{\mathcal{M}} w_2 && \text{(By } h_2 \text{ from Theorem 5.48).} \end{aligned} \tag{5.5–19}$$

On the other hand, note that walks in normal form are quasi-simple walks by definition. Therefore, it is possible to get $h_3 : \|\mathsf{nf}(w_1) \sim_{\mathcal{M}} \mathsf{nf}(w_2)\|$ by applying the spherical property of the map $\mathcal{M}$ to $\mathsf{nf}(w_1)$ and $\mathsf{nf}(w_2)$. By the elimination of the propositional truncation applied to h_1, h_2, and h_3, the required evidence of a homotopy between w_1 and w_2 can be obtained, as stated in (5.5–19). □

For the sake of completeness, let us here state Lemma 5.51 that there exists one spherical map for every C_n. This lemma together with the results of Section 4.4.2 allows us to prove later Example 6.3. The candidate map for C_n to be spherical is precisely the one given in Example 4.30.

Lemma 5.50. Let x, y be nodes in C_n. The following claims hold for the graph $\mathsf{Sym}(C_n)$. The walks $\mathrm{cw}_{\mathsf{Sym}(C_n)}(x, y)$ and $\mathrm{ccw}_{\mathsf{Sym}(C_n)}(x, y)$, referenced in Lemma 4.29, are quasi-simple walks. The total length of these walks sums up to n.

Additionally, one can prove that n is the maximum possible length of a quasi-simple walk in the graph $\mathsf{Sym}(C_n)$, as stated in Lemma 5.24. Moreover, as illustrated in Figure 4.5 for the face $\mathcal{F}$, the graph $\mathsf{Sym}(C_n)$ is completely covered by the walks $\mathrm{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ and $\mathrm{cw}_{\mathsf{Sym}(C_n)}(x, y)$. Note that for any graph G, there are at least two closed walks between any pair of nodes x, y in any face $\mathcal{F}$, respectively, $\mathrm{ccw}_{\mathcal{F}}(x, y)$ and $\mathrm{cw}_{\mathcal{F}}(x, y)$.

Lemma 5.51. The graph map for C_n given in Example 4.30 is spherical.

Proof. We must show that any pair of walks in C_n, equivalently quasi-simple walks, are walk-homotopic. Let us consider the following cases.

1. If $n = 1$, the only walk to consider is the trivial walk, which is trivially homotopic to itself.
2. If $n > 1$ and $x \neq y$, then one only needs to consider the quasi-simple walks $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ and $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, y)$ given by Lemmas 4.29 and 5.50.

   However, such walks are walk-homotopic by

   $$\mathsf{hcollapse}(\mathcal{F}, x, y, x, y, \langle x \rangle, \langle y \rangle),$$

   where $\mathcal{F}$ is the face induced by $\mathsf{Sym}(C_n)$.
3. Otherwise, if $n > 1$ and $x = y$, the only walks to consider are the trivial walk at x and $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, x)$. Remember that the $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ is by definition $\langle x \rangle$. Similarly, as in the previous case, these two walks are homotopic by the constructor $\mathsf{hcollapse}$. □

5.6 Discussion

In other areas of mathematics unrelated to type theory, considering homotopy for graph-theoretical concepts, for example, is not new. There are several proposals for the concept of homotopy for graphs using a few discrete categorical constructions (Grigor'yan, Lin, Muranov, et al. 2014). Many of these constructions use the $\times$-homotopy notion, defined as a relation based on the categorical product of graphs in the Cartesian closed category of undirected graphs. Since a walk of length n in a graph G is simply a morphism between a path graph P_n into G, the notion of homotopy for walks is defined as homotopy between graph homomorphisms. The looped path graph I_n is used to define the homotopy of these morphisms in a manner similar to the interval $[0, 1]$ for the concept of homotopy between functions in homotopy theory. As a source of more results, it is possible to endow the category of undirected graphs with a 2-category structure by considering homotopies of walks as 2-cells, as described by Chih and Scull (Chih and Scull 2020).

In terms of the reduction relation on walks and spherical maps, this work is related to polygraphs used in the context of higher-dimensional rewriting systems. Recent work by Kraus and von Raumer (Kraus and Raumer 2020, 2021) uses ideas in graph theory, higher categories, and abstract rewriting systems to approximate a series of open problems in HoTT. In the same vein, the internalisation of rewriting systems and the implementation

of polygraphs in Coq by Lucas (Lucas 2019, 2020) were found to be related to the Kraus and von Raumer approach. A fundamental object in the work of the authors mentioned above is that of an n-polygraph, also called *computad*.

A n-polygraph is a (higher dimensional) structure that can serve, for example, to analyse reducing terms to normal forms and compare reduction sequences on abstract term rewriting systems. The following is a possible correspondence to relate these ideas within the context of our work. The notion of a 1-polygraph (Kraus and Raumer 2021, §2), which is given by two sets Σ_0 and Σ_1, and two functions, $s_0, t_0 : \Sigma_1 \to \Sigma_0$ is equivalent to the type of graphs in Definition 3.1. An *object* is a node, a *reduction step* is an edge, and a *reduction sequence* $a \rightsquigarrow^* b$ is a walk between nodes a to b. A (closed) *zig-zag* is a (cycle) walk in the symmetrisation of the graph representing the reduction relation. A (generalised) 2-polygraph (Kraus and Raumer 2021, Def. 25) consists of a type A, a set of reduction steps on A, and all rewriting steps between zig-zags. Then, the notion of 2-polygraph on A will correspond to a graph G representing the type A with the set of all walks in G and the collection of walk homotopies in the symmetrisation $\mathsf{Sym}(G)$ for a given combinatorial map.

Using the previous interpretation for polygraphs, one may state that a graph with a spherical map holds properties such as *terminating*, *closed under congruence*, *cancels inverses*, and it has a *Winkler-Buchberger structure* (Kraus and Raumer 2021, Eq. 32-35). The related concept of *homotopy basis* of a 2-polygraph (Kraus and Raumer 2021, Def. 28) may be seen as the set obtained from Definition 5.43 without using propositional truncation in the corresponding type.

On the other hand, *Noetherian induction for closed zig-zags* (Kraus and Raumer 2021, § 3.5) addresses a similar issue that we investigated here. In this work, we found that to prove certain properties, such as the normalisation theorem in Theorem 5.48 for graphs with a spherical map and a discrete set, it was only necessary to consider (cycle) walks without inner loops. One can prove other properties related to walk homotopies for graphs with spherical maps, not only considering the property on a cycle walk but on any walk. This approach relies on the machinery of quasi-simple walks in Section 5.2 and the loop reduction relation on walks in Section 5.3. Our loop reduction relation is likely *locally confluent* (Kraus and Raumer 2021, § 3.3), but without the uniqueness of normal forms. Proof of these properties will be done in the future, since they are not required here. We will also investigate in depth the extent to which the constructions given by Kraus and von Raumer, as well as by Lucas, are not only related but also applicable to our main project of graph theory in HoTT (Prieto-Cubides 2023).

Finally, on the computer formalisation side, we identify only the earlier mentioned work by Kraus and von Raumer as related to our Agda formalisation. They have a for-

malised version[1] of their results in Lean's HoTT variant.

[1] https://gitlab.com/fplab/freealgstr

6
Planar Maps

In this final chapter, our aim is to combine the concepts developed in previous chapters and establish a HoTT characterisation of graph planarity, as outlined in Definition 6.1. Our comprehension of graph planarity is influenced by topological graph theory (Gross and Tucker 1987, §3), allowing us to employ combinatorial maps to represent graph embeddings in a surface up to isotopy, without needing to define a notion of surface or other topological concepts within type theory.

We begin by reviewing background information on graph planarity before presenting the type of planar maps. Subsequently, we introduce planar extensions of graph maps and analyse the Euler characteristic for finite graphs. This method of extending graphs paves the way for constructing a multitude of planar graphs.

6.1 Planarity in graph theory

In the field of graph theory, planar graphs refers to graphs that can be drawn in the two-dimensional plane without any edge crossings. Why are planar graphs important? Planar graphs, apart from the joy of studying them, are often used as convenient and efficient models to address a wide variety of real-world problems. From a practical standpoint, they aid in numerous applications ranging from geographical mapping, data visualisation, and various graph drawing algorithms to network layouts and electric circuit printing. In light of this, various characterisations have been introduced to offer alternative

perspectives and methods for understanding and identifying planar graphs.

The study of planarity criteria, which encompasses methods for identifying planar graphs, commenced with Kuratowski's work in 1930 and its theorem on graph planarity. According to this theorem, a graph is considered planar if and only if it does not contain a subgraph isomorphic to any of the forbidden minors $K_{3,3}$ or K_5. Consequently, if a graph can be transformed into one of the forbidden minors through edge deletions and contractions, then it is deemed non-planar. Another planarity criterion that mentions the forbidden minors is Wagner's theorem (Diestel 2012; Rahman 2017). Alternative approaches involve algebraic methods such as MacLane's planarity criterion (MacLane 1937) and Schnyder's theorem (Baur 2012, §3.3).

In the context of type theory and formal methods, planar graphs hold a special place. The Four Colour Theorem (FCT), a seminal result in graph theory, was proven with computer assistance by Appel and Haken in 1976. This theorem states that any finite planar graph can be coloured with no more than four colours, ensuring no two adjacent nodes share the same colour. Its proof, covering over 1900 cases, marks a significant milestone in formal verification history and sparks a debate on the role of computers in mathematics. It raises questions such as: is a computer-checked proof truly a mathematical proof? And to what extent can computers assist in theorem proving?

Addressing the concerns surrounding the validity of Appel and Haken's proof, Gonthier undertook a complete formalisation of the FCT proof using Coq (Gonthier 2008). This monumental task stands as a significant milestone in the realm of formal verification. Their work not only fortified the standing of the FCT proof but also illuminated the vast potential of computer-assisted methodologies. One such method involves elaborating mathematical statements using dependently typed theories such as the Coq's type system.

6.2 A type of planar maps for a graph

We aim to characterise graph planarity in HoTT based on the intuitive notion that edges cannot cross each other on the plane. Finding a way to define the concept of edge crossing carefully is a challenging task. If we follow the geometric nature of the intuitive description, we may end up working with real numbers to represent, for example, the coordinates in the $\mathbb{R}^2$ to represent the nodes and edges of the graph drawing. The construction of real numbers in HoTT is discussed in (Univalent Foundations Program 2013, §10). To avoid these issues, we choose to follow the combinatorial approach described in previous chapters to describe graph maps in the plane, or equivalently, the 2-sphere with a puncture.

In the context of topology, we are aware that the 2-sphere possesses two primary

invariants that we want transport to the language of graph maps: path-connectedness and simply-connectedness. The concept of path-connectedness is that a path exists connecting any two points within the 2-sphere. On the other hand, simply-connectedness suggests that if two paths share the same endpoints in the 2-sphere, they can be deformed into one another.

If we consider a walk as the path in the corresponding space induced by the graph map, we can transport these two concepts with a fixed graph. Thus, the path-connectedness property coincides with *being connected* for the embedded graph. To address simply-connectedness for the surface induced by a graph map, we need to have an equivalent notion to saying how a pair of walks can be deformed into one another. We developed this notion and called *walk homotopy* in Definition 5.41. The concept of a graph map in the 2-sphere is what we call a *spherical map*.

A spherical map for a graph G is defined as a map $\mathscr{M}$ that satisfies the property of $\mathsf{isSpherical}(\mathscr{M})$.

$$\mathsf{isSpherical}(\mathscr{M}) :\equiv \prod_{(x,y\,:\,\mathsf{N}_{\mathsf{Sym}(G)})} \prod_{(w_1,w_2\,:\,\mathsf{W}_{\mathsf{Sym}(G)}(x,y))} \| \, w_1 \sim_{\mathscr{M}} w_2 \, \| \,. \tag{6.2–1}$$

In essence, the concept of a spherical map for a graph is defined as the characterisation of graph embeddings in the 2-sphere. The non-edge-crossing condition appearing in the intuitive definition of planarity mentioned earlier is now captured in the characterisation of spherical maps with the notion of homotopy of walks. In other words, there is no edge-crossing when embedding a graph using a map if such is spherical. As a result, having spherical maps for a graph is a necessary condition for its planarity.

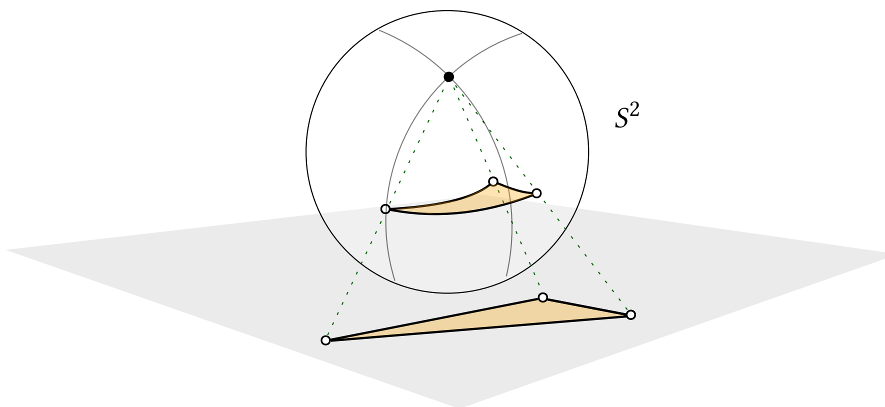


Figure 6.1: The stereographic projection of the sphere S^2 onto the two-dimensional plane.

The final observation involves determining how to obtain a graph embedding in the plane from a spherical map. This process comes naturally; recall that by puncturing the 2-sphere at a specific location and subsequently applying stereographic projection, we can transform a graph embedded in the 2-sphere into one embedded in the plane. Thus,

we need to select one face of the map to serve as the puncture point on the 2-sphere, which completes our characterisation of planarity as described below.

Definition 6.1. A connected and locally finite graph G is *planar* if the type $\mathsf{Planar}(G)$ is inhabited. Elements of $\mathsf{Planar}(G)$ are called *planar maps* of G.

$$\mathsf{Planar}(G) :\equiv \sum_{(\mathcal{M}\,:\,\mathsf{Map}(G))} \mathsf{isSpherical}(\mathcal{M}) \times \underbrace{\mathsf{Face}(G,\mathcal{M})}_{\text{outer face}}.$$

We define the type $\mathsf{Planar}(G)$ to represent all possible embeddings of G into the plane, specifically focussing on plane graphs. Although $\mathsf{Planar}(G)$ is not a planarity test in itself, it can be used to determine if a finite graph is planar or not by generating all the maps of the graph and subsequently verifying their spherical nature and the presence of an outer face, see Lemma 5.45.

Theorem 6.2. The type of all planar maps of a (finite) graph forms a (finite) set.

Proof. The type of planar maps in Definition 6.1 is not a proposition. It encompasses two sets: the set of combinatorial maps, see Lemma 4.10, and the set of faces, see Lemma 4.18. Since being spherical for a map is a mere proposition, one concludes that the Σ-type collecting all planar maps of a graph forms a set. Now, given a finite graph, the finiteness property of the collection of its planar maps is a direct consequence of the finiteness of the set of nodes, edges, maps, and faces (see Lemma 4.10 and Theorem 4.26). □

Example 6.3. To establish the planarity of C_n, we begin with the base case $n = 0$. The graph C_0 is a unit graph, a graph with a single node $\star$ and no edges. Without edges, the type of functions mapping this node to any cyclic order of its star is a contractible type, yielding a unique, trivially spherical map. The map is spherical since the only walk to consider is the empty walk, which is trivially homotopic to itself. Planarity follows as C_0 is connected by definition and possesses an outer face. To define this face, we use as the base cyclic graph, the graph C_0 itself along with identity graph homomorphism h, see that $\mathsf{Sym}(C_0) \cong C_0$. The other conditions to inhabit the type of faces for our map are thus trivially met, and the proof is detailed in Example A.1.

For $n > 0$, C_n is connected and locally finite as shown by Lemma 4.28. Its planarity is supported by Example 4.30, which confirms the existence of a unique map $\mathcal{M}$ for C_n. To show this map is spherical, it suffices to show that any two walks w_1 and w_2 with identical endpoints are homotopic. Inner loops in walks can be ignored since they are irrelevant to walk homotopy, as shown in Corollary 5.49. Let us now consider the following cases. For $n = 1$, the only walk is the trivial one, which is self-homotopic. For $n > 1$, when examining nodes x and y in C_n, we have:

▷ If $x \neq y$, the relevant walks are $\mathsf{ccw}_{\mathsf{Sym}(C_n)}(x, y)$ and $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, y)$, as per Lemma 4.29.

These walks are homotopic via $\mathsf{hcollapse}(\mathcal{F}, x, y, x, y, \langle x \rangle, \langle y \rangle)$, where $\mathcal{F}$ denotes the face associated with $\mathsf{Sym}(C_n)$ where these walks form the boundary of $\mathcal{F}$.

▷ If $x = y$, the walks under consideration are the trivial walk at x and $\mathsf{cw}_{\mathsf{Sym}(C_n)}(x, x)$. Similarly to the previous case, these walks are homotopic via hcollapse.

Finally, the outer face of $\mathcal{M}$ is naturally induced by C_n, which satisfies Definition 4.14 by construction. In fact, the definition of faces in Definition 4.14 was informed by the structure of C_n. Hence, we conclude that C_n is planar for all n.

In addition to their simple structure, cyclic graphs, and in particular C_n graphs, are building blocks in a few relevant constructions in formal systems related to the study of the planarity of graphs, such as planar triangulations and the characterisation of all 2-connected planar graphs.

In order to expand our collection of planar map examples, we will now explore the concept of planar extensions in the context of graph maps. This approach will provide a deeper understanding and additional instances of planar structures in graph theory.

6.3 Planar extensions

This subsection outlines the construction of planar maps from existing ones using the path addition operation. The inspiration for this construction derives from ear decompositions (Bang-Jensen and Gutin 2009, §5.3), reliable networks, extensions of planar graphs for undirected graphs (J. Gross, Yellen, and Anderson 2018, §5.2, 7.3), and the characterisation of 2-connected graphs (Whitney 1932).

6.3.1 Path additions

Definition 6.4. Let G be a graph with nodes u, v, and P_n denote a path graph of n nodes as defined in Definition 3.14. The (simple) path addition of P_n to G at nodes u and v in G is a new graph constructed using the function path-addition with arguments G, u, v, n, and r showing that n is positive, as illustrated in Figure 6.5 (a). For short, this new graph is denoted by $G \bullet_{u,v} P_n$. Here, u and v are referred to as the *endpoints of the addition.*

$$\mathsf{path\text{-}addition} : \prod_{(G:\mathsf{Graph})} \prod_{(u,v:\mathsf{N}_G)} \prod_{(n:\mathbb{N})} (0 < n) \to \mathsf{Graph}.$$

$$\mathsf{path\text{-}addition}\,(G, u, v, n, r) \; :\equiv (N', E', h_1, h_2).$$

The types of nodes N' and the family of edges E' are defined below. The functions h_1 and h_2 are well defined, although not elaborated here. Refer to Example A.3 for details on

these functions, their properties, and other functions related to path-additions.

$$
\begin{aligned}
N' &:\equiv \mathsf{N}_G + [\![n]\!].\\
E' &: N' \to N' \to \mathcal{U}.\\
E'(\mathsf{inl}(x), \mathsf{inl}(y)) &:\equiv \mathsf{E}_G(x, y).\\
E'(\mathsf{inl}(x), \mathsf{inr}(y)) &:\equiv (x = u) \times (y = (0, r)).\\
E'(\mathsf{inr}(x), \mathsf{inl}(y)) &:\equiv (x = \mathsf{pred}((0, r))) \times (y = v).\\
E'(\mathsf{inr}(x), \mathsf{inr}(y)) &:\equiv \mathsf{E}_{P_n}(x, y).
\end{aligned}
$$

Remember that the path graph P_n with n nodes can be defined as follows.

$$P_n :\equiv ([\![n]\!], \lambda\, u\, v.\mathsf{toNat}(u) + 1 = \mathsf{toNat}(v)),$$

where

$$
\begin{aligned}
\mathsf{toNat} &: [\![n]\!] \to \mathbb{N}.\\
\mathsf{toNat}\,(k, !) &:\equiv k.
\end{aligned}
$$

We also conveniently define the non-simple path addition of P_n to G at nodes u and v in G. This operation mirrors the symmetrisation of a simple path addition. This construction of non-simple path addition is needed for subsequent sections, as it is used to establish the planar graphs, which involve the symmetrisation of the given graph.

Definition 6.5. Let G be a graph with nodes u, v. The *non-simple path addition* of P_n to G at these nodes yields a new graph. This graph is constructed in a similar fashion as the simple path addition, by linking G and the graph $\mathsf{Sym}(P_n)$ using four edges. Two of these edges go from node u to 0 in $\mathsf{Sym}(P_n)$ and back. The other two edges link v to n in $\mathsf{Sym}(P_n)$ and back.

To ease the upcoming discussion, we must introduce the following conventions.

- ▷ G is a locally connected finite graph with decidable equality on its nodes.
- ▷ n is a positive natural number.
- ▷ In the graph $G \bullet_{u,v} P_n$, we denote the walk from u to v via the addition of P_n to G as p. This is illustrated in Figure 6.2 (a). By an abuse of notation, we may also refer to this walk as $e_0 \cdot P_n \cdot e_n$. Here, e_0 and e_n are the edges connecting nodes u to 0 and nodes $n-1$ to v respectively. The remaining edges, denoted as e_i, connect nodes $i-1$ and i and represent the new additions from the path addition.
- ▷ For brevity, we denote $G \bullet_{u,v} P_n$ by $G \bullet p$. This notation is often used below when the specifics of n and u, v are not crucial to the discussion.
- ▷ We denote $G \bullet_{u,v} \mathsf{Sym}(P_n)$ by $G \bullet \overline{p}$. Here, $\overline{p}$ represents the subgraph added to G through the non-simple path addition of P_n at nodes u and v. This is illustrated in Figure 6.2 (b).

- In $G \bullet \overline{p}$, we adopt similar notation regarding edges in the symmetrisation of a graph, as introduced in Figure 4.1. The walk $\overleftarrow{p}$ signifies the walk in $\overline{p}$ induced by the sequence $\overleftarrow{e_0} \cdot \overleftarrow{e_1} \cdot \cdots \cdot \overleftarrow{e_n}$. Conversely, $\overrightarrow{p}$ denotes the opposite direction walk, induced by the sequence $\overrightarrow{e_n} \cdot \overrightarrow{e_{n-1}} \cdot \cdots \cdot \overrightarrow{e_0}$. See Figure 6.2 (b) for an illustration.
- Both $G \bullet p$ and $G \bullet \overline{p}$ are referred to as *graph extensions*.
- The operator $(\bullet)$ is left associative.
- The variables p_i denote finite path graphs of positive length, with respective end-points u_i and v_i, adhering to the same considerations as for p in the previous items.
- A *simple cyclic addition* to G is the path addition $G \bullet_{u,u} p$ for some p, where u is a node in G.

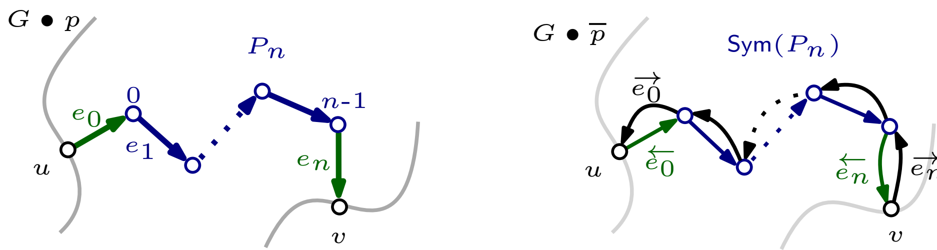


Figure 6.2: The left figure illustrates the path addition $G \bullet_{u,v} P_n$, achieved by adding path graph P_n to graph G at nodes u and v. This process introduces two new edges, e_0 and e_n, along with n new nodes from path P_n. We define p as the walk $e_0 \cdot P_n \cdot e_n$ from u to v in $G \bullet_{u,v} P_n$, simplifying notation. Similarly, the right figure depicts the non-simple path addition of P_n to G at nodes u and v, extending graph G with P_n's symmetrisation and four additional edges.

Lemma 6.6. If G is connected, then $G \bullet p$ and $G \bullet \overline{p}$ are connected.

Proof. To demonstrate the connectedness of $G \bullet_{u,v} P_n$, it is sufficient to consider connectivity between all node pairs in the augmented graph. The case for $G \bullet_{u,v} \mathsf{Sym}(P_n)$ is analogous. Additionally, we assume a walk always can be constructed to connect any two nodes in G. This is justified by eliminating the propositional truncation in the definition of connectedness, since we want to prove connectedness for a graph, which is a proposition itself. The proof is followed by cases, depending on the location of the nodes in the augmented graph.

Let x and y be distinct nodes in $G \bullet_{u,v} P_n$; for identical nodes, a trivial walk suffices. If both are in G, their connectivity is inherent. If x is in G and y is in P_n, their connectivity is established via a concatenated walk from x to u within G, followed by the subwalk of $e_0 \cdot P_n \cdot e_n$ that connects 0 to y. If x and y lie in P_n, say they correspond to i and j, we can use as the walk to connect them, $e_{i+1} \cdot \cdots \cdot e_j$ if $i < j$. Otherwise, the walk is $e_i \cdot \cdots \cdot e_n \cdot w \cdot e_0 \cdot \cdots \cdot e_j$, where w denotes a given walk from v to u in G. □

Lemma 6.7. $\mathsf{Sym}(G \bullet p) \cong \mathsf{Sym}(G) \bullet \overline{p}$.

Proof. To show these graphs are isomorphic, we compare their node and edge sets for equivalence. By definitions of Sym and path-addition, the node sets are identical:

$$\mathsf{N}_{\mathsf{Sym}(G \bullet p)} \equiv \mathsf{N}_G + [\![n]\!] \equiv \mathsf{N}_{\mathsf{Sym}(G)} + [\![n]\!] \equiv \mathsf{N}_{\mathsf{Sym}(G) \bullet \overline{p}}.$$

For the edge sets, we want to show that for given nodes x and y,

$$\mathsf{E}_{\mathsf{Sym}(G \bullet p)}(x, y) \simeq \mathsf{E}_{\mathsf{Sym}(G) \bullet \overline{p}}(x, y).$$

To address this equivalence, we notice how the path addition operation affects the edge sets of original graph. This operation affects the edges differently based on the location of x and y, but within G or P_n, and because symmetrisation does not alter the edge sets:

$$\mathsf{E}_{\mathsf{Sym}(G \bullet p)}(x, y) \equiv \mathsf{E}_{\mathsf{Sym}(G)}(x, y) \equiv \mathsf{E}_{\mathsf{Sym}(G) \bullet \overline{p}}(x, y).$$

When x is in G and y in P_n, or vice versa, symmetry allows us to consider two cases: $x \equiv u$ and $y \equiv 0$, or $x \equiv v$ and $y \equiv n$. In both scenarios, the new edges introduced by path addition result in equivalent edge sets.

$$\mathsf{E}_{\mathsf{Sym}(G \bullet_{u,v} P_n)}(u, 0) \simeq [\![2]\!] \simeq \mathsf{E}_{\mathsf{Sym}(G) \bullet \overline{p}}(u, 0).$$

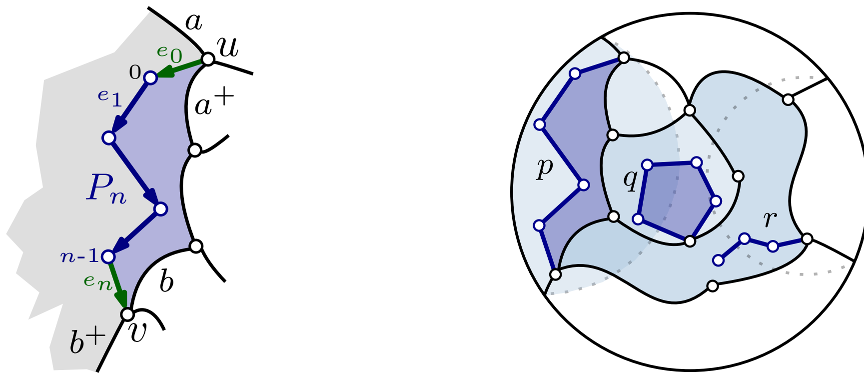
The first part of this chain, the equivalence, $\mathsf{E}_{\mathsf{Sym}(G \bullet_{u,v} P_n)}(u, 0) \simeq [\![2]\!]$ which is due to the fact that u and 0 are adjacent in $\mathsf{Sym}(G \bullet_{u,v} P_n)$. These two edges are the one induced in $G \bullet_{u,v} P_n$ and the other one from the symmetrisation process. On the other hand, the equivalence $\mathsf{E}_{\mathsf{Sym}(G) \bullet \overline{p}}(u, 0) \simeq [\![2]\!]$ follows by applying $(\bullet \overline{p})$ to $\mathsf{Sym}(G)$. The case for $x \equiv v$ and $y \equiv n$ is analogous. Consequently, the edge sets coincide, confirming the expected isomorphism. □

Lemma 6.8. Let $\mathcal{M}$ represent a planar map of G, $\mathcal{F}$ a specific face, and u and v two nodes on the boundary walk of $\mathcal{F}$. An extended planar map of $G \bullet p$ can be constructed from $\mathcal{M}$, where p is situated onto $\mathcal{F}$, splitting it into two faces.

The proof of Lemma 6.8 unfolds in several steps. We first define a map that extends $\mathcal{M}$ to a proper map of $G \bullet p$ with defined values for the nodes in p. Next, as illustrated in Figure 6.4, we establish two faces resulting from placing p onto $\mathcal{F}$. The final step involves demonstrating that the candidate map for $G \bullet p$ is planar. That is, per Definition 6.1, that all pairs of walks in the symmetrisation of $G \bullet p$ are walk homotopic with respect to the given map.

Proof of Lemma 6.8. Let $\mathcal{M}$ be a planar map of G, $\mathcal{F}$ a specific face, and u and v two nodes on the boundary walk of $\mathcal{F}$. We denote the graph $G \bullet p$ as H and the prospective planar map for this graph as $\mathcal{M}'$. In the context of Definition 4.14, within the face walk boundary $\partial\mathcal{F}$ of the given face $\mathcal{F}$, we identify an edge preceding u, represented as $a : \mathsf{E}_G(\mathsf{pred}(u), u)$, and its succeeding edge $a^+ : \mathsf{E}_G(u, \mathsf{suc}(u))$. Analogously for v, we have $b : \mathsf{E}_G(\mathsf{pred}(v), v)$ and $b^+ : \mathsf{E}_G(v, \mathsf{suc}(v))$, as depicted in Figure 6.3a.

We define the map $\mathcal{M}'$ at each node x in H. We begin with the endpoints of p, that is, $x = u$ and $x = v$. For $x = u$, we alter the cycle $\mathcal{M}(u)$ by introducing e_0 between the edges a and a^+, resulting in the cycle $\mathcal{M}'(u) = (\cdots a\, e_0\, a^+ \cdots)$. Similarly, for $x = v$, the modified cycle $\mathcal{M}'(v)$ is $(\cdots b\, e_n\, b^+ \cdots)$. For internal nodes of p, that is, nodes x in P_n, the map $\mathcal{M}'$ is defined directly. At each of these nodes, we encounter only two edges, denoted as e_i and e_{i+1}, where i ranges from 0 to $n-1$. Remember that e_0 connects nodes u and 0, e_n links nodes $n-1$ and v, and for the remaining, e_i bridges nodes $i-1$ and i.



(a) Path addition used in Lemma 6.8. (b) The embedded graph $\mathsf{Sym}(G \bullet p \bullet q \bullet r)$.

Figure 6.3: Figure (a) in the caption illustrates the path addition $G \bullet p$ as detailed in Lemma 6.8. Figure (b) presents the planar map for G from Figure 4.2 (b), showcasing three graph extensions: the path addition of p, cyclic addition of q, and spike addition of r. Though it is feasible to define the construction of r, it is not necessary for this discussion. The additions of p and q split faces F_2 and F_3 from Figure 4.2, generating two new faces each. The spike addition of r substitutes F_4 with a face of higher degree.

Assume the face $\mathcal{F}$, induced by (A, h) of degree m according to Definition 4.14. Here, h is an edge-injective graph homomorphism from A to $\mathsf{Sym}(H)$, satisfying the map-compatibility condition. Let $\partial\mathcal{F}$ be the boundary walk of $\mathcal{F}$ of length k, and define n_1, n_2 as $k + (n+1)$ and $(m-k) + (n+1)$ respectively.

Let us denote F_1, F_2 as faces induced by (C_{n_1}, h_1) and (C_{n_2}, h_2) respectively, where $h_1 = (\alpha_1, \beta_1)$ and $h_2 = (\alpha_2, \beta_2)$ are morphisms of type $\mathsf{Hom}(C_{n_i}, \mathsf{Sym}(H))$ for $i = 1, 2$. The boundary walks of these faces, ∂F_1 and ∂F_2, are defined as $\mathsf{cw}_{\mathcal{F}}(u, v) \cdot \overrightarrow{p}$ and $\mathsf{ccw}_{\mathcal{F}}(u, v) \cdot \overrightarrow{p}$ respectively. The illustration in Figure 6.4 provides a visual representation of this concept.

To establish the planarity of $\mathcal{M}'$, we must first demonstrate that for each face F_1 and F_2 of $\mathcal{M}'$, h_1 and h_2 satisfy the map-compatibility condition and uphold the edge-injectivity

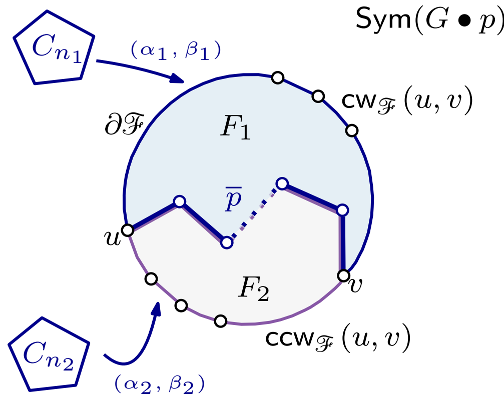


Figure 6.4: The figure demonstrates the partitioning of face $\mathcal{F}$ into two, F_1 and F_2, via $G \bullet p$ when p resides on face $\mathcal{F}$.

property. Beginning with h_1, consider the nodes in C_{n_1}, namely $0, 1, \ldots, n_1 - 1$. Each node $i : \mathsf{N}_{C_{n_1}}$ maps to a node defined by α from $\mathcal{F}$. Specifically, $\alpha_1(i)$ equals $\alpha(i)$ for $i < k$, while $\alpha(i)$ positions the node in $\text{cw}_{\mathcal{F}}(u, v)$. For the corresponding edges, $e : \mathsf{E}_{C_{n_1}}(i, i+1)$, we employ the function β from $\mathcal{F}$ to define β_1, such that $\beta_1(i, i+1, e)$ corresponds to $\beta(i, i+1, e)$.

However, if $k \leq i \leq n_1$, node i must be placed in $\overline{p}$, then $\alpha_1(i)$ is $n - i$. Correspondingly, for edges, we set $\beta_1(i, i+1, e)$ as the edge $\mathsf{inl}(e_i)$ in $\text{Sym}(H)$. It is clear by construction that h_1 is an edge-injective, map-compatible graph homomorphism with the map $\mathcal{M}'$, properties naturally inherited from h. In a similar vein, it can be proven that h_2 is well-defined and fulfills the map-compatibility condition and the edge-injectivity property.

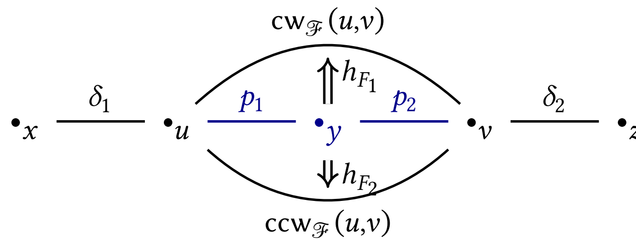


Figure 6.5: The figure shows a part of the graph $\text{Sym}(G \bullet p)$ embedded in the 2-sphere. As constructed in the proof of Lemma 6.8, the faces, F_1 and F_2, of the map $\mathcal{M}'$ are given by a face division of $\mathcal{F}$ by the path p. Such gives rise to new walk homotopies, as h_{F_1} and h_{F_2} in the picture. The walk $\overleftarrow{p}$ from u to v is the walk composition of p_1, a walk from u to y, and p_2, a walk from y to v. The walks δ_1 and δ_2 are walks in $\text{Sym}(G)$ from x to z.

To prove that $\mathcal{M}'$ is planar, we must first show that it is spherical. To see this, we rely on Lemma 5.45, which allows us to apply the elimination of the propositional truncation to the evidence that $\mathcal{M}$ is spherical. This enables us to obtain a walk homotopy for any pair of walks in $\text{Sym}(G)$ sharing endpoints, which is perhaps used henceforth without explicit mention. This entails that homotopic walks in $\text{Sym}(G)$, deforming along faces other than $\mathcal{F}$, maintain their homotopy in $\text{Sym}(H)$. Therefore, our focus narrows down

to:

(i) the set of walks in $\mathsf{Sym}(G)$ deforming along $\mathcal{F}$, and

(ii) the set of walks resulting from possible compositions of $\overline{p}$ with existing walks in $\mathsf{Sym}(G)$.

For both walks originating from set (i), their homotopy is defined by the vertical composition of homotopies along F_1 and F_2, as referenced in Lemma 5.42, (Prieto-Cubides 2022, §5).

In case (ii), we consider walks without inner loops, following Lemma 5.8 in (Prieto-Cubides 2022). We examine three subcases without loss of generality, where the walk $\overline{p}$ from u to v decomposes into p_1 and p_2. Here, p_1 is a walk from u to node y in $\mathsf{Sym}(G \bullet p)$, and p_2 from y to v, as shown in Figure 6.5. Recall that a walk homotopy for any pair of walks in $\mathsf{Sym}(G)$ sharing endpoints is always accessible by hypothesis.

(a) Either w_1, w_2, or both, include $\overleftarrow{p}$ as a subwalk from x to z. If w_1 composes as $\delta_1 \cdot \overleftarrow{p} \cdot \delta_2$, and $\overleftarrow{p}$ is not a subwalk of w_2, with δ_1 and δ_2 being walks in $\mathsf{Sym}(G)$ from x to u and v to z, a homotopy of walks can be obtained as in the calculation below. The remaining cases are demonstrated similarly.

$$
\begin{aligned}
w_1 &\equiv \delta_1 \cdot \overleftarrow{p} \cdot \delta_2 \\
&\equiv \delta_1 \cdot \mathsf{ccw}_{F_1}(u,v) \cdot \delta_2 && \text{(By construction of } F_1\text{)} \\
&\sim_{\mathcal{M}'} \delta_1 \cdot \mathsf{cw}_{F_1}(u,v) \cdot \delta_2 && \text{(By hcollapse constructor applied to } F_1, \delta_1, \text{ and } \delta_2\text{)} \\
&\equiv \delta_1 \cdot \mathsf{cw}_{\mathcal{F}}(u,v) \cdot \delta_2 && \text{(By construction of } F_1\text{)} \\
&\sim'_{\mathcal{M}} w_2 && \text{(By hypothesis: walks in Sym}(G) \text{ are homotopic).}
\end{aligned}
$$

(b) The walks w_1 and w_2 from x to y share a suffix (p_1) or a prefix (p_2). Without loss of generality, let $w_1 = \delta_1 \cdot p_1$ and $w_2 = \delta \cdot p_1$, where δ is a walk from x to u. These walks are homotopic in $\mathsf{Sym}(G)$ via the spherical map $\mathcal{M}$, i.e., $\delta_1 \sim_{\mathcal{M}} \delta$. The construction of $\mathcal{M}'$ ensures $\delta_1 \sim_{\mathcal{M}'} \delta$. Utilising right whiskering, we deduce $\delta_1 \cdot p_1 \sim_{\mathcal{M}'} \delta \cdot p_1$, thereby reaching our desired conclusion. Similarly, if w_1 is $p_2 \cdot \delta_2$ and w_2 is $p_2 \cdot \delta$, where δ is a walk from v to z, one can show that $\delta_2 \sim_{\mathcal{M}} \delta$, and hence $\delta_2 \cdot p_2 \sim_{\mathcal{M}'} \delta \cdot p_2$ by left whiskering.

(c) The walks w_1 and w_2 from x to y can be expressed as composites of $\delta \cdot p_1$ and $\delta' \cdot \overrightarrow{p_2}$, respectively. Here, δ and δ' are walks from x to u and x to v, without sharing a common prefix or suffix subwalk. We aim to show $w_1 \sim_{\mathcal{M}'} w_2$ via F_2 deformation.

$$
\begin{aligned}
w_1 &\equiv \delta \cdot \overleftarrow{p_1} \\
&\equiv \delta \cdot \mathrm{cw}_{F_2}(u, y) && \text{(By construction of } F_2) \\
&\sim_{\mathcal{M}'} \delta \cdot \mathrm{ccw}_{F_2}(u, y) && \text{(By constructor hcollapse applied to } F_2, \delta_1, \text{ and } \langle y \rangle) \\
&\equiv \delta \cdot (\mathrm{ccw}_{\mathcal{F}}(u, v) \cdot \overrightarrow{p_2}) && \text{(By construction of } F_2) \\
&\equiv (\delta \cdot \mathrm{ccw}_{\mathcal{F}}(u, v)) \cdot \overrightarrow{p_2} && \text{(By assoc. of walk concat.)} \\
&\sim_{\mathcal{M}'} \delta' \cdot \overrightarrow{p_2} && \text{(By whiskering applied to the walk htpy. by hyp.)} \\
&\equiv w_2.
\end{aligned}
$$

Concluding our proof of Lemma 6.8, we have shown that $\mathcal{M}$ extends to a spherical map $\mathcal{M}'$ of $G \bullet p$. By identifying F_1 as the outer face, we further establish that $\mathcal{M}'$ is a planar map. □

Given that $\mathcal{M}$ is a planar map, we denote its planar extension derived from Lemma 6.8 by $E(\mathcal{M}, \mathcal{F}, u, v, P_n)$. To shorten the notation, this planar extension is denoted by $\mathcal{M}$ by $E(\mathcal{M}, \mathcal{F}, p)$, when the specifics of u, v, and P_n are not really crucial to the discussion. We refer to this as the *face division* of $\mathcal{F}$ by p, since this construction results in the placement of p in $\mathcal{F}$, dividing it into two new faces.

Definition 6.9. If G is a finite graph with a map $\mathcal{M}$, then we refer to the Euler characteristic of G by $\mathcal{M}$, denoted by $\chi_{\mathcal{M}}$, as the number associated with the cardinal of the set of nodes (v), edges (e), and faces (f).

$$\chi_{\mathcal{M}} :\equiv v - e + f. \tag{6.3–2}$$

Given a graph G with a planar map $\mathcal{M}$, any planar extension of $\mathcal{M}$ preserves the Euler characteristic of G. Evidence for this is found in the construction of $E(\mathcal{M}, \mathcal{F}, p)$ as outlined in the proof of Lemma 6.8. Here, the path addition of P_n to G increases the node count v by $n+1$, edge count e by $n+2$, but only augments the face count f by one.

Lemma 6.10. For a graph G with planar map $\mathcal{M}$, any planar extension of $\mathcal{M}$ maintains Euler's characteristic. That is, for any face $\mathcal{F}$ of $\mathcal{M}$ and nodes u, v within $\mathcal{F}$ connected by a path P_n, we have $\chi_{\mathcal{M}}$ equals $\chi_{E(\mathcal{M}, \mathcal{F}, u, v, P_n)}$.

Proof. The lemma follows from the construction detailed in the proof of Lemma 6.8. The path addition of P_n between nodes u and v on face $\mathcal{F}$ increases the node count by $n+1$, edges by $n+2$, and faces by one, preserving the Euler characteristic. □

Euler characteristic serves as a planarity criterion for connected finite graphs. Specifically, according to Euler's formula, a graph G is planar under the map $\mathcal{M}$ if and only if

$\chi_{\mathcal{M}}$ equals two. The constructions detailed in this section facilitate the verification of Euler's formula for graphs constructed via path additions, an approach also employed later for biconnected planar graphs in Section 6.3.3.

However, for arbitrary graphs not derived from graph extensions, validating Euler's formula remains challenging, primarily due to the nontrivial task of determining the cardinality of the set of faces for an arbitrary map $\mathcal{M}$ of a given graph G, i.e., computing the set of elements of type $\mathsf{Face}(G, \mathcal{M})$ (Definition 4.14). Progress was made by establishing that the type of faces forms a finite set in Section 4.4.1. This suggests the feasibility of extracting this number in practise, possibly utilising the employed proof-assistant. We leave this to future work.

6.3.2 Planar synthesis of graphs

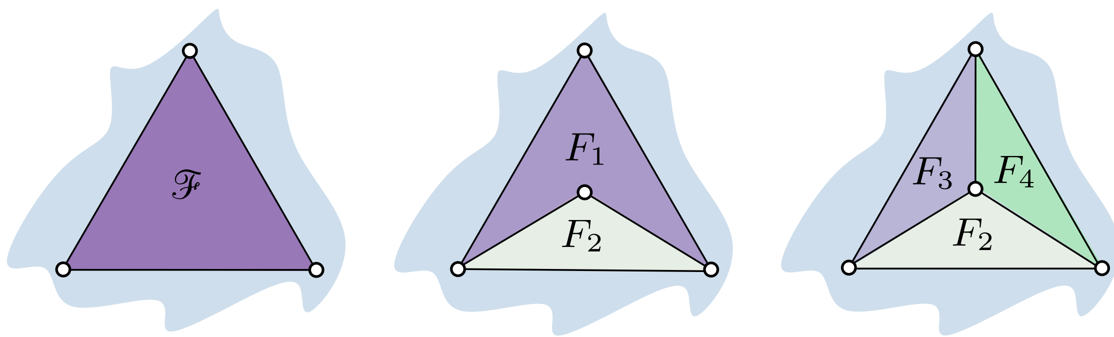


Figure 6.6: The figure illustrates a planar synthesis for constructing a K_4 planar map using a C_3 planar map. Initially, face $\mathscr{F}$ is divided into F_1 and F_2. Subsequently, F_1 is split into F_3 and F_4. The resulting map ends up with four faces, including the outer face.

Inductive graph construction methods abound, such as Whitney-Robbins synthesis, ear decomposition of a graph, and the K_4 construction depicted in Figure 6.6. Drawing inspiration from these methods and face divisions (Lemma 6.8), we propose a method to build larger planar graphs using graph extensions, ensuring that we remain within the type of planar graphs.

Definition 6.11. A Whitney *synthesis* (synthesis for short) of graph G from graph H is defined as a sequence of graphs $G_0, G_1, \cdots, G_n$, where G_0 is H, G_n is G, and each G_i results from the path addition of p_i to G_{i-1} for i in the range 1 to n. Consequently, G can be viewed as the result of adding paths $p_1, p_2, \cdots, p_n$ to H:

$$G \equiv H \bullet p_1 \bullet p_2 \bullet \cdots \bullet p_n.$$

The *length* of this synthesis is n. A *simple synthesis* refers to a sequence containing only simple additions. Conversely, a sequence composed solely of non-simple additions is termed a *non-simple synthesis*.

Lemma 6.12. Syntheses preserve graph connectedness. Specifically, if a graph H is connected and G is synthesised from H, then each intermediate graph G_i in the synthesis sequence is also connected.

Proof. We prove this by induction on the length of the synthesis, and the fact that path additions preserve connectedness, Lemma 6.6. □

Definition 6.13. Given a planar map $\mathcal{M}$ of the graph H with outer face $\mathcal{F}$, we define a *planar synthesis* of G from H of length n as a sequence

$$(G_0, \mathcal{M}_0, \mathcal{F}_0), (G_1, \mathcal{M}_1, \mathcal{F}_1) \cdots, (G_n, \mathcal{M}_n, \mathcal{F}_n),$$

where:

- ▷ $(G_0, \mathcal{M}_0, \mathcal{F}_0)$ is equivalent to $(H, \mathcal{M}, \mathcal{F})$, and
- ▷ $(G_n, \mathcal{M}_n)$ corresponds to $(G, E(\mathcal{M}_{n-1}, \mathcal{F}_{n-1}, p_{n-1}))$.

For each i in the range 1 to n, the graph G_i is $G_{i-1} \bullet p_i$, and the map $\mathcal{M}_i$ is $E(\mathcal{M}_{i-1}, \mathcal{F}_{i-1}, p_{i-1})$, where $\mathcal{F}_{i-1}$ is a face of $\mathcal{M}_{i-1}$.

Lemma 6.14. If a graph G is synthesised from a planar graph H via planar synthesis, then G and every graph in the corresponding sequence are planar.

Proof. Through planar synthesis, each G_i is derived from G_{i-1} via path addition, ensuring planarity by Lemma 6.8. □

While we have not yet employed non-simple additions, they become relevant when we characterise planar biconnected graphs in the next section. It is possible to extend the face division lemma and construction to utilise non-simple additions, Lemma 6.8, allowing us to adapt not only the planar synthesis in Definition 6.13 to *non-simple planar syntheses*, but also Lemma 6.14 to accommodate non-simple additions. Hence, given a map $\mathcal{M}$ for G with a face $\mathcal{F}$, the corresponding planar map for $G \bullet \overline{p}$ is denoted as $E(\mathcal{M}, \mathcal{F}, \overline{p})$, maintaining a similar notation as before. As with path additions, extending the map with non-simple additions introduces new faces.

Taking into account $G \bullet_{u,v} \mathsf{Sym}(P_n)$ and the map $E(\mathcal{M}, \mathcal{F}, \overline{p})$, the number of faces increases to $n + 3$, the number of nodes increases to $v + n + 1$, and the number of edges increases to $2 \cdot (n + 2)$, as illustrated in Figure 6.7. Consequently, the Euler characteristic of $E(\mathcal{M}, \mathcal{F}, \overline{p})$ is equal to the Euler characteristic of $\mathcal{M}$.

$$\chi_{E(\mathcal{M},\overline{p})} :\equiv (v + (n + 1)) - (e + 2 \cdot (n + 2)) + (f + (n + 3)) \equiv \chi_{\mathcal{M}}.$$

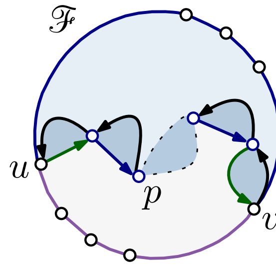


Figure 6.7: The figure illustrates the face division of $\mathscr{F}$ by a non-simple path addition.

Figure 6.5b demonstrates the construction of larger planar graphs using various path, cycle, and spike additions. A *spike addition* to G, although not precisely defined here, as it is not extensively used for further constructions, can be essentially described as a path addition sharing only one node with G. With a given map for G, a simple addition of a spike creates a new face of higher degree than the face where the spike is inserted. Consequently, non-simple spike additions also increase the number of faces for the extended map due to the emergence of new faces between edge pairs that share endpoints.

6.3.3 Biconnected planar graphs

This subsection aims to characterise the construction of all 2-connected planar graphs.

In general, a graph is k-*connected* if it cannot be disconnected by removing less than k nodes. Depending on k, there are various methods to construct the set of k-connected graphs. For instance, any undirected 2-connected graph can be constructed by applying path additions to an appropriate cyclic graph (Diestel 2012, §3). Our focus in the following will be on the construction of 2-connected planar graphs.

Definition 6.15. A graph G is defined as 2-*connected*, or biconnected, when the proposition $\mathsf{Biconnected}(G)$ holds. This is, when the resulting graph $G - x$, formed by removing a node x from G, remains connected.

Specifically, $G - x$ is the graph made up of the set of nodes, $\Sigma_{y : \mathsf{N}_G}(x \neq y)$, and their corresponding edges in G.

$$\mathsf{Biconnected}(G) :\equiv \prod_{(x : \mathsf{N}_G)} \mathsf{Connected}(G - x).$$

Lemma 6.16. If G is a cyclic graph, then $\mathsf{Sym}(G)$ is 2-connected.

Proof. The cyclic nature of G ensures that in $\mathsf{Sym}(G)$, there are two inner loop-free walks between any pair of nodes: a direct walk following the cycle of G, and a reverse walk counter to the cycle. These walks are edge-disjoint and thus preserve the graph's connectivity despite the removal of any single node—only one of the walks might be affected, leaving the other intact to sustain connectedness. □

The property of 2-connectedness in a graph does not remain invariant under simple path additions. Clearly, removing a node from the added path p disconnects $G \bullet p$. Yet, through non-simple path additions, it is possible to maintain and even augment 2-connected graphs.

Lemma 6.17. Let G denote a 2-connected graph. The graph extensions, $G \bullet \overline{p}$ and $\mathsf{Sym}(G) \bullet \overline{p}$, preserve the 2-connected graph property.

Proof. To show that $G \bullet_{u,v} \mathsf{Sym}(P_n) - x$ remains connected for any node x in $G \bullet \overline{p}$, we consider the location of x. If x is within G, then $G - x$ is connected by hypothesis. Applying Lemma 6.6, it follows that $G \bullet \overline{p} - x$ is also connected, showing the 2-connectivity of $G \bullet \overline{p}$. Otherwise, if x lies on $\overline{p}$, its removal divides $\overline{p}$ into two parts, p_1 and p_2. For any two nodes in $G \bullet \overline{p} - x$ we show they are connected. If both nodes are in G or the same part p_i, they are connected by prior arguments or direct traversal, respectively. If they are located in distinct subgraphs, say x is in p_1 and y is in p_2. We can construct a walk from x to u, another walk across G from u to v (since G is connected), and then to the second node, Hence, $G \bullet \overline{p}$ maintains 2-connectivity. □

Inspired by Yamamoto's work in (Yamamoto, Nishizaki, Hagiya, et al. 1995), our focus is on the construction of 2-connected planar graphs. Within a different theoretical setting (HOL) and using a different graph definition, Yamamoto shows that any undirected 2-connected planar graph can be inductively built by adding diverse *paths* to *circuits* (their term for *cyclic graphs*). In our context, we initiate constructions with any 2-connected graph $\mathsf{Sym}(C_n)$ and subsequently extend these graphs by non-simple planar additions.

Lemma 6.18. In a non-simple Whitney synthesis of G originating from a 2-connected graph H, with planarity ensured by a map $\mathcal{M}$, each graph in the synthesis maintains 2-connectivity and planarity via planar extension of $\mathcal{M}$ using non-simple additions.

Proof. Assuming a non-simple Whitney synthesis of G from H of length n is given, we proceed by induction on n.

- ▷ **Base case ($n = 0$):** The graph G is H, and by hypothesis, H is a 2-connected planar graph. Thus, the conclusion follows.
- ▷ **Inductive step:** For the inductive step, we assume that the claim holds for a sequence of length n, thus establishing G_n as a 2-connected planar graph via map $\mathcal{M}_n$. We then aim to demonstrate that G, defined as $G_n \bullet \overline{p_i}$ for some path p_i, also qualifies as a 2-connected planar graph.
- ▷ Given that G_n is 2-connected, it follows from Lemma 6.17 that $G_n \bullet \overline{p_i}$ also retains this property. We then extend the planar map $\mathcal{M}_n$ of G_n to a planar map $\mathcal{M}$ for

G, preserving the outer face or selecting a new outer face from the additions. This construction of $\mathcal{M}$ follows the method in Lemma 6.8, where we expand planar maps using simple additions, as required here. □

Lemma 6.19. Any graph G, synthesised from $\mathsf{Sym}(C_n)$ through non-simple Whitney syntheses is a 2-connected planar graph.

Proof. Given that C_n is planar by Example 6.3 and consequently connected, $\mathsf{Sym}(C_n)$ is 2-connected by Lemma 6.16. By repeatedly applying Lemma 6.18 to each step in the given synthesis sequence, we ensure the resulting graph's 2-connectivity and planarity. □

Lemma 3 and Proposition 4 in (Yamamoto, Nishizaki, Hagiya, et al. 1995) discuss undirected 2-connected planar graphs similar to the converse of Lemma 6.19. It is possible to follow Yamamoto's argument closely, even though it was presented in an informal way. However, this requires preliminary formalisation of several technicalities, such as maximal subgraphs, adjacent faces, and edge sequence deletion. Subsequently, one can assert that non-simple Whitney syntheses entirely determine 2-connected planar graphs, as expressed similarly in (Diestel 2012, §3). In essence, any graph defined as planar in Definition 6.1 and 2-connected in Definition 6.15, can be inductively generated from $\mathsf{Sym}(C_n)$ via iterative non-simple path additions and proper map extensions.

Further exploration of graph extensions, such as amalgamations, appendages, deletions, contractions, and subdivisions, should be considered to generate planar graphs (J. Gross, Yellen, and Anderson 2018, §7.3).

7

Concluding Remarks and Future Work

In the following, we give a brief summary of the contributions of this thesis, that are novel in HoTT, as far as we know. The order of the chapters is not accidental in the document. The final chapter, Chapter 6, present, among other things, our characterisation of planarity of connected and locally finite directed multigraphs using graph maps also referred to as combinatorial maps or rotation systems. This characterisation is significant as it wraps up our main constructions. For example, the type of planar maps of a graph G requires us to define the type of graph maps, the subtype of spherical maps, and the type of faces introduced in Definition 5.43 and Definition 4.14, respectively.

In addition to the technical definition given in Chapter 6 for the planarity of graphs, we believe that we have encoded, in a better combinatorial and more general way, the essence of the topological intuition behind it. Rather than stating planarity only as a property of the graph itself, we have defined it here as structure on the type of graphs, a different approach compared to other works, see, for example, definitions in terms of hypermaps and cyclic lists and expressed in other proof-irrelevant type theories (G. J. Bauer 2005; Gonthier 2008; Yamamoto, Nishizaki, Hagiya, et al. 1995). In other words, we characterised the identity types of the type of planar maps of a graph and showed that it forms a homotopy set, as shown in Theorem 6.2. This result is significant and common theme in HoTT when defining new types. To support this claim, we developed a few lemmata, as listed below, and proven in the same order they appear.

1. The star at any node x of G is a set, see Lemma 4.6.

2. The collection of all graph maps for G, and, in particular, the subtype of its spherical maps, forms a set, see Section 4.3 and Lemmas 5.22 and 5.46.

3. The subtype of walks without inner loops of G, here called quasi-simple walks, form a set, see Theorem 5.26.

4. The faces of any graph map of G is a set, see Lemma 4.18 and Theorem 4.26.

5. The collection of planar maps of G is a set, see Theorem 6.2.

To support the previous results, we gave new proofs in Chapter 5 to a few non-trivial facts about quasi-simple walks (walks without internal loops) and spherical maps, two key concepts introduced in (Prieto-Cubides 2022). The main contributions to this regard are Theorems 5.38 and 5.48, and especially Corollary 5.49. Briefly, the former gives a normalisation algorithm for walks. Given any walk, we can always find its normal form which removes all the internal loops with evidence that the normal form is walk-homotopic to the original walk. This occurs whenever the graph has a discrete node set and is embedded in the sphere. The latter, Corollary 5.49, on the other hand, establishes an equivalence between the two definitions for spherical maps. This result confirms that one can ignore loops and multiple edges when considering spherical maps of graphs where the node set is discrete. Except for this last result, the machinery shown in Chapter 5 was completely unexpected and developed solely to find evidence for our initial conjecture, Corollary 5.49.

Moreover, for characterising maps of finite graphs in the sphere, we found that considering only the finite set of quasi-simple walks suffices. Using the results mentioned herein, one could devise an algorithm to determine whether a graph map is spherical or not; see Lemma 5.45. Additionally, we have shown that the set of planar maps is finite, provided that the graph is also finite. We use this result in Section 6.3, where we introduce planar extensions and the Euler characteristic number for planar graphs. To this end, we presented a method for constructing planar graphs using planar extensions inductively. This method is inspired by Yamamoto's work on biconnected planar graphs (Yamamoto, Nishizaki, Hagiya, et al. 1995). See, for example, the construction of a planar map of K_4 based on a map for C_3 by using simply path additions.

As part of our contributions, we provided computer proofs of most results in the dependently typed programming language Agda, see Appendix A. The Agda formalisation turned out to be helpful on several occasions. For example, we use our formalisation to confirm that only a subset of HoTT was necessary to perform all the proofs in Chapter 5. Precisely, the formalisation of that chapter only needed the intensional Martin-Löf type theory equipped with universes, function extensionality, and propositional truncation. No other higher inductive types nor Univalence was required. Moreover, we also used the computer formalisation to identify flaws, missing assumptions, and new proofs.

7.1 Directions of further developments

There are several directions for further research on the topics of this thesis. Let us mention a few of them.

For example, there exist other criteria of planarity in literature, for instance, Kuratowski's and Wagner's characterisations for planar graphs. An interesting result would be to prove that our notion of planarity is equivalent to one of these characterisations.

Another possible direction is the study of surfaces in HoTT as the topological representation/realisation. At the moment of writing, defining the notion of a surface is still an open problem in HoTT. On this regard, consider the torus as the realisation of the bouquet graph consisting of two edges using the graph map M_c given in Figure 4.8.

Transferring this to HoTT would mean a mapping, hopefully, an equivalence, between a higher inductive type representing the torus (Univalent Foundations Program 2013, §6.6) and a type representing the topological realisation of the bouquet graph with the graph map M_c. Concerning planar maps, we would expect the correspondence between any planar map and the type of the 2-sphere as defined in (7.1−1), spotlighted by the stereographic projection, as illustrated in Figure 6.1. Let us elaborate a bit more on this regard.

Specifically, we conjecture that there is an equivalence between the 2-cell topological realisation of a graph G with a planar map $\mathscr{M}$, and the type of the 2-sphere $\mathbb{S}^2$.

$$\begin{aligned} &\textbf{data}\ \mathbb{S}^2 : \mathcal{U} \\ &\qquad \textbf{base} : \mathbb{S}^2 \\ &\qquad \textbf{surf} : \mathsf{refl}_{\mathsf{base}} = \mathsf{refl}_{\mathsf{base}}. \end{aligned} \tag{7.1−1}$$

To elaborate on this conjecture, we first need to introduce two distinct geometric realisations of graphs, the 1- and 2-cell topological representations of a grap. These construction are further explained in Appendices B and C.

The 1-cell topological realisation of a graph G is denoted by $\mathbb{T}^1(G)$ and can be defined using the following HIT.

$$\begin{aligned} &\textbf{data}\ \mathbb{T}^1\,(G : \mathsf{Graph}) : \mathcal{U} \\ &\qquad \mathbb{n} : \mathsf{N}_G \to \mathbb{T}^1(G) \\ &\qquad \mathbb{e} : \Pi_{(a\,b\,:\,\mathsf{N}_G)}\,.\,\mathsf{E}_G(a,b) \to \mathbb{n}(a) = \mathbb{n}(b). \end{aligned}$$

Given a graph map $\mathscr{M}$ for the graph G, let us consider the 2-cell topological realisation of G, $\mathbb{T}^2(G,\mathscr{M})$, which can be defined using the HIT in (7.1−2). The function $\mathbb{w}$ used in (7.1−2) maps a walk to a path, see Appendix B.5.

$$
\begin{aligned}
&\textbf{data}\ \mathbb{T}^2\ (G : \mathsf{Graph})\,(\mathcal{M} : \mathsf{Map}(G)) : \mathcal{U}\\
&\qquad \mathbb{n} : \mathsf{N}_G \to \mathbb{T}^2(G, \mathcal{M})\\
&\qquad \mathbb{e} : \Pi_{(a\,b\,:\,\mathsf{N}_G)}\,.\,\mathsf{E}_G(a,b) \to \mathbb{n}(a) = \mathbb{n}(b)\\
&\qquad \mathbb{f} : \Pi_{(\mathcal{F}\,:\,\mathsf{Face}(G,\mathcal{M}))}\,.\,\Pi_{(a\,b\,:\,\mathsf{N}_{\mathcal{F}})}\,.\,\mathbb{w}(\mathsf{cw}(\mathcal{F}, a, b)) = \mathbb{w}(\mathsf{ccw}(\mathcal{F}, a, b)).
\end{aligned}
\tag{7.1–2}
$$

Our first conjecture states that the type $\mathbb{T}^2(C_0, m)$ is equivalent to the type of the 2-sphere $\mathbb{S}^2$, where C_0 is the unit graph consisting of one node with no edges, and m denotes the only graph map for C_0, see Example A.1.

Conjecture 7.1. Let C_0 be the unit graph consisting of one node with no edges, and m be the graph map for C_0. Then,

$$\mathbb{T}^2(C_0, m) \simeq \mathbb{S}^2. \tag{7.1–3}$$

As expected, as illustrated in Figure 7.1, establishing the back-and-forth correspondence between the two types is straightforward. Recall that there is only one graph map for C_0, m, and one face for such a map. However, the difficulty lies in proving the correspondent homotopies for this equivalence. It involves two HITs, which are not easy to work with, especially because of the induction principles for these types due to their path constructors. Therefore, we believe lemmata developed on Spheres (Univalent Foundations Program 2013, § 6) would come in handy, and the work done in the induction principle for $\mathbb{T}^2(C_0, m)$ (Appendix C.1.3).

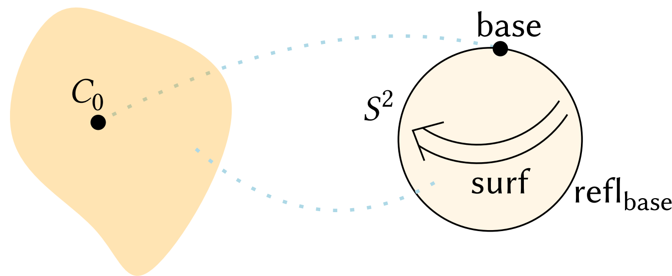


Figure 7.1: The correspondence between the sphere $\mathbb{S}^2$ and the 2-cell topological realisation of the unit graph C_0.

In Lemma B.29 we established that the 1-cell topological realisation of a tree yields a contractible type. Hence, it is reasonable to conjecture an analogous outcome for the 2-cell realisations of graphs. A tree with only one map and one face, when realised as a 2-cell topological space, should yield the same type as realising the unit graph.

Conjecture 7.2. Let G be a tree, as defined in Definition B.5, and $\mathscr{M}$ be its map, then,

$$\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{T}^2(C_0, m). \tag{7.1–4}$$

Now consider a graph map $\mathscr{M}$ for a graph G. We require an operation that can contract a face in a graph map to a singular node, a process akin to deforming a disk to a point. The proposed transformation entails contracting a face $\mathscr{F}$ within $\mathscr{M}$ for a graph G into a singular node, leading to a subgraph H of G, along with one new map $\mathscr{M}'$, a restriction of $\mathscr{M}$ to H. We denote such an operation as $(G, \mathscr{M}) \rightsquigarrow_{\mathscr{F}} (H, \mathscr{M}')$. The underlying conjecture is that such a transformation preserves the graph's planarity, ensuring that the resultant map $\mathscr{M}'$ is also planar.

Conjecture 7.3. Given graphs G and H with maps $\mathscr{M}$ and $\mathscr{M}'$ respectively, and a face contraction from $(G, \mathscr{M})$ to $(H, \mathscr{M}')$, if G is planar by $\mathscr{M}$, then H is planar by $\mathscr{M}'$.

Conjecture 7.4. Let $\mathscr{F}$ be a face of the graph map $\mathscr{M}$ and H be a subgraph of G with a graph map $\mathscr{M}'$. If $(G, \mathscr{M}) \rightsquigarrow_{\mathscr{F}} (H, \mathscr{M}')$, then,

$$\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{T}^2(H, \mathscr{M}'). \tag{7.1–5}$$

Assuming the provability of prior conjectures, we can construct an equivalence between the 2-cell topological realisation of any planar graph and the sphere $\mathbb{S}^2$. Let us state this result as a theorem.

Theorem 7.5. Let G be a nonempty finite graph with n nodes and $\mathscr{M}$ be a planar map for G. Then,

$$\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{S}^2. \tag{7.1–6}$$

Proof. We proceed by case analysis on the number of nodes of G. For $n = 1$, $G \cong C_0$ and the result follows from Conjecture 7.1. For $n > 1$, consider a planar map $\mathscr{M}$ of G. Given G's finiteness, we can proceed by induction on the number m of faces of $\mathscr{M}$, starting with the base case $m = 1$. Here, G is a tree and the desired equivalence is obtained via (7.1–4) and (7.1–3).

$$\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{T}^2(C_0, m) \simeq \mathbb{S}^2. \tag{7.1–7}$$

For the inductive step, assume we obtain the equivalence in question for graphs with $m-1$ faces. Now, contracting a face $\mathscr{F}$ from G yields a new graph G' with a corresponding map $\mathscr{M}'$, which is planar since contracting a face preserves planarity (Conjecture 7.3),

$$(G, \mathscr{M}) \rightsquigarrow_{\mathscr{F}} (G', \mathscr{M}').$$

Conjecture 7.4 provides an equivalence $\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{T}^2(G', \mathscr{M}')$. Since G' has $m-1$ faces, the induction hypothesis implies $\mathbb{T}^2(G', \mathscr{M}') \simeq \mathbb{S}^2$. We can then establish the following chain of equivalences, from which the conclusion follows.

$$\mathbb{T}^2(G, \mathscr{M}) \simeq \mathbb{T}^2(G', \mathscr{M}') \simeq \mathbb{S}^2. \qquad \square$$

7.2 Formalisation

Although formalisation is an important aspect of our thesis, it is not our primary focus. Due to the time-consuming nature of formalising concepts in a proof assistant, we have prioritised our efforts accordingly. Nonetheless, besides providing insights into the conjectures discussed earlier, the future work involves completing the elaboration of some of the contributions listed earlier. This includes proving the planarity of any cyclic graph in Agda, as outlined in Example 6.3, and expanding on the main results about planar extensions discussed in Section 6.3, such as Lemmas 6.8 and 6.19. On this regard, we expect extract an algorithm for computing the number of faces of a given map for finite graphs to validate that the Euler characteristic number for planar graphs is 2. A starting point for this must be the proof's formalisation in Section 4.4.1, demonstrating the finiteness property of face types.

"Those who cannot remember the past are condemned to repeat it."

George Santayana

Epilogue

This thesis titled *Investigations on Graph-Theoretical Constructions in HoTT* documents my research from late 2018 to 2022. It was conducted at the ICT Research School of the University of Bergen and adds to the study field of Homotopy Type Theory/Univalent Foundations. This field converges constructive mathematics, logic, type theory, category theory, homotopy theory, algebraic topology, and formalisation of mathematics. The manuscript as it stands today have been improved by the feedback of my advisors, Håkon Gylterud and Marc Bezem.

I have structured each chapter in my writing to focus on a specific subject. Some chapters include an introduction and a discussion section. To provide a contextual overview of the thesis and connect its key elements, I have included a summary of the main results and raised conjectures in the conclusion chapter. This will allow readers who want to explore further research to do so. The appendices contain additional results that did not fit well within the main chapters but are still relevant to the thesis and, in my opinion, quite interesting.

In the following, I briefly describe how the research for this thesis was carried out, which may be of interest to those considering a similar project.

The research journey for this thesis began in 2018 when I began my PhD at the University of Bergen. Given the freedom to explore my interests, I attended Marc Bezem's introductory seminar on advances topics in programming languages, which covered Homotopy Type Theory. It was during these sessions that I met Håkon Gylterud, a researcher who occasionally attended the seminars and later became my primary advisor.

In the summer of 2018, I collaborated with Marc to illustrate equivalences related to the circle and another equivalence involving the type of *pathovers*[1]. This collaboration deepened my understanding of how HoTT uses dependent types to encapsulate diverse concepts and the strictness of its constructive nature. I saw the potential of HoTT as a formal system for creating/defining new constructions in a precise and discipline way, which led me to pursue a PhD in this field. Combinatorics emerged as the primary option due to the scarcity of research in this area. Thus, I decided to explore this area.

In late 2018, I began to adapt graph theory concepts to HoTT, with a particular fo-

[1] https://jonaprieto.github.io/type-theory/2018/07/05/pathovers-hott/index.html

cus on potential characterisations of graph planarity. This interest was likely sparked by Gonthier's formalisation of the Four Colour Theorem (4CT) (Gonthier 2008), a proof verified using the proof assistant Coq, asserting that *every finite planar graph can be coloured with no more than four colours.* Håkon shared this interest, leading to our collaboration on the topic.

Our initial proposal on a type of planar embeddings for undirected graphs was presented at TYPES (Prieto-Cubides and Håkon Robbstand Gylterud 2019). Feedback from the conference led us to broaden our scope to include the planarity of directed multigraphs. The remainder of my research focused on redefining and formalising multiple constructions in Agda, a proof assistant similar to Coq.

This iterative process allowed me to prove, disprove, and refine my initial conjectures. One such conjecture, which remained unproven until 2021, was later formalised in Agda as Corollary 5.49.

In the summer of 2019, I attended the CMU HoTT Summer School in Pittsburg, where I was introduced to Cubical methods by Anders Mörtberg. Inspired by his lectures, I began using Cubical Agda for my constructions, summarised in Appendix B. This appendix, along with the work in Appendix C, focusses on graph embeddings as a way to realise graphs as spaces in HoTT. This construction was inspired by geometrical intuition and the concept underlying higher inductive types. Later I would discover that this construction is used by Swan in his proof on the Nielsen-Schreier Theorem in Homotopy Type Theory (Swan 2022).

During the 2020-2021 COVID-19 pandemic, I organised a weekly seminar[2] on type theories, Haskell, and Agda. In collaboration with fellow PhD students and friends from the PLT research group at UiB (Eli, Benji, Tam, Knut, and Max) we engaged in the study and discussion of various topics. These included content from the PLFA book (Kokke, Siek, and Wadler 2020), homotopy theory as presented in Cubical Agda (Mörtberg and Pujet 2020), and the topology of data types[3] (Escardó 2004). As an outcome of this seminar, I could prove a few lemmas on homotopy walks, which are summarised in Chapter 5. That was conceived during the preparation of talks on Lambda Calculus and Term Rewriting Systems.

Formalisation of mathematics The computational aspect of my project, the formalisation of mathematics, is another key component. Gonthier's work, which used Coq, a proof assistant based on the *Calculus of Inductive Constructions*—inspired me. However, I opted for Agda, a dependently typed functional programming language, due to its inherent support for HoTT and my previous experience with Haskell and Agda during my

[2] https://nextjournal.com/uib-types/meetings

[3] https://www.cs.bham.ac.uk/~mhe/papers/entcs87.pdf

master's studies.

When my formalisation project began in late 2018, I contemplated using the recommended HoTT-Agda library. However, compatibility issues with the latest compiler version led me to develop my own library for Agda (v2.6.0+). Despite discovering Escardo's TypeTopology library at the Midlands Graduate School in Birmingham in April 2019, I continued with my library, as it was already in progress. At the end, although my development fulfils the formalisation requirements, I underestimated the time and effort needed to develop/maintain a library. In this regard, for those considering formalisation, I suggest examining existing libraries in your field and contributing where gaps are identified. Fortunately, many libraries exist today.

Here are some libraries compatible with recent Agda versions and related to HoTT and its derivatives, listed alphabetically.

- ▷ Vanilla Agda with HoTT support:

Agda-UniMath	`http://unimath.github.io/agda-unimath`.
Swan's fork of HoTT-Agda	`https://github.com/awswan/HoTT-Agda/tree/agda-2.6.1-compatible`.
TypeTopology	`https://www.cs.bham.ac.uk/~mhe/TypeTopology/`.
Thesis's Agda library	`https://jonaprieto.github.com/synthetic-graph-theory`.

- ▷ Cubical methods in Agda:

1Lab	`https://1lab.dev/`.
Cubical Agda	`http://www.github.com/agda/cubical`.

Final Comment The exploration of graph planarity within HoTT uncovered a range of unexpected constructions, many of which emerged through iterative interactions with the proof assistant, Agda, in my case. The use of formal methods, formal systems such as HoTT, and tools such as proof assistants truly offer an engaging approach to our understanding of mathematics, promoting a more profound comprehension and stimulating novel insights. Type theories, in particular, HoTT, are a promising formal system for the study of mathematics and hold remarkable potential for new discoveries. This includes new proofs of existing results and new results that are impossible or hard to prove in other formal systems. There is still much to be done in this area, including improving the computer tooling around the formalisation process in these systems. About this work, I hope the topics developed in this manuscript serve as both a stimulus and groundwork for future research, inspiring further refinement, completion, or new (mathematical) constructions on graphs in HoTT.

"It soon became clear that the only real long-term solution to the problems that I encountered is to start using computers in the verification of mathematical reasoning."
"A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail."

Vladimir Voevodsky, Univalent Foundations, March 26, 2014.

Computer Formalisation in Agda

This thesis includes a set of mechanised proofs and constructions, verified using Agda `v2.6.2.2-442c76b`, our chosen proof assistant. The formalisation comprises its own self-contained Agda library of a subset of the HoTT book's foundations and the central elements of the thesis. For our experiments with Cubical Agda in Appendix B, we used the Cubical Agda version `v0.3` of the library. This projects can be found at the following address:

▷ `https://jonaprieto.github.io/synthetic-graph-theory/`.

A.1 Proof assistants

Proof assistants are sophisticated computer programs that function as tools to develop formal proofs, making the process of verifying correctness more efficient (up to the correctness of the proof assistant itself). When used in combination with dependently typed programming languages, these tools cater not only to the programming language community but also to those who require rigorous and trustworthy communication methods with a powerful and expressive language. Their adoption may offer a significant improvement over traditional human methods, such as the revision process of mathematical papers.

Our choice of Agda as our proof assistant to conduct this investigation stems from its modernity, robustness, and reliance on a powerful intuitionistic type theory. Crucially,

its type system fully backs this thesis' HoTT focus by desabling the Axiom K (Cockx, Devriese, and Piessens 2016) and enabling term rewriting via the REWRITE pragma[1], which allow us to define high-inductive types and their computational rules. For our experiments in Appendix B, we used Cubical Agda, which is a mode of Agda that offers backing for cubical type theories, enhancing higher inductive type definitions via pattern matching.

Despite its flexibility and power, Agda lacks maturity and robustness in certain areas such as program synthesis via auto, especially when compared to systems like Coq, Isabelle, and Lean. This can result in lengthy explicit proof terms, making the formalisation process somewhat tedious, even for small-scale developments. However, its emphasis on dependently typed programming and its commitment to incorporates the latest developments in type theory makes it a suitable tool for conducting research effectively.

A.2 Agda notation

Several examples included in this chapter and the appendices use Agda syntax. We offer below a very short description of the main constructs of the language. For an in-depth exposition on Agda, refer to its official documentation (The Agda Development Team 2023) and the sources of the this thesis for any types not explicitly defined herein.

- Type annotations in Agda are written similar as on paper. For example, `x : A` is used to indicate that `x` is a term of type `A`.
- `Type` denotes a type family of types indexed by their universe level with the following hierarchy:

  `Type 0 : Type 1 : ... : Type ℓ : Type (lsuc ℓ) : ...`

  Thus, `A : Type ℓ` indicates that `A` is a type in the universe `ℓ`. Most of the time, our definitions are universe polymorphic, i.e., they hold for any universe level `ℓ`.
- The empty type is denoted by `𝟘` or `⊥` and the unit type by `𝟙` or `⊤`. Also, we use standard type formers such as (`→`) for function types, (`×`) for product types, Σ-types through the use of the `Σ` type former, and Π-types for functions with codomain varying over the domain, e.g. the type `((x : A) → B x)` in Agda signifies a function that takes in a term `x` of type `A` and returns a term of type `B x`, given that `B` is a type family over `A`.
- The identity type on a type `A` between x and y is denoted by `Path {A} x y`, or simple as $x \equiv y$. The constructor for the identity type is `refl`.

[1] `https://jesper.sikanda.be/posts/hack-your-type-theory.html`

- Propositional truncation of a type A is denoted by ∥ A ∥, and its constructor is |_|, so | a | is a term of type ∥ A ∥ for any a : A.

- Declaring inductive data types is done through the use of the data keyword. For example, the following declaration defines the type of natural numbers.

```
data ℕ : Type 0 where
 zero : ℕ
 suc : ℕ → ℕ
```

- We can define functions by pattern matching whenever the domain is an inductive data type. For example, the following function add defines addition on natural numbers.

```
add : ℕ → ℕ → ℕ
add zero n = n
add (suc m) n = suc (add m n)
```

  Alternatively, we can define functions using the with keyword, which allows us to pattern match on an expression. An equivalent definition of add using with is as follows.

```
add' : Nat → Nat → Nat
add' n m with n
... | Z = m
... | S x = S (add' x m)
```

- Records that act as named dependent product types are declared with the record keyword, where the components are declared after the field keyword and which can be accessed with their respective projections. For example, the following declaration represents Σ-types.

```
record Σ {ℓ₁ ℓ₂} (A : Type ℓ₁) (B : A → Type ℓ₂) : Type (ℓ₁ ⊔ ℓ₂) where
  constructor _,_
  field
    π₁ : A
    π₂ : B π₁
```

- Modules encapsulate declarations, serving as namespaces. To declare a module, use the module keyword, which may include parameters of types and terms. For instance, the module M, parameterised by a type A and a term a, is defined as follows. Anonymous modules are declared with _ as the name.

```
module M (A : Type) (a : A) where
```

- To import definitions from a module into the current scope, the keyword import is used. These imported names are qualified by the module name they are imported

from. To bring all the names, unqualified, from a module into the current module, the keyword `open` is used. For example, the following statement imports the module `M` and brings all its names into the current scope.

```
open import M
```

▷ One powerful feature of Agda is the ability to infer implicit arguments for calls to functions, including data constructors. Implicit arguments in Agda are denoted by curly braces, e.g. `{A : Type}`.

A.3 Library

The Agda codebase accompanying this thesis comprises a library of 15459[2] lines of code, including the formalisation of the HoTT book's foundations used in this thesis and the central topics of the thesis. The library is structured into two main directories: *foundations* and *lib*. The former aligns with the essential background from the HoTT book, while the latter contains the central elements of the formalisation, further divided into distinct modules.

```
{-# OPTIONS --without-K --exact-split --rewriting #-}

module agda-index where

-- • Graph definitions as seen in the document
import lib.graph-definitions.Graph -- As in the document
import lib.graph-definitions.Alternative-definition-is-equiv

-- • Graph forms a univalent category
import lib.graph-definitions.Graph.EquivalencePrinciple
import lib.graph-definitions.Graph.IsomorphismInduction
import lib.graph-definitions.Graph.isGroupoid
import lib.graph-homomorphisms.Hom
import lib.graph-homomorphisms.classes.Isomorphisms
import lib.graph-homomorphisms.classes.Isomorphisms.Exponentiation

-- • Special set of graph homomorphisms
import lib.graph-homomorphisms.classes.EdgeInjective
import lib.graph-homomorphisms.classes.Injective
import lib.graph-homomorphisms.classes.EdgeInjective.Lemmas
import lib.graph-homomorphisms.Lemmas

-- • Graph isomorphisms/equivalences
import lib.graph-relations.Isomorphic
import lib.graph-relations.Isomorphic.isSet
import lib.graph-relations.Homomorphic
import lib.graph-calculation-reasoning.Isos

-- • Graph walks
import lib.graph-walks.Walk
import lib.graph-walks.Walk.Composition
import lib.graph-walks.Walk.SigmaWalks
```

[2]Calculation performed using the `loc` command from the https://github.com/cgag/loc tool.

```
import lib.graph-walks.Walk.Equality
import lib.graph-walks.Walk.isSet

-- • Quasi-simple walks
import lib.graph-walks.Walk.QuasiSimple
import lib.graph-walks.Walk.QuasiSimpleFinite

-- • Graph transformations, symmetrisation of graphs
import lib.graph-transformations.U
import lib.graph-transformations.W
import lib.graph-transformations.Inv

-- • Graph maps/embeddings
import lib.graph-embeddings.Map
import lib.graph-embeddings.Map.Face
import lib.graph-embeddings.Map.Face.isSet
import lib.graph-embeddings.Map.Face.Walk
import lib.graph-embeddings.Finiteness

-- • Walk-homotopy and whiskering
import lib.graph-embeddings.Map.Face.Walk.Homotopy
import lib.graph-embeddings.Map.Face.Walk.Whiskering

-- • Graph maps into the sphere (spherical maps)
import lib.graph-embeddings.Map.Spherical
import lib.graph-embeddings.Map.Spherical-is-enough

-- • Planar graph maps
import lib.graph-embeddings.Planar
import lib.graph-embeddings.Planar.isSet

-- • The one-point graph is planar.
import lib.graph-embeddings.Map.Face.Example

-- • One higher-inductive of a graph with 2-cells
-- And the reason we need the flag --rewriting.
import HIT
import HIT-toProp
import HIT-toSet
import Homotopic-are-equal

-- • Some families of graphs seen in the document
import lib.graph-families.CycleGraph
import lib.graph-families.CycleGraph.RotHom
import lib.graph-families.CycleGraph.Isomorphisms.IdentityType
import lib.graph-families.CycleGraph.Isomorphisms.Lemmas
import lib.graph-families.CycleGraph.Map
import lib.graph-families.CycleGraph.isCyclicGraph
import lib.graph-families.CycleGraph.isFiniteGraph
import lib.graph-families.CycleGraph.Walk
import lib.graph-families.BouquetGraph
import lib.graph-families.CompleteGraph

-- • A few graph classes used in the document
import lib.graph-classes.CyclicGraph
import lib.graph-classes.CyclicGraph.Stuff
import lib.graph-classes.CyclicGraph.isFiniteGraph
import lib.graph-classes.CyclicGraph.Walk
import lib.graph-classes.EmptyGraph
import lib.graph-classes.UnitGraph
import lib.graph-classes.StarGraph
import lib.graph-classes.CompleteGraph
```

```
import lib.graph-classes.ConnectedGraph
import lib.graph-classes.FiniteGraph
import lib.graph-classes.UndirectedGraph
import lib.graph-classes.PartiteGraph

-- Appendix. Univalent foundations
----------------------------------

import foundations.Core   -- • Part I in the HoTT book
import foundations.Nat    -- • Lemmata about ℕ
import foundations.Fin    -- • ⟦ n ⟧ :≡ { 0,1,2,⋯, n-1}.
import foundations.Finite -- • Finite types Σ[ n ] ∥ A ≃ ⟦ n ⟧ ∥.
import foundations.Cyclic -- • Cyclic types
```

A.4 Short excerpts from the library

Let us present the definitions pertinent to the type of planar maps.

▷ Cycle types as introduced in Definition 2.21:

```
record
  CyclicSet (A : Type ℓ) : Type ℓ
  where
  constructor cyclic-set
  field
    φ : A → A
    n : ℕ
    cyclicity
      : ∥ Σ[ e ∶ (A ≃ ⟦ n ⟧) ]
          ((φ :> (e ∙→)) ≡ (e ∙→) :> fin-pred) ∥
```

▷ The symmetrisation of a graph, Sym, called here U for short (see Definition 4.1):

```
U : Graph ℓ → Graph ℓ
U g@(graph Ng Eg Ng-set Eg-set)
  = graph Ng
  (UEdge g)
  Ng-set
  λ x y → +-set (Eg-set x y) (Eg-set y x)
  where
    UEdge : (G : Graph ℓ) (x y : Node G) → Type ℓ
    UEdge G x y = (Edge G x y) + (Edge G y x)
```

▷ Star at a node as introduced in Definition 4.4:

```
Star : (G : Graph ℓ) → Node G → Type ℓ
Star G x = Σ[ y ∶ Node (U G) ] Edge (U G) x y
```

▷ Graph maps as introduced in Definition 4.8 :

```
Map : (G : Graph ℓ) → Type ℓ
Map G = Π[ x ∶ Node G ] CyclicSet (Star G x)
```

▷ Faces for a graph map as an inductive record type to ensure eta-equality by default. The fields within the record type represent the conditions outlined in Definition 4.14.

```
record Face (𝕞 : Map G) : Type (lsuc ℓ) where
  inductive
  constructor face
  field
    A : Graph ℓ
    AG : CyclicGraph ℓ A
    h : Hom A (U G)
    h-is-edge-inj : isEdgeInj h
    star-cond : starFaceCond A h
    corners-cond : faceCornersPreserved 𝕞 A AG h
```

The type of these fields use the following definitions:

- Cyclic graph:

```
record
  CyclicGraph (ℓ : Level) (G : Graph ℓ) : Type (lsuc ℓ)
  where
  eta-equality
  constructor cyclic-graph
  field
    φ : Hom G G
    n : ℕ
    is-cyclic : ∥ Path {A = Σ[ H ] (Hom H H)}
                        (G , φ) (Cycle ℓ n , rot ℓ n) ∥
```

- Edge-injectivity property for graph homomorphisms:

```
isEdgeInj : Hom G H → Type (ℓ₁ ⊔ ℓ₂)
isEdgeInj (hom α β)
  = ∀ {x y} → (e₁ : Edge G x y)
  → ∀ {x' y'} → (e₂ : Edge G x' y')
  → Path {A = Σ[ x ] Σ[ y ] Edge H x y}
    (α x , α y , β x y e₁)
    (α x' , α y' , β x' y' e₂)
  → (x , y , e₁) ≡ (x' , y' , e₂)
```

- Conditions on the stars:

```
starFaceCond : Type ℓ
starFaceCond = (x : Node A) → ∥ Star G (α h x) ∥ → ∥ Star A x ∥
```

- and preservation of corners by the combinatorial map:

```
faceCornersPreserved : Type ℓ
faceCornersPreserved
  = (x : Node A) → (e₀ : Edge A (pred-G A AG x) x)
  → (e₁ : Edge A x (suc-G A AG x))
  → Path {A = Star G (α h x)}
      (CyclicSet.φ (𝕞 (α h x))
      (α h (pred-G A AG x) , flip (β h _ _ e₀)))
      ((α h (suc-G A AG x) , β h _ _ e₁))
```

▷ A planar map can then be defined as a spherical graph map with a face:

```
Planar : Graph ℓ → Type (lsuc ℓ)
Planar G = isConnectedGraph (U G) × (Σ[ 𝕞 ] (isSphericalMap G 𝕞 × Face G 𝕞))
```

Where spherical graph maps are defined as follows:

– Spherical maps as introduced in Chapter 5:

```
isSphericalMap : Map G → Type (lsuc ℓ)
isSphericalMap 𝓜
  = (x y : Node (U G))
  → (w₁ w₂ : Walk (U G) x y)
  → w₁ is-quasi-simple
  → w₂ is-quasi-simple
  → ∥ w₁ ~⟨ 𝓜 ⟩~ w₂ ∥
```

– Quasi-simple walks as introduced in Definition 5.12:

```
isQuasi : ∀ {x z : Node G} → Walk G x z → Type ℓ
isQuasi w = Π[ y ∶ Node G ] (isProp (y ∈w w))
```

– The special membership relation used above as introduced in Definition 5.10:

```
_∈w_ : Node G → Walk G x z → Type ℓ
y ∈w ⟨ z ⟩ = ⊥ ℓ
y ∈w (e ⊙ w) = (y ≡ source G e) + (y ∈w w)
```

Utilising the above definitions, we ascertain that the face type constitutes a homotopy set. This consequently implies that the planar map type is also a set.

```
{-# OPTIONS --without-K --exact-split #-}

module lib.graph-embeddings.Map.Face.isSet
  where
  open import foundations.Core
  open import foundations.HLevelLemmas
  open import foundations.NaturalsType
  open import foundations.Nat
  open import lib.graph-embeddings.Map
  open import lib.graph-embeddings.Map.Face
  open import lib.graph-definitions.Graph
  open Graph
  open import lib.graph-walks.Walk
  open import lib.graph-transformations.U
  open import lib.graph-homomorphisms.Hom
  open Hom
  open import lib.graph-classes.CyclicGraph
  open CyclicGraph
  open import lib.graph-classes.CyclicGraph.Stuff
  open import lib.graph-homomorphisms.classes.EdgeInjective

  module _ {ℓ : Level} (G : Graph ℓ) (𝓜 : Map G) where

    Face-is-set : isSet (Face G 𝓜)
    Face-is-set = equiv-preserves-sets (Face'≃Face G 𝓜) Face'-is-set
      where
      abstract
        Face'-is-set : isSet (Face' G 𝓜)
        Face'-is-set
            f1@(d1@(A , AG@(cyclic-graph φA zero pA) , f) , (p₁ , p₂) , p₃)
            f2@(d2@(B , BG@(cyclic-graph φB zero pB) , g) , (q₁ , q₂) , q₃)
            = trunc-elim prop-is-prop
              (λ {idp → trunc-elim prop-is-prop
                (λ {idp → isProp-≃
                    (≃-sym (≃-trans (Face'-Path-space G 𝓜 f1 f2) equiv))
```

```
            (Σ-prop
              (Σ-prop
                (equiv-preserves-prop
                (≃-sym
                  (≃-trans equiv-principle only-one-iso)) 𝟙-is-prop)
                       (λ {_ → Hom-is-set _ _ _ _}))
                       (λ _ → Hom-is-set A B _ _))
                       }) pB}) pA
  where
  open import lib.graph-classes.UnitGraph
  open import lib.graph-definitions.Graph.EquivalencePrinciple
  open EquivPrinciple (𝟙-graph ℓ) (𝟙-graph ℓ)

  equiv
    : (d1 ≡ d2)
      ≃ (Σ[ p ∶ Σ[ α ∶ A ≡ B ] (tr _ α f ≡ g) ] ((tr _ (π₁ p) φA) ≡ φB))
  equiv =
    begin≃
      _
        ≃⟨ qinv-≃ (λ {idp → idp})
                  ((λ {idp → idp}) , (λ {idp → idp}) , (λ {idp → idp})) ⟩
      Path {A = Σ[ _ ] Σ[ _ ] Σ[ _ ] _}
        (A , f , φA , pA)
        (B , g , φB , pB)
        ≃⟨ qinv-≃ (λ {idp → idp})
                  ((λ {idp → idp}) , (λ {idp → idp}) , (λ {idp → idp})) ⟩
      Path {A = Σ[ (X , h , φh) ∶ ((Σ[ X ] (Σ[ _ ] Hom X _))) ] _}
        ((A , f , φA) , pA)
        ((B , g , φB) , pB)
        ≃⟨ simplify-pair (λ {(_ , _ , _) → trunc-is-prop}) ⟩
      Path {A = Σ[ _ ] Σ[ _ ] _}
        (A , f , φA)
        (B , g , φB)
        ≃⟨ qinv-≃ (λ {idp → (idp , idp) , idp})
                  (( λ { ((idp , idp) , idp) → idp})
                       , (λ {((idp , idp) , idp) → idp}) , λ {idp → idp}) ⟩
        _ ≃∎
Face'-is-set ((A , AG@(cyclic-graph φA zero pA) , f) , (p₁ , p₂) , p₃)
              ((B , BG@(cyclic-graph φB (succ nB) pB) , g) , (q₁ , q₂) , q₃)
    = trunc-elim prop-is-prop (λ {idp → trunc-elim prop-is-prop
      (λ {idp → λ {()}})
      pB}) pA

Face'-is-set ((A , AG@(cyclic-graph φA (succ nA) pA) , f) , (p₁ , p₂) , p₃)
              ((B , BG@(cyclic-graph φB zero pB) , g) , (q₁ , q₂) , q₃)
    = trunc-elim prop-is-prop (λ {idp → trunc-elim prop-is-prop
      (λ {idp → λ {()}})
      pB}) pA

Face'-is-set
    f1@(d1@(A , AG@(cyclic-graph φA (succ nA) pA) , f) , (p₁ , p₂) , p₃)
    f2@(d2@(B , BG@(cyclic-graph φB (succ nB) pB) , g) , (q₁ , q₂) , q₃)
    = trunc-elim prop-is-prop
      (λ {idp → trunc-elim prop-is-prop
        (λ {idp →
          isProp-≃
            (≃-sym
              (≃-trans (Face'-Path-space G 𝕄 f1 f2) equiv))
            (Σ-prop (ℕ-is-set _ _) (λ {idp → Σ-prop
```

```
                      (Σ-Cn-isos-is-prop ℓ
                        (succ nA) (succ-n>0 ℓ {nA}) (U G) f p₁ g q₁)
                      (λ _ → Hom-is-set A B _ _)}))})
                pB}) pA
            where
            open import lib.graph-homomorphisms.classes.EdgeInjective.Lemmas
            open import lib.graph-homomorphisms.Lemmas
            open Hom-Lemma-1 A B
            open Hom-Lemma-2 A B (U G) f g
            equiv : (d1 ≡ d2) ≃ (Σ[ α : nA ≡ nB ]
                                  Σ[ p : Σ[ α : A ≡ B ] (f ≡ g∘ α) ]
                                  ((tr _ (π₁ p) φA) ≡ φB))
            equiv =
              begin≃
                  _
                  ≃⟨ qinv-≃ (λ {idp → idp})
                            ((λ {idp → idp}) , (λ {idp → idp}) , (λ {idp → idp})) ⟩
                  Path {A = Σ[ _ ] Σ[ _ ] Σ[ _ ] Σ[ _ ] _}
                  (nA , A , f , φA , pA)
                  (nB , B , g , φB , pB)
                  ≃⟨ qinv-≃ (λ {idp → idp})
                            ((λ {idp → idp}) , (λ {idp → idp}) , (λ {idp → idp})) ⟩
                Path {A = Σ[ (nX , X , h , φh) : (Σ[ _ ] (Σ[ X ] (Σ[ _ ] Hom X _))) ] _}
                  ((nA , A , f , φA) , pA)
                  ((nB , B , g , φB) , pB)
                  ≃⟨ simplify-pair (λ {(_ , _ , _ , _) → trunc-is-prop}) ⟩
              Path {A = Σ[ _ ] Σ[ _ ] Σ[ _ ] _}
                  (nA , A , f , φA)
                  (nB , B , g , φB)
              ≃⟨ qinv-≃ (λ {idp → idp , idp}) ((λ {(idp , idp) → idp}) ,
                          (λ {(idp , idp) → idp}) , λ {idp → idp}) ⟩
                ((nA ≡ nB) × Path ((A , f , φA)) ((B , g , φB)))
              ≃⟨ qinv-≃ (λ {(idp , idp) → idp , (idp , idp)})
                        ((λ {(idp , (idp , idp)) → (idp , idp)}) ,
                        (λ {(idp , (idp , idp)) → idp}) , λ {(idp , idp) → idp}) ⟩
                ((nA ≡ nB) × (Σ[ α : A ≡ B ] (Path (f , tr _ α φA) (g∘ α , φB))))
                    ≃⟨ qinv-≃ (λ {(idp , (idp , idp)) → idp , (idp , idp) , idp})
                          ((λ {(idp , ((idp , idp) , idp)) → idp , (idp , idp )}) ,
                            (λ {(idp , ((idp , idp) , idp)) → idp})
                            , λ {(idp , (idp , idp)) → idp}) ⟩
              ((nA ≡ nB) × (Σ[ p : Σ[ α : A ≡ B ] (f ≡ g∘ α) ] ((tr _ (π₁ p) φA) ≡ φB)))
              ≃∎
Planar-is-set : {G : Graph ℓ} → isSet (Planar G)
Planar-is-set {G} =
  ×-is-set
    (prop-is-set being-connected-is-prop)
    (Σ-set
      (Map-is-set G)
      (λ 𝓂 → ×-is-set (prop-is-set (isSphericalMap-is-prop G 𝓂))
    (Face-is-set G 𝓂)))
  where
  open import lib.graph-embeddings.Map.Face.isSet
```

(1)

Example A.1. Let us elaborate on a basic example, the planar map of the one-point graph. This instance sheds light on the stringent process necessary to affirm a graph is planar within the formal setting of this thesis. Herein, we construct an Agda term step-by-step. First, affirming that the trivial graph map for the one-point graph is spherical and that it has an outer face, thus fulfilling the required data.

```
{-# OPTIONS --without-K --exact-split #-}

module lib.graph-embeddings.Map.Face.Example
  where
  open import foundations.Core
  open import lib.graph-definitions.Graph
  open import lib.graph-transformations.U
  open Graph

  open import lib.graph-embeddings.Map
  open import lib.graph-classes.UnitGraph

  open import foundations.Cyclic using (CyclicSet; cyclic-set)
  open import foundations.Fin
  open import foundations.Nat

  open import lib.graph-embeddings.Map
  open import lib.graph-definitions.Graph
  open import lib.graph-homomorphisms.Hom
  open import lib.graph-homomorphisms.classes.EdgeInjective
  open import lib.graph-transformations.U
  open import lib.graph-classes.CyclicGraph
  open import lib.graph-classes.CyclicGraph.Stuff
  open import lib.graph-embeddings.Map.Face

  open import lib.graph-embeddings.Planar
  open import lib.graph-classes.EmptyGraph

  open import foundations.Fin
  open import lib.graph-classes.UnitGraph
  open import lib.graph-classes.EmptyGraph
  open import foundations.Cyclic
  open import foundations.UnivalenceAxiom
  open CyclicGraph-is-set
  open import foundations.FunExtAxiom

  ℓ : Level
  ℓ = lzero

  star-𝟙 : Star (𝟙-graph ℓ) ⋆ ≃ Fin ℓ 0
  star-𝟙
    = qinv-≃
        (λ { (_ , inl ()) ; (_ , inr ())})
        ((λ { (zero , ()) ; (succ _ , ())}) ,
        (λ { (zero , ()) ; (succ _ , ())}) , λ { (_ , inl ()) ; (_ , inr ())})

  zero-is-only-once-cyclic : isProp (CyclicSet ℓ (Fin ℓ 0))
  zero-is-only-once-cyclic p q = rapply (lemma-2-13 ℓ {A = Fin ℓ 0} p q)
      (pi-is-prop (λ _ → isProp-≃ (𝟘-≃-⟦0⟧ ℓ) 𝟘-is-prop) _ _)

  -- There is only one map.
  -- Let's prove that by showing that the corresponding type is contractible.
  -- Or equivalently, that it is a proposition and that the type is inhabited.
```

```
𝟙-map : Map (𝟙-graph ℓ )
𝟙-map ⋆ = cyclic-set id 0 | star-𝟙 , funext (λ { (fst₁ , inl ()) ; (fst₁ , inr ())}) |

𝟙-has-prop-map : isProp (Map (𝟙-graph ℓ))
𝟙-has-prop-map = (pi-is-prop (λ {* → tr (λ o → isProp (CyclicSet ℓ o)) (! ua star-𝟙)
                   zero-is-only-once-cyclic}))

-- Similarly, we prove that the type Hom(𝟙-graph,U(𝟙-graph)) is contractible.
𝟙-hom : Hom (𝟙-graph ℓ) (U (𝟙-graph ℓ))
𝟙-hom = hom id (λ _ _ abs → inl abs)

𝟙-hom-prop' : isProp (Hom (𝟙-graph ℓ) (𝟙-graph ℓ))
𝟙-hom-prop' = isProp-≃ (≃-sym (Hom-≃-Σ (𝟙-graph ℓ) (𝟙-graph ℓ)))
  (Σ-prop (pi-is-prop (λ _ → 𝟙-is-prop))
    (λ {_ → pi-is-prop
    (λ _ → pi-is-prop (λ _ → pi-is-prop
    (λ _ → 𝟘-is-prop )))}))

𝟙-hom-prop : isProp (Hom (𝟙-graph ℓ) (U (𝟙-graph ℓ)))
𝟙-hom-prop = isProp-≃ (≃-sym (Hom-≃-Σ (𝟙-graph ℓ) (U (𝟙-graph ℓ))))
    (Σ-prop (pi-is-prop (λ _ → 𝟙-is-prop))
      (λ {_ → pi-is-prop
      (λ _ → pi-is-prop (λ _ → pi-is-prop
      (λ _ → +-prop 𝟘-is-prop 𝟘-is-prop (λ {()})))))}))

-- CyclicGraph ℓ (𝟙-graph ℓ) is contractible.
𝟙-graph-is-cyclic : CyclicGraph ℓ (𝟙-graph ℓ)
𝟙-graph-is-cyclic = cyclic-graph (id-hom (𝟙-graph ℓ)) 0 | idp |

𝟙-graph-cyclic-prop : isProp (CyclicGraph ℓ (𝟙-graph ℓ))
𝟙-graph-cyclic-prop = isProp-≃ CyclicGraph-≃-Σs
    (Σ-prop 𝟙-hom-prop' λ {_ (n , c) (m , d)
      → pair= (instances-have-same-natural ℓ (𝟙-graph ℓ)
          (cyclic-graph _ n c)
          (cyclic-graph _ m d)
            , trunc-is-prop _ d) })

𝟙-face : Face (𝟙-graph ℓ) 𝟙-map
𝟙-face = face (𝟙-graph ℓ) 𝟙-graph-is-cyclic 𝟙-hom
              (λ {()}) (λ _ → id) (λ {_ _ ()})

helper : ∀ {A : Graph ℓ} → isProp (Hom A (U (𝟙-graph ℓ)))
helper {A} =
  isProp-≃ (≃-sym (Hom-≃-Σ A _))
    (Σ-prop (pi-is-prop λ _ → 𝟙-is-prop)
      λ _ → pi-is-prop (λ _ → pi-is-prop
      λ _ → pi-is-prop (λ _ → +-prop 𝟘-is-prop 𝟘-is-prop (λ {()}))))

open import lib.graph-relations.Isomorphic

U𝟙-≅-𝟙 : U (𝟙-graph ℓ) ≅ 𝟙-graph ℓ
U𝟙-≅-𝟙 = idEqv ,
  λ _ _ → prop-ext-≃
          (+-prop 𝟘-is-prop 𝟘-is-prop (λ {()}))
          𝟘-is-prop
  ((λ { (inl ()) ; (inr ())}) , λ {()})

open import lib.graph-embeddings.Map.Face.Walk.Homotopy
open HomotopyWalks
open import lib.graph-embeddings.Map.Spherical
open import lib.graph-walks.Walk
```

```
    M-is-spherical : isSphericalMap (𝟙-graph lzero) 𝟙-map
    M-is-spherical ⋆ .⋆ ⟨ .⋆ ⟩ ⟨ .⋆ ⟩ _ _ = ∣ hwalk-refl ∣
    M-is-spherical x .x ⟨ .x ⟩ (inl e ⊙ w) = 𝟘-elim e
    M-is-spherical x .x ⟨ .x ⟩ (inr e ⊙ w) = 𝟘-elim e
    M-is-spherical x y (inl e ⊙ w1) w2 = 𝟘-elim e
    M-is-spherical x y (inr e ⊙ w1) w2 = 𝟘-elim e

    open import lib.graph-classes.ConnectedGraph
    𝟙-graph-is-connected : (U (𝟙-graph lzero)) is-connected-graph
    𝟙-graph-is-connected ⋆ ⋆ = ∣ ⟨ ⋆ ⟩ ∣

    𝟙-graph-is-planar : Planar (𝟙-graph lzero)
    𝟙-graph-is-planar =
      𝟙-graph-is-connected ,
      𝟙-map ,
      M-is-spherical ,
      𝟙-face
```

Example A.2. The following is the elaboration of the proof of Lemma 3.20.

```
{-# OPTIONS --without-K --exact-split #-}
module lib.graph-families.CycleGraph.Isomorphisms.Lemmas
  where
  open import foundations.Core
  module _ (ℓ : Level) where
    open import lib.graph-definitions.Graph
    open import lib.graph-homomorphisms.Hom
    open import lib.graph-relations.Isomorphic
    open import lib.graph-families.CycleGraph.Isomorphisms.IdentityType
    open import foundations.FunExtAxiom using (happly)
    open import foundations.UnivalenceAxiom using (idtoeqv)
    open import foundations.Fin ℓ
    open import foundations.Nat ℓ
    open import foundations.Cyclic ℓ
    open import lib.graph-families.CycleGraph.RotHom ℓ
    open import lib.graph-families.CycleGraph ℓ

    open Graph
    open Hom

    order-of-an-iso
        : (n : ℕ)
        → (n>0 : ℕ-ordering._>_ ℓ n 0)
        → (φ : Cycle n ≅ Cycle n)
        → ∑[ (k , _) ∶ ⟦ n ⟧ ] ((rot n ^-hom k) ≡ hom-from-iso φ)

    order-of-an-iso zero ()
    order-of-an-iso n@(succ _) ⋆ φ = ((Isos-→-Fin ℓ n ⋆) φ) ,
        (begin
          (rot n ^-hom π₁ (((Isos-→-Fin ℓ n ⋆) φ)))
            ≡⟨⟩
          hom-from-iso ((Fin-→-Isos ℓ n ⋆ ) (((Isos-→-Fin ℓ n ⋆) φ)))
            ≡⟨ ap hom-from-iso (rlmap-inverse-h (Isos-≃-Fin ℓ n ⋆ ) φ) ⟩
          hom-from-iso φ ∎)

    open import lib.graph-homomorphisms.Lemmas
    module _ (n : ℕ) (n>0 : n > 0)
        where
        hom-from-≡ = Hom-Lemma-1.hom-from-≡ (Cycle n) (Cycle n)
        ≡-from-iso = Hom-Lemma-1.≡-from-iso (Cycle n) (Cycle n)
        same-hom-from-≡-or-≅ = Hom-Lemma-1.same-hom-from-≡-or-≅ (Cycle n) (Cycle n)
```

```
rot^k-from-iso
  : (n : ℕ) (n>0 : n > 0)
  → (φ : Cycle n ≅ Cycle n)
  → (G : Graph ℓ)
  → (f g : Hom (Cycle n) G)
  → let k = π₁ (((Isos-≃-Fin ℓ n n>0 ∙) φ))
    in
      (f ≡ ( ((hom-from-≡ n n>0) (≡-from-iso n n>0 φ)) ∘Hom g))
    ≡ (f ≡ ((rot n ^-hom k) ∘Hom g))

rot^k-from-iso zero () φ G f g
rot^k-from-iso n@(succ _) ⋆ φ G f g
    = ap (λ w → f ≡ (w ∘Hom g))
        ((same-hom-from-≡-or-≅ n ⋆ φ) · ! π₂ (order-of-an-iso n ⋆ φ))

m₁
  : (n : ℕ) (n>0 : n > 0) (G : Graph ℓ)(f g : Hom (Cycle n) G)
  → (Σ[ α : Cycle n ≅ Cycle n ]
      (f ≡ (((hom-from-≡ n n>0) ((≡-from-iso n n>0) α)) ∘Hom g)))
      ≃ (Σ[ (k , _) : ⟦ n ⟧ ] (f ≡ ((rot n ^-hom k) ∘Hom g)))

m₁ zero ()
m₁ n@(succ _) ⋆ G f g
    = sigma-maps-≃ (Isos-≃-Fin ℓ n ⋆ ) $
        λ φ → idtoeqv (rot^k-from-iso n ⋆ φ G f g)

abstract
    L1-hom
      : (n : ℕ) (n>0 : n > 0) { k₁ k₂ : ⟦ n ⟧}
      → ((x y : Node (Cycle n))
      → (e : Edge (Cycle n) x y)
      → let
        h₁ = rot n ^-hom (π₁ k₁)
        h₂ = rot n ^-hom (π₁ k₂)
        in
        Path {A = Σ[ x ] Σ[ y ] Edge (Cycle n) x y}
          (α h₁ x , α h₁ y , β h₁ x y e)
          (α h₂ x , α h₂ y , β h₂ x y e))
      → k₁ ≡ k₂

    L1-hom zero ()
    L1-hom n@(succ m) ⋆ {k₁ = k₁}{k₂} f
      = fin-exp-is-unique k₁ k₂ (fin-pred (0' ℓ m)) eq₁
      where
      open Sigma

      h₁ h₂ : Hom (Cycle n) (Cycle n)
      h₁ = rot n ^-hom (π₁ k₁)
      h₂ = rot n ^-hom (π₁ k₂)

      eq₁ : (fin-pred {k = n} ^ π₁ k₁) (m , <-succ m)
          ≡ (fin-pred ^ π₁ k₂) (m , <-succ m)
      eq₁ =
        (fin-pred ^ (π₁ k₁)) (m , <-succ m)
          ≡⟨ happly (lemma-on-nodes-hom-expo (rot n) (π₁ k₁)) (m , <-succ m) ⟩
        (α h₁ (m , <-succ m) )
          ≡⟨ π₁ (Σ-componentwise (f (m , <-succ m) (0' ℓ _) idp)) ⟩
        (α h₂ (m , <-succ m))
          ≡⟨ ! happly
               (lemma-on-nodes-hom-expo (rot n) (π₁ k₂))
```

```
          ((fin-pred (0' ℓ _))) ⟩
      (fin-pred ^ π₁ k₂) _
      ∎
```

Example A.3. We present a few excerpts on the definition of the path addition of a graph and a few related lemmas as discussed in Section 6.3.1.

▹ The path addition of P_n to a graph G, where $n > 0$.

```
path-addition : (a b : Node G) → (n : ℕ) → 0 < n → Graph ℓ
path-addition a b n p = graph N' E' N'-is-set E'-forms-sets
  where
  N' : Type ℓ
  N' = Node G + Fin n
  E' : N' → N' → Type ℓ
  E' (inl x) (inl y) = Edge G x y
  E' (inl x) (inr y) = (x ≡ a) × (y ≡ (0 , p))
  E' (inr x) (inl y) = (x ≡ fin-pred (0 , p)) × (y ≡ b)
  E' (inr x) (inr y) = x ≡ fin-pred y

  N'-is-set : N' is-set
  N'-is-set = +-set (Node-is-set G) Fin-is-set

  E'-forms-sets : (x y : N') → (E' x y) is-set
  E'-forms-sets (inl x) (inl y) = Edge-is-set G _ _
  E'-forms-sets (inl x) (inr y) =
    Σ-set (prop-is-set (Node-is-set G _ _))
          (λ _ → prop-is-set (Fin-is-set _ _))
  E'-forms-sets (inr x) (inl y) =
    Σ-set (prop-is-set (Fin-is-set _ _))
          (λ _ → prop-is-set (Node-is-set G _ _))
  E'-forms-sets (inr x) (inr y) = prop-is-set (Fin-is-set _ _)
```

▹ The graph resulting from the path addition contains the walks of the original graph.

```
path-addition-has-original-walks
  : (a b : Node G) → (n : ℕ) → (p : 0 < n)
  → (x y : Node G) → (Walk G x y)
  → Walk (path-addition a b n p) (inl x) (inl y)
path-addition-has-original-walks a b n p x .x ⟨ .x ⟩ = ⟨ inl x ⟩
path-addition-has-original-walks a b n p x y (_⊙_ {y = y₁} e w) = e ⊙ w'
  where
  w' : Walk (path-addition a b n p) (inl y₁) (inl y)
  w' = path-addition-has-original-walks a b n p _ _ w
```

▹ In addition, the graph resulting from the path addition contains the inner walks in the path graph P_n.

```
path-addition-has-new-walks
  : (a b : Node G) → (n : ℕ) → (p : 0 < n)
  → (x y : ℕ) → (x< : x < n) → (y< : y < n)
  → ((x ≡ y) + (x < y))
  → Walk (path-addition a b n p) (inr (x , x<)) (inr (y , y<))
path-addition-has-new-walks a b zero ()
path-addition-has-new-walks a b n@(succ n') ⋆ zero zero x< y< (inl idp)
  rewrite <-prop {n = n}{m = zero} x< y< = ⟨ inr (0 , y<) ⟩
path-addition-has-new-walks a b n@(succ n') ⋆ zero (succ y) x< y< (inr ⋆)
  with _≟fin_ {n} (0 , x<) (fin-pred (succ y , y<))
... | yes p = p ⊙ ⟨ _ ⟩
... | no ¬p   = walk-0-to-y ·w (pair= (idp ,
```

```
      ⊲-prop {succ n'}{y} (⊲-inj {n}{y} y<) (n⁺<k→n<k {y}{succ n'} y<)) ⊙ ⟨ _ ⟩)
    where
    open •-walk (path-addition a b n ⋆)
    walk-0-to-y
      : Walk (path-addition a b n ⋆) (inr (0 , x<)) (inr (y , ⊲-inj {n}{y} y<))
    walk-0-to-y with zero ≟nat y
    ... | yes idp
      rewrite ⊲-prop {n = n}{m = zero} (⊲-inj {n}{y} y<) ⋆ = ⟨ inr ((0 , x<)) ⟩
    ... | no ¬p = path-addition-has-new-walks a b (succ n') ⋆ 0 y ⋆
          ((⊲-inj {n}{y} y<)) (inr (n≠0 (λ p → ¬p (sym p))))
path-addition-has-new-walks a b n@(succ n') ⋆ (succ x) (succ .x) x< y< (inl idp)
  rewrite ⊲-prop {n = n}{m = succ x} x< y< = ⟨ inr (succ x , y<) ⟩
path-addition-has-new-walks a b n@(succ n') ⋆ (succ x) (succ y) x< y< (inr x<y)
  with _≟fin_ {n} (succ x , x<) (fin-pred (succ y , y<))
... | yes p = p ⊙ ⟨ _ ⟩
... | no ¬p = walk-0-to-y •w
      (pair= (idp , ⊲-prop {succ n'}{y} (⊲-inj {n}{y} y<)
             (n⁺<k→n<k {y}{succ n'} y<)) ⊙ ⟨ _ ⟩)
  where
  open •-walk (path-addition a b n ⋆)
  walk-0-to-y
    : Walk (path-addition a b n ⋆) (inr (succ x , x<)) (inr (y , ⊲-inj {n}{y} y<))
  walk-0-to-y with (succ x) ≟nat y
  ... | yes idp
      rewrite ⊲-prop {n = n}{m = succ x} (⊲-inj {n}{y} y<) x< = ⟨ inr (succ x , x<) ⟩
  ... | no p =
    path-addition-has-new-walks a b (succ n') ⋆ (succ x) y
      (mono-succ {x}{n'} x<) (⊲-inj {n}{y} y<)
      (inr (⊲-suc-suc< {ℓ} {x}{y} x<y λ o → ¬p
           (pair= (sym o , ⊲-prop {succ n'}{y} _ _))))
```

▷ As expected, one can prove that the graph resulting from the path addition is connected if the original graph is connected. To prove this lemma, we need a few extra lemmas, which are proved below.

```
path-addition-preserves-connectedness
  : G is-connected-graph
  → (a b : Node G) → (n : ℕ) → (p : 0 < n)
  → (path-addition a b n p) is-connected-graph
path-addition-preserves-connectedness G-is-connected a b zero ()
path-addition-preserves-connectedness G-is-connected a b n@(succ n') p
  = helper
  where
  G' : Graph ℓ
  G' = path-addition a b n p
  N' = Node G + Fin n
  open •-walk (path-addition a b n p)

  helper : (x y : N') → ‖ Walk G' x y ‖
  helper (inl x) (inl y)
    = trunc-elim trunc-is-prop
          (λ w → | path-addition-has-original-walks a b n p _ _ w |)
          (G-is-connected x y)
  helper (inl x) (inr y@(naty , y<))
    with x ≟Node a
  ... | inl idp = | path-addition-walk-from-first-endpoint a b n p y |
  ... | inr p' = trunc-elim trunc-is-prop
      (λ w → | path-addition-has-original-walks a b n p x a w •w walk-a-finy |)
      (G-is-connected x a)
      where
      walk-a-finy = path-addition-walk-from-first-endpoint a b n p y
  helper (inr x@(natx , x<)) (inl y)
    with (x ≟fin fin-pred (0 , p)) | (y ≟Node b)
```

```
    ... | yes idp | inl idp = | (((idp , idp)) ⊙ ⟨ inl b ⟩) |
    ... | yes idp | inr y≠b = trunc-elim trunc-is-prop
      (λ w → | ((((idp , idp)) ⊙ ⟨ inl b ⟩))
        ·w path-addition-has-original-walks a b n p _ _ w |)
      (G-is-connected b y)
    ... | no ¬p | inl idp = | walk-fin0-finn-1 ·w ((idp , idp) ⊙ ⟨ inl b ⟩) |
        where
        walk-fin0-finn-1
            : Walk (path-addition a b n p) (inr x) (inr (fin-pred (0 , p)))
        walk-fin0-finn-1 = path-addition-walk-to-the-end a b n p x
    ... | no ¬p | inr y≠b
        with (x ≟fin fin-pred (0 , p))
    ... | yes idp = trunc-elim trunc-is-prop (λ w →
            | ((((idp , idp)) ⊙ ⟨ inl b ⟩))
               ·w path-addition-has-original-walks a b n p _ _ w |)
            (G-is-connected b y)
    ... | no ¬p₁ = trunc-elim trunc-is-prop (λ w →
            | walk-finx-b ·w (((idp , idp))
               ⊙ path-addition-has-original-walks a b n p _ _ w) |)
            (G-is-connected b y)
        where
        walk-finx-b : Walk (path-addition a b n p) (inr x) (inr (fin-pred (0 , p)))
        walk-finx-b = path-addition-walk-to-the-end a b n p x
    helper (inr x@(natx , x<)) (inr y@(naty , y<))
      with trichotomy natx naty
    ... | inl (inl idp) =
      | path-addition-has-new-walks a b n p natx naty x< y< (inl idp) |
    ... | inl (inr natx<naty) =
      | path-addition-has-new-walks a b n p natx naty x< y< (inr natx<naty) |
    ... | inr naty<natx =
          trunc-elim trunc-is-prop (λ w
            → | path-addition-walk-to-last-endpoint a b n p x
              ·w path-addition-has-original-walks a b n p _ _ w
              ·w path-addition-walk-from-first-endpoint a b n p y
              |)
            (G-is-connected b a)
```

▷ Walks from the head of a path addition.

```
  path-addition-walk-from-the-head
    : (a b : Node G) → (n : ℕ) → (p : 0 < n)
    → (x : Fin n)
    → Walk (path-addition a b n p) (inr (0 , p)) (inr x)
  path-addition-walk-from-the-head a b zero ()
  path-addition-walk-from-the-head a b n@(succ n') p x@(natx , x<)
    with natx ≟nat 0
  ... | yes idp = ⟨ inr x ⟩
  ... | no ¬p = path-addition-has-new-walks a b n p 0 natx p x< (inr (n≠0 {natx} ¬p))
```

▷ Walks to the tail of a path addition.

```
  path-addition-walk-to-the-end
    : (a b : Node G) → (n : ℕ) → (p : 0 < n)
    → (x : Fin n)
    → Walk (path-addition a b n p) (inr x) (inr (fin-pred (0 , p)))
  path-addition-walk-to-the-end a b zero ()
  path-addition-walk-to-the-end a b n@(succ n') p x@(natx , x<)
    with <s-to-<= natx n' x<
  ... | (inl idp) rewrite <-prop {succ n'}{n'} x< (<-succ n') = ⟨ inr (natx , _) ⟩
  ... | (inr natx<n') =
    path-addition-has-new-walks a b n p natx n' x< (<-succ n') (inr natx<n')
```

▷ Walks from the first endpoint of a path addition.

```
path-addition-walk-from-first-endpoint
  : (a b : Node G) → (n : ℕ) → (p : 0 < n)
  → (x : Fin n)
  → Walk (path-addition a b n p) (inl a) (inr x)
path-addition-walk-from-first-endpoint a b n p x =
  (idp , idp) ⊙ (path-addition-walk-from-the-head a b n p x)
```

▷ Walks to the last endpoint of a path addition.

```
path-addition-walk-to-last-endpoint
  : (a b : Node G) → (n : ℕ) → (p : 0 < n)
  → (x : Fin n)
  → Walk (path-addition a b n p) (inr x) (inl b)
path-addition-walk-to-last-endpoint a b n p x
  = path-addition-walk-to-the-end a b n p x ·w (((idp , idp)) ⊙ ⟨ inl b ⟩)
  where
  open ·-walk (path-addition a b n p)
```

B
On Trees and Their Topological Realisation

B.1 Introduction

This appendix explores the concepts of rooted trees and oriented spanning trees in HoTT. A rooted tree is a tree with one node singled out, while an oriented spanning tree is defined as a subgraph that not only forms a tree but also encompasses all nodes of the original graph (Diestel and Kühn 2004). Although there is no universal method for constructing spanning trees in infinite connected graphs, we can establish the mere existence of oriented spanning trees for finite, connected graphs, provided that the set nodes are finite and the edge family comprises homotopy sets. Some notions, including the type of a graph, differ slightly from the one used in the main text in this part. For example, we define a graph as a type of nodes equipped with a binary type-valued relation of edges.

Furthermore, we explore these notions of connected graphs rooted trees in the context of higher-inductive types (HITs) in HoTT. Towards this end, we define the 1-cell topological realisation of a graph, a construction also seen in Swan's proof on the proof of Nielsen-Schreier theorem (Swan 2022). This construction, which considers only the 0-cells and 1-cells ––nodes and edges—excluding higher-dimensional cells, can be expressed in HoTT as a coequalizer that models the topological space generated by a graph where the nodes are points, and the edges are paths glued to their endpoints in the space. In this context of HITs, we define a graph as connected if its realisation is a connected type. We also define a tree as a loop-free graph whose topological realisation is a con-

tractible type. Then, later, we reconnect with the notion of rooted trees, which are initially defined independently from topological realisation as trees with a designated node, and prove that realisation of rooted trees corresponds to contractible types, i.e. also trees in the sense of HITs.

The connection and constructions presented in this appendix , although novel at the moment of writing, as we did not find any similar work in the literature, were influenced by Swan's constructions, especially the sections on enlarging subtrees and spanning trees (see Appendices B.4 and B.4.1), which draw upon Lemma 4.6 (Swan 2022, §4). The primary distinction between our approach and Swan's lies in our focus on the combinatorial constructions of graphs. We construct (spanning) trees through the expansion of graphs, as similarly outlined in the main text, rather than directly employing higher inductive types on the topological realisations of graphs.

B.2 Computer formalisation in Cubical Agda

We use Cubical Agda to formalise this part, an extension of Agda that supports Cubical type theory and a large family of HITs, making it easy to prove principles like function and propositional extensionality, and define higher inductive types such as the circle and the torus. Cubical Agda lets us define (dependent) functions on HITs by pattern matching. Although we could use vanilla Agda, it is rather tedious without the Cubical Mode. In recent versions of Agda, we can extend the type-checker's capabilities by adding custom rewriting rules, which can alleviate the lack of support for HITs. To type-check the proofs in this appendix, we use the following versions of Agda and the Cubical Agda library.

- ▷ Agda version `2.6.2.1-59c7944` with the flag `--cubical`.
- ▷ Cubical Agda library version 0.3.

```
{-# OPTIONS --cubical #-}
open import Cubical.Core.Everything
open import Base
```

B.3 Basic concepts

Let us start defining a few basic concepts about graphs.

B.3.1 The type of graphs

Definition B.1. A graph consists of a type of nodes equipped with a binary type valued rela-

tion of edges.

$$\text{Graph} :\equiv \sum_{(N:\mathcal{U})} (N \to N \to \mathcal{U}). \tag{B.3–1}$$

The type of graphs is defined in Agda as follows.

```
record Graph : Type (ℓ-suc ℓ) where
  constructor graph
  field
    N : Type ℓ
    E : N → N → Type ℓ
open Graph
```

B.3.2 The type of walks

We can define the type of walks in a graph as an indexed inductive data type, similar to the polymorphic type for lists. This type is useful for formalising results on walks, as it allows us to define walk functions through pattern matching in an easy and convenient way (Prieto-Cubides 2022). Unfortunately, pattern matching is not supported in Cubical Agda for such inductive data types at the moment of writing. We, therefore, consider the following equivalent types from where the former type is chosen for the convenience of the lemmas stated in this chapter. In particular, walks here grow by attaching edges at their ends, as in Lemma B.11. In what follows, we denote by $W^n_G(x, y)$ the type of walks from x to y of length n in a graph G.

1. Walks formed by backwards edge addition.

```
W : ℕ → N G → N G → Type ℓ
W 0 x y = x ≡ y
W (suc n) a c = Σ[ b ∈ N G ] (W n a b) × (E G b c)
```

2. Walks formed by forward edge addition.

```
W' : ℕ → N G → N G → Type ℓ
W' 0 x y = x ≡ y
W' (suc n) a c = Σ[ b ∈ N G ] (E G a b) × (W' n b c)
```

The concatenation of two walks p and q of n and m edges respectively, is a walk of $n+m$ edges denoted by $p \cdot q$ and defined in Agda as follows. To not clash with Cubical notation, we denote the new walk by `p ·w q`.

```
module walk-concat (G : Graph {ℓ}) where
  open Walks G
  _·w_ : ∀ {x y z : N G} {n m : ℕ}
    → W n x y → W m y z
```

```
      → W (n +N m) x z
    _·w_ {x = x} {z = z} {n} {zero} p q
      = subst ((λ o → W o x z)) (sym (+-zero n)) (subst (λ o → W n x o) q p)
    _·w_ {x = x} {z = z} {n} {suc m} p (b , q , e)
      = subst (λ o → W o x z) (sym (+-suc _ _)) (b , p ·w q , e)
```

As typical in HoTT, once a type is defined, one would like to characterise its identity type. In the case of walks, we compute the identity type point-wise. Since we are only interested in the case where graphs consist of sets, the type of walks of such graphs turns out to be a set.

Lemma B.2. Let G be a graph such that the type of nodes is a set and the family of edges consists of sets. Then, the type of walks of length n from x to y is a set, for any $x, y : N_G$ and $n : \mathbb{N}$.

A proof term for this lemma in Agda is the following.

```
module _ (V-is-set : isSet (N G))
         (E-is-set : (x y : N G) → isSet (E G x y)) where
  W-is-set : (n : ℕ) → (x y : N G) → isSet (W n x y)
  W-is-set zero _ _ = isProp→isSet (V-is-set _ _)
  W-is-set (suc n) _ _ = isOfHLevelΣ 2 V-is-set λ _ →
    (isOfHLevel× 2 (W-is-set n _ _ ) (E-is-set _ _))
```

We work with strongly connected graphs throughout the following lemmas unless otherwise stated. Let us define such a property as the mere existence of a walk between any pair of nodes.

Definition B.3 (isGConnected). A graph G is strongly connected if the type in (B.3–2) is inhabited.

$$\text{isGConnected}(G) :\equiv \prod_{(x,y:N_G)} \left\| \sum_{(n:\mathbb{N})} W_G^n(x,y) \right\|. \tag{B.3–2}$$

In Agda, the type above is defined as follows.

```
isGConnected : Graph → Type ℓ
isGConnected G = (x y : N G) → ∥ Σ[ n ∈ ℕ ] W G n x y ∥
```

Lemma B.4. Being connected for a graph is a proposition.

B.3.3 Rooted trees and subgraphs

Trees are usually defined as undirected graphs with a single path between any pair of nodes. However, we prefer to use a more suitable notion of a tree for working directly

with directed multigraphs. Therefore, we consider the class of rooted trees, which are directed graphs with a single node acting as the root of the tree and a single walk between any pair of nodes.

Let us now define the type of rooted trees in a directed multigraph G. We refer to rooted trees as trees in the rest of this work unless otherwise stated.

Definition B.5 (`isTree`). A graph G is a *tree* if the type in (B.3–3) is contractible. The node in the centre of contraction is referred to as the *root* of the tree.

$$\sum_{(r:\mathrm{N}_G)} \prod_{(x:\mathrm{N}_G)} \mathrm{isContr}\left(\sum_{(n:\mathbb{N})} \mathrm{W}^n_G(r,x)\right) \tag{B.3–3}$$

The notion of trees for directed graphs can also be defined in terms of zig-zags, which are walks formed by edges of different possible orientations. In this view, a tree is then a graph if the corresponding type of zig-zag walks is contractible. Finally, it is worth mentioning that the definition of the type of undirected graphs and other derived concepts, including trees and trails, can be found in Agda–UniMath (Rijke et al. 2023). In this Agda library, an undirected graph consists of a type V of nodes and a family E of types over the unordered pairs of V. Lastly, an unordered pair of elements in a type A consists of a two-element type X and a map of type $X \to A$.
In Agda, the type of rooted trees is defined as follows.

```
isTree : Graph → Type ℓ
isTree G = isContr(Σ[ r ∈ N G ] (∀ x → isContr(Σ[ n ∈ ℕ ] W G n r x)))
```

Lemma B.6 (`isProp-isTree`). Being a tree is a proposition.

We need now to define the notions of subgraph and subtree. Recall that we are interested in defining and constructing spanning trees out of strongly connected graphs, which are trees containing all nodes of the original graph. If the graph is finite and strongly connected, such trees can be obtained by traversing the graph using a depth-first search or a breadth-first search algorithm. For a more general class of graphs, a principle of choice may be needed to guide the search. In Appendix B.4.1, we prove that a spanning tree merely exists if the node set of the graph is an inhabited type and the graph is strongly connected with a family of discrete sets as the type of edges.

Definition B.7 (`Subgraph`). A subgraph of G is a graph H with an embedding into G, denoted by $H \hookrightarrow G$. The type of subgraphs of G is $\mathrm{Subgraph}(G)$.

$$\mathrm{Subgraph}(H, G) :\equiv \sum_{((h,g):\mathrm{Hom}(H,G))} \mathrm{isEmbedding}(h) \times \prod_{(x,y:\mathrm{N}_H)} \mathrm{isEmbedding}(g(x,y)),$$

where $\mathrm{Hom}(H, G)$ is the type of graph homomorphisms from H to G and isEmbedding is the property that the function `ap/cong` is an equivalence, as defined in the HoTT Book.

Almost faithfully, we define in Agda the above structure on graphs as follows.

```
module _ {ℓ : Level} (G : Graph {ℓ}) where
  record Subgraph (H : Graph {ℓ}) : Type ℓ where
    field
      h : N H → N G
      g : (x y : N H) → E H x y → E G (h x) (h y)
      h-is-emb : isEmbedding h
      g-is-emb : (x y : N H) → isEmbedding (g x y)
```

Definition B.8 (`isSubtree`). A (decidable) subtree of G is a tree and subgraph of G equipped with a mechanism for checking whether a node in G is in it or not.

```
record isSubtree (H : Graph {ℓ}) : Type ℓ where
  constructor subtree
  field
    is-subgraph : Subgraph H
    is-tree : isTree H
    dec-fiber : (x : N G) → Dec (fiber (Subgraph.h is-subgraph) x)
```

B.4 Enlarging rooted subtrees

In this section we develop a few lemmas about the notion of a subgraph and subtree about how to construct larger subtrees out of subgraphs. The main result of this section is Lemma B.11, which requires first to state the following lemma.

Lemma B.9 (`∃-edgecut`). Let G be a connected graph such that its node set is partitioned into two disjoint nonempty types V_1 and V_2. Then, it merely exists an edge connecting a node of V_1 to some node of V_2 and vice versa.

Proof. Since we want to prove a proposition, let us apply the elimination principle of the propositional truncation to the fact of G being connected. One can obtain a function f, which returns a walk connecting any two nodes of G. Let v_1, v_2 be nodes in V_1 and V_2, respectively, and w be the walk obtained by $f(v_1, v_2)$.

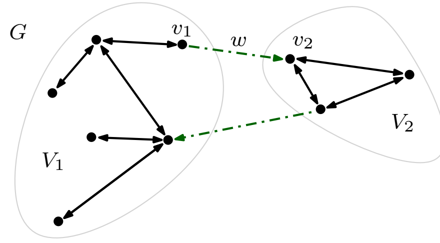


Figure B.1: The walk w in Lemma B.9's proof.

Let us proceed by induction on the length of w. We will exhibit an edge in the walk w that must have one node in V_1 and the other node in V_2, as illustrated in Figure B.1. If the walk has zero length, then there is nothing to prove since such a case is impossible by construction. Then, we can assume the induction hypothesis holds for a walk of length n. Let $p \cdot e$ be a walk of length $n + 1$ where p is a walk from x to y and e is an edge from y to v_2. Since the node set of G is equivalent to $V_1 + V_2$, the node y is either in V_1 or V_2. If y is in V_1, the required edge is e. Otherwise, we get the required edge by induction on the walk p. □

Figure B.2: The term ∃-edgecut defined below is the Agda term for the Lemma B.9's proof.

```
module EdgeCutLemma {ℓ : Level} {V₁ V₂ : Type ℓ}
  (G : Graph {ℓ}) (G-is-connected : isGConnected G)
  (e : N G ≃ V₁ + V₂)
  (v₁ : V₁) (v₂ : V₂) where
  ∃-edgecut : ∥ Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) ∥
  ∃-edgecut = trunc-elim isPropPropTrunc (λ {(n , w) → f v₁ v₂ n w}) w
    where
    isoN : Iso (N G) (V₁ + V₂)
    isoN = equivToIso e

    w : ∥ Σ[ n ∈ ℕ ] W G n (from-V₁ v₁) (from-V₂ v₂) ∥
    w = G-is-connected _ _

    f : (a : V₁) (b : V₂) (n : ℕ) → W G n (from-V₁ a) (from-V₂ b)
      → ∥ Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) ∥
    f _ _ zero w = ⊥-elim (inlinr-↛-⊥ (isoInvInjective isoN _ _ w))
    f v₁ v₂ (suc n) (b , w , ed)
      with from-NG b | inspect from-NG b
    ... | inl x       | [ from-NGb≡inlx ]
      = | x , v₂ , subst (λ o → E G o _) helper ed |
        where
        helper : b ≡ from-V₁ x
        helper = sym (retEq e b) · cong to-NG from-NGb≡inlx
    ... | inr x | [ from-NGb≡inrx ]
      = f v₁ x n (subst (λ o → W G n _ o) helper w)
        where
        helper : b ≡ from-V₂ x
        helper = sym (retEq e b) · cong to-NG from-NGb≡inrx
```

Lemma B.10 (`decompose-image`). Let $A, B : \mathcal{U}$ and f be an embedding from A to B such that the type of fibers $\mathsf{fib}_f(x)$ is a decidable set for any $x : B$. Then, the following equivalence holds.

$$B \simeq A + \sum_{(x:B)} \neg\, \mathsf{fib}_f(x),$$

where $\mathsf{fib}_f(b) :\equiv \sum_{(a:A)} f(a) = b$.

Lemma B.11 (`∃-subtree`). Let G be a connected graph with a discrete node set such that each type of edges $\mathsf{E}_G(x, y)$ is a set for any pair of nodes x, y. If H is a subtree of G such that there is a node u in H and a node v in G but not in H, then there merely exists a subtree of G enlarging H with one additional node.

Proof. Since H is a subtree, then, there must be a pair $(h, g) : H \hookrightarrow G$. We can decompose the set of nodes of G as in (B.4–4) by applying Lemma B.10 to the embedding h and the fact that the set of nodes of H is a discrete set. We write $\mathsf{N}_{G\setminus H}$ for the set $\sum_{(x:\mathsf{N}_H)} \neg\, \mathsf{fib}_h(x)$.

$$\mathsf{N}_G \simeq \mathsf{N}_H + \mathsf{N}_{G\setminus H}. \tag{B.4–4}$$

Let p be of type $\|\Sigma_{(x:\mathsf{N}_H)} \Sigma_{(y:\mathsf{N}_{G\setminus H})} \mathsf{E}_G(x, y)\|$, obtained by applying Lemma B.9 to the fact that G is connected, and the node set of G is partitioned as the coproduct of two nonempty sets. The sets N_H and $\mathsf{N}_{G\setminus H}$ are nonempty by assumption. Now, since the goal of this proof is a proposition, by eliminating of the propositional truncation applied to p, we can assume that there is an edge e from a node in H to some node in $\mathsf{N}_{G\setminus H}$. Finally, by Lemma B.20, the graph H can be extended by adding to it the edge e to get the subgraph H^* of G, similarly as illustrated in Figure B.4. The definition of H^* is given in Definition B.12. The proof H^* is a subtree of G is given in Lemma B.20. □

The remainder of this section is devoted to supporting the construction of the extended subtree H^* of G, which is crucial for the proof of Lemma B.11. The definition of H^* is given in Definition B.12. The proofs that H^* is a subgraph and a subtree are given in Lemmas B.14 and B.20, respectively. We assume below that H is a subgraph of G, defined by $(h, g) : H \hookrightarrow G$. Additionally, there is a designated edge $\hat{e}$ from $\hat{x}$ in H to $\hat{y}$ in G. The node $\hat{y}$ is not in H, as illustrated in Figure B.4. As a matter of notation, the singleton graph formed by the node x with no edges is denoted by $\{x\}$.

Definition B.12. The graph obtained from adding to H the edge $\hat{e}$ is referred as to H^*. Formally speaking, the set of nodes N_{H^*} is the set $\mathsf{N}_H + \{\hat{y}\}$ and the family of edges in H^* is defined below. Recall that the function h, appearing below in (B.4–5), is the embedding from

Figure B.3: An excerpt of the Agda term for Lemma B.11.

```
module _ (G : Graph {ℓ})
         (G-is-connected : isGConnected G)
         (_≟Node_ : (x y : N G) → Dec (x ≡ y))
         (E-is-set : (x y : N G) → isSet (E G x y)) where
  ∃-subtree
    : (H : Graph)
    → (H-subtree : isSubtree G H)
    → (u : N H) → (v : N G)
    → ¬ (fiber (Subgraph.h (isSubtree.is-subgraph H-subtree)) v)
    → ∥ Σ[ H* ∈ Graph ] isSubtree G H* × (N H* ≃ (N H + 𝟙)) ∥
  ∃-subtree H H-subtree u v v-not-in-H =
    trunc-elim isPropPropTrunc helper ∃-edgecut
    where
    H-subgraph = isSubtree.is-subgraph H-subtree
    h = Subgraph.h H-subgraph
    h-is-emb = Subgraph.h-is-emb H-subgraph
    h-has-dec-image = isSubtree.dec-fiber H-subtree
    V₁ = N H
    isoN : N G ≃ V₁ + V₂
    isoN = decompose-image _ _ h h-is-emb h-has-dec-image

    open EdgeCutLemma G G-is-connected
                      isoN u (v , v-not-in-H) hiding (E*)

    helper : Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) → _
    helper (x , y , ed) = ∣ H* , H*-subtree , e' ∣
      where
      -- H* is the graph obtained by adding an edge to H.
      -- H*-subtree is a term constructed in Lemma 4.5-4.16
```

N_H to N_{H^*} given by the fact that H is a subgraph of G.

$$\mathsf{E}_{H^*}(x,y) :\equiv \begin{cases} \mathsf{E}_H(a,b) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inl}(b), \\ h(a) = h(\hat{x}) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inr}(b), \\ \mathbb{0} & \text{otherwise.} \end{cases} \tag{B.4–5}$$

Lemma B.13. Let H^* be the graph defined in Definition B.12. The following properties hold for $a, b : \mathsf{N}_H$ and $c : \mathsf{N}_{\{\hat{y}\}}$.

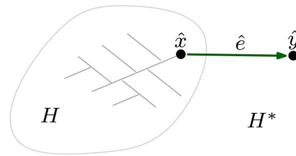


Figure B.4: The graph H^*, mentioned in Lemmas B.13 to B.20, obtained by adding an edge $\hat{e}$ to H. The edge $\hat{e}$ is given by Lemma B.9.

1. The type $\mathsf{E}_{H^*}(\mathsf{inl}(a), \mathsf{inr}(b))$ is a proposition.
2. The type $\mathsf{E}_{H^*}(\mathsf{inl}(\hat{x}), \mathsf{inr}(c))$ is contractible.
3. The type $\Sigma_{(a : \mathsf{N}_H)} \mathsf{E}_{H^*}(\mathsf{inl}(a), \mathsf{inr}(\hat{y}))$ is contractible.

Lemma B.14 (`H*-subgraph`). The graph H^* is a subgraph of G.

Proof. To show that H^* is a subgraph of G, it suffices to provide an embedding $h^* : \mathsf{N}_{H^*} \to \mathsf{N}_G$ and a function $g^* : \Pi_{(x,y : N_H)} \mathsf{E}_{H^*}(x, y) \to \mathsf{E}_G(h(x), h(y))$ such that for all $x, y : N_H$, the function $g^*(x, y)$ is an embedding.

Since H is a subgraph of G, let $(h, g) : H \hookrightarrow G$, as stated in Definition B.7.

$$h^*(x) :\equiv \begin{cases} h(a) & \text{if } x :\equiv \mathsf{inl}(a) \text{ for } a : \mathsf{N}_H, \\ \hat{y} & \text{otherwise.} \end{cases}$$

It is clear that h^* is an embedding, since when restricting to H, it is the embedding h. Otherwise, it is a map from a contractible domain, which is clearly an embedding.

Finally, let $g^* : \Pi_{(a,b : \mathsf{N}_{H^*})} \mathsf{E}_{H^*}(a, b) \to \mathsf{E}_{H^*}(h^*(a), h^*(b))$ be the mapping on edges in H^* defined as follows.

$$g^*(x, y, e) :\equiv \begin{cases} g(a, b, e) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inl}(b), \\ \mathsf{tr}(\mathsf{ap}_h(h^{-1}(e)), \hat{e}) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inr}(b), \\ & \text{and } e : h(a) = h(\hat{x}), \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

By definition, the function g^* restricted to H is the embedding g. Otherwise, the next corresponding nontrivial case is $g^*(\mathsf{inl}(a), \mathsf{inr}(b))$. By Lemma B.13-(1), it is possible to show that any fiber of $g^*(\mathsf{inl}(a), \mathsf{inr}(b))$ is a proposition, and it is then an embedding. In any case, we conclude that $g^*(x, y)$ is an embedding, from where the conclusion follows. □

To prove Lemmas B.19 and B.20, we need to show a few intermediate results, which we now state. In Lemmas B.15 to B.17, let $n : \mathbb{N}$ and a, b be two nodes in H.

Lemma B.15. The following equivalence holds.

$$W^n_H(a, b) \simeq W_{H^*}(\mathsf{inl}(a), \mathsf{inl}(b)). \tag{B.4–6}$$

Lemma B.16. The following types are empty.

1. $W^n_{H^*}(\hat{y}, \mathsf{inl}(a))$.

2. $\Pi_{(v:\mathsf{N}_H)}\ \mathsf{isContr}(W^n_{H^*}(\mathsf{inl}(a), \mathsf{inl}(v)))$.
3. $\Sigma_{(n:\mathbb{N})}\ W^{n+1}_{H^*}(\hat{y}, \hat{y})$.

Lemma B.17. The following types are contractible.

1. $W^0_{H^*}(\hat{y}, \hat{y})$.
2. $\Sigma_{(n:\mathbb{N})}\ W^n_{H^*}(\hat{y}, \hat{y})$.

Lemma B.18. The type in (B.4–7) is empty.

$$\sum_{(y:\{\hat{y}\})} \prod_{(v:\mathsf{N}_{H^*})} \mathsf{isContr}\left(\sum_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inr}(y), v)\right). \tag{B.4–7}$$

Proof. It suffices to show that there is no walk from y to some node in H. Let y be a node in $\{\hat{y}\}$ and v be a node in H^*.

$$P(y, v) :\equiv \mathsf{isContr}\left(\sum_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inr}(y), v)\right).$$

Then,

$$\begin{aligned}
\sum_{(y:\{\hat{y}\})} \prod_{(v:\mathsf{N}_{H^*})} P(y, v) &\simeq \prod_{(v:\mathsf{N}_{H^*})} P(\hat{y}, v) \\
&\simeq \prod_{(v:\mathsf{N}_H)} P(\hat{y}, \mathsf{inl}(v)) \times \prod_{(v:\{\hat{y}\})} P(\hat{y}, \mathsf{inr}(v)) \\
&\simeq \mathbb{0} \times \prod_{(v:\{\hat{y}\})} P(\hat{y}, \mathsf{inr}(v)) \\
&\simeq \mathbb{0}.
\end{aligned}$$

□

Lemma B.19 (Bottleneck). Let G be a connected graph, H be a subtree of G with root $\mathbf{r}_H$. Then, there is a unique walk in the graph H^* from $\mathsf{inl}(\mathbf{r}_H)$ to $\hat{y}$.

Proof. It suffices to show that the following type is contractible.

$$\sum_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}). \tag{B.4–8}$$

Then,

$$\sum_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}) \simeq \mathbb{0} + \sum_{(n:\mathbb{N})} W^{n+1}_{H^*}(\mathbf{r}_H, \hat{y})$$

$$
\begin{aligned}
&\simeq \sum_{(n:\mathbb{N})} \sum_{(v:\mathsf{N}_{H^*})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), v) \times \mathsf{E}_{H^*}(v, \hat{y}) \\
&\simeq \sum_{(n:\mathbb{N})} \left(\sum_{(v:\mathsf{N}_H)} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \mathsf{inl}(v)) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \right) \\
&\qquad + \left(\sum_{(v:\{\hat{y}\})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \mathsf{inr}(v)) \times \mathsf{E}_{H^*}(\mathsf{inr}(v), \hat{y}) \right) \\
&\simeq \sum_{(n:\mathbb{N})} \sum_{(v:\mathsf{N}_H)} W^n_H(\mathbf{r}_H, v) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\qquad + \left(W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}) \times \mathbb{0} \right) \\
&\simeq \sum_{(v:\mathsf{N}_H)} \left(\sum_{(n:\mathbb{N})} W^n_H(\mathbf{r}_H, v) \right) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \sum_{(v:\mathsf{N}_H)} \mathbb{1} \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \sum_{(v:\mathsf{N}_H)} \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \mathbb{1}.
\end{aligned}
$$

□

Lemma B.20 (`H*-subtree`). The graph H^* is a subtree of G.

Proof. To show that H^* is a subtree, the following must hold:

1. The graph H^* is a connected subgraph of G, i.e., there is an embedding from H^* to G given as a pair of mappings (h^*, g^*), as in Definition B.7.

2. The type of fibers $\mathsf{fib}_{h^*}(x)$ is a decidable set for any node x in G.

3. The following type is contractible.

$$\sum_{(r:\mathsf{N}_{H*})} \prod_{(v:\mathsf{N}_{H^*})} \mathsf{isContr}\left(\sum_{(n:\mathbb{N})} W^n_{H^*}(u, v) \right). \tag{B.4–10}$$

The first condition is satisfied by Lemma B.14. Since H is a subgraph of G, we have access to the embedding given by $(h, g) : H \hookrightarrow G$. Then, the second condition follows, since the type in question is equivalent to the $\mathsf{fib}_h(b) + (\hat{y} = b)$ for any b in G, by the following calculation, and any equivalence of types preserve any property.

$$\mathsf{fib}_{h^*}(b) :\equiv \sum_{(a:\mathsf{N}_{H^*})} h^*(a) = b$$

$$
\begin{aligned}
&\simeq \left(\sum_{(a:\mathsf{N}_H)} h^*(\mathsf{inl}(a)) = b \right) + \sum_{(a:\{\hat{y}\})} h^*(\mathsf{inr}(a)) = b \\
&\simeq \left(\sum_{(a:\mathsf{N}_H)} h(a) = b \right) + (\hat{y} = b) \\
&\simeq \mathsf{fib}_h(b) + (\hat{y} = b).
\end{aligned}
$$

The mapping h^* has a decidable image inherited from h, since H is a tree, and the nodes of H form a discrete set. Finally, for the third condition, we have the following calculation. For brevity, let P be a shorthand for the type family in (B.4–10).

$$
\begin{aligned}
&\sum_{(r:\mathsf{N}_{H*})} \prod_{(v:\mathsf{N}_{H^*})} P(H^*, r, v) \\
&\simeq \sum_{(r:\mathsf{N}_H)} \prod_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) + \sum_{(r:\{\hat{y}\})} \prod_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inr}(r), v) \\
&\simeq \sum_{(r:\mathsf{N}_H)} \prod_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) + \mathbb{0} \\
&\simeq \sum_{(r:\mathsf{N}_H)} \prod_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) \\
&\simeq \sum_{(r:\mathsf{N}_H)} \Big(\prod_{(v:\mathsf{N}_H)} P(H^*, \mathsf{inl}(r), \mathsf{inl}(v)) \times \prod_{(v:\{\hat{y}\})} P(H^*, \mathsf{inl}(r), \mathsf{inr}(v)) \Big) \\
&\simeq \sum_{(r:\mathsf{N}_H)} \left(\prod_{(v:\mathsf{N}_H)} P(H, r, v) \times P(H^*, \mathsf{inl}(r), \hat{y}) \right) \\
&\simeq \sum_{\left((r,!):\sum_{(a:\mathsf{N}_H)} \prod_{(v:\mathsf{N}_H)} P(H,a,v)\right)} P(H^*, \mathsf{inl}(r), \hat{y}) \\
&\simeq P(H^*, \mathsf{inl}(\mathbf{r}_H), \hat{y}) \\
&\equiv \mathsf{isContr}\left(\sum_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}) \right) \\
&\simeq \mathsf{isContr}(\mathbb{1}) \\
&\simeq \mathbb{1}. \qquad \square
\end{aligned}
$$

B.4.1 Oriented spanning trees

In graph theory, any connected undirected graph has at least one spanning tree. In our setting, we can prove that any strongly connected and directed multigraph has at least one oriented spanning tree.

Definition B.21 (isSpanningTree). An oriented spanning tree of G is a subtree that contains all the vertices of G.

```
record isSpanningTree (H : Graph) : Type ℓ where
  open isSubtree; open Graph
  field
    is-subtree : isSubtree G H
  h = Subgraph.h (is-subgraph is-subtree)
  g = Subgraph.g (is-subgraph is-subtree)
  field
    cover-all-nodes : isEquiv h
```

We are ready now to prove the main result of this section.

Lemma B.22. Let G be a nonempty strongly connected graph such that the node set of G is finite and the family of edges of G consists of sets. Then there merely exists an oriented spanning tree of G.

Proof. Let n be the cardinality of the node set of G. We proceed by induction on n. If $n = 1$, then the graph has only one node, and its spanning tree is the same one-point graph with no edges. Otherwise, let $n > 1$. We state the induction hypothesis as the mere existence of a subtree of G with k nodes where $k < n$. Since the goal of the lemma is a proposition, we can apply the elimination principle of the truncation to the induction hypothesis to get a subtree of G with $n-1$ nodes, namely, H_{n-1}. Finally, since there is a missing node of G not in H_{n-1}, we can apply Lemma B.11 to G and H_{n-1} to obtain the required spanning tree, a graph H_n including all the nodes of G. □

The previous proof suggests that Lemma B.22 can be generalised to the case where the node set of G has a principle of choice. One can construct a chain of subtrees, ordered by the subgraph relation, using a construction similar to the argument in Lemma B.22's proof. Then, the spanning tree of the infinite graph is the maximal element in such a chain, assuming the axiom of choice, see Lemma 4.7 (Swan 2022, §4). However, we do not attempt to formalise this generalisation here.

On the other hand, one version of the Kőnig's lemma states that if an infinite graph is locally finite and connected, then the graph contains a *ray*. A ray is a simple walk that starts at one node and continues from it through infinitely many nodes. It seems natural to consider a proof of this result using Lemma B.9 and the axiom of choice. This direction is, however, left for future work. Here we only give a first proposal for the type of rays. A ray in the current setting can be defined as an infinite walk starting at the node x such that the type of occurrences of x in the walk is contractible. We can define these definitions in Agda as follows.

```
record InfiniteWalk (x : N G) : Type ℓ where
  coinductive
  field
    head : Σ[ y ∈ N G ] E G x y
    tail : InfiniteWalk (fst head)
open InfiniteWalk

{-# TERMINATING #-}
_∈w_ : (x : N G) → {y : N G} → (w : InfiniteWalk y) → Type ℓ
_∈w_ x {y} w = (x ≡ y) + (x ∈w tail w)

isRay : (x : N G) → InfiniteWalk x → Type ℓ
isRay x w = isContr (x ∈w w)
```

B.5 Topological realisation of graphs

The topological realisation of a graph can be represented by the coequalizer of the corresponding source and target functions. Every node in the graph is mapped to a point in the space. Moreover, any edge in the graph gives rise to a path in the space glued to the endpoints.

This topological point of view for representing graphs is further described in type theory by Swan (Swan 2022). It is worth noting that the type of graphs in this appendix is equivalent to the type of graphs in their setting, as the following equivalence shows.

$$\begin{aligned}
\mathsf{Graph} \; :\equiv & \sum_{(N : \mathcal{U})} (N \to N \to \mathcal{U}) \\
\simeq & \sum_{(N : \mathcal{U})} (N \times N \to \mathcal{U}) \\
\simeq & \sum_{(N,E : \mathcal{U})} (E \to (N \times N)) \\
\simeq & \sum_{(N,E : \mathcal{U})} ((E \to N) \times (E \to N)).
\end{aligned}$$

Therefore, one benefit of working in Univalent mathematics is that one can transport their results to the setting of this appendix and vice versa. Now, back to Cubical Agda, let us define the topological realisation of a graph G as the following higher inductive data type.

```
module realisation {ℓ : Level} (G : Graph {ℓ}) where
  data 𝕋¹ : Type ℓ where
```

```
    𝕟 : N G → 𝕋¹
    𝕖 : ∀ {a b} → E G a b → 𝕟 a ≡ 𝕟 b
```

To prove a few properties of this geometric realisation below, we define two handy elimination principles into propositions.

```
elimProp
  : {B : 𝕋¹ → Type ℓ}
  → ((x : 𝕋¹) → isProp (B x))
  → ((a : N G) → B (𝕟 a))
  → (x : 𝕋¹) → B x
elimProp _ f (𝕟 a) = f a
elimProp B-fiber-prop f (𝕖 {a}{b} e i) =
  isOfHLevel→isOfHLevelDep 1 B-fiber-prop (f a) (f b) (𝕖 e) i
```

For the particular case of relations, we obtain the following elimination principle.

```
elimPropRel
  : {R : 𝕋¹ → 𝕋¹ → Type ℓ}
  → ((x y : 𝕋¹) → isProp (R x y))
  → ((a b : N G) → R (𝕟 a) (𝕟 b))
  → (x y : 𝕋¹) → R x y
elimPropRel Rprop f = elimProp (λ x → isPropΠ (λ y → Rprop x y))
                               (λ x → elimProp (λ y → Rprop (𝕟 x) y) (f x))
```

The walks in the graph give rise to paths in the geometric realisation, as shown in the following Agda code. As a consequence, the connectedness of a graph implies the connectedness of its geometric realisation.

```
𝕨 : {n : ℕ} {a b : N G} → W G n a b → 𝕟 a ≡ 𝕟 b
𝕨 {zero} a=b = cong 𝕟 a=b
𝕨 {suc _} (_ , w , e) = (𝕨 w) · 𝕖 e
```

The realisation of walks using the function 𝕨 respects the concatenation of walks. In particular, it respects backward edge addition, as in the Agda code below.

```
comp-edge
  : {a b c : N G} {n : ℕ}
  → (w : W G n a b) (e : E G b c)
  → 𝕨 ((_ , w , e)) ≡ (𝕨 w) · (𝕖 e)
comp-edge {n = zero} w e = reflc
comp-edge {n = suc n} (_ , w , e₁) e₂ = cong (λ x → x · (𝕖 e₂) ) (comp-edge w e₁)
```

Let us introduce the following notions to not clash with the names of some definitions defined earlier.

Definition B.23. A graph is topologically connected if its geometric realisation is connected.

```
isConnected : Type ℓ → Type ℓ
isConnected A = (x y : A) → ∥ x ≡ y ∥
isTConnected : Graph → Type ℓ
isTConnected G = isConnected (realisation.𝕋¹ G)
```

Lemma B.24. Being connected for the realisation of a graph is a proposition.

```
isProp-isTConnected : (G : Graph) → isProp (isTConnected G)
isProp-isTConnected _ = isPropΠ λ _ → isPropΠ λ _ → isPropPropTrunc
```

Lemma B.25. Being connected for a graph implies its geometric realisation is connected.

```
isGConnected-isTConnected : (G : Graph) → isGConnected G → isTConnected G
isGConnected-isTConnected G G-is-connected =
  elimPropRel (λ _ _ → isPropPropTrunc) helper
  where
    open realisation G
    helper : (a b : N G) → ∥ 𝕟 a ≡c 𝕟 b ∥
    helper a b = trunc-elim isPropPropTrunc (λ {(_ , w) → ∣ 𝕨 w ∣}) (G-is-connected a b)
```

Definition B.26. A graph is a topological tree if its geometric realisation is contractible.

```
isTopTree : Graph → Type ℓ
isTopTree G = isContr (realisation.𝕋¹ G)
```

Using this topological point of view for graphs, we can prove that any tree, as in Definition B.5 is topologically connected and tree in a topological way. The converse is not true; see, for example, the triangle graph, where an edge connects any pair of nodes. The realisation of such a graph contains a non-trivial loop and thus is not contractible.

Lemma B.27. If the graph is a tree then it is topologically connected.

Proof. For this proof, we are only interested in what happens when we apply the geometric realisation on nodes and how the nodes are glued. Since the graph is a tree, we have access to its root node equipped with a walk to every other node, see Definition B.5. Finally, one can use the walks given by the tree to connect the nodes in the geometric realisation, as illustrated in Figure B.5 and proved in Agda code below. □

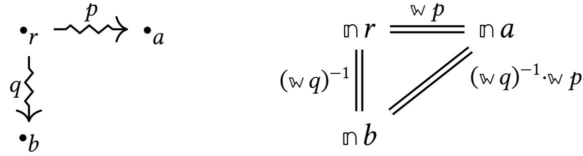


Figure B.5: It is shown the walks and paths mentioned in Lemma B.27's proof. The node r on the left represents the root of the given tree. The node a is the node connected to r by the walk p, and similarly, the node b is the node connected to r by the walk q. Then, we can connect the realisation of a and b by the walk $(\mathbb{w}(q))^{-1} \cdot \mathbb{w}(p)$.

```
module _ {ℓ : Level}(G : Graph {ℓ}) where
  open realisation
  open walk-concat G

  isTree-isTConnected : isTree G → isConnected (𝕋¹ G)
  isTree-isTConnected ((r , unique-walk-from-r-to) , _) =
    elimPropRel G ((λ _ _ → isPropPropTrunc)) helper
    where
      helper : (a b : N G) → ∥ 𝕟 {G = G} a ≡c 𝕟 b ∥
      helper a b = ∣ (sym (w G (snd p))) · w G (snd q) ∣
        where
          p : Σ[ n ∈ ℕ ] W G n r a
          p = fst (unique-walk-from-r-to a)
          q : Σ[ n ∈ ℕ ] W G n r b
          q = fst (unique-walk-from-r-to b)
```

Lemma B.28. If the graph is a tree and its topological realisation is a set, then it can be concluded that the graph is a topological tree.

```
isTree-isSet-isTopTree : isTree G → isSet (𝕋¹ G) → isTopTree G
isTree-isSet-isTopTree
  G-is-graph-tree@((r , unique-walk-from-r-to) , _)
  𝕋¹G-is-set = 𝕟 r , λ y →
    trunc-elim (𝕋¹G-is-set (𝕟 r) y)
               (λ nr=y → nr=y)
               (isTree-isTConnected G-is-graph-tree (𝕟 r) y)
```

Finally, we can prove that a tree, in a combinatorial way, is topologically a tree.

Lemma B.29. Being a tree for a graph implies its realisation is a contractible type.

Proof. Let G be a graph tree. Then, we must show that $\mathbb{T}^1(G)$ is a contractible type. To show that, let $\mathbb{n}(r)$ be the centre of contraction of $\mathbb{T}^1(G)$, where r is the root of G. Then,

we must construct a function that returns a path from $\mathbb{n}(r)$ to a for any $a : \mathbb{T}^1(G)$. We do this by induction on the constructors of $\mathbb{T}^1(G)$. The first case is the point constructor $\mathbb{n}(x)$ for $x : \mathsf{N}_G$, for which we can just return the realisation of the unique walk from r to x given by the proof that G is a tree. The second and last case is the path constructor case. Given a path $\mathbb{e}(e)$, where e is an edge from a to b in G, we must construct a path from $\mathbb{n}(r)$ to every point in the path $\mathbb{e}(e)$. Since G is a tree, we have access to a unique walk from the root r to the nodes a and b, respectively. Let p and q be such walks, as illustrated in Figure B.6. Then, the required path can be obtained considering the path $\mathbb{w}(p) \cdot \mathbb{e}(e)$.

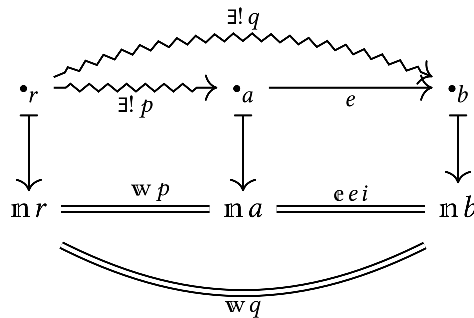


Figure B.6: The construction of a path from $\mathbb{n}(r)$ to any point in the path $\mathbb{e}(e)$.

However, for coherence, we must make sure that there is a homotopy between the paths $\mathbb{w}(p) \cdot \mathbb{e}(e)$ and $\mathbb{w}(q)$, which is the right face of the cube as illustrated in Figure B.7. The back face is the whole square of deforming the path $\mathbb{w}(p)$ to $\mathbb{w}(p) \cdot \mathbb{w}(q)$, which is precisely Lemma `compPath-filler` in the Cubical Agda library. □

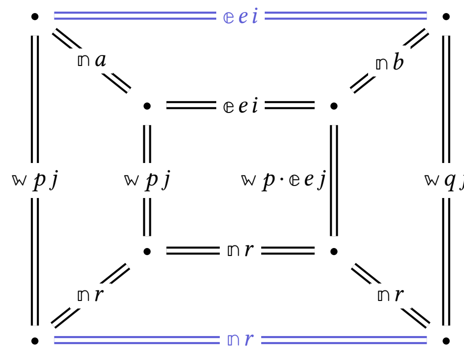


Figure B.7: The constructed cube for Lemma B.29's proof.

B.6 Discussion

Here we present one short example of transferring some concepts and results from graph theory in a classical setting to Cubical type theory. As part of this process, we have used a

Figure B.8: An Agda term for Lemma B.29.

```
isTree-isTopTree : isTree G → isTopTree G
isTree-isTopTree ((r , unique-walk-from-r-to) , _) =
  𝕟 r , helper
  where
    walk = snd
    helper : (x : 𝕋¹ G) → 𝕟 r ≡c x
    helper (𝕟 x) = 𝕨 G (walk (fst (unique-walk-from-r-to x)))
    helper (𝕖 {a}{b} e i) j
      = hcomp (λ k → λ { (i = i0) → wp j
                       ; (i = i1) → wp·𝕖e≡wq k j
                       ; (j = i0) → reflc {x = 𝕟 r} i
                       ; (j = i1) → 𝕖 e i
                       })
        (compPath-filler wp (𝕖 e) i j)
      where
      p : Σ[ n ∈ ℕ ] 𝕎 G n r a
      p = fst (unique-walk-from-r-to a)
      length-walk-p = fst p

      q : Σ[ n ∈ ℕ ] 𝕎 G n r b
      q = fst (unique-walk-from-r-to b)

      wp : 𝕟 r ≡ 𝕟 a
      wp = 𝕨 G (walk p)

      wq : 𝕟 r ≡ 𝕟 b
      wq = 𝕨 G (walk q)

      q-is-unique : q ≡c (suc (length-walk-p) , _ , walk p , e)
      q-is-unique = snd (unique-walk-from-r-to b) _

      wp·𝕖e≡wq : (wp · 𝕖 e) ≡ wq
      wp·𝕖e≡wq =  𝕨 G (walk p) · 𝕖 e
              ≡⟨ sym (comp-edge G (walk p) e) ⟩
                𝕨 G ((_ , walk p , e))
              ≡⟨ cong (λ w → 𝕨 G (walk w)) (sym q-is-unique) ⟩
                𝕨 G (walk q) ∎
```

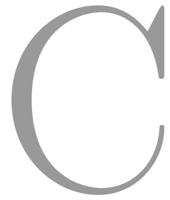
proof assistant to support this goal. Precisely, we have characterised the notion of rooted trees to construct oriented spanning trees for directed multigraphs. These concepts are the generalisation of the notion of a tree and spanning tree for undirected graphs, respectively. A proof is given for the existence of an oriented rooted spanning tree for any strongly connected graph with a finite node set and a family of edges consisting of sets. To this end, we introduce a few lemmas that suggest algorithms for constructing spanning trees. Furthermore, we show that any rooted tree is a tree in the topological sense, inspired by Swan's work on defining free groups in HoTT and using higher—inductive types to model the topological realisation of graphs (Swan 2022). The results here can

then be used to study free groups, particularly the fundamental group of a graph. In this direction, the realisation of a graph maps any of its spanning trees to a point in the space, and the remaining edges not in such a tree, become loops around the point. The loop edges then correspond to the elements of the group associated with the graph, sometimes called the fundamental group. We left this investigation for future work.

Most results here are formalised in Agda (The Agda Development Team 2023). Except for proofs in Appendix B.5, we conjecture it is only required intensional Martin-Löf type theory equipped with universes, function extensionality, and propositional truncation. To ease the work with higher–inductive types, especially in Appendix B.5, we used the Cubical mode (Vezzosi, Mörtberg, and Abel 2021) in Agda and the Cubical Agda library (Mörtberg, Andrea, and Evan 2021). Nevertheless, the type theory as presented in the HoTT Book (Univalent Foundations Program 2013) suffices to prove the results in this appendix.

Even when graph theory has been formalised before in type theory with proof-assistants, as the formalisation of the 4CT in Coq (Gonthier 2008), there are still a few works in homotopy type theory (Kraus and Raumer 2020, 2021; Prieto-Cubides 2022). As far as we know, the proofs and some types given here are original in this context. We believe this development contributes to the project of this thesis and the formalisation of mathematical content in type theories alike. We expect more contributions in this direction in the future.

A notable work close to ours is the recent work in Agda–UniMath (Rijke et al. 2023), an Agda library for Univalent mathematics. Their authors formalised the notion of trees, rooted and quasi–rooted trees, for the case of undirected graphs. In future, we plan to transfer the results shown here to Agda–UniMath and make them available to a broader audience. In addition, ongoing work explores other topics, such as the 2-cells realisation of a graph, where 2-cells correspond to faces of a graph map.



Yet Another HIT for Graphs

In Appendix B, we describe the topological realisation of graphs considering 0-cells (nodes) and 1-cells (edges). Building upon this, we pair a graph G with a graph map $\mathscr{M}$, and adds 2-cells to the realisation of G for considering faces within $\mathscr{M}$. This yields an enhanced higher inductive type $\mathbb{T}^2(G, \mathscr{M})$, referred to as the 2-cell realisation of G with respect to $\mathscr{M}$, extending the type $\mathbb{T}^1(G)$ detailed in Appendix B.5.

The first two constructors for $\mathbb{T}^2(G, \mathscr{M})$, $\mathbb{n}$ and $\mathbb{e}$, are the same as those for $\mathbb{T}^1(G)$. The 2-path constructor $\mathbb{f}$ in (C.0−1) yields identifications between the realisations of walks on the boundary of each face. Precisely, given two nodes a and b in the boundary of a face, we identify the realisations of the counter-clockwise walk and the clockwise walk from a to b, lifting the notion of homotopy of walks in the graph map to actual paths in the space. The function $\mathbb{w}$ used to lift walks into the space is defined similarly as in Appendix B.5.

$$
\begin{aligned}
&\mathsf{data}\ \mathbb{T}^2\ (G : \mathsf{Graph})\,(\mathscr{M} : \mathsf{Map}(G)) : \mathscr{U}\\
&\qquad \mathbb{n} : \mathsf{N}_G \to \mathbb{T}^2(G, \mathscr{M})\\
&\qquad \mathbb{e} : \Pi_{(a\,b\,:\,\mathsf{N}_G)}\,.\,\mathsf{E}_G(a,b) \to \mathbb{n}(a) = \mathbb{n}(b)\\
&\qquad \mathbb{f} : \Pi_{(\mathscr{F}\,:\,\mathsf{Face}(G,\mathscr{M}))}\,.\,\Pi_{(a\,b\,:\,\mathsf{N}_{\mathscr{F}})}\,.\,\mathbb{w}(\mathsf{cw}(\mathscr{F}, a, b)) = \mathbb{w}(\mathsf{ccw}(\mathscr{F}, a, b)).
\end{aligned}
\tag{C.0−1}
$$

Considering walk homotopies as defined in Section 5.4, we show that any pair of walks in a graph G with a discrete node set and subject to the graph map $\mathscr{M}$, when homotopic,

yield identical 2-cell topological realisations (refer to Lemma C.1). Moreover, when such homotopies are restricted to the plane, one can show that the corresponding realisations of homotopic walks are merely equal (see Corollary C.3).

In the rest of this appendix, we present definitions and theorems using Agda syntax, specifically employing Agda (v2.6.2.2) for type-checking our constructions. To ensure compatibility with HoTT, we invoke the flag --without-K.

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
module HIT where
  open import foundations.Core
  open import lib.graph-definitions.Graph
  open Graph


  open import lib.graph-embeddings.Map
  open import lib.graph-walks.Walk
  open import lib.graph-walks.Walk.Composition
  open import lib.graph-embeddings.Map.Face
  open import lib.graph-homomorphisms.Hom


  open import lib.graph-classes.UndirectedGraph
  open import lib.graph-transformations.U
  open import foundations.Rewriting
```

To ease the work with higher inductive types in this setting, we use the flag --rewriting that alow us to define custom rewriting rules for the HIT path constructions and the corresponding computation rules. Agda uses these reduction rules during the type-checking process. As an example, the following rule, the runit law for path allows us to treat as definitionally equal the paths p and $p \cdot \mathsf{refl}$, where p is a path in a type A.

```
  postulate
    runit : ∀ {ℓ} {A : Type ℓ} {a a' : A} {p : a ≡ a'} → p · idp ↦ p        (2)
  {-# REWRITE runit #-}
```

The reduction relation, denoted by (↦), is defined as follows.

```
infix 30 _↦_
postulate
  _↦_ : ∀ {ℓ} {A : Type ℓ} → A → A → Type ℓ
{-# BUILTIN REWRITE _↦_ #-}
```

Careful attention is needed when augmenting Agda with manual rewriting rules as done above. However, the rule in (2) has been proven to enhance type-checking efficiency without inducing confluence problems. Lastly, some im- ports and definitions are hidden in the following Agda excerpts for brevity. We refer the reader to the corresponding files for the complete detail. See Appendix A.

C.1 The 2-cell topological realisation of graphs

```
module construction {ℓ : Level} (G : Graph ℓ) (M : Map G) where
  open import foundations.Core
  open import lib.graph-embeddings.Map.Face.Walk
  open FaceWalks G
```

The 2-cell topological realisation of a graph with a graph map is defined as the HIT with three constructors, one for each type of cells. The 0-cells are constructed by 𝕟, 1-cells by 𝕖 and 2-cells by 𝕗 below. These constructors need to be defined as postulates in Agda since there is no support for defining HITS natively.

```
  postulate
    𝕋² : Type ℓ
    𝕟 : Node G → 𝕋²
    𝕖 : ∀ {x y} → Edge G x y → 𝕟 x ≡ 𝕟 y
```

To define the 2-cell constructor 𝕗, we need to consider the faces of the graph map and the walks in them promoted as paths in the geometric realisation, for which a few auxiliary functions are needed and stated below.

C.1.1 Promoting walks to equalities

As part of the construction of the geometric realisation of a graph, we need to be able to lift a walk into an arbitrary space. To do so, we require a function mapping nodes to points and another function mapping edges to paths. Then, it is possible to define edge-by-edge a function mapping a walk into a path in the space as follows.

```
    to-eq : (f : Node G → A)
          → ({x y : Node G} → Edge G x y → f x ≡ f y)
          → {x y : Node G} → Walk (U G) x y → f x ≡ f y
    to-eq f g = λ {⟨ x ⟩ → refl (f x)
      ; (inl xz ⊙ w) → g xz · to-eq f g w
      ; (inr zx ⊙ w) → ! (g zx) · to-eq f g w
      }
```

As suggested in Appendix B, one can prove that lifting walks into the space is an operation that respects the composition of walks. Meaning that the function to-eq maps a composition of walks to the composition of the corresponding paths.

```
    to-eq-comp-·w : (f : Node G → A)
                  → (g : {x y : Node G} → Edge G x y → f x ≡ f y)
                  → {x y z : Node G} → (p : Walk (U G) x y) (q : Walk (U G) y z)
                  → to-eq f g (p ·w q) ≡ (to-eq f g p) · (to-eq f g q)
    to-eq-comp-·w f g = λ {
```

```
⟨ x ⟩ w₂ → idp
; (inl a ⊙ w₁) w₂ → ap (λ w → (g a) · w) (to-eq-comp-·w f g w₁ w₂)
                  · ( ! (·-assoc (g a) (to-eq f g w₁) (to-eq f g w₂)))
; (inr a ⊙ w₁) w₂ → ap (λ w → (! (g a)) · w) (to-eq-comp-·w f g w₁ w₂)
                  · ( ! (·-assoc (! (g a)) (to-eq f g w₁) (to-eq f g w₂)))
}
```

We use a shorthand for the above function to-eq specialised to type $\mathbb{T}^2(G, \mathcal{M})$ for convenience.

```
w : ∀ {x y} → Walk (U G) x y → n x ≡ n y
w = to-eq n e
```

Finally, we define the 2-cell constructor f, which identifies the realisation of walks on the boundary of each face. Precisely, given two nodes x and y in the boundary of a face, the 2-cell constructor of $\mathbb{T}^2(G, \mathcal{M})$ is the homotopy between the counter-clockwise walk and the clockwise walk from x to y.

```
postulate
  f : (𝓕 : Face G M) → (a b : Node (Face.A 𝓕))
    → w (cw-walk 𝓕 a b) ≡ w (ccw-walk 𝓕 a b)
```

C.1.2 Recursion principle

The non-dependent eliminator 𝕋²-rec for $\mathbb{T}^2(G, \mathcal{M})$ allows us defining a function of type $\mathbb{T}^2(G, \mathcal{M}) \to A$ for any $A : \mathcal{U}$.

```
module Recursion {ℓ₂} (A : Type ℓ₂)
      (A-n : Node G → A)
      (A-e : ∀ {x y} → (e : Edge G x y) → A-n x ≡ A-n y)
      (A-f : (𝓕 : Face G M) → (a b : Node (Face.A 𝓕)) → let A-w = to-eq A-n A-e in
          A-w (cw-walk 𝓕 a b) ≡ A-w (ccw-walk 𝓕 a b)) where
  postulate
    𝕋²-rec : 𝕋² → A
```

The corresponding computation rules for nodes, edges and faces are introduced as rewriting rules and named as 𝕋²-β-rec-nodes, 𝕋²-β-rec-edges, and 𝕋²-β-rec-faces, respectively.

```
  postulate
    𝕋²-β-rec-nodes : (x : Node G) → 𝕋²-rec (n x) ↦ A-n x
    {-# REWRITE 𝕋²-β-rec-nodes #-}
    𝕋²-β-rec-edges : {x y : Node G} → (e : Edge G x y) → ap 𝕋²-rec (e e) ↦ A-e e
    {-# REWRITE 𝕋²-β-rec-edges #-}
```

The computation rule for faces, 𝕋²-β-rec-faces, is a bit more involved since we need to consider the functorial application of a function on two-dimensional paths.

```
lhs : ∀ {x y} → (g : Walk (U G) x y) → ap 𝕋²-rec (𝕨 g) ≡ A-𝕨 g
lhs ⟨ x ⟩ = idp
lhs (inl e ⊙ w) =
  ap 𝕋²-rec (𝕖 e · 𝕨 w)             ≡⟨ ap-· _ (𝕖 e) _ ⟩
  ap 𝕋²-rec (𝕖 e) · ap 𝕋²-rec (𝕨 w) ≡⟨ ap (A-𝕖 e ·_) (lhs w) ⟩
  (A-𝕖 e) · A-𝕨 w                   ≡⟨⟩
  A-𝕨 (inl e ⊙ w)                   ∎
lhs (inr e ⊙ w) =
  ap 𝕋²-rec (! 𝕖 e · 𝕨 w)             ≡⟨ ap-· _ (! 𝕖 e) _ ⟩
  ap 𝕋²-rec (! 𝕖 e) · ap 𝕋²-rec (𝕨 w) ≡⟨ ap (_· _) (ap-inv 𝕋²-rec (𝕖 e)) ⟩
  ! A-𝕖 e · ap 𝕋²-rec (𝕨 w)           ≡⟨ ap (! A-𝕖 e ·_) (lhs w) ⟩
  ! A-𝕖 e · A-𝕨 w                     ≡⟨⟩
  A-𝕨 (inr e ⊙ w)                     ∎
postulate
  𝕋²-β-rec-faces
    : (𝓕 : Face G M) → (a b : Node (Face.A 𝓕))
    → ap (ap 𝕋²-rec) (𝕗 𝓕 a b)
      ↦ lhs (cw-walk 𝓕 a b) · A-𝕗 𝓕 a b · ! lhs (ccw-walk 𝓕 a b)
{-# REWRITE 𝕋²-β-rec-faces #-}
```

C.1.3 Induction principle

We define in this subsection the dependent eliminator for 𝔾, for which, we must generalise the walk lifting function defined above. We believe that Figure C.2 may help to understand how this generalisation works. Additionally, the notation for heterogeneous equalities introduced by Licata and Brunerie (Licata and Brunerie 2015), namely pathovers, is used below, similarly as defined in the HoTT-Agda library (Brunerie, Hou (Favonia), Cavallo, et al. n.d.), see the intuition behind it in Figure C.1.

```
to-deq : {ℓ' : Level} {A : 𝕋² → Type ℓ'}
       → (f : (x : Node G) → A (𝕟 x))
       → (g : ∀ {x y : Node G} → (e : Edge G x y)
            → f x ≡ f y [ A ↓ (𝕖 e) ])
       → {x y : Node G} → (w : Walk (U G) x y)
       → f x ≡ f y [ A ↓ 𝕨 w ]
to-deq f _ ⟨ x ⟩ = refl (f x)
to-deq f g (inl e ⊙ w) = pathover-comp {p = 𝕖 e} {q = 𝕨 w} (g e) (to-deq f g w)
to-deq f g (inr e ⊙ w) = pathover-comp {p = (𝕖 e) ⁻¹} {q = 𝕨 w}
                           (! move-transport {α = 𝕖 e} (g e))
                           (to-deq f g w)
```

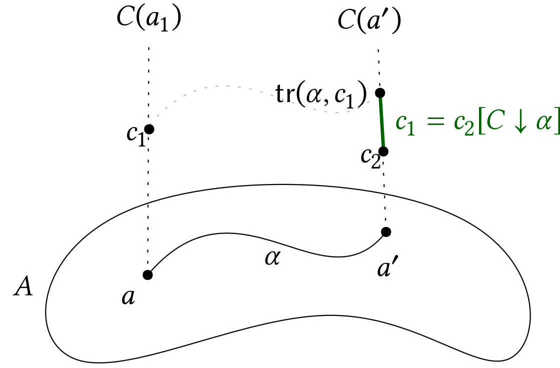


Figure C.1: The type denoted by $c_1 = c_2[C \downarrow \alpha]$ is a shorthand for the type of paths between $\mathrm{tr}^C(\alpha, c_1)$ and c_2, where $\alpha : a = a'$ is a path in $A : \mathcal{U}$, and $c_1 : C(a)$ and $c_2 : C(a')$ are points in the fibre of the type family C over A.

Finally, it is now possible to declare the required data for defining a dependent function of type $\Pi_{(x:\mathbb{T}^2(G,\mathcal{M}))} A(x)$ for any type family A over $\mathbb{T}^2(G, \mathcal{M})$, which are the dependent functions A-𝕟, A-𝕖, and A-𝕗 of type as in (3).

```
module Induction
  (A : 𝕋² → Type ℓ)
  (A-𝕟 : (x : Node G) → A (𝕟 x))
  (A-𝕖 : ∀ {x y} → (e : Edge G x y) → A-𝕟 x ≡ A-𝕟 y [ A ↓ 𝕖 e ])
  (A-𝕗 : (𝓕 : Face G M) → (a b : Node (Face.A 𝓕))
      → let wᵈ = to-deq A-𝕟 A-𝕖
        in wᵈ (cw-walk 𝓕 a b) ≡ wᵈ (ccw-walk 𝓕 a b)                    (3)
          [ (λ x≡y → A-𝕟 (Hom.α (Face.h 𝓕) a) ≡ A-𝕟 (Hom.α (Face.h 𝓕) b)
                    [ A ↓ x≡y ])
            ↓ 𝕗 𝓕 a b
          ])
  where
```

The dependent eliminator is defined below as `𝕋²-ind`. The corresponding computation rules are `𝕋²-β-ind-nodes`, `𝕋²-β-ind-edges`, and `𝕋²-β-ind-faces`, respectively, and stated as rewriting rules using the `REWRITE` pragma.

```
  postulate
    𝕋²-ind : (x : 𝕋²) → A x
    𝕋²-β-ind-nodes : (x : Node G) → 𝕋²-ind (𝕟 x) ↦ A-𝕟 x
    {-# REWRITE 𝕋²-β-ind-nodes #-}

    𝕋²-β-ind-edges : ∀ {x y} → (e : Edge G x y) → apd 𝕋²-ind (𝕖 e) ↦ A-𝕖 e
    {-# REWRITE 𝕋²-β-ind-edges #-}
```

However, the computation rule for the face constructor requires a more involved equality reasoning as one has to consider the path over the 2-cell associated with the face F and the walks (`cw-walk` and `ccw-walk`) in the definition. This requires us to construct

two additional paths, `rhs` and `lhs`, given below. The function `apd²` is defined in Lemma 6.4.6 in the HoTT Book (Univalent Foundations Program 2013, § 6).

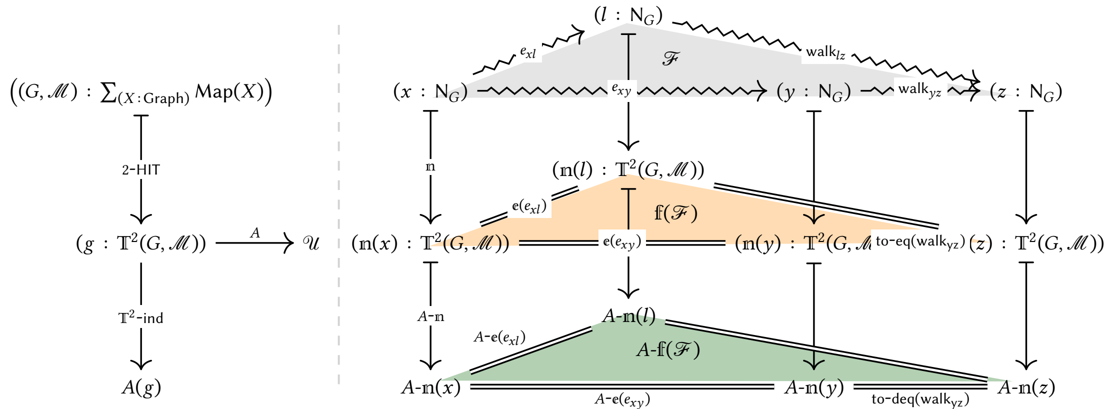
```
module _ (𝓕 : Face G M) (a b : Node (Face.A 𝓕)) where
  F' : ∀ {x y} p → Type _
  F' {x} {y} p = A-𝕞 x ≡ A-𝕞 y [ A ↓ p ]
  rhs : ∀ {x y} → (w : Walk (U G) x y)
      → apd 𝕋²-ind (𝕨 w) ≡ (𝕨ᵈ w) [ (F' {x}{y}) ↓ refl (𝕨 w) ]
  rhs ⟨ x ⟩ = idp
  rhs w'@(inl e ⊙ w) = begin
    tr F' (refl (𝕨 w')) (apd 𝕋²-ind (𝕨 w'))   ≡⟨⟩
    apd 𝕋²-ind (𝕨 w')                          ≡⟨⟩
    apd 𝕋²-ind (𝕖 e · 𝕨 w)                     ≡⟨ i ⟩
    apd 𝕋²-ind (𝕖 e) ·d apd 𝕋²-ind (𝕨 w)       ≡⟨⟩
    A-𝕖 e ·d apd 𝕋²-ind (𝕨 w)                  ≡⟨ ii ⟩
    A-𝕖 e ·d 𝕨ᵈ w                              ≡⟨⟩
    𝕨ᵈ w'                                      ∎
    where
      i  = apd-· 𝕋²-ind (𝕖 e) (𝕨 w)
      ii = ap (λ o → pathover-comp {p = (𝕖 e)} {q = 𝕨 w} _ o ) (rhs w)

  rhs w'@(inr e ⊙ w) = begin
    tr F' (refl (𝕨 w')) (apd 𝕋²-ind (𝕨 w'))  ≡⟨⟩
    apd 𝕋²-ind (𝕨 w')                         ≡⟨⟩
    apd 𝕋²-ind (((𝕖 e) ⁻¹) · 𝕨 w)             ≡⟨ i ⟩
    apd 𝕋²-ind ((𝕖 e) ⁻¹) ·d apd 𝕋²-ind (𝕨 w) ≡⟨ ii ⟩
    (! move-transport {α = 𝕖 e} (apd 𝕋²-ind (𝕖 e))) ·d apd 𝕋²-ind (𝕨 w)
                                               ≡⟨ iii ⟩
    (! move-transport {α = 𝕖 e} (apd 𝕋²-ind (𝕖 e))) ·d 𝕨ᵈ w
                                               ≡⟨⟩
    𝕨ᵈ w'                                      ∎
    where
      i   = apd-· 𝕋²-ind ((𝕖 e) ⁻¹) (𝕨 w)
      ii  = ap (λ o → pathover-comp {p = (𝕖 e) ⁻¹} o _) (apd-! 𝕋²-ind (𝕖 e))
      iii = ap (λ o → pathover-comp {p = (𝕖 e) ⁻¹}{q = 𝕨 w} _ o ) (rhs w)

  lhs : ∀ {x y} → (w : Walk (U G) x y)
      → 𝕨ᵈ w ≡ apd 𝕋²-ind (𝕨 w) [ (F' {x}{y}) ↓ refl (𝕨 w) ]
  lhs w = ! rhs w
  pathover
```

```
      : apd 𝕋²-ind (w (cw-walk 𝓕 a b)) ≡ apd 𝕋²-ind (w (ccw-walk 𝓕 a b))
                                        [ F' ↓ 𝕗 𝓕 a b ]
    pathover = pathover-comp {p = refl (w (cw-walk 𝓕 a b))} {q = 𝕗 𝓕 a b}
      (rhs (cw-walk 𝓕 a b))
      (pathover-comp {p = 𝕗 𝓕 a b} {q = refl (w (ccw-walk 𝓕 a b))}
        (A-𝕗 𝓕 a b)
        (lhs (ccw-walk 𝓕 a b)))
    postulate
      𝕋²-β-ind-faces
        : apd² 𝕋²-ind (𝕗 𝓕 a b) ↦ pathover
      {-# REWRITE 𝕋²-β-ind-faces #-}
```

Figure C.2: The figure shows on the top a face F of a graph map $\mathscr{M}$ for graph G. The face can be seen as the highlighted region between two walks from x to z in a graph G. Such walks are promoted into equalities in the 2-cell realisation of G by using the function `to-eq`. The 2-cell associated with F is denoted by $\mathbb{f}(\mathscr{F})$. Later, one can define a depending function of type $\Pi_{(x\,:\,\mathbb{T}^2(G,\mathscr{M}))}\, A(x)$ for a type family A. The required data is the dependent functions A-$\mathbb{n}$, A-$\mathbb{e}$, and A-$\mathbb{f}$ of type as in (3).



Let us look at the elimination principle in some particular situations.

C.1.4 Eliminating into propositions

```
module toProp {ℓ : Level} (G : Graph ℓ) (𝕄 : Map G) where
  open •-walk (U G)
  open construction G 𝕄
```

The recursion principle and the induction principle, Appendices C.1.2 and C.1.3, are the main tools to prove lemmas about the 2-cell topological realisation of a graph. However, in the particular case, where a lemma is a proposition, another simpler principle can be used, since, the path space of a proposition is a proposition.

Recursion Principle

```
𝕋²-rec : (A : Type ℓ) → isProp A → (Node G → A) → (𝕋² → A)
𝕋²-rec A A-is-prop A-𝔫 = Recursion.𝕋²-rec A A-𝔫 A-𝔢 A-𝔣
  where
  A-𝔢 : ∀ {x y} → (e : Edge G x y) → A-𝔫 x ≡ A-𝔫 y
  A-𝔢 {x = x}{y} _ = A-is-prop (A-𝔫 x) (A-𝔫 y)
  A-𝔣 : _
  A-𝔣 𝓕 a b = isProp-isSet A-is-prop _ _ _ _
```

Induction principle

```
𝕋²-ind : (A : 𝕋² → Type ℓ )
  → ((x : Node G) → A (𝔫 x))
  → ((x : Node G) → isProp (A (𝔫 x)))
  → (x : 𝕋²) → A x
𝕋²-ind A A-𝔫 A-forms-props = Induction.𝕋²-ind A A-𝔫 A-𝔢 A-𝔣
  where
  A-𝔢 : {x y : Node G} → (e : Edge G x y) → A-𝔫 x ≡ A-𝔫 y [ A ↓ 𝔢 e ]
  A-𝔢 {y = y} e = A-forms-props y _ _
  A-𝔣 : _
  A-𝔣 𝓕 a b = isProp-isSet (A-forms-props _) _ _ _ _
```

C.1.5 Eliminating into sets

```
module toSet {ℓ : Level} (G : Graph ℓ) (𝓜 : Map G) where
  open •-walk (U G)
  open construction G 𝓜
```

Recursion principle

```
𝕋²-rec : (A : Type ℓ) → isSet A
  → (A-𝔫 : Node G → A)
  → (∀ {x y} → Edge G x y → A-𝔫 x ≡ A-𝔫 y)
  → (𝕋² → A)
𝕋²-rec A A-is-set A-𝔫 A-𝔢 = Recursion.𝕋²-rec A A-𝔫 A-𝔢 (λ _ _ _ → A-is-set _ _ _ _)
```

Induction principle

```
𝕋²-ind : (A : 𝕋² → Type ℓ )
  → (A-𝔫 : (x : Node G) → A (𝔫 x))
```

```
    → (A-𝕖 : ∀ {x y} → (e : Edge G x y) → A-𝕟 x ≡ A-𝕟 y [ A ↓ 𝕖 e ])
    → (∀ x → isSet (A (𝕟 x)))
    → (x : 𝕋²) → A x
  𝕋²-ind A A-𝕟 A-𝕖 A-sets = Induction.𝕋²-ind A A-𝕟 A-𝕖 (λ _ _ _ → A-sets _ _ _ _ _)
```

C.2 Promoting walk homotopies to 2-paths

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
module Homotopic-are-equal
  where
  open import foundations.Core
  open import foundations.Rewriting
  open import lib.graph-definitions.Graph
  open import lib.graph-embeddings.Map
  open import lib.graph-embeddings.Map.Spherical
  open import lib.graph-transformations.U
  open Graph
```

In Section 5.4, we establish a combinatorial definition for the notion of homotopy between walks using a graph map. This notion is a congruence relation on the set of walks of the same endpoints, to say that walks are related if one can deform one into another. In this subsection, we focus on relating the walk homotopy notion with the internal notion of path homotopy in HoTT. First, in Lemma C.1, we prove that if there is a walk homotopy between any pair of walks, assuming that the node set of the graph is discrete, then their corresponding topological realisation are equal. Second, if one only considers walk homotopies in the plane, then the corresponding realisations of such walks are merely equal; see Corollary C.3

Lemma C.1. Let G be a graph with discrete set of nodes and $\mathscr{M}$ be a map for G. If p and q are walks in the symmetrisation of G such that $p \sim_{\mathscr{M}} q$, then p and q are merely equal.

Proof. It follows by induction on the given walk homotopy $p \sim_{\mathscr{M}} q$ for any walk p and q in the symmetrisation of G. The detailed proof is given below in (4). □

```
  module _ {ℓ : Level} (G : Graph ℓ)
    (_≟Node_ : (x y : Node (U G)) → (x ≡ y) + (x ≢ y)) (𝓜 : Map G)
    where
    open import lib.graph-embeddings.Map.Face.Walk
    open import lib.graph-embeddings.Map.Face.Walk.Homotopy
    open import lib.graph-embeddings.Map.Spherical-is-enough
```

```
open import lib.graph-walks.Walk hiding (length)
open import lib.graph-walks.Walk.Composition
open import lib.graph-walks.Walk.QuasiSimple
open import HIT
open ·-walk (U G)
open FaceWalks
open HomotopyWalks
open WR (U G) _≟Node_ hiding (P)
open construction G 𝕄

w[_] = to-eq 𝕟 𝕖
walk-homotopy-gives-homotopy
  : ∀ {x y} {p q : Walk (U G) x y}
  → p ~⟨ 𝕄 ⟩~ q → w p ≡ w q
walk-homotopy-gives-homotopy = λ {
    hwalk-refl → idp
  ; (hwalk-symm q~p) → ! walk-homotopy-gives-homotopy q~p
  ; (hwalk-trans {w₂ = r} p~r r~q) → let
      p=r = walk-homotopy-gives-homotopy {q = r} p~r
      r=q = walk-homotopy-gives-homotopy {p = r} r~q
      in p=r · r=q
  ; (collapse 𝓕 {a}{b} p q) →
    let
      i   = to-eq-comp-·w 𝕟 𝕖 p (cw-walk _ 𝓕 a b ·w q)                        (4)
      ii = ap (w[ p ] ·_) (to-eq-comp-·w 𝕟 𝕖 (cw-walk _ 𝓕 a b) q)
      iii = ap (λ r → w[ p ] · (r · w q)) (𝕗 𝓕 a b)
      iv = ap (λ r → w[ p ] · r) (! to-eq-comp-·w 𝕟 𝕖 (ccw-walk _ 𝓕 a b) q)
      v   = ! to-eq-comp-·w 𝕟 𝕖 p (ccw-walk _ 𝓕 a b ·w q)
    in begin
      w[ p ·w cw-walk _ 𝓕 a b ·w q ]                ≡⟨ i ⟩
      w[ p ] · w[ cw-walk _ 𝓕 a b ·w q ]            ≡⟨ ii ⟩
      w[ p ] · (w[ cw-walk _ 𝓕 a b ] · w[ q ])  ≡⟨ iii ⟩
      w[ p ] · (w[ ccw-walk _ 𝓕 a b ] · w[ q ]) ≡⟨ iv ⟩
      w[ p ] · w[ ccw-walk _ 𝓕 a b ·w q ]           ≡⟨ v ⟩
      w[ p ·w ccw-walk _ 𝓕 a b ·w q ]               ∎
  }
```

Corollary C.2. Under the same assumptions as in Lemma C.1, if p and q are walks in the symmetrisation of G such that $p \sim_{\mathcal{M}} q$, then any normal form of p is equal to any normal form of q in the geometric realisation.

Proof. Let $P(x, y, w)$ be the collection of normal forms for the walk w in the symmetrisation of G from x to y, defined as follows.

$$P(w) :\equiv \sum_{(r : W_{U(G)}(x,y))} (w \sim_{\mathcal{M}} r) \times \mathrm{Normal}(r).$$

The proof of this lemma follows, almost inmediately, from Lemma C.1, as in the following Agda proof.

```
P : ∀ {x y} → Walk (U G) x y → Type (lsuc ℓ)
P {x}{y} w = Σ[ r ∶ Walk (U G) x y ] (w ~⟨ 𝓜 ⟩~ r) × Normal r

corollary₁ : ∀ {x y} (p q : Walk (U G) x y)
  → ((nf-p , _) : P p) → ((nf-q , _) : P q) → nf-p ≡ nf-q → w p ≡ w q
corollary₁ p q (nf-p , (p~nf-p , _)) (nf-q , (q~nf-q , _)) nf-p≡nf-q = begin
  w[ p ] ≡⟨ walk-homotopy-gives-homotopy p~nf-p ⟩
  w[ nf-p ] ≡⟨ cong w nf-p≡nf-q ⟩
  w[ nf-q ] ≡⟨ ! walk-homotopy-gives-homotopy q~nf-q ⟩
  w[ q ] ∎
```

□

Corollary C.3. Let G be a graph with discrete set of nodes and $\mathcal{M}$ be a spherical map for G. Then, any pair of walks p and q in the symmetrisation of G are merely equal.

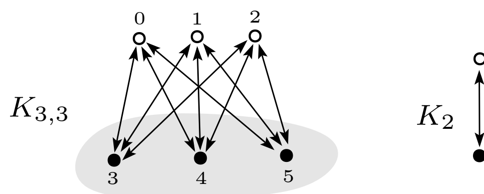
Proof. Since the graph has a discrete set of nodes, by Corollary 5.49, we can freely use the most general definition of spherical maps to obtain the mere existence of a walk homotopy for any pair of walks (Prieto-Cubides 2022). The conclusion then follows by applying the elimination principle of the propositional truncation to Lemma C.1 and the walk homotopy obtained earlier. □

```
corollary₂ : isSphericalMap G 𝓜 → ∀ {x y} → (p q : Walk (U G) x y) → ∥ w p ≡ w q ∥
corollary₂ 𝓜-is-spherical p q = trunc-elim trunc-is-prop
  (λ p~q → ∣ walk-homotopy-gives-homotopy p~q ∣) |p~q|
  where
  |p~q| : ∥ p ~⟨ 𝓜 ⟩~ q ∥
  |p~q| = lemap (spherical-equiv G (_≟Node_) 𝓜) 𝓜-is-spherical _ _ p q
```

D
Other Constructions

The complete bipartite graph $K_{3,3}$ is a well-known example of the smallest non-planar graph. It comprises six nodes, evenly divided into two independent sets. Herein, we define $K_{3,3}$, its automorphism group $\mathrm{Aut}(K_{3,3})$, and one of its maps into the torus.

A graph G with an n-colouring is described by a homomorphism of type $\mathsf{Hom}(G, K_n)$. Each node in K_n signifies a unique colour for the nodes in G. If G has an n-colouring, we denote G as n-colourable or n-partite. Consequently, a *bipartite* graph possesses a 2-colouring. The graph $K_{3,3}$ is such a bipartite complete graph with six nodes. We illustrate this graph in Appendix D, each arrow symbolises a pair of edges, one in each direction.



The collection of all n-colourings of a graph forms a set by Lemma 3.4, and the collection of n-partite graphs forms a 1-groupoid. Since there are some n-partite graphs that are equal up to isomorphism, we have the following distinction. Two graph colourings of G, namely, $f, g : \mathsf{Hom}(G, K_n)$ are *essentially* equal if a nontrivial isomorphism $\sigma : K_n \cong K_n$ exists and if the functions f and $\sigma \circ g$ are equal. The type of essentially equal colourings of a graph G is (D.0−1).

$$\text{EssentiallyPartite}(n, G) :\equiv \sum_{(A\,:\,\text{Graph})} \text{Hom}(G, A) \times \|A \cong K_n\|. \tag{D.0–1}$$

Example D.1. We compute the identity type of the essentially equal colourings of the path graph P_3 in Calculation (D.0–2). As we will see, there can only be two graph homomorphisms from P_3 to K_2, namely φ_0 and φ_1 as in Figure D.1. Let c_1 and c_2 be of type $\text{EssentiallyPartite}(2, P_3)$.

$$(c_1 = c_2) \simeq ((K_2, \varphi_0, !) = (K_2, \varphi_1, !)) \tag{D.0–2a}$$

$$\simeq \sum_{(\tau : K_2 = K_2)} \text{tr}^{\lambda X.\text{Hom}(P_3, X)}(\tau, \varphi_0) = \varphi_1 \tag{D.0–2b}$$

$$\simeq \sum_{(\tau : K_2 = K_2)} \text{coe}\,(\tau) \circ \varphi_0 = \varphi_1. \tag{D.0–2c}$$

In Equivalence (D.0–2b), the equality $\tau : K_2 = K_2$ is one of two alternatives: the trivial path or the path from the equivalence that swaps the only two nodes in K_2. Only the latter possibility, the equation, $\text{coe}\,(\tau) \circ \varphi_0 = \varphi_1$ can hold.

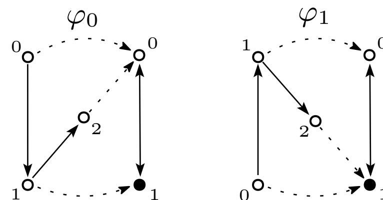


Figure D.1: Two graph homomorphisms φ_0 and φ_1 from P_3 to K_2. The dashed arrows represent how φ_0 and φ_1 map the nodes of P_3 into K_2. We represent the colours of the 2-coloring of P_3 by the nodes black and white in K_2.

Thus, $\text{Aut}(K_{3,3})$ can be identified as the subgroup $\mathbb{Z}_2 \times S_3 \times S_3$ in S_6. This is due to the nodes of $K_{3,3}$ being partitionable into two independent sets of three, which can be permuted independently. Furthermore, these two partitions are interchangeable.

We now outline a graph map, $\mathcal{M}$, for $K_{3,3}$ as per (D.0–3), along with its faces F_1, F_2, and F_3. Although surface holds no significance in our context, as there is no a type that define such a concept, the map $\mathcal{M}$ described above would correspond to the torus in a traditional setting. This can be depicted using the polygonal schema shown in Figure D.2.

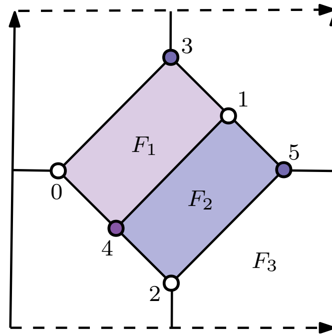


Figure D.2: A map for $K_{3,3}$ in the surface of the torus.

$$
\begin{aligned}
\mathscr{M} &:\equiv (0 \mapsto ((03)\,(04)\,(05)), 1 \mapsto ((13)\,(15)\,(14)),\\
&\quad 2 \mapsto ((24)\,(25)\,(23)), 3 \mapsto ((32)\,(31)\,(30)),\\
&\quad 4 \mapsto ((40)\,(41)\,(42)), 5 \mapsto ((51)\,(50)\,(52))).\\
F_1 &:\equiv ((30)\,(04)\,(41)\,(13)).\\
F_2 &:\equiv ((14)\,(42)\,(25)\,(51)).\\
F_3 &:\equiv ((03)\,(32)\,(24)\,(40)\,(05)\,(52)\,(23)\,(31)\,(15)\,(50)).
\end{aligned}
\tag{D.0–3}
$$

Bibliography

Ahrens, Benedikt and Paige Randall North (2019). "Univalent Foundations and the Equivalence Principle." In: *Reflections on the Foundations of Mathematics: Univalent Foundations, Set Theory and General Thoughts*, pp. 137–150. URL: https://doi.org/10.1007/978-3-030-15655-8_6 (cit. on pp. 20, 49).

Ahrens, Benedikt, Paige Randall North, Michael Shulman, et al. (2021). The Univalence Principle. URL: https://arxiv.org/abs/2102.06275 (cit. on p. 17).

Ahrens, Benedikt, Paige Randall North, Michael Shulman, and Dimitris Tsementzis (2020). A Higher Structure Identity Principle. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. URL: https://doi.org/10.1145/3373718.3394755 (cit. on p. 49).

Avigad, Jeremy and John Harrison (2014). Formally Verified Mathematics. Communications of the ACM 57.4, pp. 66–75 (cit. on p. 30).

Awodey, Steve (2012). Type Theory and Homotopy. In: Epistemology versus Ontology, pp. 183–201. URL: https://doi.org/10.1007/978-94-007-4435-6_9 (cit. on p. 33).

— (2018). Univalence as a principle of logic. Indagationes Mathematicae 29.6, pp. 1497–1510. URL: https://doi.org/10.1016/j.indag.2018.01.011 (cit. on pp. 17, 33).

Awodey, Steve and Michael A. Warren (2009). Homotopy theoretic models of identity types. Mathematical Proceedings of the Cambridge Philosophical Society 146.1, pp. 45–55. URL: https://doi.org/10.1017/s0305004108001783 (cit. on p. 17).

Baez, John C., Alexander E. Hoffnung, and Christopher D. Walker (2009-08). Higher-Dimensional Algebra VII: Groupoidification. Theory and Applications of Categories 24, pp. 489–553. URL: http://arxiv.org/abs/0908.4305 (cit. on p. 34).

Bagaria, Joan (2021). Set Theory. In: The Stanford Encyclopedia of Philosophy. Winter 2021. URL: https://plato.stanford.edu/archives/win2021/entries/set-theory/ (cit. on p. 2).

Bang-Jensen, Jørgen and Gregory Z. Gutin (2009). Digraphs. Springer London. URL: https://doi.org/10.1007/978-1-84800-998-1 (cit. on p. 108).

Barendregt, Henk (1997). The Impact of the Lambda Calculus in Logic and Computer Science. Bulletin of Symbolic Logic 3.2, pp. 181–215. URL: https://doi.org/10.2307/421013 (cit. on p. 9).

Barendregt, Henk, Wil Dekkers, and Richard Statman (2013). Lambda Calculus with Types. Cambridge University Press. URL: https://doi.org/10.1017/CBO9781139032636 (cit. on p. 4).

Bauer, Andrej (2017). Five stages of accepting constructive mathematics. Bulletin of the American Mathematical Society 54.3, pp. 481–498. URL: https://doi.org/10.1090/bull/1556 (cit. on p. 4).

Bauer, Andrej, J. Gross, Peter LeFanu Lumsdaine, et al. (2017). The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In: Proceedings of the 6th ACM SIGPLAN Conference on

Certified Programs and Proofs, pp. 164–172. URL: https://doi.org/10.1145/3018610.3018615 (cit. on p. 41).

Bauer, Gertrud and Tobias Nipkow (2002). The 5 Colour Theorem in Isabelle/Isar. In: Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings, pp. 67–82. URL: https://doi.org/10.1007/3-540-45685-6_6 (cit. on p. 31).

Bauer, Gertrud Josefine (2005). Formalizing Plane Graph Theory: Towards a Formalized Proof of the Kepler Conjecture. PhD thesis. Technische Universität München. URL: https://mediatum.ub.tum.de/doc/601794/document.pdf (cit. on pp. 31, 121).

Baur, Melanie (2012). Combinatorial Concepts and Algorithms for Drawing Planar Graphs. PhD thesis. Universität Konstanz. URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-202281 (cit. on p. 105).

Bezem, Marc, Ulrik Buchholtz, Pierre Cagne, et al. (19, 2022). Symmetry. https://github.com/UniMath/SymmetryBook. Commit: 870cb20. URL: https://gitub.com/UniMath/SymmetryBook (cit. on pp. 19, 44).

Bezem, Marc, Thierry Coquand, and Simon Huber (2017). The univalence axiom in cubical sets. URL: https://doi.org/10.1007/s10817-018-9472-6 (cit. on p. 18).

Bishop, Errett and Douglas Bridges (1985). Constructive Analysis. Springer Berlin Heidelberg. URL: https://doi.org/10.1007/978-3-642-61667-9 (cit. on p. 3).

Brady, Edwin (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23 (05), pp. 552–593. URL: https://journals.cambridge.org/article_S095679681300018X (cit. on p. 29).

Bruijn, N. G. de (1983). AUTOMATH, a Language for Mathematics. In: Automation of Reasoning, pp. 159–200. URL: https://doi.org/10.1007/978-3-642-81955-1%5F11 (cit. on p. 13).

Brunerie, Guillaume, Kuen-Bang Hou (Favonia), Evan Cavallo, et al. (n.d.). Homotopy Type Theory in Agda. URL: https://github.com/HoTT/HoTT-Agda (cit. on p. 173).

Chih, Tien and Laura Scull (2020). A homotopy category for graphs. Journal of Algebraic Combinatorics. URL: https://doi.org/10.1007/s10801-020-00960-5 (cit. on p. 101).

Church, Alonzo (1932). A Set of Postulates for the Foundation of Logic. The Annals of Mathematics 33.2, p. 346. URL: https://doi.org/10.2307/1968337 (cit. on p. 9).

— (1940). A formulation of the simple theory of types. Journal of Symbolic Logic 5.2, pp. 56–68. URL: https://doi.org/10.2307/2266170 (cit. on pp. 5, 13).

Cockx, Jesper, Dominique Devriese, and Frank Piessens (2016). Eliminating dependent pattern matching without K. Journal of Functional Programming 26, e16. URL: https://doi.org/10.1017/s0956796816000174 (cit. on p. 131).

Cohen, Cyril, Thierry Coquand, Simon Huber, et al. (2017). Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. FLAP 4.10, pp. 3127–3170. URL: http://collegepublications.co.uk/ifcolog/?00019 (cit. on p. 18).

Coquand, Thierry and Nils Anders Danielsson (2013). Isomorphism is equality. Indagationes Mathematicae 24.4, pp. 1105–1120. URL: https://doi.org/10.1016/j.indag.2013.09.002 (cit. on p. 49).

Coquand, Thierry, Simon Huber, and Anders Mörtberg (2018). On Higher Inductive Types in Cubical Type Theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in

Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, pp. 255–264. URL: https://doi.org/10.1145/3209108.3209197 (cit. on p. 18).

Coquand, Thierry and Gérard Huet (1988). The calculus of constructions. Information and Computation 76.2-3, pp. 95–120. URL: https://doi.org/10.1016/0890-5401(88)90005-3 (cit. on p. 4).

Diestel, Reinhard (2012). Graph Theory, 4th Edition. Vol. 173. Springer. URL: https://doi.org/10.1007/978-3-662-53622-3 (cit. on pp. 83, 105, 118, 120).

Diestel, Reinhard and Daniela Kühn (2004). Topological paths, cycles and spanning trees in infinite graphs. European Journal of Combinatorics 25.6, pp. 835–862. URL: https://doi.org/10.1016/j.ejc.2003.01.002 (cit. on p. 148).

Doczkal, Christian (2021). A Variant of Wagner's Theorem Based on Combinatorial Hypermaps. working paper or preprint. URL: https://hal.archives-ouvertes.fr/hal-03142192 (cit. on pp. 30, 31).

Doczkal, Christian and Damien Pous (2020). Graph Theory in Coq: Minors, Treewidth, and Isomorphisms. J. Autom. Reason. 64.5, pp. 795–825. URL: https://doi.org/10.1007/s10817-020-09543-2 (cit. on p. 30).

Dubois, Catherine, Alain Giorgetti, and Richard Genestier (2016). Tests and Proofs for Enumerative Combinatorics. In: Tests and Proofs - 10th International Conference, TAPSTAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings. Vol. 9762, pp. 57–75. URL: https://doi.org/10.1007/978-3-319-41135-4%5C_4 (cit. on p. 30).

Dufourd, Jean François (2009). An intuitionistic proof of a discrete form of the Jordan curve theorem formalised in Coq with combinatorial hypermaps. Journal of Automated Reasoning 43.1, pp. 19–51. URL: https://doi.org/10.1007/s10817-009-9117-x (cit. on pp. 30, 31).

Dufourd, Jean-François and François Puitg (2000). Functional specification and prototyping with oriented combinatorial maps. Computational Geometry 16.2, pp. 129–156. URL: https://www.sciencedirect.com/science/article/pii/S0925772100000043 (cit. on pp. 30, 31).

Ellis-Monaghan, Joanna and Iain Moffatt (2013). Graphs on Surfaces: Dualities, Polynomials, and Knots. 1st ed. Springer (cit. on p. 56).

Escardó, Martín (2004). Synthetic Topology. Electronic Notes in Theoretical Computer Science 87, pp. 21–156. URL: https://doi.org/10.1016/j.entcs.2004.09.017 (cit. on p. 128).

— (2018). A self-contained, brief and complete formulation of Voevodsky's Univalence Axiom. URL: https://arxiv.org/abs/1803.02294 (cit. on p. 33).

— (2019). Introduction to Univalent Foundations of Mathematics with Agda. URL: http://arxiv.org/abs/1911.00580 (cit. on pp. 19, 50).

Gentzen, Gerhard (1964). Investigations Into Logical Deduction. American Philosophical Quarterly 1.4, pp. 288–306 (cit. on p. 5).

Gonthier, Georges (2008). The Four Colour Theorem: Engineering of a Formal Proof. In: Computer Mathematics, pp. 333–333. URL: https://doi.org/10.1007/978-3-540-87827-8_28 (cit. on pp. 30, 105, 121, 128, 168).

Grayson, Daniel (2018). An introduction to univalent foundations for mathematicians. Bulletin of the American Mathematical Society 55.4, pp. 427–450. URL: https://doi.org/10.1090/bull/1616 (cit. on p. 19).

Grigor'yan, Alexander, Yong Lin, Yuri Muranov, et al. (2014). Homotopy theory for digraphs. URL: http://arxiv.org/abs/1407.0234 (cit. on p. 101).

Gross, J., Jay Yellen, and Mark Anderson (2018). Graph Theory and Its Applications. Chapman and Hall/CRC. URL: https://doi.org/10.1201/9780429425134 (cit. on pp. 108, 120).

Gross and Tucker (1987). Topological graph theory. A Wiley-Interscience Publication. John Wiley & Sons Inc., pp. xvi+351 (cit. on pp. 24, 60, 104).

Hales, Thomas, Mark Adams, Gertrud Bauer, et al. (2017). A Formal Proof Of The Kepler Conjecture. Forum of Mathematics, Pi 5, e2. URL: https://doi.org/10.1017/fmp.2017.1 (cit. on p. 30).

Hofmann, Martin and Thomas Streicher (1998). The groupoid interpretation of type theory. In: Twenty-five years of constructive type theory (Venice, 1995). Vol. 36, pp. 83–111 (cit. on p. 17).

Howard, William Alvin (1980). The Formulae-as-Types Notion of Construction. In: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism (cit. on p. 9).

Kamareddine, Fairouz, Twan Laan, and Rob Nederpelt (2005). A Modern Perspective on Type Theory: From its Origins Until Today. Kluwer Academic Publishers. URL: https://doi.org/10.1007%2F1-4020-2335-9 (cit. on p. 5).

Kokke, Wen, Jeremy G. Siek, and Philip Wadler (2020). Programming language foundations in Agda. Sci. Comput. Program. 194, p. 102440. URL: https://doi.org/10.1016/j.scico.2020.102440 (cit. on pp. 90, 91, 128).

Kraus, Nicolai and Jakob von Raumer (2020). Coherence via Well-Foundedness. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. URL: https://doi.org/10.1145/3373718.3394800 (cit. on pp. 101, 168).

— (2021). A Rewriting Coherence Theorem with Applications in Homotopy Type Theory (cit. on pp. 101, 102, 168).

Licata, Daniel R. and Guillaume Brunerie (2015). A Cubical Approach to Synthetic Homotopy Theory. URL: https://doi.org/10.1109/lics.2015.19 (cit. on p. 173).

Linsky, Bernard and Andrew David Irvine (2022). Principia Mathematica. In: The Stanford Encyclopedia of Philosophy. Spring 2022. URL: https://plato.stanford.edu/archives/spr2022/entries/principia-mathematica/ (cit. on p. 5).

Lucas, Maxime (2019). An implementation of polygraphs. working paper or preprint. URL: https://hal.archives-ouvertes.fr/hal-02385110 (cit. on p. 102).

— (2020). Abstract rewriting internalized (cit. on p. 102).

MacLane, Saunders (1937). A combinatorial condition for planar graphs (cit. on p. 105).

Martin-Löf, Per (1975). An Intuitionistic Theory of Types: Predicative Part. In: Logic Colloquium '73, Proceedings of the Logic Colloquium, pp. 73–118. URL: https://doi.org/10.1016/s0049-237x(08)71945-1 (cit. on pp. 4, 5, 11).

McBride, Conor (n.d.). A polynomial testing principle. URL: https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf (cit. on p. 90).

Mohar, Bojan (1988). Embeddings of infinite graphs. Journal of Combinatorial Theory, Series B 44.1, pp. 29–43. URL: https://doi.org/10.1016/0095-8956(88)90094-9 (cit. on p. 60).

Mörtberg, Anders, Vezzosi Andrea, and Cavallo Evan (2021). A Standard library for Cubical Agda. URL: https://github.com/agda/cubical (cit. on p. 168).

Mörtberg, Anders and Loïc Pujet (2020). Cubical Synthetic Homotopy Theory. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 158–171. URL: https://doi.org/10.1145/3372885.3373825 (cit. on p. 128).

Moura, Leonardo de, Soonho Kong, Jeremy Avigad, et al. (2015). The Lean theorem prover (System Description). In: International Conference on Automated Deduction. Springer, pp. 378–388. URL: https://doi.org/10.1007/978-3-319-21401-6_26 (cit. on pp. 4, 29, 30).

Nordström, Bengt (1988). Terminating general recursion. Bit 28.3, pp. 605–619. URL: 10.1007/bf01941137 (cit. on p. 80).

Nordström, Bengt, Kent Petersson, and Jan M. Smith (1990). Programming in Martin-Lo¨f's Type Theory: An Introduction. Clarendon Press (cit. on pp. 4, 11, 16).

Norrell, Ulf (2007). Towards a practical programming language based on dependent type theory. PhD thesis. Chalmers University of Technology. URL: https://research.chalmers.se/en/publication/46311 (cit. on p. 34).

Noschinski, Lars (2014). A Graph Library for Isabelle. Mathematics in Computer Science 9.1, pp. 23–39. URL: https://doi.org/10.1007/s11786-014-0183-z (cit. on p. 30).

— (2015). Formalizing Graph Theory and Planarity Certificates. PhD thesis. Technischen Universität München. URL: https://d-nb.info/1104933624/34 (cit. on pp. 19, 30).

Petrakis, Iosif (2019). Dependent Sums and Dependent Products in Bishop's Set Theory. In: 24th International Conference on Types for Proofs and Programs (TYPES 2018). Vol. 130, 3:1–3:21. URL: http://drops.dagstuhl.de/opus/volltexte/2019/11407 (cit. on p. 3).

Prawitz, Dag (1967). Natural deduction. A proof-theoretical study. Journal of Symbolic Logic 32.2, pp. 255–256. URL: https://doi.org/10.2307/2271676 (cit. on p. 13).

Prieto-Cubides, Jonathan (2022). On Homotopy of Walks and Spherical Maps in Homotopy Type Theory. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 338–351. URL: https://doi.org/10.1145/3497775.3503671 (cit. on pp. 114, 122, 150, 168, 180).

— (2023). *Artefact for the manuscript "Investigations in Graph-theoretical Constructions in Homotopy Type Theory"*. URL: https://jonaprieto.github.io/synthetic-graph-theory (cit. on p. 102).

Prieto-Cubides, Jonathan and Håkon Robbestad Gylterud (2022). On Planarity of Graphs in Homotopy Type Theory. Submitted, *Mathematical Structures in Computer Science*. URL: https://arxiv.org/abs/2112.06633 (cit. on p. 94).

Prieto-Cubides, Jonathan and Håkon Robbstand Gylterud (2019). Planar graphs in HoTT. 25th International Conference on Types for Proofs and Programs, TYPES. URL: http://www.ii.uib.no/~bezem/abstracts/TYPES%5F2019%5Fpaper%5F37 (cit. on p. 128).

Rahman, Md. Saidur (2017). "Planar Graphs." In: *Basic Graph Theory*, pp. 77–89. URL: https://doi.org/10.1007/978-3-319-49475-3_6 (cit. on p. 105).

Reck, Erich and Georg Schiemer (2020). Structuralism in the Philosophy of Mathematics. In: The Stanford Encyclopedia of Philosophy. Spring 2020. URL: https://plato.stanford.edu/archives/spr2020/entries/structuralism-mathematics/ (cit. on p. 1).

Rijke, Egbert, Elisabeth Bonnevier, Jonathan Prieto-Cubides, Fredrik Bakke, et al. (2023). *Univalent mathematics in Agda*. URL: https://github.com/UniMath/agda-unimath/ (cit. on pp. 30, 152, 168).

Schütte, K. (1972). The collected papers of Gerhard Gentzen. 37.4, pp. 752–753. URL: https://doi.org/10.2307/2272429 (cit. on p. 13).

Stahl, Saul (1978). The embeddings of a graph–A survey. Journal of Graph Theory 2.4, pp. 275–298. URL: https://doi.org/10.1002/jgt.3190020402 (cit. on pp. 56, 60).

Suppes, Patrick (1972). Axiomatic set theory. Dover publications (cit. on p. 12).

Swan, Andrew W (2022). On the Nielsen-Schreier Theorem in Homotopy Type Theory. Logical Methods in Computer Science Volume 18, Issue 1. URL: https://doi.org/10.46298%2Flmcs-18%281%3A18%292022 (cit. on pp. 128, 148, 149, 161, 162, 167).

The Agda Development Team (2023). Agda 2.6.3 documentation. URL: https://agda.readthedocs.io/en/v2.6.3/ (cit. on pp. 4, 29, 131, 168).

The Coq Development Team (2021). The Coq Proof Assistant. en. URL: https://zenodo.org/record/4501022 (cit. on pp. 4, 29).

Troelstra, A. S. (2011). History of constructivism in the 20th century. In: Set Theory, Arithmetic, and Foundations of Mathematics, pp. 150–179. URL: https://doi.org/10.1017/cbo9780511910616.009 (cit. on pp. 1, 3, 5).

Univalent Foundations Program, The (2013). Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book. URL: https://homotopytypetheory.org/book (cit. on pp. 13, 18, 33, 34, 36, 44, 48, 80, 105, 123, 124, 168, 175).

Vezzosi, Andrea, Anders Mörtberg, and Andreas Abel (2021). Cubical Agda: A dependently typed programming language with univalence and higher inductive types. Journal of Functional Programming 31, e8. DOI: 10.1017/s0956796821000034 (cit. on pp. 18, 168).

Voevodsky, Vladimir (2010). The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010). URL: https://arxiv.org/abs/1402.5556 (cit. on pp. 17, 33).

Whitney, Hassler (1932). Non-separable and planar graphs. Transactions of the American Mathematical Society 34.2, pp. 339–339. URL: https://doi.org/10.1090/s0002-9947-1932-1501641-2 (cit. on p. 108).

Yamamoto, Mitsuharu, Shin-ya Nishizaki, Masami Hagiya, et al. (1995). Formalization of Planar Graphs. In: Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings. Vol. 971, pp. 369–384. URL: https://doi.org/10.1007/3-540-60275-5_77 (cit. on pp. 26, 31, 119–122).

Yorgey, Brent Abraham (2014). Combinatorial Species And Labelled Structures. PhD thesis. University of Pennsylvania, p. 206. URL: https://www.cis.upenn.edu/~sweirich/papers/yorgey-thesis.pdf (cit. on p. 34).