Computing Spaces in Type Theory



Maximilian Doré
Wolfson College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Michaelmas 2023

Acknowledgements

I am deeply grateful to my supervisors Samson Abramsky and Sam Staton for taking me in as a doctoral student—Samson for giving me the chance to return to computer science after having strayed in philosophy, and Sam for stepping in without hesitation two years into my DPhil. Samson's suggestion to look at formalising discrete Morse theory has put this research project on its tracks. It is very special for a researcher of Samson's stature to be as charitable towards other people's ideas, a character trait which allows his students to develop their own viewpoints. Sam has been deeply inspiring with his energy and enthusiasm, it is exciting to see how he brings into existence a school of thought.

Being isolated during the pandemic has made it apparent that science is fundamentally a social endeavour. Only once I could return to meet people in person, at the department and at workshops, was pursuing research meaningful. My heartfelt thanks go out to the fellow researchers around me, first and foremost to Evan Cavallo and Anders Mörtberg for supporting my idea of a cube solver and providing crucial ideas and guidance. Without them, the second half of this thesis could not have come about. Thanks also to the patient audiences at talks I gave and to several anonymous reviewers, your feedback was very helpful.

I am indebted to Thorsten Altenkirch, Jeremy Gibbons, Sean Moss and Vidit Nanda, who have assessed this manuscript and the different milestones leading up to it. The ensuing discussions were extremely illuminating and shaped this thesis. I am very grateful to all who have supported me in my academic upbringing, in particular Krysia Broda and Hannes Leitgeb.

I am lucky that I can call some of my fellow researchers my friends; spending time with Swaraj Dash, Axel Ljungström, Maximilian Pflüger, Amy Searle, Nihil Shah, Samuel Speight, Fabian Zaiser, Max Zeuner, and

many others has made this journey a fun one. Many thanks also to Swaraj for proof-reading my thesis.

I am grateful for the funding that has allowed me to purse research—the Scatcherd European Scholarship and the Engineering and Physical Sciences Research Council for funding my DPhil, and the COST Action EuroProofNet (CA20111) and Wolfson College for generously funding my academic travels.

Abstract

In this thesis, we develop multiple case studies computing with and in cubical type theory.

We begin by setting up a framework to compute with sets in the theorem prover Cubical Agda. Using the example of the powerset, we show how both mathematical *definitions* and *programs* can be expressed and related with one another in type theory. We show how a functional program implementing the powerset can be shown to satisfy the mathematical definition of the powerset, thereby allowing for using interchangeably the program and its mathematical specification.

We then develop in Cubical Agda *discrete Morse theory* for directed graphs, a central method in the field of topological data analysis. We show how Morse matchings, gradient paths and Morse reducts can be expressed in dependent type theory. Using the novel reasoning principles introduced by Cubical Agda, we prove that a graph is homotopy equivalent to its Morse reduct. This allows us to quickly establish topological properties of many graphs, which we demonstrate with some examples. Finally, we show how the Morse reduct can be computed explicitly, using the previously developed framework for computing with sets. This formalisation provides a crucial stepping stone towards a verified tool for topological data analysis.

Afterwards, we study reasoning principles of cubical type theory from the point of view of automated reasoning, in particular proof synthesis. We develop a term language which allows us to capture *boundary problems*, a new kind of proof obligation that cubical type theory poses to its users. These boundary problems can be solved using either *contortions* or *Kan filling* (which is called homogeneous composition in Cubical Agda). We develop an algorithm that can derive many instances of Kan filling automatically. The algorithm works for a wide range of cubical type theories,

which vary in the class of contortions they support; in particular for those based on *cartesian*, *disjunctive*, *Dedekind* and *De Morgan* cubes.

We then refine our approach for Dedekind cubes. Contortions for Dedekind cubes can be represented as certain poset maps, we use this perspective to obtain a memory-efficient and fast solver for boundary problems in Dedekind cubical type theory. We introduce an implementation of our approach in Haskell which can automatically derive many statements in cubical type theory, notably the Eckmann-Hilton argument. The solver can be used for all variations of cubical type theory that we have discussed, we compare the different theories with respect to how fast our solve can resolve boundary problems in each respective theory. We find that Dedekind cubical type theory can be solved well in practice.

Our case studies highlight how we can compute spaces in type theory—both in the formalisation of a particular mathematical domain such as topological data analysis; and for the automation of cubical type theory, which is one proposal for a logic of proof-relevant equality.

Contents

1 Introduction **1**

2 Computing with sets **9**

2.1 Sets in dependent type theory 11

2.1.1 Background on Cubical Agda 12

2.1.2 Sets as ordered lists 17

2.1.3 Membership and subsets 22

2.2 Computing the powerset 25

2.2.1 The powerset program in Agda 25

2.2.2 Soundness 27

2.2.3 Completeness 29

2.2.4 Showing that the powerset is a set 30

2.2.5 A correct by construction powerset 31

2.3 Discussion 32

2.3.1 Finite sets in homotopy type theory 32

2.3.2 Powersets in Lean 33

3 Computing smaller spaces **35**

3.1 Background on discrete Morse theory 38

3.2 Background on higher inductive types 40

3.3 Realising relations and graphs 43

3.3.1 Directed graphs 45

3.4 Morse matchings on graphs 47

3.5 Graphs and their reducts are isomorphic 51

3.5.1 Mapping between the graph and its reduct 51

3.5.2 ι is a right inverse of ϕ 53

3.5.3 ι is a left inverse of ϕ 57

3.6 The Morse theorem for graphs 59

3.7 Computing the reduct explicitly . . . 62
3.8 Towards verified topological data analysis . . . 64
3.8.1 Implementing a tool for topological data analysis . . . 65
3.8.2 Using more nuanced complexes . . . 66

4 Automating reasoning in cubical type theory 69
4.1 A syntax for cubes . . . 71
4.1.1 Contortion theory . . . 73
4.1.2 Term theory . . . 79
4.1.3 The Kan term theory . . . 83
4.2 Boundary problems . . . 87
4.3 Finding terms in the Kan theory . . . 88
4.3.1 Finding Kan filling terms as open cubes . . . 89
4.3.2 Fitting Kan filling terms into sides of an open cube . . . 92
4.3.3 An algorithm for finding Kan terms . . . 97

5 Dedekind cubes & a practical cube solver 102
5.1 An alternative characterisation of Dedekind contortions . . . 103
5.1.1 Dedekind contortions as poset maps . . . 104
5.1.2 Finding Dedekind contortions with potential poset maps . . . 108
5.1.3 Potential poset maps in the Kan solver . . . 113
5.2 A practical cube solver . . . 115
5.2.1 An experimental solver . . . 115
5.2.2 Benchmarking the solver . . . 118

6 Conclusions 121

Index of Agda symbols 126

Bibliography 129

Chapter 1

Introduction

From its very conception, dependent type theory (DTT) had a dual character: it was intended as a logic able to express all of mathematics, and as a programming language with a well-behaved reduction of terms to normal forms. Martin-Löf devised DTT to extend the correspondence that Curry [17] and Howard [42] found between the simply typed lambda calculus and derivations in propositional logic to predicate logic. He had the goal to develop a foundation for mathematics, while his *meaning explanations* required that all types should have computational content—the introduction rules for a type describe all terms of a type, while the computation rules ensure that any term reduces to a canonical form. In the intensional formulation of DTT [55], there was only a single introduction rule for identity types, capturing reflexivity, which made it natural to expect only a single inhabitant in any identity type. However, this principle of *uniqueness of identity proofs* (UIP) could not proved inside the theory. Martin-Löf attempted to rectify this perceived incompleteness by definitionally equating all inhabitants of the identity type, leading to an extensional version of DTT [56]. However, this additional computation rule makes type-checking undecidable, which conflicts with the type-theoretic principle that we should always be able to recognise a proof of a proposition if we are given one.

Hofmann [39] later showed that intensional DTT has models in which identity types contain multiple inhabitants; in fact, identity types can be seen to have the structure of a groupoid. It was unclear what to make of this observation until Awodey and Warren [4] and Voevodsky [79] worked out that there was even more interesting structure in the identity type: the identity type between identity proofs forms itself a groupoid, etc., leading to the notion of an ∞-groupoid. This structure is well studied in homotopy theory and higher category theory, and the connection between these areas and DTT has sparked a highly fruitful exchange of ideas epitomised with Homotopy Type Theory (HoTT) [77]. The classification of types according to

the complexity of their identity types gives rise to the stratification into *homotopy-n-types*; the use of higher-dimensional constructors in inductive types allows for defining *higher inductive types*; and the homotopy-theoretic principle that equivalent structures should be considered equal was carried over to type theory as *univalence* by Voevodsky [80] (the idea that observationally equal objects should be equated in type theory already goes back to Hofmann and Streicher [40]).

When formulating mathematical theories in HoTT instead of set theory, some constructions become more direct; redeveloping mathematics in this way is often called *synthetic*. While it is possible to develop synthetic mathematics by mimicking HoTT in intensional DTT with some axioms and tricks, the resulting theory is not computationally well-behaved. Cubical type theory (CTT) [6, 16] set out to remedy this by redeveloping type-theoretic identity based on the ideas of HoTT. Drawing inspiration from cubical sets [48], which are a combinatorial model of homotopy types of topological spaces, CTT introduces a novel *path* type with special primitive terms, thereby replacing the identity type of Martin-Löf [55]. Identifications in CTT are paths, i.e., functions out of (an abstract version of) the unit interval. A logical manifestation of *Kan composition*, a property that cubical sets possess, becomes a central equational reasoning principle. Sterling and Angiuli [73] have established that all terms in CTT reduce to a normal form, which means that CTT does indeed give computational content to higher inductive types and univalence.

In summary, CTT is a well-behaved (total) programming language, which allows us to reason directly about (homotopy types) of spaces. The goal of this thesis to emphasise these two aspects of CTT through several case studies. We develop these in the theorem prover Cubical Agda [78] which provides an implementation of CTT. The ability to compute in a logical framework allows for novel things not possible when doing mathematics informally or in a non-constructive logic. We think that formalising in a dependently typed theorem prover changes how and what kind of mathematics is developed, which we hope to highlight with this thesis.

Computation in mathematics. In mathematics, the term "computation" is used with various meanings. We will classify two different notions and illustrate each notion with some classical examples. We will then see how reasoning in a dependently typed theorem prover can be used for the respective kind of computation; introducing the different case studies carried out in this thesis along the way.

A relatively weak notion of computation commonly used by mathematicians is the following:

(1) Given some object a and relation R, construct an object b such that $R(a, b)$ holds.

Examples:

- Given a topological space X, "compute" a group G which is the n-th homotopy group of X, i.e., $R(X, G) \coloneqq \pi_n(X) \cong G$.
- Given a function $f(x)$, "compute" its integral $F(x)$, i.e., $R(f, F) \coloneqq \frac{d}{dx}F = f$.
- Given a group G, "compute" a subgroup H, i.e., $R(G, H) \coloneqq H < G$.
- Given an elliptic curve E over a field K, "compute" its solutions $E(K)$, i.e., $R(E, E(K)) \coloneqq$ for all $x \in E(K), x$ is a solution to E.

Computation in the sense of (1) is the main business of mathematicians, understanding the properties of a mathematical object by relating it to another object in a certain way.

In the context of HoTT, homotopy groups (and many other properties) of various spaces have been established. Most prominently, Brunerie [9] has proved synthetically that the 4th homotopy group of the 3-sphere, denoted S^3, is equivalent to the integers modulo some n. The reduction rules of a computationally sound type theory then reduce n to the concrete numeral for which the statement holds. Brunerie's result was proved in Cubical Agda by Ljungström and Mörtberg [53]; in their formalisation n indeed reduces to 2 (or rather -2, which is just as good). Therefore, working in type theory allows for establishing a relation such as $\pi_4(S^3) \cong \mathbb{Z}/n\mathbb{Z}$ only for some term n of the integers, the reduction rules then yield the concrete instance of n. Reasoning in a calculus with normalisation can therefore make it easier to perform a computation in the sense of (1) as some steps of the calculation might follow by reduction.

As another case study of (1) in HoTT, we will establish homotopy levels and cohomology groups for some graphs in Chapter 3. We do this by formalising discrete Morse theory [26] for graphs in Cubical Agda, which is a general recipe for reducing the size of a topological space while retaining its homotopy type. This makes it possible to easily study the topology of large and infinite graphs, contributing to the development of synthetic topology in Cubical Agda.

We now turn to a stronger notion of computation which captures most of mathematical research concerned with algorithms:

(2) Given some object a and relation R, algorithmically generate an object b such that $R(a, b)$ holds.

Examples:

- The Kenzo system [21] computes algebraic invariants of topological spaces.
- Numerical analysis tools such as MATLAB [44] compute integrals for many functions.
- Computer algebra systems such as Maple [54] compute subgroups of a group.
- Birch and Swinnerton-Dyer [7] used a computer to establish solutions to many elliptic curves, allowing them to devise their famed conjecture.

When performing computations in the sense of (2), one uses a computer to mechanise calculations that otherwise would have to be done manually. Much research is going into devising algorithms which derive invariants of mathematical objects, some are based on heuristics to ease the computational cost of these automatic calculations. This, and the fact that both the algorithms and their implementations might contain errors, means that the output of tools computing invariants cannot always be trusted. As a result, it is often necessary to check the outputs of these tools by performing manual "computations" in the sense of (1) to establish that the suggested object does indeed stand in the presumed relation to the original object.

Much research in computer science is devoted to the verification of algorithms and systems, it is hence natural to verify the algorithms and tools which mathematicians use to carry out computations in the sense of (2). There is a fruitful interplay between interactive and automated theorem proving and automated algebra systems, see for instance Homann and Calmet [41], Adams et al. [1] and Grégoire, Pottier, and Théry [29]; we will primarily be interested in one particular paradigm for developing correct code: Dijkstra [18] envisioned that programs could be developed in way that they were *correct by construction*, this idea was later continued by Gries [30]. The central idea of this paradigm is to develop an algorithm by turning its specification step by step into an executable sequence of operations. If all these steps are sound transformations, correctness of the implementation follows immediately. In type theory, the role of the specification is taken by a type. If we can construct a term of that type, the type-checker ensures that we have indeed produced something satisfying the specification—we thereby get a certificate that our implementation is correct. This provides a very strong manifestation of Dijkstra's idea, and we think that mathematical algorithms are a natural application for this paradigm.

We will discuss three different case studies developing an example of (2) in a correctness-by-construction fashion in type theory. The first two case studies are

instances of the following general pattern. When defining a function $f\colon A \to B$, a mathematician usually describes $f(a)$ with an abstract definition, say $R(a, f(a))$, which specifies the relation between input and output. The human mathematician then has to envision what $f(a)$ precisely looks like for a concrete a. In type theory, we can compute $f(a)$ explicitly, while also verifying that $R(a, f(a))$ does indeed hold. We thereby have a strong manifestation of the correct by construction paradigm applied to mathematics.

In Chapter 2, we will develop as the first case study the powerset function, which takes a set S to a family of sets $\mathcal{P}(S)$. This family is usually only given in terms of its specification, namely, $R(S, \mathcal{P}(S))$ states that $\mathcal{P}(S)$ contains all the subsets of S. In a programming language, we can actually compute the powerset of a finite set S, which means we can list the family of sets $\mathcal{P}(S)$ explicitly. In DTT, we can implement functional programs and use normalisation to compute the family of sets $\mathcal{P}(S)$, while we can also prove that this object satisfies the usual specification, i.e., that $\mathcal{P}(S)$ contains precisely the subsets of S.

The second case study discussed in Chapter 3 provides another instance of the pattern sketched above. Discrete Morse theory is about constructing a function f from a larger complex C to a smaller complex, preserving relevant topological properties. In this case, we will turn a graph into a smaller complex which is shown to have the same homotopy type as the graph—this provides the specification of this function, i.e., $R(C, f(C))$ states that $f(C)$ is homotopy equivalent to C. We can directly characterise $f(C)$ as a certain dependent type. Using the framework developed in Chapter 2, we can moreover explicitly output the cells of the smaller complex, thereby giving rise to an automatic procedure to compute the object $f(C)$. This case study hence is, quite literally, about computing spaces—namely Morse reducts of graphs—in type theory. We use the internal notion of equality offered by CTT in our formalisation, which is thought of to coincide with homotopy equivalence and thereby gives us a direct way to reason about homotopy types. CTT is therefore especially well-suited to implement and verify tools concerned with topological spaces. The results in Chapter 3 are intended as a stepping stone for a verified tool for *topological data analysis*, which aims to give topological meaning to datasets; and for implementing tools similar to Kenzo [21] fully inside a theorem prover based on type theory.

The third case study presented in Chapters 4 and 5 is concerned with the following problem: given some boundary description ϕ in a cubical set, can we find a cell t which has this boundary? This problem provides another instance of (2), where

the relation $R(\phi, t)$ specifies that t has boundary ϕ. Since t is not unique, we are not looking to compute a function, but instead need to search for any such t. This problem is of central importance in CTT since equational proof goals manifest itself as such boundary problems. Devising algorithms which can solve it hence amounts to automating proof search in CTT. We present an implementation of our findings in Chapter 5 which makes it significantly easier to resolve equational proof goals in Cubical Agda. The results presented in Chapters 4 and 5 provide another manifestation of the title of this thesis, as we are automatically deriving cells in cubical representations of spaces.

We thereby hope to provide evidence for the usefulness of reasoning inside a theorem prover based on a constructive logic with a well-behaved notion of reduction. We can do things mathematicians using just pen and paper and imperative programming languages cannot, and can combine both programs and mathematical theory in the same language.

As a side argument, we also hope to make another point with the developments presented in Chapters 4 and 5. The practicability of a type theory depends not only on the powerfulness of its reduction rules, but also on how well one can automatically search for terms in that theory. CTT as an attempt to construct a higher-dimensional type theory is based on cubical sets, which are a relatively weak model of homotopy types, while globular approaches such as homotopy.io [33] rest on stricter models, i.e., associative n-categories [20]. In a globular theory, some equalities hold by reduction while they have to be shown explicitly in CTT. However, most of such proofs can also be resolved automatically using our solver, which means that from a user perspective, there might be little practical difference between tools based on stricter or weaker models. The present thesis therefore provides an argument that we ought to study higher-dimensional type theories also from the point of view of automated reasoning to provide another dimension along which we can compare different proposals for a logic of proof-relevant equality.

Lastly, we hope to illustrate that computational mathematics is an ideal domain for correct by construction programming. Adoption of this paradigm in real-world programs has been impeded by the fact that the specification of a program is often not known from the start, making it difficult to develop large code bases with this paradigm. In computational mathematics, the specification of a program is usually understood beforehand and does not change over time, making the use of this programming paradigm efficacious.

Structure of this thesis. In Chapter 2, we introduce (Cubical) Agda alongside a case study to compute the powerset of a set in a way that we can show that it satisfies the usual definition of the powerset. We use the notion of a listable type to allow for inspecting and computing with concrete representations of types, and compare our approach to formalisations in other theorem provers.

We then study higher-dimensional types in Chapter 3, namely homotopy types of graphs. We formalise the correctness of discrete Morse theory for graphs, which allows for reducing the size of a graph while retaining its homotopy type. We characterise the Morse reduct in DTT and use the framework developed in Chapter 2 to inspect its cells for concrete examples.

In the formalisation in Chapter 3 we often have to use a novel reasoning principle introduced by CTT, namely Kan composition. We study this logical principle from the standpoint of automated reasoning in Chapter 4. We give a term language capturing the fragment of CTT relevant for Kan compositions, this language is applicable to several versions of CTT which vary in which kind of *contortions* they support, which are basic topological transformations. We then devise a general algorithm which searches for Kan compositions over all contortion theories studied in the literature.

In Chapter 5, we study more closely CTT based on the *Dedekind* contortions, and find this theory particularly apt for automation. We present an implementation of our methods as a practical solver which automatically solves many equational proof goals in CTT.

Contributions and collaborators. Chapters 2 and 3 are entirely my own work. I have presented preliminary work towards Chapter 3 at several workshops:

- *Towards Simplicial Complexes in Homotopy Type Theory*, Workshop on Homotopy Type Theory/Univalent Foundations at FCSD'20, online, 07/07/2020
- *Discrete Morse theory in Cubical Agda*, Logic Seminar at Stockholm University, 08/09/2021
- *Elementary Simplicial Collapses in Cubical Agda*, Workshop on Homotopy Type Theory/Univalent Foundations at LICS'22, online, 31/07/2022

Chapters 4 and 5 result from a collaboration with Evan Cavallo and Anders Mörtberg. I had the initial idea to use constraint satisfaction programming to search for Kan compositions. Evan devised the term language presented in Chapter 4, which

generalised and fixed earlier attempts of mine to capture the fragment of CTT we are interested in. Evan and Anders suggested to look at Dedekind cubes for a version of CTT which might be particularly amenable for automation, and provided numerous examples which guided the development of the presented algorithms. Some passages in Chapters 4 and 5 have been written jointly, in particular Section 4.1. Our joint paper [19] contains several other results not presented in this thesis.

I have presented this line of work at a number of workshops:

- *Automatic proof search in Cubical Agda*, Midlands Graduate School 2022 at University of Nottingham, 11/04/2022
- *Automating Kan composition*, EuroProofNet Workgroup meeting on Type Theory at Stockholm University, 20/05/2022
- *Searching for Kan fillers with poset maps*, Logic Seminar at Stockholm University, 28/09/2022
- *Filling cubes in Cubical Agda with poset maps*, Agda Implementor's Meeting XXXI at Edinburgh University, 14/11/2022
- *Algorithmically finding cells in Kan cubical sets*, Computational Mathematics Seminar at Stockholm University, 01/03/2023
- *Automating reasoning in cubical type theory*, EuroProofNet Workgroup meeting on Type Theory at Technical University Vienna, 25/04/2023

Artefacts. Chapters 2 and 3 have been formalised in Agda (version 2.6.3) using the Cubical library (version 0.5)[1]. All results have been type-checked using the `--safe` flag. We will discuss excerpts from the formalisations; the Index of Agda symbols on page 126 contains a list of all relevant Agda names and where they have been introduced. The full formalisation can also be found online:

`https://zenodo.org/doi/10.5281/zenodo.10966896`

The solver for Cubical Agda presented in Chapters 4 and 5 will be described in pseudo-code, the implementation in Haskell can be found online:

`https://zenodo.org/doi/10.5281/zenodo.10966924`

[1]Cubical Agda library: `https://github.com/agda/cubical`

Chapter 2

Computing with sets

When you ask a mathematician what the powerset of a finite set is, they will answer "it's the set of all its subsets", or more formally:

$$Y \in \mathcal{P}(X) \text{ iff } Y \subseteq X$$

If you ask a Haskell programmer about the powerset, they will use lists to represent sets and devise something like the following program to compute all subsets of the input set:

```
powerset :: [a] -> [[a]]
powerset [] = [[]]
powerset (x:xs) = powerset xs ++ map (x:) (powerset xs)
```

The function works as follows. The powerset of the empty set is the set containing the empty set. If we are given a list `x:xs`, we will compute the powerset of `xs`, and then for each `ys` in the powerset of `xs` add both `ys` and `x:ys` to the powerset of `x:xs`.

The mathematician thereby gives an abstract definition singling out the crucial property of elements of the powerset of X, while the programmer gives an implementation of the powerset as an operation taking a set to a family of sets. Both characterisations have advantages. The mathematical definition allows for quickly deriving properties of the powerset, e.g., using the above definition it is easy to show that for two sets X and X' we have $X \subseteq X'$ precisely if $\mathcal{P}(X) \subseteq \mathcal{P}(X')$. This is not as easy to see for the Haskell program, but the program allows for looking at specific powersets concretely. For example, if we ask Haskell to compute `powerset [1,2,3]`, we get a list of the sets in the powerset:

```
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
```

In this chapter, we will see how we can use the theorem prover Agda [65], which is based on dependent type theory [57], to express both the abstract and algorithmic

version of the powerset in the same language, assuming that we have a total order on the set in question. In fact, the distinction between $\mathcal{P}$ and `powerset` disappears in our setting. We write an analogue of the `powerset` program in Agda, called powerset, and *prove* that powerset satisfies the statement we gave as a definition for $\mathcal{P}$ above. By using the Cubical extension of Agda [78] which implements cubical type theory, a formulation of homotopy type theory (HoTT) [77], we can moreover prove that this correspondence holds for a strong notion of "iff", namely for *equivalence between types* [77, Ch. 4]. We denote an equivalence between types A and B with $A \simeq B$. Establishing that powerset satisfies its mathematical specification allows us to freely switch between the concretely computed powerset and its abstract definition in a formalization.

We can aggregate our constructions to give a term of the following type:

$$\Pi(X : \mathsf{set}), \Sigma(P : \mathsf{family}), \Pi(Y : \mathsf{set}), Y \in P \simeq Y \subseteq X$$

In words: given a set X, we produce a family of sets P (which is powerset X) such that P satisfies the following specification: for any set Y, the types $Y \in P$ and $Y \subseteq X$ are equivalent.

Our construction is an instance of the following pattern: if we are given an input instance for an algorithm X and want to produce an output instance Y such that the dependent type $P(X, Y)$ specifies the relation between Y and X, then giving an algorithm that satisfies its specification amounts to producing a term of the following type:

$$\Pi X, \Sigma Y, P(X, Y)$$

When carrying out this construction in a theorem prover such as Agda, its type-checker ensures correctness of the algorithm. DTT therefore allows for developing algorithms following a very strong version of the *correct by construction* paradigm proposed by Dijkstra [18]. Not only is our development driven by the specification—i.e., the type—of our program, in the end we also get a guarantee that we meet the specification.

In summary, we can use concepts developed in dependently typed programming and apply them to the development of formalized mathematics. We thereby hope to emphasize a benefit of formal reasoning in a dependently type language: it allows for a combined treatment of mathematical theory and programs.

The purpose of this chapter is threefold. First, we provide background on how to reason in (Cubical) Agda. Second, we give a tutorial on using the correct by

construction paradigm for a basic mathematical program. Finally, we develop a framework which will allow us to conveniently compute Morse reducts in Chapter 3.

In Section 2.1, we introduce a notion of sets which—by its very construction—does not have higher structure in its identity type. This makes our notion of sets closely related to lists, which allows us to reason about sets and lists interchangeably. This is used crucially in Section 2.2, where we write an analogue to the `powerset` program in Agda and show that it satisfies the abstract mathematical definition discussed above. In Section 2.3, we compare our development with approaches to representing sets in other theorem provers, namely Coq and Lean. We argue that our approach strikes a useful balance between generality and practicability, not brushing over any complexities while also assuming structure where it is helpful.

The full formalisation of this chapter can be found online[1].

2.1 Sets in dependent type theory

The fundamental building block in modern mathematics is a *set*, which allows for grouping together objects. For instance, we can group the natural numbers 0, 1, and 2 together in a set:

$$S \coloneqq \{0, 1, 2\}$$

Crucially, the elements of the set are considered to exist independently of the set S, and could also be part of a different set. DTT, developed by Martin-Löf [57] as an alternative mathematical foundation to set theory, has at its core a different notion of collections of elements, the eponymous *types*. The elements of a type are not thought of to exist independently of their type—instead, the rules for constructing terms have to be given at the same time as the formation rule of a type. To create a type T with three elements, we can write the following to emphasise that 0_T, 1_T and 2_T have been introduced as the elements of T:

$$0_T, 1_T, 2_T : T$$

In general, the construction rules for types can be recursive, mutually recursive with other types, or parametrised by other types. The construction rules for a type are hence very expressive, and coming up a with succinct definition of a type is often the crucial step towards an elegant formalisation.

The differentiation between sets and types raises the question: what makes a type a set? The answer put forward by HoTT, which we discuss in Section 2.1.1, is that a

[1] Folder `Set` of `https://zenodo.org/doi/10.5281/zenodo.10966896`

type A should be considered a set if there are only trivial identifications on its terms. In other words: every term is equal to itself in only one way. We can strengthen this characterisation by also requiring a procedure which can *decide* for any two terms of A whether they are equal or not. Hedberg's theorem [32] establishes that a type with decidable equality has unique identity proofs, which implies that there are only trivial identity proofs on terms of A.

In the following, we develop a notion of sets which is even more tightly controlled, allowing us to conveniently mirror in type theory the traditional notion of sets as groups of elements taken from a background universe. In our setting, the background universe is a type and all selected elements are saved in a data structure, namely a list. In Section 2.1.2, we introduce what structure we require on the background type, giving rise to *listable* types. This allows us to obtain unique representations of sets, significantly simplifying our theory. Moreover, listable types are closed under the dependent pair type which we will use in Chapter 3. We show in Section 2.1.3 how to develop usual notions such as set-membership and subsets in our framework.

In the following, we will take "set" to mean our very structured notion of sets.

2.1.1 Background on Cubical Agda

In this thesis, we develop all results in the Cubical extension [78] of the dependently typed theorem prover Agda [65]. In the following, we will assume the basics of dependent type theory and only introduce the notions newly introduced by Cubical Agda in more detail. We base our formalisation on the Cubical library[2].

In Agda, any type lives in a universe Type ℓ, where the universe levels ℓ form a cumulative hierarchy. The universe levels behave like the natural numbers, where the bottom level is denoted ℓ_0 and the successor operator ℓ-suc. We usually write Type for Type ℓ_0.

Function types from A to a type B depending on A are denoted with $(x : A) \to B\ x$. If B does not depend on A, we write $A \to B$. Inhabitants of the function type are defined by abstracting over a variable in the definition of the function, or by λ-abstraction. We also use dependent pair types $\Sigma\ A\ B$, for which we have the syntactic sugar $\Sigma[\ x \in A\ ]\ (B\ x)$ standing for $\Sigma\ A\ (\lambda\ x \to B\ x)$. Non-dependent pairs are denoted $A \times B$. Inhabitants of a pair type are denoted $(x\ ,\ y)$; we can access the components with projection functions fst and snd.

[2]Cubical Agda library: `https://github.com/agda/cubical`

We also make use of some standard features of Agda. We sometimes use implicit arguments which are surrounded by curly brackets { }. These can be omitted and will be inferred by the type-checker. We use with abstraction which allows for pattern matching on the result of a computation.

We use the following standard data types:

- We can form the sum of two types, $A \uplus B$, which has injection functions $\mathsf{inl} : A \to A \uplus B$ and $\mathsf{inr} : B \to A \uplus B$.
- The empty type $\bot$ and the unit type $\top$, with no terms and the unique term tt, respectively. These types live in the base universe Type, we sometimes need to lift these types to a higher universe level which we denote with $\bot^*$ and $\top^*$, where the latter type has inhabitant tt^*.

  Negation of a type $\neg\ A$ is defined as $A \to \bot$.

  When pattern matching on an empty type, the absurd pattern () can be used to establish that the respective case cannot occur.
- The two-element type Bool with terms true and false.
- The natural numbers type $\mathbb{N}$ with two terms $\mathsf{zero} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.
- The finite type $(n : \mathbb{N}) \to \mathsf{Fin}\ n$ which contains n many inhabitants, namely $\mathsf{zero} : \{n : \mathbb{N}\} \to \mathsf{Fin}\ (\mathsf{suc}\ n)$ and $\mathsf{suc} : \{n : \mathbb{N}\} \to \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n)$.
- The type $\mathsf{List}\ A$ captures lists on A with the empty list $[] : \mathsf{List}\ A$ and a recursive constructor $_::_ : A \to \mathsf{List}\ A \to \mathsf{List}\ A$.

  We have standard functions $\mathsf{map} : (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B$ as well as $_++_ : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A$.
- A type A is $\mathsf{Discrete}$ if it has decidable equality, i.e., given $P : \mathsf{Discrete}\ A$, we have for all $x, y : A$ that $P\ x\ y$ reduces to either $\mathsf{yes}\ p : x \equiv y$ or to $\mathsf{no}\ \neg p : \neg\ x \equiv y$ (we discuss equality shortly).[3]

When devising dependent type theories, attention is paid so that they have *computational content*, which means that all terms reduce to a normal form. We can leverage reduction in Agda by asking the type-checker to normalise a term. For example, we can inspect the result of adding 1 to each element of a list of numbers as follows:

[3]Note that it is common in Agda to use unicode for identifiers, like $\neg p$ above. The negation symbol here is not an operation, but just part of the name of the variable.

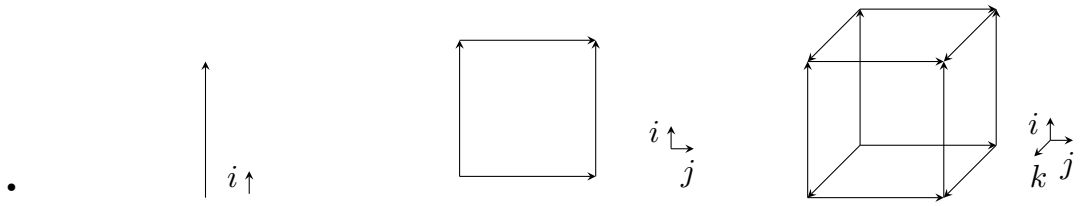
```
normalise: map (_+ 1) (1 :: 2 :: 3 :: [])
>  2 :: 3 :: 4 :: []
```

The normalisation process therefore allows us to inspect normal forms under judgmental equality, i.e., the above reduction relies on the fact that the following judgmental equality holds in Agda:

$$\mathsf{map}\ (_+1)\ (1 :: 2 :: 3 :: []) = 2 :: 3 :: 4 :: []$$

When working in a theorem prover, judgmental equality is usually implicit, but we will at times depict a chain of judgmental equalities to emphasise why a particular statement holds.

Judgmental equality is to be contrasted with *propositional* equality, which we treat differently from Agda's standard approach to equality which is based on Martin-Löf's intensional propositional identity type [55]. We use Cubical Agda, an implementation of cubical type theory [6, 16] which gives computational content to HoTT [77]. Cubical Agda gives primitives for the equality type, which it calls the *path* type. Inspired by the connection of type theory to topology, an abstract unit interval I is introduced which has two endpoints $\mathsf{i0}$ and $\mathsf{i1}$. Interval variables $i, j, \ldots : \mathsf{I}$ allow for talking about cubes of arbitrary dimension, i.e., a term in the empty interval context is thought of as a point; a term in context i as a 1-dimensional path; a term in context i, j as a 2-dimensional path, i.e., a square; a term in context i, j, k as a 3-dimensional path, i.e., a cube; etc. We can hence think of elements of the path type as fillers for the following geometric objects:



The interval I is endowed with the structure of a De Morgan algebra which consists of an involution operation $\sim i$ and connections $i \vee j$ and $i \wedge j$ for $i, j : \mathsf{I}$. A path between two elements living in the same type A parametrised over I lives in the heterogeneous path type:

$$\mathsf{PathP} : (A : \mathsf{I} \to \mathsf{Type}) \to A\ \mathsf{i0} \to A\ \mathsf{i1} \to \mathsf{Type}$$

If the type A is not dependent on the interval, the type above simplifies to the homogeneous path type:

Path : $(A : \mathsf{Type}) \to A \to A \to \mathsf{Type}$

We write _≡_ for Path A, which means that $x \equiv y$ is the type of paths between x and y. Inequality $x \not\equiv y$ is defined as $\neg\ x \equiv y$. One can show that the path type is reflexive and symmetric by abstracting over and applying (the reversal of) an interval variable:

refl : $x \equiv x$
refl $\{x = x\}\ i = x$
sym : $x \equiv y \to y \equiv x$
sym $p\ i = p\ (\sim i)$

In order to show that _≡_ is transitive, another reasoning principle is necessary: homogeneous composition. We will not apply this reasoning principle in this chapter and will introduce it properly in Chapter 3. We will study novel reasoning principles of Cubical Agda from the point of automated reasoning in Chapters 4 and 5, in particular homogeneous composition and the structure on the interval I alluded to above. In this chapter, we only need features of Cubical Agda which have already been introduced in the HoTT book [77].

Given a type A with terms $x, y : A$, we can study paths between x and y; given two paths $p, q : x \equiv y$, we can study paths between p and q, etc., leading to the structure of an ∞-groupoid [79, 4], which we can think of as a higher-dimensional generalisation of equivalence relations. We classify types by the level at which their iterated path type trivialises. In homotopy theory, this corresponds to the homotopy level of a type, and in HoTT, the concept is called homotopy-n-type. In this thesis, we call it h-level, and as follows for some specific levels:

- If A has an inhabitant x and all other elements $y : A$ are equal to x via a path $p : x \equiv y$, it is said to be *contractible*, i.e, isContr A.
- If there is a path $p : x \equiv y$ between any two inhabitants $x, y : A$, then A is said to be an *h-proposition*, i.e., isHProp A.
- If there is a path $\alpha : p \equiv q$ between any two proofs of equality $p, q : x \equiv y$ for some $x, y : A$, then A is said to be an *h-set*, i.e., isHSet A.
- If there is a path $\psi : \alpha \equiv \beta$ between any two $\alpha, \beta : p \equiv q$ for $p, q : x \equiv y$ and $x, y : A$, then A is said to be a *groupoid*, i.e., isGroupoid A.

The stratification of types according to the complexity of their path types carries on with higher groupoids. In this chapter, we only reason about h-sets, while in the next Chapter 3 we will also reason about groupoids. In Chapters 4 and 5, we will automatically derive terms in higher groupoids.

Naturally, h-levels are closed upwards, for instance, an h-proposition is also an h-set. Dependent function and pair types are closed under h-levels.

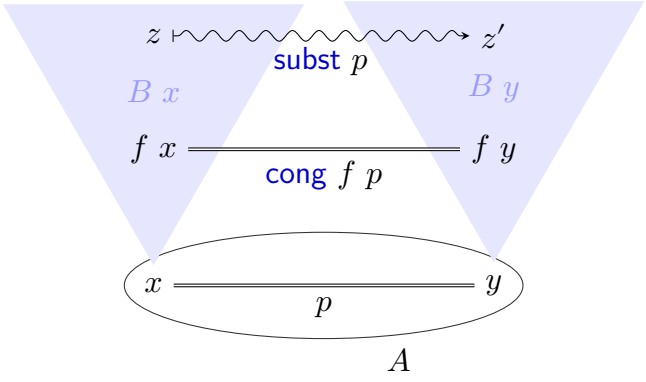
In order to show that two types A and B are equal to each other, one can establish an isomorphism Iso between the two types. One has to construct functions between the two types, $F : A \to B$ and $G : B \to A$, and show that these are mutually inverse, i.e., construct two proofs $R : (x : A) \to G\ (F\ x) \equiv x$ and $S : (y : B) \to F\ (G\ y) \equiv y$. The tenet of univalence states that equivalence and equality of types coincide, which has computational content in Cubical Agda such that an Iso can be turned into an equality between types (after turning it into an equivalence, which is a certain kind of coherent isomorphism, which can always be done in Cubical Agda):

$\mathsf{isoToPath} : \mathsf{Iso}\ A\ B \to A \equiv B$

We also make use of the following functions from the Cubical library, which hold for all types A and terms x, $y : A$ and enable us to use an equality in A in a type depending on A:

$\mathsf{subst} : (B : A \to \mathsf{Type}\ \ell)\ (p : x \equiv y) \to B\ x \to B\ y$
$\mathsf{cong} : (B : A \to \mathsf{Type}\ \ell)\ (f : (a : A) \to B\ a)\ (p : x \equiv y)$
$\quad \to \mathsf{PathP}\ (\lambda\ i \to B\ (p\ i))\ (f\ x)\ (f\ y)$

We can visualise both functions with the following picture:



We are given an equality p between elements x and y which live in type A. Since we are in a type theory where equality is proof-relevant, types $B\ x$ and $B\ y$ indexed by both elements are distinct and we have to explicitly use p to mediate between

them. The function subst allows us to turn a witness $z : B\ x$ into some element $z' : B\ y$, i.e., we *substitute* x with y in the property B. With cong we can apply a function f to $p : x \equiv y$ to obtain a path between the images of the endpoints, i.e., we take the *congruence* with respect to f. Since $f\ x$ and $f\ y$ live in different types, this path lives in $\lambda\ i \to B\ (p\ i)$. When $i =$ i0, this type is $B\ x$, when $i =$ i1, this type is $B\ y$.

Sterling and Angiuli [73] have established normalisation for cartesian cubical type theory, which can be thought of as a sublanguage of Cubical Agda as discussed in Chapter 4. For De Morgan cubical type theory [16], on which Cubical Agda is based, so far only *canonicity* could be established [43], which states that every inhabitant of the natural numbers reduces to canonical form.

2.1.2 Sets as ordered lists

We define sets as lists of elements taken from a background type C. To make it easier to perform and reason about computations, we assume that we have a partial order on C, sets are then chains taken from this background poset. In Agda, we characterise such background types with the following Listable type:

```
record Listable (C : Type ℓ) : Type (ℓ-suc ℓ) where
  constructor listable
  field
    dec : Discrete C
    _⊏_ : C → C → Type ℓ
    ⊏-prop : (x y : C) → isHProp (x ⊏ y)
    ⊏-irreflexive : (x : C) → ¬ (x ⊏ x)
    ⊏-trans : (x y z : C) → x ⊏ y → y ⊏ z → x ⊏ z
```

A record type is a dependent product type where all components get names that can be used as projection functions. Listable is parametrised over the type C on which we impose the structure.

The constructor listable can be used to introduce terms of the record type. The first field— the name of a component of a record type—of Listable is dec which establishes that equality between two elements of C is decidable.

The relation $x \sqsubset y$ is assumed to be an h-proposition, which means that, up to equality, there is at most one term witnessing that two elements are related. Moreover, $\sqsubset$ is irreflexive and transitive, i.e., it is a partial order. It is straightforward to derive

from irreflexivity and transitivity that $\sqsubset$ is antisymmetric, a fact we refer to as $\sqsubset$-antisym.

While C might be a type with infinitely many terms, any set that we consider in this thesis has finite size so that we can list it in Agda. We capture that a list of elements of a Listable type C is totally ordered if it satisfies the following predicate:

```
ordered : List C → Type ℓ
ordered [] = ⊤
ordered (x :: []) = ⊤
ordered (x :: y :: xs) = (x ⊏ y) × ordered (y :: xs)
```

The above predicate is defined by pattern matching on the input list. If the list is empty or contains a single element, it is trivially ordered. If the list has two elements x and y in the front, we require that they are ordered by $\sqsubset$ and that the rest of the list starting at y is ordered. We can now define the notion of a finite set over a Listable type:

```
set : Type ℓ
set = Σ[ xs ∈ List C ] ordered xs
```

The simplest example of a set is the empty set:

```
emptySet : set Cs
emptySet = [] , tt*
```

The empty list captures that we have no elements in our set, and proving that it is ordered is done by giving the single inhabitant of the unit type. The empty set can be constructed for any type C such that Cs : Listable C. For more involved examples of sets we have to turn to a concrete instance of Listable. For example, it is straightforward to show that the Bool type is Listable by imposing an ordering on its two values, say:

```
_⊏₂_ : Bool → Bool → Type
true ⊏₂ false = ⊤
_ ⊏₂ _ = ⊥
```

We can then instantiate sets over the Bool type containing either no, one, or both terms of Bool. The set containing both values, along with proofs that the carrying list ordered, looks as follows:

```
setBool : set listBool
setBool = true :: false :: [] , tt , tt*
```

Another natural example of a type which is totally ordered and has decidable equality is the natural numbers under the greater-than relation; call listℕ the proof that the natural numbers ℕ are Listable under this ordering. We can only list finitely many elements of ℕ, which means that any set over listℕ is a subset of ℕ. We capture all finite sets $\{n-1,\dots,0\}$ for any n with the following function:

```
setℕ : ℕ → set listℕ
setℕ 0 = emptySet
setℕ (suc n) = (n :: fst (setℕ n)) , setℕ-ordered n
```

In the recursive step of the definition, we have to prove that the list n :: fst (setℕ n) is ordered. We can prove this with the following function, which has a type which refers to setℕ and hence is mutually recursive with setℕ:

```
setℕ-ordered : (n : ℕ) → ordered (n :: fst (setℕ n))
setℕ-ordered zero = tt*
setℕ-ordered (suc n) = (0 , refl) , setℕ-ordered n
```

By asking Agda to normalise some expression we can compute with our function setℕ. For instance, we can compute the three-element set S from above by asking Agda to normalise the following term:

```
normalise: setℕ 3
>  2 :: 1 :: 0 :: [] , (0 , refl) , (0 , refl) , tt*
```

The first component of this tuple is the list carrying the elements of the set, afterwards we have a proof that the list is ordered: (0 , refl) shows that $2 > 1$ and $1 > 0$, respectively, and the list 0 :: [] is trivially ordered.

Using the definition of finite types as a subtype of ℕ, we can straightforwardly adapt the above definitions to show that Fin n is listable for any n. We write setFin n for the listed set of elements of Fin n in the following.

By Hedberg's theorem [32], a type with decidable equality has unique identity proofs. This means that the carrying type C of a Listable type is an h-set, a fact we refer to as carrier-isHSet. Consequently, all fields of the Listable type have the h-level of a set or a proposition, which means that Listable C is itself an h-set.

Moreover, all witnesses of two elements being ordered are inhabitants of the unit type and hence unique, and since h-propositions are closed under products, we can easily prove that being an ordered list is also an h-proposition; we refer to this fact as ordered-isHProp.

This means that set is the product of an h-set and an h-proposition; we can hence give a witness of set-isHSet which states that any set is an h-set. The converse does not hold. An h-set is not necessarily a set since we might not be able to decide equality or have an ordering on an h-set. However, any h-proposition P with witness $P\text{-}isHProp : \mathsf{isHProp}\ P$ can be seen to be Listable. Deciding equality on a proposition is trivial—if we have two inhabitants of an h-proposition, they must be equal—and we can use the empty ordering on P, which vacuously satisfies irreflexivity and transitivity:

```
listProp : Listable P
dec listProp a b = yes (P-isHProp a b)
_⊏_ listProp _ _ = ⊥*
⊏-prop listProp _ _ ()
⊏-irreflexive listProp _ ()
⊏-trans listProp _ _ _ ()
```

Therefore any h-proposition is listable. In order to turn an h-proposition into a set, we furthermore need to be able to decide whether it holds. This way we can either create the singleton, or the empty list:

```
setProp : Dec P → set (listProp P-isHProp)
setProp (yes p) = [ p ] , tt*
setProp (no ¬p) = emptySet
```

Listable sets are furthermore closed under dependent products, a fact we will use in Chapter 3 to compute Morse reducts. Given a listable type C and a type D depending on C such that $D\ x$ is listable for any $x : C$, we can show that $\Sigma\ C\ D$ is listable:

```
listΣ : {C : Type ℓ} (Cs : Listable C) {D : C → Type ℓ}
  → ((x : C) → Listable (D x)) → Listable (Σ C D)
_⊏_ (listΣ Cs {D} Ds) (x , y) (x' , y') with dec Cs x x'
... | no ¬p = _⊏_ Cs x x'
... | yes p = _⊏_ (Ds x) y (subst D (sym p) y')
```

We order two elements (x, y) , $(x', y') : \Sigma\ C\ D$ by first checking whether x and x' are equal. If this is not the case, we take the order on C, otherwise we compare y and y'. Since y' lives in $D\ x'$, we have to turn it into an element of $D\ x$ using substitution using that x and x' are equal. It is straightforward to show that this order satisfies all properties of Listable by referring to the appropriate instances of C and D being listable.

In order to turn two sets over C and D into a set over $\Sigma\ C\ D$, we compute their cartesian product:

```
setΣ : (xs : List C) → ordered Cs xs →
  (ys : (x : C) → List (D x)) → ((x : C) → ordered (Ds x) (ys x))
  → set (listΣ Cs Ds)
setΣ [] ds ys es = emptySet
setΣ (x :: xs) ds ys es = x:: (ys x) ++ fst IH ,
  ++-ordered (x::-ord (ys x) (es x)) (snd IH) x⊏setΣxs
  where
  x:: : (ys' : List (D x)) → List (Σ C D)
  x:: ys' = map (λ y → (x , y)) ys'
  IH : set (listΣ Cs Ds)
  x::-sound : (ys' : List (D x)) → (xy : Σ C D) → (xy ∈ₗ x:: ys') → fst xy ≡ x
  x::-ord : (ys' : List (D x)) → ordered (Ds x) ys' → ordered (listΣ Cs Ds) (x:: ys')
```

We do induction on the first set over C. If this set is empty, we can just return the empty set. Otherwise, the set starts with an element x. We want to have all (x, y) for any $y : D\ x$ in the resulting set. This list of pairs is computed with the helper function x::, and prepended to computing the cartesian product with the remaining list xs. In order to show that the thereby constructed list is ordered requires several steps. First, we have to show that the list of (x, y)'s is ordered, which is done in x::-ordered. We then use the following lemma:

```
++-ordered : {xs : List C} (ds : ordered xs) {ys : List C} (es : ordered ys)
  → ({x : C} → (x ∈ₗ xs) → x ⊏L ys) → ordered (xs ++ ys)
```

The lemma states we can directly concatenate two ordered lists into an ordered list provided that any element in the first list comes prior to the second list, where the relation $x \sqsubset\mathsf{L}\ ys$ encodes that x is prior to the head of list ys (which might be the empty list). Since we chose a lexicographic ordering for $\Sigma\ C\ D$, we can establish

this with the following lemma, which is proved by double induction on xs and, if xs starts with an element x', $ys\ x'$:

```
x⊏setΣ : ((x , y) : Σ C D) (xs : List C) → (ds : ordered Cs (x :: xs))
  → (ys : (x' : C) → List (D x')) → (es : ((x' : C) → ordered (Ds x') (ys x')))
  → (x , y) ⊏L fst (setΣ xs (⊏-tail Cs ds) ys es)
```

Thereby we have devised a general method to compute a dependent set. As an example, we can compute all even numbers simply by taking setΣ of setℕ and the predicate isEven (which is defined in the Cubical library and can readily be shown to be propositionally valued and decidable) as follows:

```
setEven : (k : ℕ) → set {C = Σ ℕ isEven} _
setEven k = setΣ (setℕ k) λ n → setProp (isHPropIsEven n) (decIsEven n)
```

Since we stipulated how dependent pairs compute for all instances setΣ, we do not have to do any more work to compute the even numbers, but can directly retrieve all even numbers, e.g., those in the set $\{9, \ldots, 0\}$:

```
normalise: fst (setEven 10)
>  (8 , tt) :: (6 , tt) :: (4 , tt) :: (2 , tt) :: (0 , tt) :: []
```

Each element in the list consists of a number and a proof that it is even, which is always tt.

2.1.3 Membership and subsets

The properties of Listable, over which we defined our sets, gives us a strong link between lists and sets, which will ease the formalisation work when reasoning about computations in the following sections. The following lemmas capture that a set is uniquely determined by its underlying list and that we can therefore switch freely between an equality between sets and an equality between lists:

```
toList≡ : {X Y : set Cs} → X ≡ Y → fst X ≡ fst Y
toSet≡ : {X Y : set Cs} → fst X ≡ fst Y → X ≡ Y
```

We use the following standard data type to capture that given a list, there is at least one element in the list which satisfies some property P:

```
data Any (P : A → Type ℓ) : List A → Type ℓ where
  here : ∀ {x xs} (px : P x) → Any P (x :: xs)
  there : ∀ {x xs} (pxs : Any P xs) → Any P (x :: xs)
```

The first constructor here establishes that if x has property P, then any list $x :: xs$ satisfies Any P; the second constructor there says that Any P is closed under prepending elements. Using the data type Any we can define list membership:

```
_∈ₗ_ : A → List A → Type ℓ
_∈ₗ_ x = Any (x ≡_)
```

Given a list xs and an object x, the type $x \in_l xs$ contains a witness for any y in the list which is equal to x. Therefore we have a proof-relevant notion of membership in a list. This makes it apparent how lists differ from sets: a list might contain duplicate elements, which give rise to multiple proofs that an element is in the list. Moreover, reordering elements in a list yields a different list, but for a set it does not matter in which order we give the elements. Since we only consider strictly ordered lists to be sets, we do not have to worry about duplicates and reordering however, and can safely define set membership in terms of list membership on the carrier list:

```
_∈_ : C → set Cs → Type ℓ
x ∈ X = x ∈ₗ fst X
```

The following lemma establishes that set membership works in the expected way and that there is only a single way in which an element can be part of a set:

```
∈-isHProp : {y : C} {X : set Cs} → isHProp (y ∈ X)
∈-isHProp {y} {x :: xs , ds} (here P) (here Q) = cong here (carrier-isHSet y x P Q)
∈-isHProp {y} {x :: xs , ds} (there P) (there Q) =
  cong there (∈-isHProp {y} {xs , ⊏-tail ds} P Q)
```

We perform induction on the two different proofs that y is an element of X. If it is the head x of the carrying list, we are given two different proofs that x and y are equal and make use of the fact that the carrying type of a Listable type is an h-set. By using cong with the constructor here, this proof of equality is injected into the Any datatype. In case y is part of the tail of the carrying list of X, we can inject the induction hypothesis using cong there. To apply the induction hypothesis, we use the lemma ⊏-tails, which establishes that removing the first element from an ordered list

results in an ordered list. The other two cases of the induction, where we are given one here and one there term, can be shown absurd using the irreflexive order on X.

In order to define the subset-relation, we again first define what it means for one list to be subsumed by another list, and then define the subset relation in terms of the sublist relation:

```
_⊆ₗ_ : List A → List A → Type ℓ
ys ⊆ₗ xs = (y : A) → y ∈ₗ ys → y ∈ₗ xs

_⊆_ : set Cs → set Cs → Type ℓ
X ⊆ Y = fst X ⊆ₗ fst Y
```

Since h-propositions are closed under dependent functions, we can conclude from set-membership being an h-proposition that the sublist and subset relations are h-propositions; we refer to the latter fact as ⊆-isHProp.

Reflexivity ⊆ₗ-refl and transitivity ⊆ₗ-trans can be established for the sublist relation, which carry over to proofs that the subset relation is reflexive and transitive. However, only the subset relation is antisymmetric, in contrast to the sublist relation: consider for instance the lists 1 :: [] and 1 :: 1 :: [], which are sublists of each other, but not the same list. Since our representations of listed sets are ordered by an irreflexive relation, the second list does not represent a set and our subset relation works as expected. We can show with the following proof that ⊆ is antisymmetric:

```
⊆-antisym : {xs : List C} → (ds : ordered xs) {ys : List C} → (es : ordered ys)
  → (xs , ds) ⊆ (ys , es) → (ys , es) ⊆ (xs , ds) → xs ≡ ys
⊆-antisym {[]} ds {[]} es p q = refl
⊆-antisym {x :: xs} ds {y :: ys} es P Q with ⊆-dec es ds P | ⊆-dec ds es Q
... | inl p | inl q =
  Listη q (⊆-antisym (⊏-tail ds) (⊏-tail es) (⊆-step es ds P p) (⊆-step ds es Q q))
```

Given two sets and proofs that they are mutually subsets, we perform induction on the carrying lists of the sets. If both lists are empty, we trivially have that they are equal. In case they start with elements x and y, respectively, we use the lemma ⊆-dec, which establishes that if two lists are subsets of each other, their heads must either be equal, or the head of the first list must be ordered before the head of the second list. The only non-absurd case is that both heads x and y are equal. We can hence apply the induction hypothesis and resolve the proof goal with Listη, which takes proofs that $x \equiv y$ and $xs \equiv ys$ to a proof $x :: xs \equiv y :: ys$. In order to apply

the induction hypothesis, we have also used the following lemma, which establishes that if the heads of two lists are equal and one is a subset of the other, then their tails must also be in the subset relation:

```
⊆-step : (y :: ys) ⊆l (x :: xs) → x ≡ y → ys ⊆l xs
```

2.2 Computing the powerset

In this section, we implement the powerset program in Agda. We can implement the program similar to the Haskell function `powerset` on page 9, the program is only complicated by the fact that we have to show that we compute ordered lists, which we discuss Section 2.2.1. We then show that our powerset operation satisfies the usual mathematical definition of a powerset, i.e., powerset computes precisely the subsets of the input set. We prove the forward direction of the specification in Section 2.2.2, which we call *soundness* as we ensure that any set computed by powerset is indeed a subset of the input set. We then prove the backward direction in Section 2.2.3, called *completeness* as we want to show that any subset is computed by the program. Finally, we establish in Section 2.2.4 that the result of the powerset program is itself an ordered list and therefore forms a set. We aggregate the whole formalization in a type inspired by the correct by construction paradigm in Section 2.2.5. In the following, we assume that we are given types C and Cs : Listable C.

2.2.1 The powerset program in Agda

When porting the powerset program to Agda, we want to show that we only produce sets, i.e., that each computed sublist is ordered. Moreover, we want to show that the resulting list is itself a set, which means that we need to establish an ordering between all computed sublists. We leave this part of the proof to Section 2.2.4 and for now focus on computing a *list* of sets, which we do with the following powerset program:[4]

```
powerset : (xs : List C) → ordered xs → List (set Cs)
powerset [] ds = [ emptySet ]
powerset (x :: xs) ds = powerset xs (⊏-tail ds) ++ powerset-insert x xs ds
```

[4]It would be nicer to have a single input $(xs\ ,\ ds)$: set for the powerset function, but this is not possible since the termination checker of Agda does not recognise that a function defined by recursion on one component of a Σ type is structurally recursive. We therefore have to uncurry the set in this and the following definitions.

The program is structurally similar to the `powerset` program in Haskell from page 9. The main difference is that in the recursive step, we cannot simply insert x into every set with `map` since we have to show that we produce an ordered list in this step. We do this in the function powerset-insert, but before we can construct this we first derive a crucial lemma: x is the first element of any set computed in the recursive call with respect to $\sqsubset$. To express this, we use the relation $\sqsubset$LS, where $x \sqsubset\text{LS } Ys$ means that x is prior to any set in the list of sets Ys. Equipped with this, we can prove that in the recursive step, x is prior to all sets computed in the recursive call:

```
x⊏Lpowerset : (x : C) (xs : List C) → (ds : ordered (x :: xs))
  → x ⊏LS (powerset xs (⊏-tail ds))
x⊏Lpowerset x xs ds ([] , es) p = tt*
x⊏Lpowerset x (x' :: xs) (d , ds) (y :: ys , es) p
  with ⊆-dec ds es (powerset-sound ds es p)
... | inl q = subst _ q d
... | inr q = ⊏-trans _ _ _ d q
```

The proof works by induction on both the input set (xs, ds) and an arbitrary set (ys, es) computed by powerset xs ds. If ys is empty, then x is trivially in front of ys. Otherwise we have two non-empty sets $x' :: xs$ and $y :: ys$ and have to prove that $x \sqsubset y$ holds. By using soundness of the powerset function, which we will prove in Section 2.2.2, we know that $y :: ys$ is a subset of $x' :: xs$. We decide with $\subseteq$-dec whether y is equal to x' or whether we have $x' \sqsubset y$. In the first case, we can simply substitute x' for y in the proof $d : x \sqsubset x'$; otherwise we use transitivity of $\sqsubset$.

We split up the definition of powerset-insert as we will need to prove certain lemmas about the insertion process later. It is straightforward to define the following functions for inserting an element into a set and inserting it into a list of sets, we only give the types of the functions here:

```
insert : (x : C) → (Z : set Cs) → x ⊏L fst Z → set Cs
insertL : (x : C) (Ys : List (set Cs)) (desc : x ⊏LS Ys) → List (set Cs)
```

Equipped with these helper functions and lemmas we can finally define the insertion function for the powerset:

```
powerset-insert : (x : C) (xs : List C) → (ordered (x :: xs)) → List (set Cs)
powerset-insert x xs ds = insertL x (powerset xs (⊏-tail ds)) (x⊏Lpowerset x xs ds)
```

Our powerset function is defined mutually recursively with x⊏Lpowerset; which in turn makes use of the function powerset-sound; which we will define shortly by recourse to powerset. Therefore the powerset program requires mutual induction between three functions to establish that it only produces ordered lists. We can test that we have reconstructed the original functional program `powerset` by mapping the first projection to the result of powerset, for instance for the three-element set:

```
normalise: map fst (powerset (setℕ 3))
>  [] ∷ (0 ∷ []) ∷ (1 ∷ []) ∷ (1 ∷ 0 ∷ []) ∷ (2 ∷ []) ∷ (2 ∷ 0 ∷ []) ∷
(2 ∷ 1 ∷ []) ∷ (2 ∷ 1 ∷ 0 ∷ []) ∷ []
```

2.2.2 Soundness

In our construction of the powerset program we have already used that it only computes subsets of the input set, which is the soundness direction of our specification:

$$\begin{aligned}&\text{powerset-sound} : \{xs : \text{List } C\}\ (ds : \text{ordered } xs) \to \{ys : \text{List } C\}\ (es : \text{ordered } ys)\\ &\quad \to (ys\ ,\ es) \in_l \text{powerset } xs\ ds \to (ys\ ,\ es) \subseteq (xs\ ,\ ds)\end{aligned}$$

We prove soundness by induction on both the input set, whose underlying list we call xs, and the set computed by powerset, which we call ys. The base cases can be established as follows:

$$\begin{aligned}&\text{powerset-sound}\ \{[]\}\ ds\ \{ys\}\ es\ P =\\ &\quad \text{subst}\ (_\subseteq_l\ [])\ (\text{sym}\ (\text{toList}\equiv\ (\in_l\text{-singl-extract}\ P)))\ (\subseteq_l\text{-refl}\ [])\\ &\text{powerset-sound}\ \{x ∷ xs\}\ ds\ \{[]\}\ es\ P = []\subseteq_l\text{-all}\end{aligned}$$

If the input list xs is empty, the proof P witnessing that ys is computed by powerset reduces to a proof that ys is in the family of sets containing only the empty set. We then use a lemma $\in_l$-singl-extract, which says that if an element is in a set containing a single element, it must be this very element, to establish that ys is empty. We can therefore use reflexivity of the subset relation, appropriately substituted.

If the list ys is empty, it is a sublist of any list, a fact which is established in the lemma $[]\subseteq_l$-all.

Proving the inductive step is more laborious, we only give the structure of the proof and the types of necessary lemmas here (the interested reader can find details in the code):

```
powerset-sound {x :: xs} ds {y :: ys} es P
  with ++-dec discreteSet _ (powerset xs (⊏-tail ds)) (powerset-insert x xs ds) P
... | inl Q = ⊆ₗ-weaken (powerset-sound (⊏-tail ds) es Q)
... | inr Q = subst (_⊆ₗ (x :: xs)) (cong (_:: ys) (sym headLemma)) (⊆ₗ-insert x IH)
  where
  tailLemma : (ys , ⊏-tail es) ∈ₗ powerset xs (⊏-tail ds)
  IH : (ys , ⊏-tail es) ⊆ (xs , ⊏-tail ds)
  headLemma : y ≡ x
```

We do a case distinction using ++-dec, which says that if an element is in the concatenation of two lists, it must be either in the first or in the second list.

In the first case, $y :: ys$ was computed by the recursive call powerset xs, which means we can weaken the induction hypothesis using $\subseteq_l$-weaken. This lemma establishes that the sublist relation is closed under prepending an element to the larger list.

In the second case, $y :: ys$ was computed by inserting x in front of that recursive call. The induction hypothesis yields that the tail of the computed list must be a sublist of the tail of the input list, but in order to use the induction hypothesis we have to prove that ys is computed by the recursive call of powerset. This is done with the tailLemma. By proving that the first element in the computed list y must be x with headLemma, we can substitute y for x in a version of the induction hypothesis. More precisely, the induction hypothesis is extended using the lemma $\subseteq_l$-insert, which establishes that the sublist relation is closed under prepending an element to both lists.

In the inductive step we have used two simple lemmas about inserting an element x into a list of sets. If we insert x into a list of sets Zs with the function insertL, then any set in this list has x in front, and the tail of each set was already a set in Zs. The types of the lemmas are the following:

```
insertL-head : (Zs : List (set Cs)) (desc : x ⊏LS Zs)
  → (y :: ys , es) ∈ₗ insertL x Zs desc → y ≡ x

insertL-tail : (Zs : List (set Cs)) (desc : x ⊏LS Zs)
  → (y :: ys , es) ∈ₗ insertL x Zs desc → (ys , ⊏-tail es) ∈ₗ Zs
```

Thereby we have proved that everything that the powerset function computes is indeed a subset of the input set.

2.2.3 Completeness

We now prove that any subset of the input set is computed by powerset, i.e., that the program is complete. Unlike the soundness proof, completeness is not required to define the program in the first place, which means that its term is defined without any mutual recursion. The type and inductive bases of this statement are as follows:

```
powerset-comp : {xs : List C} (ds : ordered xs) {ys : List C} (es : ordered ys)
  → (ys , es) ⊆ (xs , ds) → (ys , es) ∈ₗ powerset xs ds
powerset-comp {[]} ds {ys} es P = here (toSet≡ (⊆[]→≡[] es P))
powerset-comp {x :: xs} ds {[]} es P =
  ∈ₗ++-inj₁ discreteSet (powerset-comp (⊏-tail ds) es ([]⊆ₗ-all))
```

If the input set xs is empty, we know that the output set ys is empty since it is a sublist of the empty list, a statement which is proved in the lemma ⊆[]→≡[]. Computing the powerset of the empty set yields the set containing the empty set, we can therefore use the here constructor to establish that powerset has computed ys.

If the output set is empty, we know that it must have been computed in the left side of the list concatenation in the recursive step of powerset, we use the lemma ∈ₗ++-inj₁ to conclude that its in the concatenation of this list with the second list in the definition of powerset.

Again, the inductive step requires more work:

```
powerset-comp {x :: xs} ds {y :: ys} es P with ⊆-dec ds es P
... | inr p =
  ∈ₗ++-inj₁ discreteSet (powerset-comp (⊏-tail ds) es (⊆-skip-fst ds es P (⊏→≢ p)))
... | inl p = ∈ₗ++-inj₂ (subst
    (_∈ₗ insertL x (powerset xs (⊏-tail ds)) (x⊏Lpowerset x _ (ds)))
    (toSet≡ (insxys≡x::ys ▪ cong (_:: ys) p)) insxys∈insxPs) where
  IH : (ys , _) ∈ₗ powerset xs _
  insxys∈insxPs : insert x (ys , _) _ ∈ₗ insertL x (powerset xs _) _
  insxys≡x::ys : fst (insert x (ys , _) _) ≡ x :: ys
```

In order to prove the inductive step with a non-empty input $x :: xs$ and output $y :: ys$, we have to do a case distinction on whether the first elements of the input and output sets coincide, which we do with ⊆-dec.

In case we have a proof $p : x \sqsubset y$, we know that the computed set $y :: ys$ must have already been in the previously computed powerset xs. In order to apply the

induction hypothesis, we use $\subseteq$-skip-fst to weaken P to a witness that $y :: ys$ must have already been a subset of xs, where we turn p into a proof that x and y must be unequal using $\sqsubset \to \not\equiv$. We can then again use $\in_l$++-inj$_1$ to show that $y :: ys$ is in the concatenated list.

Otherwise we have a proof $p : x \equiv y$ and the induction hypothesis only gives us that ys is a sublist of xs. By substituting x and y appropriately we can derive that $y :: ys$ has been computed by powerset in the right list, we can then complete the proof using $\in_l$++-inj$_2$ which is dual to $\in_l$++-inj$_1$.

2.2.4 Showing that the powerset is a set

So far, we have only computed a *list* of sets, we now show that the result of the powerset program is itself a set. For that, we have to show that it is totally ordered using an apt notion of ordering. We use an irreflexive lexicographic ordering, where two sets of unequal length such that one is a prefix of the other will have the shorter set ordered before the longer set:

```
setOrder : {xs : List C} → ordered xs → {ys : List C} → ordered ys → Type ℓ
setOrder {[]} _ {[]} _ = ⊥*
setOrder {[]} _ {y :: ys} _ = ⊤
setOrder {x :: xs} _ {[]} _ = ⊥*
setOrder {x :: xs} ds {y :: ys} es with dec x y
... | yes p = setOrder (⊏-tail ds) (⊏-tail es)
... | no ¬p = y ⊏ x
```

Proving that this ordering is irreflexive and transitive is straightforward.

We can package setOrder to a relation between sets $X \ll Y$, and hence have established that the type of sets is itself Listable, which we call listFamily in the following.

To show that the powerset program produces a list ordered by $\ll$, we have to show that the union of finite sets is itself ordered. This can be done similarly as in the definition of setΣ, we just give the type of the statement we thereby establish:

```
powerset-ordered : (xs : List C) (ds : ordered xs) → IF.ordered (powerset xs ds)
```

2.2.5 A correct by construction powerset

We have constructed a program powerset which, given an input set X, computes its powerset such that we can prove that it contains all and only the subsets of X. Thereby we can put our developments in the shape of a correct by construction program as follows:

```
Powerset : (X : set Cs) → Σ[ P ∈ family ] ((Y : set Cs) → Iso (Y ⊆ X) (Y F.∈ P))
fst (Powerset (xs , ds)) = powerset xs ds , powerset-ordered xs ds
snd (Powerset X) Y = ϕ where
  ϕ : Iso (Y ⊆ X) (Y F.∈ fst (Powerset X))
  fun ϕ = powerset-comp (snd X) (snd Y)
  inv ϕ = powerset-sound (snd X) (snd Y)
  rightInv ϕ _ = F.∈-isHProp {Y} { fst (Powerset X) } _ _
  leftInv ϕ _ = ⊆-isHProp Y X _ _
```

In order to establish an isomorphism as introduced in Section 2.1.1, we have to give maps back and forth between $Y \subseteq X$ and Y F.$\in$ P, which we can do with the completeness and soundness proofs developed in Sections 2.2.2 and 2.2.3. Furthermore, we must show that both functions have a section and retraction, which amounts to showing that going back and forth with the two functions leads yields an element that is equal to the original element. This can be shown easily since both the subset relation and membership in a family are h-propositions, which means that all inhabitants are equal.

Therefore our careful definition of sets, membership and subsets allows us to derive a strong statement about the powerset program. We can use the powerset program both to compute the powerset of concrete sets, and to develop usual mathematical theory. For example, we can show that $X \subseteq X'$ implies $\mathcal{P}(X) \subseteq \mathcal{P}(X')$ in a natural way, by writing $\mathcal{P}$:= fst ∘ Powerset this looks as follows:

```
𝒫-closed : (X X' : set Cs) → X ⊆ X' → 𝒫 X F.⊆ 𝒫 X'
𝒫-closed X X' P Y Q = X'fun (⊆-trans Y X X' (Xinv Q) P) where
  X'fun : Y ⊆ X' → Y F.∈ 𝒫 X'
  X'fun = fun (snd (Powerset X') Y)
  Xinv : Y F.∈ 𝒫 X → Y ⊆ X
  Xinv = inv (snd (Powerset X) Y)
```

Given some set Y which is in $\mathcal{P}$ X, we use the inverse direction of the isomorphism established in Powerset for X to show that Y must be a subset of X. Since X is a

subset of X', we have that Y is a subset of X' by transitivity, and using the forwards direction of the isomorphism for X' that Y must hence be in $\mathcal{P}\ X'$.

2.3 Discussion

We have seen how we can both compute with, and reason about sets in Cubical Agda. Imposing a transitive and irreflexive order on the type from which we build sets allows us to use sets and lists interchangeably, which means we can reason about list based computations to establish properties of the output of such computations. The development in this section could be carried out in any theorem prover based on DTT; reasoning in Cubical Agda makes transparent the complexity of each identity type—in contrast to, e.g., extensional DTT where this complexity is hidden. Imposing the structure of "listability" externally on a type (instead of, e.g., phrasing all results in terms of finite types) has the advantage that we can develop mathematical formalisations independently of computational considerations, and only afterwards compute with listable instances as necessary. We will follow this approach in the next Chapter 3.

We could have relaxed the assumption of an ordering on the background type and used the concept of *truncations* to make sure that the sets have no higher homotopical content; we compare our framework with such an approach in Section 2.3.1. Leaving HoTT behind, it is also possible to reconstruct our results in the theorem prover Lean [61] as discussed Section 2.3.2. In Lean, soundness of the powerset function can be established significantly quicker than in Agda, which comes at the cost of Lean being a theorem prover which rests on a less well-understood foundation than Cubical Agda. We believe that our approach strikes a good balance between the two presented alternatives, taking seriously foundational concerns about type theory while also assuming additional structure where it is helpful for carrying out computations.

2.3.1 Finite sets in homotopy type theory

Frumin et al. [27] have developed finite sets over a background type in HoTT. They study and show equivalent *Kuratowski finite sets*, which are in essence free join semilattices axiomatizing union of sets, and *listed finite sets*, which quotient lists by duplication and reordering of elements. In order to establish the cardinality of a thereby defined finite set, it is necessary to assume that the base type has decidable equality, while an ordering on the base type is not necessary for the developments by Frumin et al. [27].

Our notion of sets as totally ordered lists provides an *implementation* of Kuratowski finite sets [27, Definition 5.2.]. Crucially, the finite sets of Frumin et al. [27] are h-set-truncated, which means that they have a constructor which introduces a path between any two identity proofs in a set. Otherwise, their finite sets would not be h-sets since the higher path constructors needed both for Kuratowski and listed finite sets introduce higher homotopical structure. In our development, a set requires no truncation since it has no higher homotopical structure. This significantly simplifies our developments, as otherwise the truncation constructor has to be dealt with whenever mapping out of a set.

The ordering assumption on the base type therefore allows us to reason more easily about the powerset program and many proofs make crucial use of it. When reasoning about a program, it is necessary to reason about the implementation details of the objects on which this program operates, and we chose the notion of sets which is easiest to reason about. We also note that any finite set is isomorphic to a subset of the natural numbers set$\mathbb{N}$, we therefore do not lose any generality.

2.3.2 Powersets in Lean

In Lean, finite sets are implemented as multisets which are accompanied by a proof that they do not contain duplicates. This makes it possible to implement the powerset function as follows in mathlib4, Lean's standard library[5]:

```
def powerset (s : finset α) : finset (finset α) :=⟨
 s.1.powerset.pmap finset.mk
  $ λ t h, nodup_of_le (mem_powerset.1 h) s.nodup,
  s.nodup.powerset.pmap $ λa ha b hb, congr_arg finset.val ⟩
```

Similarly to our program powerset, the Haskell `powerset` works by mutual induction on a proof that the function is correct, i.e., the lemma `mem_powerset` used in the construction of `powerset` establishes that the program computes precisely the subsets of the input set:

```
@[simp] theorem mem_powerset {s t : finset α} :
  s ∈ powerset t ↔s ⊆t :=
  by cases s;
  simp only [powerset, mem_mk, mem_pmap, mem_powerset,
            exists_prop, exists_eq_right];
  rw ← val_le_iff
```

[5] `https://leanprover-community.github.io`

The main work in the proof is done with the `simp` tactic, which uses a selection of lemmas to derive the specification of the `powerset` function.

Crucially, the statement above is proved with respect to logical equivalence $\leftrightarrow$, which only requires functions back and forth between two types. This is the strongest notion of equivalence available in Lean—an equivalence of types as shown in Powerset cannot be expressed in Lean since higher homotopical structure is not captured in the logic underlying Lean.

Identity and quotients have been implemented with a focus on usability in Lean, but lack a well-understood theoretical underpinning. Using tactics, proofs about programs can be carried out quickly in Lean , but it is unclear which normalisations can be carried out by the evaluator and where it might break. Notably, subject reduction fails in Lean when using quotients, which means that reducing a term might change its type.[6]

[6] `https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/What.20is.20the.20subject.20reduction.20debate.3F`

Chapter 3

Computing smaller spaces

After having set up a framework in which we can compute with sets, we will now develop discrete Morse theory (DMT) [26], a crucial method in the field of topological data analysis (TDA). TDA aims to give topological meaning to datasets, giving rise to tools to study the "shape of data". TDA is a natural domain for our project of demonstrating the usefulness of both the computational and topological features of Cubical Agda since the field is about devising algorithms using advanced mathematical methods involving homotopy types. Moreover, TDA has already been applied in critical areas such as the detection of cancer [64], increasing the need to ensure the reliability of its methods and tools.

We formalise the main result of DMT for graphs, which states that a graph and its Morse reduct are homotopy equivalent, making crucial use of the language that Cubical Agda offers for reasoning about homotopy types. The full formalisation can be found online[1].

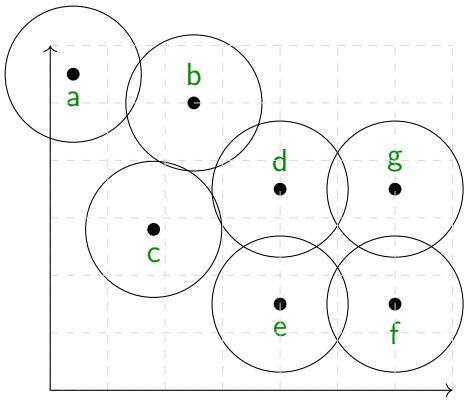


Figure 3.1: Associating a space with points in a metric space

[1]Folder `Morse` of `https://zenodo.org/doi/10.5281/zenodo.10966896`

The basic tenet of TDA is to associate a topological space with a dataset and study topological invariants of that space to understand the dataset. Consider Figure 3.1, which depicts points a, ... , g in a 2-dimensional metric space.

We can compute the Čech complex [28] of this dataset for a given parameter ϵ as follows. Take as 0-cells the data points. Consider the ϵ-balls around each point and add a cell to the complex for any collection of points whose balls have a non-empty intersection. The Čech complex of this dataset, depicted in Figure 3.2, has 8 1-dimensional cells which we draw as lines between the 0-cells (the arrows stem from the fact that we will develop our results for *directed* structures in the following). We can compute invariants of the complex such as its cohomology groups to detect that there are holes in the dataset between points b, c, d and d, e, f, g. However, computing these invariants is complicated by the fact that there are many cells in the complex which have no impact on the topology. To rectify this, tools for TDA often preprocess the space using DMT, which is a method for reducing the size of a complex without changing its topological properties. DMT works by collapsing a number of cells according to a partial matching between adjacent cells. We have given an exemplary matching on the graph in Figure 3.2, where all cells which are matched are depicted hollow/dashed and in blue. The direction of an edge leads away from the vertex that has been matched to that edge, e.g., the 0-cell f is matched to the edge between f and g. The only non-matched cells are the 0-cell d and the 1-cells from f to e and b to c, which are depicted solid and in red.

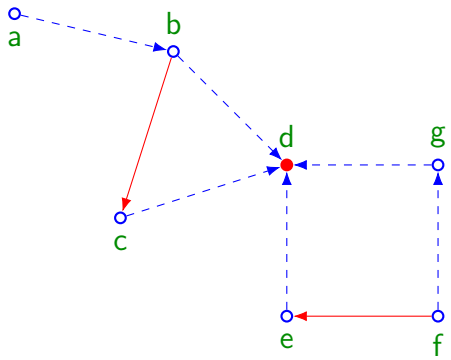


Figure 3.2: A graph associated with a dataset

When collapsing a complex according to a partial matching, we arrive at a topological space which has as many cells as there were unmatched cells in the graph. For the graph in Figure 3.2; the collapsed complex consists of a single point and two loops around it as depicted in Figure 3.3. This complex can also be constructed as the topological wedge sum [31], which joins two spaces in a single point, of the circle

with itself. The central result of DMT establishes that the original complex has the same homotopy type as the reduced complex, which justifies computing invariants of the latter space to study the original dataset.

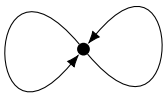


Figure 3.3: The wedge of two 1-spheres

In this chapter, we show in general that performing such reductions is sound, i.e., that reducing a graph along a Morse matching to its critical cells preserves the homotopy type of the graph. Homotopy equivalence can be turned into equality by univalence such that our developments will culminate in establishing the following result, where E is the edge relation of a directed graph, crit_1 are the 1-dimensional paths of its Morse reduct, and $|_|$ turns the combinatorial description of a space to the corresponding homotopy type (this operation is usually called geometric realisation):

$$\mathsf{MorseTheorem} : |\ E\ | \equiv |\ \mathsf{crit}_1\ |$$

In the context of homotopy type theory, Swan [74] has developed machinery similar to ours to prove the Nielsen–Schreier theorem, a classical result from group theory; indeed [74, Lemma 4.8] can be seen as a case of discrete Morse theory where a whole spanning tree is removed. In contrast to this work, our result is formalised using the internal logic of homotopy types offered by Cubical Agda, using concrete derivations in the cubical calculus instead of more abstract homotopy theory. As a result, our work is also significantly different from non-formalised proofs [26, 63, 13]. Formalising discrete Morse theory in a theorem prover which also acts as a programming language is a crucial step towards developing a verified tool for TDA fully inside a theorem prover.

We discuss the mathematical theory behind DMT in Section 3.1, and the relevant features of Cubical Agda which have not already been introduced in Section 3.2. In Section 3.3, we introduce a higher inductive type to take the geometric realisation of 1-dimensional structures, i.e., relations, which allows us to directly study the homotopy type captured by a combinatorial description. We develop Morse matchings and give an explicit description of the Morse complex of a graph, which is a relation, in Section 3.4. We then prove our main theorem in Section 3.5, where we establish an isomorphism between a graph and its Morse reduct. We can turn this isomorphism into an equality between types using univalence. This allows us to use results from the

Cubical library to establish topological properties of groups which we demonstrate with some examples in Section 3.6. Using the framework of listable types developed in Chapter 2, we can furthermore inspect the Morse complex of a listed graph explicitly, which we demonstrate in Section 3.7. We close with an outlook on what is required to turn the presented work into a tool for TDA in Section 3.8.

3.1 Background on discrete Morse theory

Discrete Morse theory (DMT) [26] was developed as a combinatorial analogue of Morse theory from differential topology. Roughly, DMT allows to perform at once a sequence of Whitehead collapses [82], which remove a single cell from a complex to obtain a homotopy-equivalent subcomplex. The theory is often presented for simplicial complexes, such as in Forman [25], whereas the complex after the reduction is a more general kind of complex, namely a CW complex. We will shortly outline this traditional presentation before specialising our considerations to directed graphs, the subject matter of this chapter (in the context of graphs, there has only been work on understanding the structure of all matchings on a graph [5], but not a dedicated development of Morse reductions).

The input to a Morse reduction is the combinatorial description of a simplicial complex and a matching on its cells. Simplicial complexes are certain well-behaved spaces that can be built by gluing together simpler spaces, called simplices. To capture these kinds of spaces, one only has to keep track of which simplices exist and how they are glued together, which can be done with the following combinatorial gadget, which is in essence a downwards-closed family of sets:

Definition 3.1. An abstract simplicial complex is a set V, called the vertices, with a set K of finite non-empty subsets of V, called the cells, s.t.

1. $V \in K$ and
2. $\tau \in K$ and $\sigma \subseteq \tau$ implies $\sigma \in K$.

In the second condition, cell σ is said to be a *face* of τ. Denote with $\sigma \prec \tau$ and $\tau \succ \sigma$ that σ is a face of τ and that $\mid \sigma \mid = \mid \tau \mid - 1$.

Definition 3.2. An *acyclic partial matching* (also called *gradient vector field*) μ on a complex K is given by a set of pairs (σ, τ), where $\sigma \prec \tau$, s.t.

1. every cell of K is in at most one pair of μ and

2. for every set of distinct matched pairs $(\sigma_i, \tau_i) \in \mu$ with $1 \leq i \leq k$ for $k > 1$, which are adjacent in the complex, i.e.,

$$\sigma_1 \prec \tau_1 \succ \sigma_2 \prec \tau_2 \succ \ldots \prec \tau_k,$$

we have that σ_1 is not a face of τ_k.

The first condition in Definition 3.2 states that any cell is part of at most one collapse, while the second condition ensures that we do not get rid of any cycles in the complex, as this would fundamentally change the topology. The sequences of cells in the second condition are often called *paths*, if a path satisfies the given condition it is said to be *gradient*. Consider the two triangle complexes in Figure 3.4. In the complex on the left, we have matched a with (a, b), b with (b, c), and c with (a, c). Since we have a sequence $a \prec (a, b) \succ b \prec (b, c) \succ c \prec (a, c)$ such that a is a face of (a, c), this matching gives rise to a path which is not gradient, the matching is hence not acyclic. In contrast, in the triangle on the right we have matched a with (a, b) and c with (b, c), and all paths are gradient. The partial matching is hence acyclic.

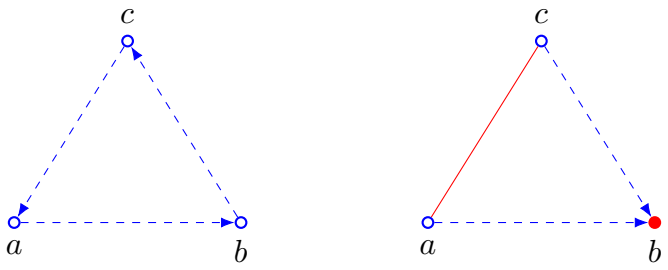


Figure 3.4: Examples of a cyclic and an acyclic partial matching

The cells that are not in μ are called *critical* since they determine the topology of the complex. In the right triangle in Figure 3.4, the cells b and (a, c) are critical. The central result of Forman [26] allows for reconstructing the homology groups—an important algebraic invariant of topological spaces—of the complex just by looking at the critical cells of a Morse matching on it. Moreover, Forman [26] proves the existence of a CW complex consisting only of the critical cells which is homotopy equivalent—a stronger algebraic invariant than homological equivalence—to the original complex. However, this smaller Morse complex is not constructed explicitly. To rectify this, Nanda, Tamaki, and Tanaka [63] offer a generalization of acyclic partial matchings which allow for exactly describing the homotopy type of the reduced complex.

DMT has been useful in many contexts; for this project we are interested in its applications for TDA. The main idea of TDA is to associate a topological space with a

dataset, which is given by points in a metric space. The two most common methods of turning a dataset into a space are the Čech complex (Č) and the Vietoris-Rips complex (VR) [28], which both construct a cell for every collection of data points which lies closely together. As discussed on page 36, the Č is constructed by studying intersections of balls around the points. The VR is constructed as follows. A cell is constructed if the *pairwise* distances of all points are less than a certain parameter. For example, the VR of the dataset in Figure 3.1 with distance 2ϵ has a triangle filler for the three points b, c , d as they are pairwise within 2ϵ.

In general, Č is a subcomplex of VR and is a more topologically accurate representation of the dataset. However, the VR is cheaper to construct, which is why it is often used in practice [28]. Crucially, the VR is determined by the 0- and 1-dimensional cells as a higher-dimensional cell σ exists in a VR iff $\{x, y\} \in K$ for all $x, y \in \sigma$. This means one only has to save the 1-skeleton of a VR, making it easier to store than the Č.

In the following, we develop DMT for *directed* graphs, which are a generalisation of 1-dimensional simplicial or cubical complexes since we have no symmetry condition for the edges.

3.2 Background on higher inductive types

Results about DMT such as [26, 70] are usually presented using CW complexes and simple homotopy theory. In this chapter, we formalise our results in Cubical Agda [78], and use its internal logic for homotopy equivalence inspired by cubical sets to develop DMT.

The path types introduced in Section 2.1.1 can be used in constructors of inductive types, giving rise to *higher* inductive types (HITs) [77]. For example, one can define a HIT representing the 1-dimensional sphere, i.e., the circle, as follows:

```
data S¹ : Type where
  base : S¹
  loop : base ≡ base
```

Since S^1 has a path loop which is different from refl at the base point, S^1 is not an h-set. However, it is a groupoid, a fact established in the Cubical library as isGroupoidS^1.

We can build more complex homotopy types by gluing together more elementary HITs. A central construction for this is the Pushout type, which is inspired by the

categorical pushout construction and is defined as follows in Cubical Agda for a given span $B \overset{f}{\leftarrow} A \overset{g}{\rightarrow} C$:

```
data Pushout {A B C : Type} (f : A → B) (g : A → C) : Type where
  inl : B → Pushout f g
  inr : C → Pushout f g
  push : (a : A) → inl (f a) ≡ inr (g a)
```

The first two constructors inject elements of the two codomains of the span into the pushout, and the third constructor equates all images of elements from A. We can use the pushout type to mimic the join construction [31, p. 9] in type theory, where the following function creates a wedge of two types, joining types A and B in the given base points:

```
_∨_ : Σ[ A ∈ Type ] A → Σ[ B ∈ Type ] B → Type
_∨_ (A , a) (B , b) = Pushout {A = ⊤} (λ _ → a) (λ _ → b)
```

The above function is used in the Cubical library to create the wedge of two 1-spheres as introduced in Figure 3.3 with (S¹ , base) ∨ (S¹ , base) by joining the circle S¹ with itself, taking the base points as their intersection. The type (S¹ , base) ∨ (S¹ , base) is comprised of two loops inl loop and inr loop around inl base and inr base, respectively. Both base points are equated by push tt : inl base ≡ inl base, giving rise to the intended homotopy type.

Using HITs to represent homotopy types of topological spaces, Mörtberg and Pujet [60] develop homotopy theory in Cubical Agda. Cavallo [14] develops cohomology, a computationally more benign topological invariant, synthetically. Brunerie, Ljungström, and Mörtberg [10] develop this approach further in Cubical Agda. We will use results from this line of work which are part of the Cubical library and establish group isomorphisms ≅ between cohomology groups coHomGr of (wedges of) S¹ and (products ×G of) the trivial group ⊤Group and the integers ℤGroup in Section 3.6.

When reasoning about HITs, we have to use the De Morgan structure of the interval variables as well as *homogeneous composition*. This reasoning principle is inspired by Kan composition [48], a criterion devised by Kan to make cubical sets a model of (homotopy types of) topological spaces. Intuitively, Kan composition states that any open cube can be filled in a uniform way. We will discuss the cubical reasoning principles—and work on automatically applying them—in more detail in the following Chapters 4 and 5, in the following we just sketch how to apply these principles to carry out the formalisation in this chapter.

Homogeneous composition can be used to show that $\equiv$ is transitive, as we expect from a structure capturing equality. Suppose we are given three paths $p : w \equiv x$, $q : x \equiv y$ and $r : y \equiv z$ and consider the square on the left of Figure 3.5, which is spanned by interval variables i and j. The three paths are fitted in the left, bottom and right side of the square by inverting p and leaving q and r as is. Homogeneous composition now states that the square must have a path on the top, which we call $p \blacksquare\blacksquare q \blacksquare\blacksquare r : w \equiv z$.

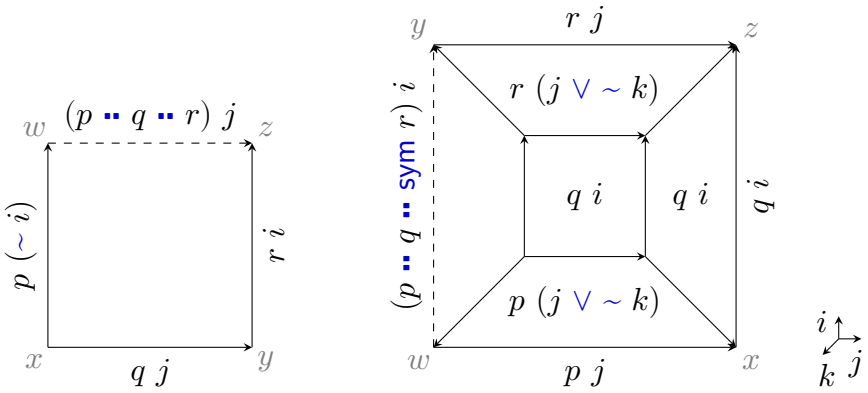


Figure 3.5: Constructing paths and squares with composition

In Cubical Agda, homogeneous composition is implemented as hcomp. The primitive takes as a first argument the sides of the open cube that we are filling, where we can pattern match over the interval variables in the context. The second argument captures the back side of the cube. The following term corresponds to the construction depicted on the left of Figure 3.5:

```
_▪▪_▪▪_ : w ≡ x → x ≡ y → y ≡ z → w ≡ z
(p ▪▪ q ▪▪ r) j = hcomp (λ i → λ {
      (j = i0) → p (~ i)
  ; (j = i1) → r i
  }) (q j)
```

If we take the first path to be refl, we get composition of two paths, which demonstrates that equality in Cubical Agda, captured with paths, is transitive:

```
_▪_ : x ≡ y → y ≡ z → x ≡ z
p ▪ q = refl ▪▪ p ▪▪ q
```

PathP types can be iterated to capture higher-dimensional equalities. Suppose we want to prove that going along the composite p ▪▪ q ▪▪ sym r and then r is the same as only going along p and then q. One way to capture this is with the square which is depicted on the outside of the cube on the right of Figure 3.5. On the left side of the square we have the composed path, after which we follow along the top path r. This should be equal to following p along the bottom side of the square and then q along the right side. To show that this square can be filled we again employ homogeneous composition, this time on an open 3-dimensional cube. We depict the back square in the inside of the cube and travel to the front of the cube with dimension k. We leave the left square of the cube unspecified, which leads to the composed path being unfolded inside this square. This means that the bottom square of the cube has p as a common boundary with the left square and we can use De Morgan operations on the interval variables j and k, which are spanning the bottom square, to provide a p-filler. The top square works similarly with r, and the back and right square are constant q squares in the appropriate directions. In Cubical Agda, the described homogeneous composition looks as follows:

```
pqr⁻¹r≡pq : (p : w ≡ x) (q : x ≡ z) (r : y ≡ z)
  → PathP (λ i → (p ▪▪ q ▪▪ sym r) i ≡ q i) p r
pqr⁻¹r≡pq p q r i j = hcomp (λ k → λ {
  (i = i0) → p (j ∨ ~ k)
  ; (i = i1) → r (j ∨ ~ k)
  ; (j = i1) → q i
  }) (q i)
```

In the following, we will also use another filler for path composition defined in the Cubical library:

```
compPath-filler' : (p : x ≡ y) (q : y ≡ z) → PathP (λ j → p (~ j) ≡ z) q (p ▪ q)
```

We can turn a witness of a type being an h-set into a square using isHSet→SquareP from the Cubical library, i.e., this function constructs a term of the appropriate PathP type from a witness that any two paths in this type are equal.

3.3 Realising relations and graphs

In DMT—and many other areas of topology, both abstract and applied— the working principle is to study spaces via some combinatorial description. In this section we

introduce the combinatorial descriptions we will use, and how we can pass from this description to the topological space described by them. To this end, we use a higher inductive type to turn the 1-dimensional description of a cell complex, i.e., a relation, into the homotopy type it represents. We will then introduce a special kind of relation, a directed graph, which will be the object on which we perform DMT.

A relation is given by a base type c_0 and a family of types $c_1 : c_0 \to c_0 \to \mathsf{Type}$. We can also think of c_0 and c_1 as a semi-simplicial type [37] restricted to level 1, and call c_0 the 0-cells and c_1 the 1-cells.

We use the following HIT to construct the geometric realisation of a relation:[2]

```
data |_| {c₀ : Type ℓ} (c₁ : c₀ → c₀ → Type ℓ) : Type ℓ where
  |_|₀ : c₀ → | c₁ |
  |_⇒_∋_|₁ : (x y : c₀) → c₁ x y → | x |₀ ≡ | y |₀
```

The first constructor includes any element x of the base type into the realisation, denoted $|\ x\ |_0$. The second constructor $|\ x \Rightarrow y \ni f\ |_1$ takes a witness f that x and y are related to a path between their realisations.

For example, consider the empty relation on the unit type:

```
⊥₁ : ⊤ → ⊤ → Type
⊥₁ tt tt = ⊥
```

The realisation of $\bot_1$ has only one 0-cell, but no 1-cells. We expect it to be equal to the unit type $\top$, which we can establish by constructing an isomorphism and lifting it to an equality between types using univalence:

```
⊤≡|⊥₁| : ⊤ ≡ | ⊥₁ |
⊤≡|⊥₁| = isoToPath (iso (λ _ → | tt |₀) (λ _ → tt)
  (λ {| tt |₀ → refl ; (| tt ⇒ tt ∋ () |₁ i) }) (λ _ → refl))
```

Another basic relation is the unit relation on the unit type, which has a single 1-cell starting and ending at the 0-cell tt:

```
⊤₁ : ⊤ → ⊤ → Type
⊤₁ tt tt = ⊤
```

[2]This HIT is known in homotopy type theory as a *type quotient* [50], and is part of the Cubical library as TypeQuotient. In the library, the purpose of the HIT is to quotient an algebraic structure and deal with the resulting coherences. In our work, we want to study the homotopy type induced by R and have chosen to use a different name and notation.

Since the realisation of $\top_1$ has exactly one 0- and one 1-cell, we expect it to be equal to the 1-sphere S^1. Again, one can construct an isomorphism in Cubical Agda to establish this. We have omitted the proof as it follows directly by pattern matching and refer to it as $\mathsf{S}^1{\equiv}|\top_1|$ in the following.

Of interest in the following will also be the relation which assigns two loops to the one-element type, which we can define by assigning Bool to the unit inhabitant:

```
Bool₁ : ⊤ → ⊤ → Type
Bool₁ tt tt = Bool
```

Since the realisation of Bool_1 has one 0-cell with two loops, we expect it to behave like the wedge of spheres from Figure 3.3, which is defined in the Cubical library as $(\mathsf{S}^1 , \mathsf{base}) \bigvee (\mathsf{S}^1 , \mathsf{base})$. Establishing this requires some work with the De Morgan algebra on the interval, similar to proofs by Mörtberg and Pujet [60]; we have omitted this here and refer the interested reader to $\mathsf{S}^1{\vee}\mathsf{S}^1{\equiv}|\mathsf{Bool}_1|$ in the formalisation.

In the following, we will need to show that a relation c_1 is equal to another relation c_1'. One can do this by establishing an isomorphism on the base type, and then on any of the type-values of the relation. These isomorphisms can then be lifted to an equality between types $| c_1 | \equiv | c_1' |$. We call this lift isoToReal in the following.

3.3.1 Directed graphs

We now turn to a special case of a type endowed with a relation: directed graphs. These are comprised of a set of vertices V and an edge relation E:

```
record Graph (V : Type ℓ) (E : V → V → Type ℓ) : Type ℓ where
  field
    V-discrete : Discrete V
    E-isIrrefl : (u : V) → ¬ (E u u)
    E-isHProp : (u v : V) → isHProp (E u v)
```

By requiring that V is discrete, we can decide equality between vertices. From Hedberg's theorem [32] it follows that V is a set; we refer to this fact as V-isHSet. Because simplicial complexes have no loops, we require that E is irreflexive. Requiring that E is an h-proposition means that we cannot have multiple edges between the same vertices (we can however have two edges between two vertices in opposing directions.)

Note that a Graph V E is always a set since it is a dependent pair of a set and a family of h-propositions. In contrast, a graph's realisation $\mid E \mid$ is in general a groupoid.

Example 3.3. We can define a graph on Bool with edges going in both directions:

```
two₁ : Bool → Bool → Type
two₁ false false = ⊥
two₁ false true = ⊤
two₁ true false = ⊤
two₁ true true = ⊥

two : Graph Bool two₁
```

We have omitted the definition of two since it follows directly from the properties of Bool and pattern matching on two_1. We can depict the graph two as follows:



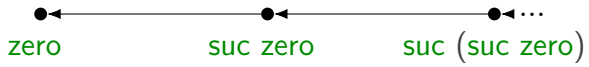
Since we have a loop going from true to itself via false, we expect the realisation of two to have the same homotopy type as the circle, a fact that we will be able to prove in Section 3.6. △

Example 3.4. Our setting also allows us to define graphs on infinite types such as the natural numbers $\mathbb{N}$. We can create a graph line on the natural numbers by pointing to the predecessor of a number:

```
line₁ : ℕ → ℕ → Type
line₁ zero n = ⊥
line₁ (suc m) n = m ≡ n

line : Graph ℕ line₁
```

Again, discreteness of the vertices follows from results in the Cubical library. The edge type is propositionally valued since $\mathbb{N}$ is a set. Irreflexivity of the graph can be established by pattern matching and using that no number is smaller than itself. The graph line can be depicted as follows:



Even though this graph has infinitely many vertices, it has no holes and we expect it to be contractible. We will be able to establish this in Section 3.6 using an appropriate Morse matching. △

3.4 Morse matchings on graphs

In the following, we are given an arbitrary G : Graph V E and develop an account of acyclic partial matchings in dependent type theory. As discussed in Section 3.1, matchings are pairs of a vertex and an edge starting at that vertex. Since any vertex can be matched to at most one edge, we represent matchings with a function $\mu : V \to V$. We then use $\mu\ u \equiv v$ to mean that u is matched with the edge $E\ u\ v$. Since we also need to make sure that this edge exists, we introduce the following notions to capture matched and critical edges:

$_\Rightarrow^m_$ $_\Rightarrow^c_$: $(u\ v :\ V) \to$ Type ℓ
$u \Rightarrow^m v = E\ u\ v \times (\mu\ u \equiv v)$
$u \Rightarrow^c v = E\ u\ v \times (\mu\ u \not\equiv v)$

Since V is a set, equality between elements of V is an h-proposition. As edges are also h-propositions, it follows that matched and non-matched edges are h-propositions; we refer to this as $\Rightarrow^m$IsHProp and $\Rightarrow^c$IsHProp, respectively. Since V is discrete, we can decide whether two vertices u and v are connected with a matched edge with $u \Rightarrow^m ?\ v$.

The critical vertices are those which are not matched to any edge, which are precisely the fixpoints of μ in our setting (recall that G does not allow self-loops). We also introduce an alias for the first projection to quickly refer to the underlying vertex of a critical vertex:

isCrit $u = \mu\ u \equiv u$
crit $= \Sigma[\ u \in V\ ]$ isCrit u
$__v = $ fst

Since V is a set, being a critical vertex is an h-proposition, a fact that we refer to as isCritIsHProp. It also follows that the type of critical vertices crit is an h-set, a fact that we refer to as critIsHSet.

By using functions to represent matchings, we can omit the first condition of Definition 3.2 requiring that any cell is part of only one matching. The second condition of Definition 3.2 requires a bit more work to establish—recall that we need to prevent that matchings remove cycles of vertices and edges since this would otherwise change the topology. For this, we introduce the following data type to represent sequences of matched pairs:

```
data _⇝_ : (u v : V) → Type ℓ where
  end : {u v : V} → isCrit u → u ≡ v → u ⇝ v
  mp : {u : V} (v : V) {w : V} → (u ⇒^m v) → (v ⇝ w) → (u ⇝ w)
```

The end constructor captures the most basic sequences of matched pairs, which consist just of a self-matched, i.e., critical, vertex. We have to introduce two separate vertices u and v with a witness that they are equal instead of simply stating $u \rightsquigarrow u$ in the antecedent of this constructor to prevent "green slime" [58], which makes it difficult to see for the type-checker if a pattern matching on this data type covers all cases.

The second constructor mp (for matched pair) prepends a matched edge to a sequence of matched edges. This definition of sequences allows us to concisely state the acyclicity condition on matchings: for any vertex, we need to be able to reach a critical vertex using a sequence of matched pairs:

```
acyclic : Type ℓ
acyclic = (u : V) → Σ[ c ∈ crit ] u ⇝ c [ v ]
```

In the terminology of Section 3.1, we require that all paths in the graph are gradient. We will hence refer to acyclic sequences of matched edges as gradient paths. In summary, an acyclic partial matching is any acyclic $\mu : V \to V$:

```
Matching : (G : Graph V E) → Type ℓ
Matching G = Σ[ μ ∈ (V → V) ] acyclic G μ
```

For the proof of the Morse theorem we need to derive some lemmas about gradient paths. Naturally, the subgraph of gradient paths of a graph is acyclic. This means that we can straightforwardly perform induction on gradient paths; and do not have to take into account the intricacies studied by Kraus and Raumer [50].

Assume that we are given a matching μ, with μ-acy a proof that it is acyclic.

We introduce notation for the endpoints of a gradient path, which we call sinks in the following, as well as the gradient path leading to the sink of some vertex:

```
sink : V → crit
sink u = fst (μ-acy u)
_⇝sink : (u : V) → u ⇝ (sink u) [ v ]
u ⇝sink = snd (μ-acy u)
```

Note that our dependently typed formulation of the acyclicity condition directly yields the sink of any vertex. We can show that any vertex has a unique sink by induction on the gradient paths to two putatively different sinks:

```
sinkUniq : {u : V} (c d : V) → (u ⇝ c) → (u ⇝ d) → c ≡ d
sinkUniq c d (end μ a) (end μ' b) = sym a ▪ b
sinkUniq c d (mp v μ p) (mp v' μ' q) =
  sinkUniq c d p (subst (_⇝ d) (sym (snd μ') ▪ snd μ) q)
```

In the base case, both gradient paths are simply a proof that the vertex is critical and we can concatenate the witnesses that they are fixpoints of μ. When both paths end with a matched edge, we can recurse on the smaller paths and substitute the types of this induction hypothesis appropriately. The other cases where one path is a critical vertex and the other is a matched edge are absurd since a vertex cannot simultaneously be critical and part of a matched edge. In the following, we sometimes use sinkUniq^c, which lifts sinkUniq to an equality between critical vertices.

We also need that paths are unique, i.e., that ⇝ is propositionally valued:

```
⇝IsHProp : {u : V} → (c : crit) → isHProp (u ⇝ c .v)
⇝IsHProp {u} c (end μ a) (end μ' b) i =
  end (isCritIsHProp u μ μ' i) (V-isHSet u (c .v) a b i)
⇝IsHProp {u} c (mp v μ p) (mp v' μ' q) i = mp (v≡v' i) (μ≡μ' i) (IH' i)
```

The proof works similar to the one of sinkUniq, we only have to perform some substitutions on the result of the induction hypothesis, here called IH'. The interested reader can find the details in the formalisation.

Lastly, we want to define the 1-cells of the Morse complex. This has to be a relation on the *critical* vertices, which means that we cannot simply use the type $u \Rightarrow^c v$ as it is a relation between vertices of the underlying graph. Instead, we consider as 1-cells between two critical vertices c and d all critical edges f in the graph such that c and d are connected to the endpoints of f:

```
crit₁ : crit → crit → Type ℓ
crit₁ c d = Σ[ u ∈ V ] Σ[ v ∈ V ] ((u ⇒^c v) × (u ⇝ c .v) × (v ⇝ d .v))
```

Note that crit$_1$ is in general neither irreflexive nor propositionally valued. The Morse reduct in our setting is hence not a Graph, but just a relation. The fact that the Morse reduct of a graph is a more general structure corresponds to the Morse

reduct of a simplicial complex being a CW complex, as discussed in Section 3.1. We can only prove that the 1-cells of the Morse complex form a set since the vertices of the original graph form a set and the type of matched edges and paths are h-propositions, a fact we refer to as $\mathsf{crit}_1\mathsf{IsHSet}$ in the following.

Example 3.5. We can create the following matching on the graph two from Example 3.3:

```
two-μ : Bool → Bool
two-μ true = true
two-μ false = true
```

We can depict the matched graph as follows:



In order to establish that this matching is acyclic, we have to give paths from each vertex of two to a critical cell. The only critical cell is true, we hence show that two-μ is acyclic by giving the trivial path from true to itself, and the 1-step path from false to true:

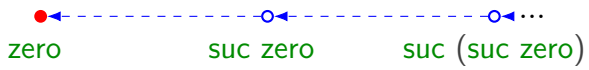
```
two-μ-acy : acyclic
two-μ-acy false = (true , refl) , mp true (tt , refl) (end refl refl)
two-μ-acy true = (true , refl) , (end refl refl)
```

In the Morse reduct given by crit and crit_1, we have one 0-cell (true , refl) and one 1-cell from (true , refl) to itself, which is induced by the critical edge from true to false. The realisation of critical cells crit_1 is hence equal to $\top_1$, which can be established using isoToReal and extensive pattern matching, resolving absurd cases as necessary. We refer with $\top_1\equiv$Mtwo to the fact that the Morse reduct of two is equal to the realisation of $\top_1$.

△

Example 3.6. We can create a matching on line from Example 3.4 encompassing every edge as follows:

```
line-μ : ℕ → ℕ
line-μ zero = zero
line-μ (suc n) = n
```



The only critical vertex of line under this matching is zero, we hence have to establish a path from each natural number to zero, which we can do by recursion:

```
line-acy : acyclic
line-acy zero = (zero , refl) , (end refl refl)
line-acy (suc m) = let IH = line-acy m in fst IH , (mp m (refl , refl) (snd IH))
```

Since the Morse reduct contains only one 0-cell (zero , refl) and no edges, we expect its realisation to be equal to $\bot_1$. This can again be established using isoToReal, extensive pattern matching and refuting absurd cases using properties of the natural numbers established in the Cubical library. We refer to this proof as $\bot_1$≡Mline. △

In both examples, we have established that the Morse reducts have simpler description in terms of relations introduced in Section 3.3, which allows us to easily prove properties of Morse reducts. To carry these statements over to the original graphs, we have to establish that the realisation of a graph and its Morse reduct have the same homotopy type.

3.5 Graphs and their reducts are isomorphic

We can now go about proving our main theorem: the realisation | E | of a Graph V E is homotopy equivalent to its Morse reduct | crit_1 |. We do this by constructing an isomorphism in Cubical Agda, giving functions back and forth in Section 3.5.1 and showing that they are mutually inverse in Sections 3.5.2 and 3.5.3.

3.5.1 Mapping between the graph and its reduct

Let us define the collapsing function from the realisation of the graph | E | to the reduct | crit_1 |. We map any vertex in the graph to its sink, i.e., the unique critical cell that its gradient path is leading to. In order to map an edge in the original graph to the Morse reduct, we have to make a case distinction depending on whether the edge was matched:

```
ϕ : | E | → | crit₁ |
ϕ (| u |₀) = | sink u |₀
ϕ (| u ⇒ v ∋ f |₁ i) with u ⇒ᵐ? v
... | yes μ = | sinkUniqᶜ (sink u) (sink v) (u ⇝sink) (mp v (f , μ) (v ⇝sink)) i |₀
... | no ¬μ = | sink u ⇒ sink v ∋ (u , v , (f , ¬μ) , (u ⇝sink) , (v ⇝sink)) |₁ i
```

If an edge $f : u \Rightarrow v$ is matched, witnessed by μ, the reduct will have collapsed u and v into a single point, namely the sink of u. This sink is the same critical vertex as the sink of v, and by concatenating the path from v to its sink with the matched edge we get another path from u to its sink. Employing sinkUniq^c for these two paths, we get a path between the images of $| u |_0$ and $| v |_0$ under ϕ, which we can lift to the realisation of the Morse complex.

If the edge f was not matched, it is still part of the reduct and we can simply send the edge to its occurrence in the reduct, which has the sinks of u and v as endpoints.

In order to define the inclusion function from the Morse reduct to the graph, we first construct a function which turns a gradient path (i.e., a sequence of matched pairs) into a path in $| E |$ (i.e., an equality in Cubical Agda). We construct this by induction on the gradient path, taking the 0-cell induced by the critical vertex in the base case and inductively precomposing realisations of edges:

```
|_|⇝ : {u v : V} → u ⇝ v → | u |₀ ≡ | v |₀
| end μ a |⇝ = cong |_|₀ a
|_|⇝ {u} (mp v μ p) = | u ⇒ v ∋ fst μ |₁ ∙ | p |⇝
```

Using this realisation of gradient paths we can construct our inclusion. Any critical cell was already a vertex in the original graph, so we can simply project out the critical vertex. When including a critical edge f between c and d, we have to construct a path from the underlying vertex of c to the underlying vertex of d, which we can do by composing the realisations of the gradients paths and f appropriately:

```
ι : | crit₁ | → | E |
ι (| c |₀) = | c v |₀
ι (| c ⇒ d ∋ (u , v , ¬μ , p , q) |₁ i) =
  ((sym | p |⇝) ∙∙ (| u ⇒ v ∋ fst ¬μ |₁) ∙∙ | q |⇝) i
```

This completes our definition of functions between the graph and its Morse reduct. We now have to show that these functions are mutually inverse (up to homotopy type, which is precisely the notion of equality in Cubical Agda).

3.5.2 ι is a right inverse of ϕ

In this section we show that collapsing the inclusion of a cell of the Morse reduct does not change the homotopy type:

```
rinv : (y : | crit₁ |) → ϕ (ι y) ≡ y
```

In order to prove the above statement, we first prove one crucial lemma. Intuitively, the lemma says that collapsing a gradient path does not close any cycles in the graph. In other words, we want to show that given a gradient path $p : u \rightsquigarrow c$, the path $\phi\ (|\ p\ |{\rightsquigarrow})$ can be continuously deformed into the reflexive path at $|\ c\ |_0$. For this statement to make sense, we construct paths between the endpoints of $\phi\ (|\ p\ |{\rightsquigarrow})$ and $|\ c\ |_0$ using $\mathsf{sinkUniq}^c$, leading to the following square:

```
||⇝Contr : (u : V) (c : crit) (p : u ⇝ c ᵥ) →
  PathP (λ i → Path (| crit₁ |)
    | sinkUniq^c (sink u) c (u ⇝sink) p i |₀
    | sinkUniq^c (sink (c ᵥ)) c (c ᵥ ⇝sink) (end (snd c) refl) i |₀)
    (λ j → ϕ (| p |⇝ j))
    (λ j → | c |₀)
```

In order to give a filler for the square with the boundary defined above, we perform induction on the gradient path. In the base case, the gradient path is $\mathsf{end}\ \mu\ a$ where a witnesses that $u \equiv c_v$. The collapsing function sends this path to a 0-dimensional cell $|\ \mathsf{sink}\ (a\ i)\ |_0$, which means that all sides of the square reduce to 0-dimensional cells. We can therefore use that the critical cells form a set to construct an appropriate square, which just needs to be lifted to the realisation of the Morse reduct.

```
||⇝Contr u c (end μ a) = λ i j →
  | isHSet→SquareP (λ _ _ → critIsHSet)
    (sinkUniq^c (sink u) c (u ⇝sink) (end μ a))
    (sinkUniq^c (sink (c ᵥ)) c (c ᵥ ⇝sink) (end (snd c) refl))
    (cong sink a)
    (λ _ → c) j i |₀
```

In the inductive step of the lemma, the gradient path starts with a matched edge from u to some vertex v. We use homogeneous composition to construct the corresponding square, as depicted on the outside of the cube in Figure 3.6.

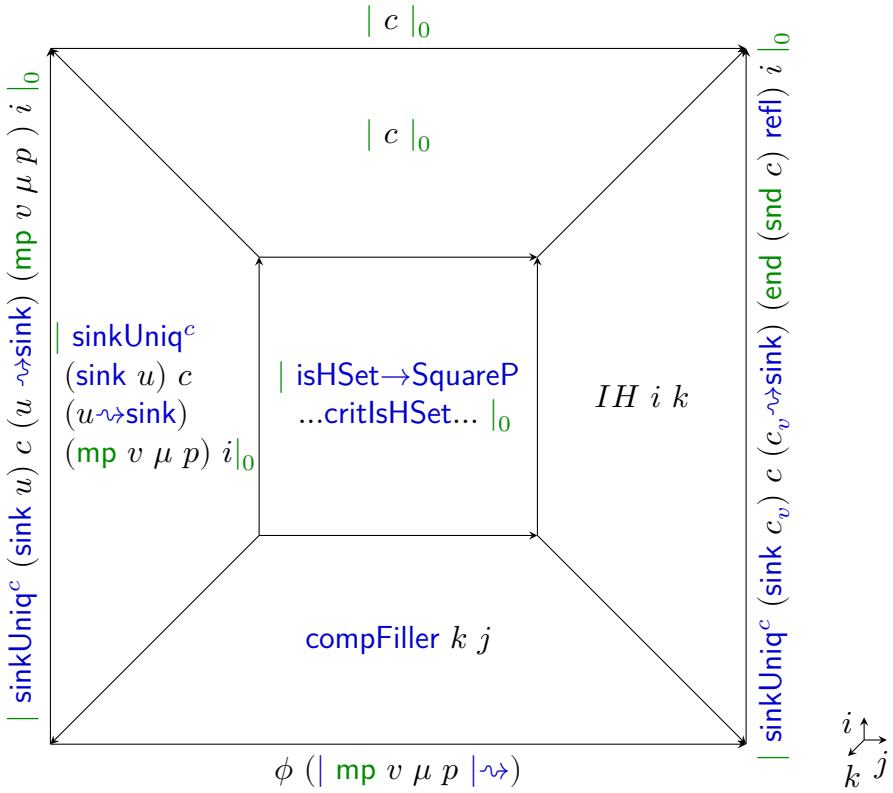


Figure 3.6: Showing that gradient paths are contractible, inductive step

To fill the sides of the open cube, we first observe that collapsing a gradient path starting with the matched edge between u and v reduces to a composition of paths:

$$\begin{aligned}
&\phi(|\mathsf{mp}\ v\ \mu\ p|\rightsquigarrow) \\
&= \phi(|u \Rightarrow v \ni \mathsf{fst}\ \mu\ |_1 \cdot |p|\rightsquigarrow) \\
&= |\ \mathsf{sinkUniq}^c\ (\mathsf{sink}\ u)\ (\mathsf{sink}\ v)\ (u\rightsquigarrow\mathsf{sink})\ (\mathsf{mp}\ v\ \mu\ (v\rightsquigarrow\mathsf{sink}))\ i\ |_0 \cdot \phi\ (|\ p\ |\rightsquigarrow)
\end{aligned}$$

Unfolding this composition of paths in the inside of the cube leads to the right side of the cube having $\phi\ (|\ p\ |\rightsquigarrow)$ on its bottom boundary. This means we can use the induction hypothesis $IH := ||\rightsquigarrow\mathsf{Contr}\ v\ c\ p$ to fill this square. The other sides of the open cube can be filled by appropriately lifting paths in the Morse reduct to its realisation:

```
||⇝Contr u c (mp v μ p)
  = λ i j → hcomp (λ k → λ {
     (i = i0) → compFiller k j
  ; (i = i1) → | c |₀
  ; (j = i0) → | sinkUniq^c (sink u) c (u ⇝sink) (mp v μ p) i |₀
  ; (j = i1) → ||⇝Contr v c p i k
  }) (| isHSet→SquareP (λ _ _ → critIsHSet)
          (sinkUniq^c (sink u) (sink v) (u ⇝sink) (mp v μ (v ⇝sink)))
          (λ _ → c)
          (sinkUniq^c (sink u) c (u ⇝sink) (mp v μ p))
          (sinkUniq^c (sink v) c (v ⇝sink) p) i j |₀)
```

The only complications in this proof are due to two technicalities of Cubical Agda. Even though we know that u and v are matched as witnessed by μ, we still have to pattern match on whether $u \Rightarrow^m ?\ \ v$ holds so that the type checker unfolds the with abstraction in the definition of ϕ. Moreover, unfolding the composition of paths cannot be done by simply leaving the side at $i = \mathsf{i1}$ open. The type checker of Cubical Agda cannot reduce all terms into the homogeneous composition that defines path composition, which is why we have to manually reduce certain terms. Since the proof of compFiller is not informative we have omitted it here and refer the interested reader to the formalisation.

Using this crucial lemma, we can go about showing that sending an element of the Morse reduct to the original graph and back will not change the homotopy type. We do this by pattern matching on the realisation of the reduct, i.e., we first give a proof for the 0-dimensional cells and then for the 1-dimensional cells.

In the 0-dimensional case, we have to construct a path between $| \ c \ |_0$ and its image under first including it in the graph and then collapsing it back into the Morse reduct, which is $| \ \mathsf{sink}\ c_v \ |_0$. Since we have a gradient path from c to its sink using the acyclicity of μ, as well as the trivial path to c, we can use the lemma $\mathsf{sinkUniq}^c$ to construct an appropriate path:

```
rinv (| c |₀) = cong |_|₀ (sinkUniq^c (sink (c ᵥ)) c (c ᵥ ⇝sink) (end (snd c) refl))
```

For the inductive step, recall that a 1-cell $(u, v, \neg\mu, p, q)$ between two critical cells c and d is sent by ι to a composition of gradient paths, on which ϕ acts componentwise:

$$\begin{aligned}\alpha &:= \phi\ ((\mathsf{sym}\ |\ p\ |{\rightsquigarrow})\ \blacksquare\ (|\ u \Rightarrow v \ni \mathsf{fst}\ \neg\mu\ |_1)\ \blacksquare\ |\ q\ |{\rightsquigarrow}) \\ &= \phi\ (\mathsf{sym}\ |\ p\ |{\rightsquigarrow})\ \blacksquare\ |\ \mathsf{sink}\ u \Rightarrow \mathsf{sink}\ v \ni (u\ ,\ v\ ,\ \neg\mu\ ,\ (u \rightsquigarrow \mathsf{sink})\ ,\ (v \rightsquigarrow \mathsf{sink}))\ |_1 \\ &\qquad \blacksquare\ \phi(|\ q\ |{\rightsquigarrow})\end{aligned}$$

We therefore have to construct a path between $|\ c \Rightarrow d \ni (u\ ,\ v\ ,\ \neg\mu\ ,\ p\ ,\ q)\ |_1$ and α, i.e., a square, where the other two sides are given by the action of rinv on $|\ c\ |_0$ and $|\ d\ |_0$. The boundary of this square is depicted on the outside of the cube in Figure 3.7; in order to fill the goal boundary, we apply homogeneous composition to this open cube.

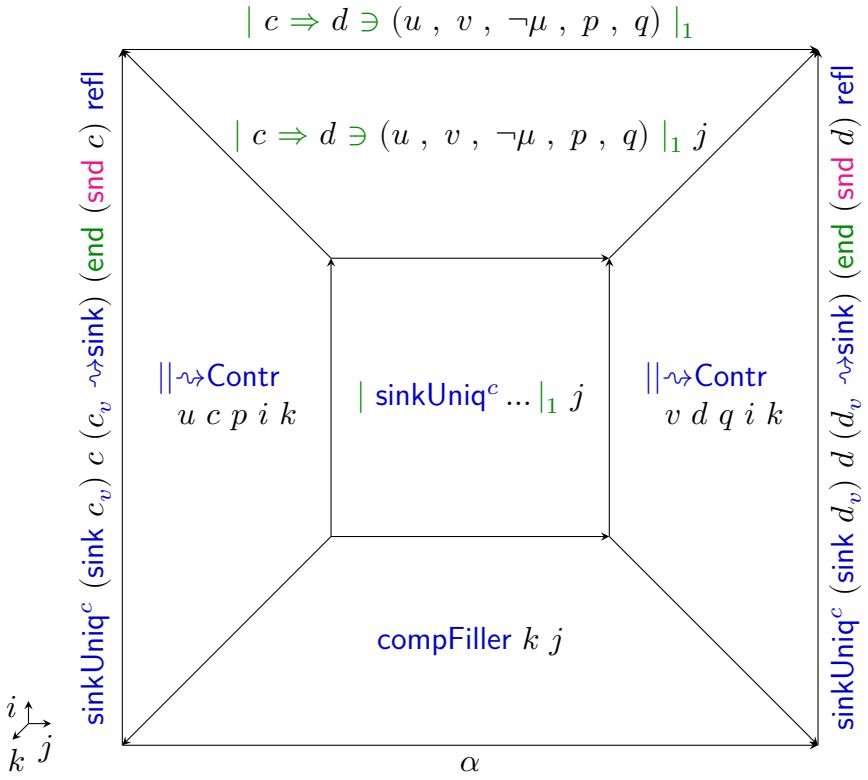


Figure 3.7: Inclusion is right inverse to collapsing

The bottom square of the cube can be filled by unfolding the path composition of α inside the cube. Similar to unfolding the path composition in ||⇝Contr, constructing compFiller is complicated by some technicalities of Cubical Agda that we omit here. For the top side of the cube we take constantly the original critical edge. For the left and right side of the cube we can use the lemma ||⇝Contr, which states that $\phi(|\ p\ |{\rightsquigarrow})$ and $\phi(|\ q\ |{\rightsquigarrow})$ contract into $|\ c\ |_0$ and $|\ d\ |_0$, respectively. The only thing

which remains is to fill the back side of the cube, where we have to show that the original critical edge $|\ c \Rightarrow d \ni (u\ ,\ v\ ,\ \neg\mu\ ,\ p\ ,\ q)\ |_1$ and its collapsed inclusion $|\ \mathsf{sink}\ u \Rightarrow \mathsf{sink}\ v \ni (u\ ,\ v\ ,\ \neg\mu\ ,\ (u \rightsquigarrow \mathsf{sink})\ ,\ (v \rightsquigarrow \mathsf{sink}))\ |_1$ are equal. This can be done using two applications of $\mathsf{sinkUniq}^c$ and the fact that gradient paths are h-propositions. In summary, the homogeneous composition we have constructed is the following:

```
rinv (| c ⇒ d ∋ (u , v , (f , ¬μ) , p , q) |₁ j) i
  = hcomp (λ k → λ {
      (i = i0) → compFiller k j
    ; (i = i1) → | c ⇒ d ∋ (u , v , (f , ¬μ) , p , q) |₁ j
    ; (j = i0) → ||↝Contr u c p i k
    ; (j = i1) → ||↝Contr v d q i k
    }) (| sinkUniq^c (sink u) c (u ↝sink) p i ⇒ sinkUniq^c (sink v) d (v ↝sink) q i
          ∋ (u , v , (f , ¬μ) ,
            ↝IsHPropS (sink u) c (u ↝sink) p i ,
            ↝IsHPropS (sink v) d (v ↝sink) q i) |₁ j)
```

3.5.3 ι is a left inverse of ϕ

We now prove that if we collapse an element of the realisation of the graph into the Morse reduct and include this back into the original complex, we can connect this element to the original element:

```
linv : (x : | E |) → ι (ϕ x) ≡ x
```

In the 0-dimensional case, a vertex u is sent by ϕ to its sink and this sink is projected back by ι. We therefore have to construct a path between $|\ u\ |_0$ and $|\ \mathsf{sink}\ u\ |_0$, which is given by the realisation of the gradient path from u:

```
linv (| u |₀) = sym (| u ↝sink |↝)
```

For an edge in the graph we do a case distinction on whether the edge is critical or matched. If the edge is critical, it still exists in the reduct and is just projected back, which means that the goal we have to prove is a square which corresponds to a rearrangement of three paths as introduced on page 43, such that we can directly prove this statement as follows:

linv (| $u \Rightarrow v \ni f$ |$_1$ j) with $u \Rightarrow^m ?$ v
... | no $\neg\mu$ = pqr^{-1}r≡pq (sym | u ↝sink |↝) | $u \Rightarrow v \ni f$ |$_1$ (sym | v ↝sink |↝) j

If the edge f is matched, it is removed in the Morse complex and its endpoints u and v of f are identified, including this identification back into the graph leads to the cell | sinkUniq (sink u) (sink v) (mp v (f , μ) (u ↝sink)) j |$_0$. We therefore have to prove an equality between this path and | $u \Rightarrow v \ni f$ |$_1$. The other sides of the resulting square stem from the action of linv on | u |$_0$ and | v |$_0$, which is sym | u ↝sink |↝ and sym | v ↝sink |↝, respectively. Once more, we obtain this square using homogeneous composition of an appropriate open cube, which is depicted in Figure 3.8.

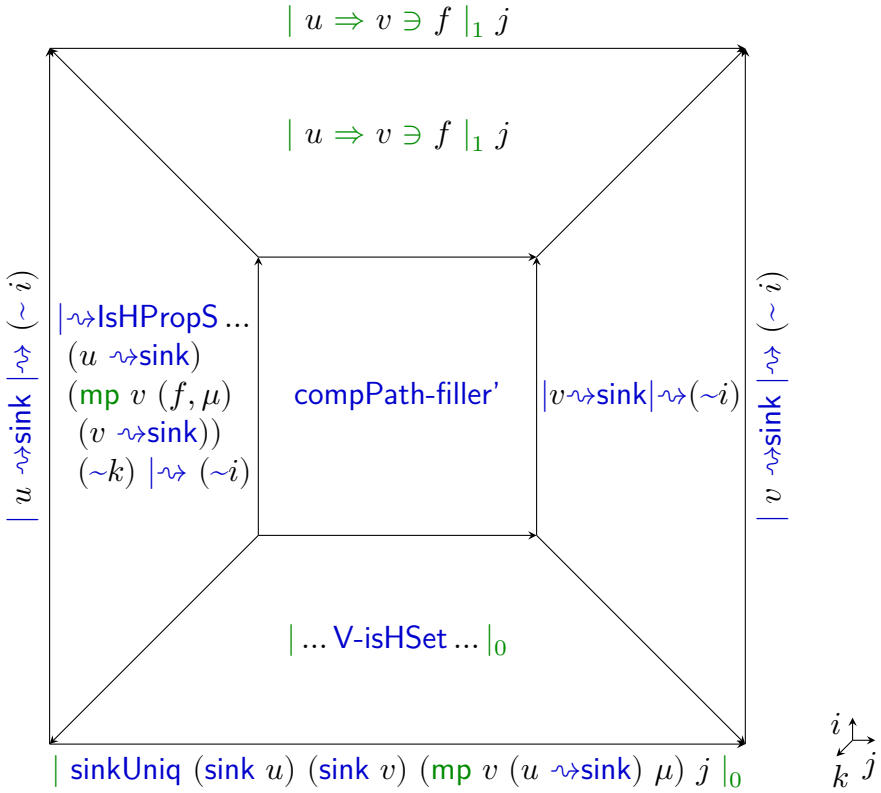


Figure 3.8: Inclusion is a left inverse to collapsing

The crucial insight that we use to construct this open cube is that gradient paths are h-propositions, which means that we have a square filler between the two gradient paths u ↝sink and mp v (f , μ) (v ↝sink) as they start at the same vertex u. We lift this square filler to the realisation of the graph and use it for the left side of the cube. The left boundary of the back square of the cube is then | mp v (f , μ) (v ↝sink) |$_1$, which by the definition of realisation of gradient paths

reduces to $\mid u \Rightarrow v \ni f \mid_1 \cdot \mathsf{sym} \mid v \rightsquigarrow\mathsf{sink} \mid\!\rightsquigarrow$. We unfold this composition of paths in the back side of the cube, which means that the top and right square can be filled by constantly stretching the goal boundary into the cube. To fill the bottom square we observe that it only has 0-dimensional cells on its boundary, so we can use that vertices in a graph are a set and lift the appropriate square to the realisation. In summary, we perform the following homogeneous composition:

```
... | yes μ = λ i → hcomp (λ k → λ {
  (i = i0) → | isHSet→SquareP (λ i j → V-isHSet)
        (λ _ → (sink v)ᵥ) ρ (sym ρ) (λ _ → (sink v)ᵥ) k j |₀
  ; (i = i1) → | u ⇒ v ∋ f |₁ j
  ; (j = i0) → | ⇝IsHPropS (sink u) (sink v)
        (u ⇝sink) (mp v (f , μ) (v ⇝sink)) (~ k) |⇝ (~ i)
  ; (j = i1) → | v ⇝sink |⇝ (~ i)
  }) (compPath-filler' | u ⇒ v ∋ f |₁ | v ⇝sink |⇝ (~ j) (~ i))
  where
  ρ : sink u ᵥ ≡ sink v ᵥ
  ρ = sinkUniq ((sink u) ᵥ) ((sink v) ᵥ) (u ⇝sink) (mp v (f , μ) (v ⇝sink))
```

3.6 The Morse theorem for graphs

We can now establish the main result of this chapter. Lifting the isomorphism constructed in Section 3.5 with univalence, we get that the realisation of a graph and its Morse reduct are equal (up to homotopy type):

```
MorseTheorem : | E | ≡ | crit₁ |
MorseTheorem = isoToPath (iso ϕ ι rinv linv)
```

We can also view the Morse reduction as an algorithm which computes a reduct from a graph. The specification of this algorithm is that the reduct is homotopy equivalent. Our result can hence be expressed in the correct by construction paradigm as follows:

```
Morse-Program : (G : Graph V E) (μ : Matching G) →
  Σ[ c₀ ∈ Type ℓ ] Σ[ c₁ ∈ (c₀ → c₀ → Type ℓ) ] | E | ≡ | c₁ |
Morse-Program G μ = crit , crit₁ , MorseTheorem
```

We now illustrate the Morse theorem by verifying properties of some graphs.

Example 3.7. The graph two from Example 3.3 is equal to the circle S^1, which means that it is a groupoid and that its first cohomology group is the integers, while all higher cohomology groups are trivial.

```
|two₁|-Groupoid : isGroupoid | two₁ |
|two₁|-Groupoid = subst isGroupoid
  (S¹≡|⊤₁| ▪▪ ⊤₁≡Mtwo ▪▪ sym MorseTheorem) isGroupoidS¹

H¹-two≅ℤ : GroupIso (coHomGr 1 (| two₁ |)) ℤGroup
H¹-two≅ℤ = subst (λ A → GroupIso (coHomGr 1 A) ℤGroup)
  (S¹≡|⊤₁| ▪▪ ⊤₁≡Mtwo ▪▪ sym MorseTheorem) H¹-S¹≅ℤ

Hⁿ⁺²-two≅0 : (n : ℕ) → coHomGr (2 + n) | two₁ | ≅ ⊤Group
Hⁿ⁺²-two≅0 n = subst (λ A → GroupIso (coHomGr (2 + n) A) ⊤Group)
  (S¹≡|⊤₁| ▪▪ ⊤₁≡Mtwo ▪▪ sym MorseTheorem) (Hⁿ-S¹≅0 n)
```

△

Example 3.8. The infinite graph line from Example 3.4 is contractible.

```
line-isContr : isContr | line₁ |
line-isContr = subst isContr (⊤≡|⊥₁| ▪▪ ⊥₁≡Mline ▪▪ sym MorseTheorem) isContrUnit
```

Contractible types are thought to be the "trivial" types which have no structure as far as homotopy theory is concerned. We can hence conclude our study of the homotopy type of line. △

Example 3.9. We close with computing the cohomology groups of the example introduced on page 36 in the introduction of this chapter. We take as vertices of the graph the finite type with 7 inhabitants and rename its inhabitants to a, ... , g. For every edge in the graph we assign the unit type with exp_1, and the empty type otherwise. The matching on the graph is captured with exp-μ:

```
exp₁ : Fin 7 → Fin 7 → Type
exp₁ a b = ⊤
exp₁ b c = ⊤
exp₁ b d = ⊤
exp₁ c d = ⊤
exp₁ e d = ⊤
exp₁ f e = ⊤
exp₁ f g = ⊤
exp₁ g d = ⊤
```

```
exp-μ : Fin 7 → Fin 7
exp-μ a = b
exp-μ b = d
exp-μ c = d
exp-μ d = d
exp-μ e = d
exp-μ f = g
exp-μ g = d
```

Showing that exp_1 gives rise to a graph exp is straightforward using that Fin is discrete as proved in the Cubical library. The acyclicity of exp-μ can also be proved straightforwardly by listing all gradient paths.

Showing that the Morse reduct of exp is isomorphic to Bool_1 via isoToReal requires lengthy case distinctions to establish that there is only a single critical cell (d , refl) with 1-cells b , c , (tt , d≠c) , snd (exp-μ-acy b) , snd (exp-μ-acy c) and f , e , (tt , g≠e) , snd (exp-μ-acy f) , snd (exp-μ-acy e). Using this simplification of the Morse complex, called Bool_1≡Mexp, and results about cohomology groups established in the Cubical library, we can prove that the graph has $\mathbb{Z} \times \mathbb{Z}$ as the first cohomology group, corresponding to the two holes in the graph:

```
H¹-exp≅ℤ×ℤ : coHomGr 1 | exp₁ | ≅ ℤGroup ×G ℤGroup
H¹-exp≅ℤ×ℤ = subst (λ A → coHomGr 1 A ≅ ℤGroup ×G ℤGroup)
  (S¹VS¹≡|Bool₁| ▪▪ Bool₁≡Mexp ▪▪ sym MorseTheorem)
  (Hⁿ-V S¹▪ S¹▪ 0 ▪≅ H¹-S¹≅ℤ ×≅ H¹-S¹≅ℤ)
```

Similarly, using also that the trivial group is a unit with respect to product, we can establish that all higher cohomology groups of the graph are trivial:

```
Hⁿ⁺²-exp≅0 : (n : ℕ) → coHomGr (2 + n) | exp₁ | ≅ ⊤Group
Hⁿ⁺²-exp≅0 n = subst (λ A → coHomGr ((2 + n)) A ≅ ⊤Group)
  (S¹VS¹≡|Bool₁| ▪▪ Bool₁≡Mexp ▪▪ sym MorseTheorem)
  (Hⁿ-V S¹▪ S¹▪ (suc n) ▪≅ Hⁿ-S¹≅0 n ×≅ Hⁿ-S¹≅0 n ▪≅ I⊤GroupIso)
```

△

3.7 Computing the reduct explicitly

So far, we have established properties of spaces using reductions to simpler descriptions of the same spaces—a topologist might say that we have "computed" these properties. We still had to manually prove that the Morse reduct of a graph is equal to some more succinct description of the same space—e.g., in Example 3.9, we had to perform a lengthy case distinction in $\mathsf{Bool}_1{\equiv}\mathsf{Mexp}$ to show that the Morse reduct of exp is equal to Bool_1. In this section, we use normalisation to compute the Morse reduct automatically, setting the groundwork for carrying out proofs such as the one for $\mathsf{Bool}_1{\equiv}\mathsf{Mexp}$ automatically. In the language of Chapter 2, we want to compute the object that is defined in abstract by crit and crit_1. Conveniently, these types are just (iterations of) dependent pair types and h-propositions, such that we can use the results from Section 2.1.2. To be able to compute with graphs, we need to be able to list its vertices, so assume in the following that we have a Graph V E such that V can be listed, and Vs is a set listing its vertices.

We can quickly establish that crit is decidable, denoted crit?, since the vertices in any graph have decidable equality. This allows us to construct the set crit for a listed graph using setΣ, which is the closure of sets under Σ, and setProp, which shows that decidable h-propositions are sets:

```
compCrit : set {C = crit} _
compCrit = setΣ Vs (λ v → setProp (isCritIsHProp v) (v crit?))
```

To show that the critical edges in the Morse reduct are listable, we have to assume that the edge relation is decidable in order to construct $\Rightarrow^c?$, a decider for whether two vertices are connected by a critical edge. Gradient paths are always decidable using discreteness of V. Again, we just need to use our lemmas developed in Section 2.1.2 for h-propositions and dependent pairs to get a listing of the gradient paths between any two critical cells:

```
compGrad : (c d : crit) → set {C = crit₁ c d} _
compGrad c d = setΣ Vs λ u → setΣ Vs λ v →
  setΣ (setProp (⇒ᶜIsHProp {u} {v}) (u ⇒ᶜ? v))
  (λ _ → setΣ (setProp (⇝IsHProp {u} c) (u ⇝? (c ᵥ)))
  (λ _ → setProp (⇝IsHProp {v} d) (v ⇝? (d ᵥ))))
```

For all the examples discussed in this chapter, the base types of the graphs have been shown to be Listable in Section 2.1.2. We can hence compute all the Morse reducts explicitly.

Example 3.10. We already argued in Section 3.4 that the Morse reduct of two from Example 3.3 has a single cell with a loop around it. We can inspect the set of critical cells by computing the following, where we use the first projection to erase the proof that this list is ordered:

```
normalise: fst (compCrit setBool two twoM)
> (true , refl) :: []
```

We therefore know that all gradient paths in the Morse reduct must start and end at the single critical vertex true. After establishing with $\mathsf{two}_1?$ that the edge relation of two is decidable, we can inspect the set of gradient paths by computing the following (we have omitted a lengthy proof term which witnesses that true is different from false with [...]):

```
normalise: fst (compGrad setBool two twoM two₁? (true , refl) (true , refl))
> (true , false , (tt , [...]) , end refl refl , mp true (tt , refl)
(end refl refl)) :: []
```

By inspecting the gradient path, we can see that it is induced by the edge from true to false, with the trivial path from true to itself as the left connecting path and the path going back to true as the right connecting path. △

Example 3.11. For the infinite graph line from Example 3.4, we can only compute the critical cells for a fixed number of cells; for instance, we can inspect with the following computation that there is a single critical cell within the first 100 vertices:

```
normalise: fst (compCrit (setℕ 100) line lineM)
> (0 , refl) :: []
```

Since (0 , refl) is the only critical cell in the Morse reduct, we again only need to inspect the gradient paths starting and ending at this critical cell, which we can do after deciding the edge relation with $\mathsf{line}_1?$. Normalisation then tells us that there are no gradient paths:

```
normalise: fst (compGrad (setℕ 100) line lineM line₁? (0 , refl) (0 , refl))
> []
```

△

Example 3.12. For the graph exp from Example 3.9, we again have a single critical cell:

```
normalise: fst (compCrit (setFin 7) exp expM)
>  (d , refl) :: []
```

This time, we have two gradient paths, which we retrieve with the following computation (using that the edge relation of the graph is decidable with $\exp_1?$):

```
normalise: fst (compGrad (setFin 7) exp expM exp₁? (d,refl) (d,refl))
>  (b , c , (tt , [...]) , mp d (tt , refl) (end refl refl) , mp d (tt
, refl) (end refl refl)) :: (f , e , (tt , [...]) , mp g (tt , refl) (mp
d (tt , refl) (end refl refl)) , mp d (tt , refl) (end refl refl)) :: []
```

△

3.8 Towards verified topological data analysis

We have proved in Cubical Agda the main result of DMT for directed graphs. Given a graph and an acyclic partial matching on that graph, its Morse reduct is homotopy equivalent to the original graph. This work is crucial towards an implementation of a TDA pipeline which is *correct by construction* since the Morse reduct is in general significantly smaller, making it feasible to compute topological invariants also for large datasets. We have therefore formalised step 3 of a typical TDA pipeline as depicted in Figure 3.9.

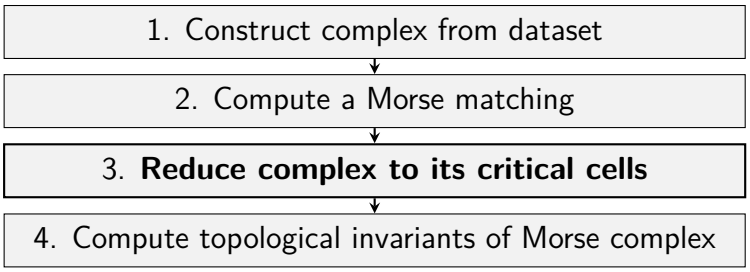


Figure 3.9: Pipeline for topological data analysis

In the following, we review what remains to be done to implement the full pipeline in Cubical Agda in Section 3.8.1, and how one can vary the type of complex associated with the dataset to obtain a more fine-grained study of the dataset in Section 3.8.2.

3.8.1 Implementing a tool for topological data analysis

Step 1 of the pipeline in Figure 3.9 can be carried out straightforwardly in any dependent type theory. As discussed in Section 3.1, there are different ways in which one can associate a topological space with a dataset, the most common ones being the Vietoris-Rips and the Čech complexes. Following the approach in Section 2, the type of a function implementing the topologisation has to ensure that all and only the necessary cells are present in the resulting complex.

In this chapter we assumed a given Morse matching along which we can reduce a graph, but a TDA tool has to automatically generate appropriate matchings for a complex, which is step 2 in the pipeline. Since computing optimal Morse matchings, i.e., matchings which lead to the smallest possible reducts, is NP-hard [47], many TDA tools use heuristics. It will be interesting to implement and compare several heuristics, where we will have to prove that each heuristic only yields admissible matchings, i.e., those which satisfy the acyclicity condition. We can take inspiration from Heras, Poza, and Rubio [36] who verify the correctness of an algorithm generating Morse matchings. The verification is carried out by first constructing the algorithm in Haskell and then manually translating the algorithm to Coq. We will attempt to formalise the algorithm directly in Agda, and then extract a compiled Haskell program for efficient execution.

While our verification of step 3 is fully constructive, we had to prove the equivalence of the Morse reduct to a smaller reduct by explicitly refuting all absurd cases. The explicit computation of the Morse reduct in Section 3.7 allows us to carry out such proofs automatically using Agda's reflection capabilities.

Finally, we had to manually establish the cohomology groups of concrete graphs in this chapter, while in a tool, we want to carry out step 4 automatically. Synthetic cohomology as proposed by Cavallo [14] and Brunerie [9] has not been studied in-depth from a computational point of view; in practice it might be expedient to take inspiration from some more traditional, computationally benign accounts of (co)homology.

The following works cannot be used directly for implementing a TDA tool in a synthetic way, but might contain relevant insights for such a project. Poza et al. [68] formalise in Coq that Morse reductions preserve homology, thereby establishing the correctness of Morse reductions for a specific algebraic invariant. The formalisation is

based on a connection between homological perturbation theory and DMT [71], which is also at the heart of the Kenzo tool [21], a computer algebra system tailored to the computation of homology and homotopy groups. Heras et al. [35] compute homology of simplicial complexes by performing certain linear transformations directly on the incidence matrices of the complexes.

3.8.2 Using more nuanced complexes

Once the basic pipeline for 1-dimensional complexes works, it will be expedient to associate more nuanced complexes with the dataset and generalise the pipeline to work on such complexes.

In order to make TDA more robust to noise in the data, it is common to construct a *filtered* simplicial complex by varying the distance parameter ϵ, leading to the idea of persistent homology [59]. Heras et al. [34] have developed an account of persistent homology in Coq, implementing an algorithm to compute Betti numbers, which are the main topological invariant of filtered simplicial complexes. They formalise the fundamental lemma of persistent homology, which states that barcodes, a representation of homology groups, encode all information of homology groups [22]. Based on this formalisation, it should be relatively straightforward to generalise the pipeline in Cubical Agda to filtered complexes.

In order to have a more fine-grained topological understanding of the dataset, it is necessary to associate higher-dimensional complexes with the dataset. Carrying out such a generalisation requires the most work with Cubical Agda's logic for homotopy types.

It will be interesting to generalise our MorseTheorem to 2-dimensional cubical complexes, which are used in [70] to study the topology of grayscale images. This generalisation should be possible with the methods used in this chapter, employing at times higher-dimensional homogeneous compositions.

It is unclear to the author if this work can be extended to complexes of arbitrary dimension (simplicial or cubical). While it has been notoriously difficult to internalise simplicial types with higher homotopical structure in homotopy type theory [49], abstract simplicial complexes from Definition 3.1 can be defined straightforwardly in dependent type theory since they are simply combinatorial descriptions of spaces without any inherent homotopical structure. In the framework developed in Chapter 2, we can define abstract simplicial complexes as follows:

```
record SimplicialComplex (Cs : Listable C) : Type (ℓ-suc ℓ) where
  constructor simpcomp
    cells : set listFamily
    closed : (X : set Cs) → X F.∈ cells → (Y : set Cs) → Y ⊆ X → Y F.∈ cells
```

Instead of having a base set V of vertices as in Definition 3.1, the definition in Agda uses the listable type C as the underlying collection from which the cells can be built. The field cells is then a family of sets over this background type, and closed ensures that the cells are closed under taking subsets. We can use the subset relation to define a face relation $\succcurlyeq$, which is the reflexive and transitive closure of $\succ$ from Definition 3.1:

```
cell : Type ℓ
cell = Σ[ X ∈ set Cs ] (X F.∈ cells)

_≽_ : cell → cell → Type ℓ
X ≽ Y = (fst Y) ⊆ (fst X)
```

For example, we can generate the standard simplicial complex for an arbitrary dimension n using the powerset function from the previous Chapter 2. The specification of the powerset functions allows us to quickly establish that the powerset does indeed produce an abstract simplicial complex:

```
Δ : ℕ → SimplicialComplex listℕ
Δ n = simpcomp cells closed where
  ps = Powerset (setℕ n)
  cells = fst ps

  corr : (X : set listℕ) → X F.∈ cells → X ⊆ setℕ n
  corr X = inv (snd ps X)

  compl : (X : set listℕ) → X ⊆ setℕ n → X F.∈ cells
  compl (X) = fun (snd ps X)

  closed : (X : set listℕ) → X F.∈ cells → (Y : set listℕ) → Y ⊆ X → Y F.∈ cells
  closed X X∈ps Y Y⊆X = compl Y (is-trans Y X (setℕ n) Y⊆X (corr X X∈ps))
```

When generalising discrete Morse theory for such arbitrary simplicial complexes, the difficulty lies in getting a good grip on their homotopy types and keeping the proofs manageable. The author has explored two ways in which that might be possible, neither of which has been fruitful.

The first attempt has been based on the works of Buchholtz and Hou (Favonia) [11] who have developed cohomology for arbitrary cell complexes. Cell complexes are defined by gluing together n-spheres with the Pushout type introduced in Section 3.2. Since cell complexes are more general than simplicial complexes, we could in principle use this to study the homotopy types of simplicial complexes of an arbitrary dimension. However, it is tricky to reason about the resulting HITs as they give rise to many more paths than the original combinatorial description would suggest. Instead, it might be necessary to further develop other tools developed in the context of homotopy type theory, e.g., from Swan [74] and Wärn [81], to reason about arbitrary simplicial complexes.

An alternative is to study simplicial complexes via their face posets [69]. The face poset (as well as an enriched version of it) is used crucially by Nanda [62] to establish the equivalence between a CW complex and its Morse reduct. We can easily construct the face poset for our simplicial complexes using that the subset relation is a partial order:

```
FacePoset : SimplicialComplex Cs → Poset _ _
FacePoset K = cell , posetstr _≻_ ≻-isPoset
```

It is not known to the author how to turn this poset into the appropriate homotopy type. This would allow us to reason about arbitrary simplicial complexes, paving the way for a formalisation of discrete Morse theory as presented by Nanda, Tamaki, and Tanaka [63] and Nanda [62].

Chapter 4

Automating reasoning in cubical type theory

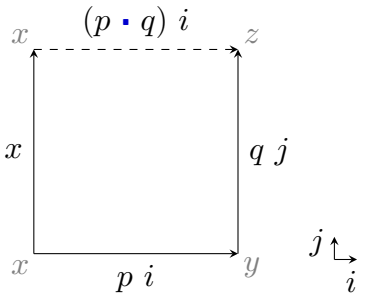
In this and the following Chapter 5, we present a novel solver which automates an important class of equality proofs in cubical type theories (CTT) such as Cubical Agda. In CTT, equational reasoning is carried out in the *path type*, which is inspired by a combinatorial model of (homotopy types of) topological spaces, namely cubical sets. An equality in a type A is hence a *path*, i.e., a function $p : \mathsf{I} \to A$ out of an abstract interval type I into A. The values of p at the endpoints $\mathsf{i0}$ and $\mathsf{i1}$ of I correspond to the elements of A which are being equated, i.e., if $p\ \mathsf{i0} = x$ and $p\ \mathsf{i1} = y$ then p equates elements $x, y : A$, denoted $p : x \equiv y$. Using the topological intuition, we sometimes call $x \equiv y$ the *boundary* of the *cell* p. In CTT, an equality proof is therefore akin to a cubical cell, and the proposition that the proof establishes is the boundary of this cubical cell.

In summary, equality problems in Cubical Agda amount to *boundary problems*: we are given a boundary in a type and try to construct a term with that boundary. We study two main methods to do this in Cubical Agda.

First, we can use that the interval type I is endowed with the structure of a De Morgan algebra, which means that we can apply certain operators to the interval variables in the context. For example, suppose that we are faced with the boundary problem $y \equiv x$, i.e., we want to construct a path from y to x. We can use the reversal operator $\sim$ to turn p into a term which solves this boundary problem, i.e., $\lambda\ i \to p\ (\sim i) : y \equiv x$ is the required term. We call these kinds of transformations of interval variables *contortions* in the following.

Second, we can use the reasoning principle of homogeneous composition, which is captured with the primitive hcomp in Cubical Agda. Intuitively, this composition principle gives the missing side of an open cube. For example, suppose that we have

another path $q : y \equiv z$ in our type A and we are faced with the boundary problem $x \equiv z$. We can fit p and q into an open 2-dimensional cube, i.e., a square, by putting p on the bottom and q on the right side of the square (note that the right endpoint of p and the left endpoint of q agree). As the remaining left side we can stay constantly at x, the left endpoint of p. Performing homogeneous composition on this open square will then give us a term with the required boundary x and $q\ \mathsf{i1} = z$, depicted with a dashed line below:



In Cubical Agda, the construction described above looks as follows, where the first argument of hcomp represents the adjacent sides, and the second argument the back side of the cube that we are composing:

```
_▪_ : x ≡ y → y ≡ z → x ≡ z
(p ▪ q) i = hcomp (λ j → λ {
      (i = i0) → x
    ; (i = i1) → q j
    }) (p i)
```

Solving boundary problems can be quite tricky in practice, in particular as the dimension of the problem grows. For example, showing that path composition as defined with _▪_ is associative—which amounts to filling a 2-dimensional boundary—requires several applications of homogeneous composition on three-dimensional open cubes, which are intricate to devise by hand. The purpose of the remainder of this thesis is to devise algorithms which can automatically solve such boundary problems. In particular, we will be able to derive automatically the mentioned associativity proof with our methods.

In this chapter, we will focus on the second kind of reasoning principle: we generalise Cubical Agda's homogeneous composition to a principle we call *Kan filling* after Kan [48], who came up with this notion of composition when devising cubical sets.

Our notion of Kan filling is the most basic principle for filling cubes and is shared by all CTTs (with the exception of the filling operator of Cohen et al. [16], but the difference is superficial and it is easy to simulate our filling operator in their theory).

The different proposed cubical type theories also vary in what kind of contortions they support, i.e., with which kind of structure the interval $\mathbb{I}$ is endowed. We give a general syntax which supports a large class of different *contortion theories* such that our methods are applicable to all variations of CTT which are currently being investigated. While including or omitting certain operators does not seem to affect the fundamental expressivity of CTT—we know of no boundary problem that can only be solved in the presence of reversals or connections—they do make a significant practical difference, providing concise terms for problems that are more laborious to prove using only Kan filling (we will discuss these aspects in more detail in the next Chapter 5, where we also refine the solver based on the Dedekind contortion theory, and discuss how the syntax introduced in this chapter relates to Cubical Agda).

In this chapter, we carry out the first systematic study of automated reasoning in CTT. In Section 4.1 we single out the sublanguage of CTT relevant for the above mentioned cubical reasoning principles. This allows us in Section 4.2 to classify the boundary problems corresponding to different cubical type theories. Then in Section 4.3 we develop a practical algorithm which can resolve many proof goals occurring in practice.

4.1 A syntax for cubes

For our solver, we are interested in solving only boundary problems that concern a single type, and now develop syntax which allows us to specify its higher homotopical structure. This higher-dimensional structure is captured with *interval variables* which make it possible to talk about the n-cube cells in a type. This allows us to state and prove theorems about the paths in a type. In brief, an n-cell in a type is represented as a term varying in n interval variables, just as an n-cube in a topological space X can be represented as a continuous map from the unit n-cube. Contortions are then reparameterizations of an n-cube to an m-cube, and applying a contortion to an n-cell gives rise to an m-cell with a slightly contorted boundary. We will in the following give a term language which captures this topological intuition purely syntactically.

Following the usual type-theoretic methodology, our syntax consists of a set of *judgment forms*, the forms of assertion we can make in the theory, and *inference rules* which explain how we can deduce that a judgment holds. We can break down

the syntax into two layers: the *contortion theory* introduced in Section 4.1.1, which axiomatizes the topological unit n-cubes and certain maps between them, and the *term theory* introduced in Section 4.1.2 on top of it, which axiomatizes maps from these cubes into a fixed space.

The contortion theory itself has an inner and outer layer: we make a distinction between the basic *substitutions*, which are those contortions needed to get the theory off the ground, and the more general *contortions*, which include the substitutions but can be extended with operators such as connections and reversals. The syntactic separation of substitutions from general contortions allows us to consider extended contortion theories without complicating the Kan filling operator.

The term theory also has two layers, an inner *contortion term theory* and an outer *Kan term theory* which will be introduced in Section 4.1.3. In the contortion term theory, we have the notion of a *cell context*, which enumerates the hypotheses of a boundary problem in the form of a sequence of cells possibly with specified boundaries, and of a *contortion term*, which is a cell obtained by applying some contortion to a cell from the context. Finally, the *Kan terms* are those formed by iterated application of the Kan filling operator to contortion terms.

We collect the judgment forms of the syntax in Figures 4.1 and 4.2; we explain their intended meaning and introduce their inference rules in the following sections.

$\Psi\ \mathsf{dctxt}$	Cube context
$\Psi \vdash r\ \mathsf{dim}$	Atomic interval term
$\xi \colon \Psi' \to \Psi$	Substitution
$\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim}$	Contortion interval term
$\sigma \colon \Psi' \rightsquigarrow \Psi$	Contortion

Figure 4.1: Judgment forms for the contortion theory.

$\Gamma\ \mathsf{ctxt}$	Cell context
$\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}$	Contortion boundary
$\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$	Kan boundary
$\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm}$	Contortion term
$\Gamma \mid \Psi \vdash_{\mathrm{k}} t\ \mathsf{tm}$	Kan term
$\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$	Contortion term with boundary
$\Gamma \mid \Psi \vdash_{\mathrm{k}} t : [\,\phi\,]$	Kan term with boundary

Figure 4.2: Judgment forms for the term theory.

4.1.1 Contortion theory

Cube contexts: Ψ dctxt

$$\frac{}{() \text{ dctxt}} \qquad \frac{}{\emptyset \text{ dctxt}} \qquad \frac{\Psi \text{ dctxt}}{\Psi, i \text{ dctxt}} \qquad \frac{\Psi \text{ dctxt} \qquad \Psi \vdash r \text{ dim} \qquad e \in \{\mathbf{0}, \mathbf{1}\}}{\Psi[r{=}e] \text{ dctxt}}$$

Equations:

$$\frac{e \in \{\mathbf{0}, \mathbf{1}\}}{\Psi[e{=}e] = \Psi \text{ dctxt}} \qquad \frac{e \in \{\mathbf{0}, \mathbf{1}\}}{\Psi[\overline{e}{=}e] = \emptyset \text{ dctxt}} \qquad \frac{e \in \{\mathbf{0}, \mathbf{1}\}}{(\Psi, i)[i{=}e] = \Psi \text{ dctxt}}$$

$$\frac{\Psi \vdash r \text{ dim}}{(\Psi, i)[r{=}e] = \Psi[r{=}e], i \text{ dctxt}}$$

Figure 4.3: Inference rules for cube contexts

A *cube context* as defined in Figure 4.3 represents a unit n-cube by a list of *interval variables*, which we can think of as names for its axes. For example, the cube context $()$ with no variables represents the unit 0-cube (i.e., a point), while the cube context (i, j, k) represents a 3-cube with axes labeled i, j, and k. In addition to these cubes, there is also the cube context $\emptyset$, which represents the empty space. Finally, we can apply a *constraint* $r{=}e$ to a cube context Ψ to obtain a new context $\Psi[r{=}e]$; we introduce this operation later on.

Atomic interval terms: $\Psi \vdash r$ dim

$$\frac{e \in \{\mathbf{0}, \mathbf{1}\}}{\Psi \vdash e \text{ dim}} \qquad \frac{\Psi \text{ dctxt}}{\Psi, i \vdash i \text{ dim}} \qquad \frac{\Psi \vdash r \text{ dim}}{\Psi, i \vdash r \text{ dim}}$$

Figure 4.4: Inference rules for atomic interval terms

As defined in Figure 4.4, an *atomic interval term* $\Psi \vdash r$ dim in a cube context Ψ represents a map from Ψ to the unit interval. The atomic interval terms are always either variables, such as $i, j, k \vdash j$ dim, which represent the projection of a single coordinate, or are constant at one of the endpoints ($\mathbf{0}$ or $\mathbf{1}$) of the unit interval, such as $i, j, k \vdash \mathbf{0}$ dim. Going forward we use the letter e to stand for any arbitrary endpoint, $\mathbf{0}$ or $\mathbf{1}$, and write $\overline{e}$ for the opposite endpoint (so $\overline{\mathbf{0}} := \mathbf{1}$ and $\overline{\mathbf{1}} := \mathbf{0}$).

Substitutions: $\xi \colon \Psi' \to \Psi$

$$\frac{}{\mathbf{id}_\Psi \colon \Psi \to \Psi} \qquad \frac{\xi \colon \Psi' \to \Psi \qquad \xi' \colon \Psi'' \to \Psi'}{\xi \circ \xi' \colon \Psi'' \to \Psi} \qquad \frac{}{!_\Psi \colon \emptyset \to \Psi}$$

$$\frac{\xi \colon \Psi' \to \Psi \qquad \Psi' \vdash r\ \mathsf{dim}}{(\xi, i \mapsto r) \colon \Psi' \to (\Psi, i)} \qquad \frac{}{\mathbf{s}_i \colon (\Psi, i) \to \Psi} \qquad \frac{\Psi \vdash r\ \mathsf{dim} \qquad e \in \{\mathbf{0}, \mathbf{1}\}}{r{=}e \colon \Psi[r{=}e] \to \Psi}$$

Equations:

$$\mathbf{id}_\Psi \circ \xi = \xi \qquad \xi \circ \mathbf{id}_\Psi = \xi \qquad \xi \circ (\xi' \circ \xi'') = (\xi \circ \xi') \circ \xi'' \qquad \mathbf{s}_i \circ (\xi, i \mapsto r) = \xi$$

$$\frac{\xi \colon \Psi' \to (\Psi, i)}{\xi = (\mathbf{s}_i \circ \xi, r[\xi]) \colon \Psi' \to (\Psi, i)} \qquad \frac{\xi \colon \emptyset \to \Psi}{\xi = !_\Psi \colon \emptyset \to \Psi} \qquad \frac{e \in \{\mathbf{0}, \mathbf{1}\}}{(e{=}e) = \mathbf{id}_\Psi \colon \Psi \to \Psi}$$

$$\frac{e \in \{\mathbf{0}, \mathbf{1}\}}{(i{=}e) = (\mathbf{id}_\Psi, i \mapsto e) \colon \Psi \to (\Psi, i)} \qquad \frac{\Psi \vdash r\ \mathsf{dim}}{(r{=}e) = (r{=}e, i \mapsto i) \colon (\Psi[r{=}e], i) \to (\Psi, i)}$$

Figure 4.5: Inference rules for substitutions

As defined in Figure 4.5, substitutions $\xi \colon \Psi' \to \Psi$ are maps from the cube Ψ' to the cube Ψ whose components are atomic interval terms. There is an identity substitution $\mathbf{id}_\Psi \colon \Psi \to \Psi$ from any cube context to itself and a unique substitution $!_\Psi \colon \emptyset \to \Psi$ from the empty space into any other context; we can compose substitutions with the $\circ$ operator. The remaining basic substitutions are the *faces* $(i \mapsto r)$, *degeneracies* $(\mathbf{s}_i)$ and *constraints* $(r{=}e)$.

Given a substitution $\xi \colon \Psi' \to \Psi$ and some atomic interval term $\Psi' \vdash r\ \mathsf{dim}$, we can form a new substitution $(\xi, i \mapsto r) \colon \Psi' \to (\Psi, i)$, using r to specify the value of a new coordinate. In the case that ξ is the identity substitution and we have $\Psi \vdash r\ \mathsf{dim}$, we use the abbreviated notation $(i \mapsto r) \colon \Psi \to (\Psi, i)$. We call these *face* substitutions because they represent the inclusion of one cube as a face or diagonal of a higher-dimensional cube.

Example 4.1. The face substitutions $i \mapsto \mathbf{0} \colon () \to (i)$ and $i \mapsto \mathbf{1} \colon () \to (i)$ specify the two 0-cube endpoints of the 1-cube (i). The face substitution $j \mapsto i \colon (i) \to (i, j)$ specifies the 1-cube diagonal of the 2-cube (i, j). △

The *degeneracy* substitution $\mathbf{s}_i \colon (\Psi, i) \to \Psi$ is the map from (Ψ, i) to Ψ that collapses the larger cube along its final coordinate (here labelled i). We can collapse a cube along an arbitrary coordinate by combining the degeneracy and face substitutions as in the example below.

Example 4.2. From the 3-cube (i, j, k), we have three substitutions $\mathbf{s}_k \colon (i, j, k) \to (i, j)$, $(\mathbf{s}_j \circ \mathbf{s}_k, k \mapsto k) \colon (i, j) \to (i, k)$, and $(\mathbf{s}_i \circ \mathbf{s}_j \circ \mathbf{s}_k, j \mapsto j, k \mapsto k) \colon (i, j, k) \to (j, k)$, which collapse the k-, j-, and i-axis respectively. △

Going forward, we make the slight abuse of notation of writing $\mathbf{s}_i$ for the substitution that collapses the i-axis of a cube even when i is not the final coordinate, for example writing $\mathbf{s}_j$ and $\mathbf{s}_i$ for the latter two substitutions in the preceding example.

We will also sometimes write $j \mapsto i$ for $(\mathbf{s}_i, j \mapsto i) \colon i \to j$ which renames dimension j to i in a 1-cube. Since the codomain of a substitution is usually understood, it is clear whether we mean the diagonal as in Example 4.1 or rename a single dimension variable.

The substitution judgment deserves its name in the sense that we can apply a substitution $\xi \colon \Psi' \to \Psi$ to any judgment indexed by a cube context. For example, we have a rule (omitted from Figure 4.1)

$$\frac{\Psi \vdash r\ \mathsf{dim} \qquad \xi \colon \Psi' \to \Psi}{\Psi' \vdash r[\xi]\ \mathsf{dim}}$$

expressing that we can apply a substitution ξ to an atomic interval term r to get another atomic interval term $r[\xi]$, together with equations

$$\frac{\Psi \vdash r\ \mathsf{dim}}{\Psi \vdash i[i \mapsto r] = r\ \mathsf{dim}} \qquad\qquad \frac{\Psi \vdash r\ \mathsf{dim} \qquad \Psi \vdash s\ \mathsf{dim}}{\Psi \vdash s[i \mapsto r] = s\ \mathsf{dim}}$$

computing the results of substitutions. For concision's sake we omit rules for applying and computing substitutions in our presentation of the theory, as these are entirely standard.

Constraining contexts and substitutions To specify the Kan filling operator, we will need a way of identifying the part of a cube context Ψ where a dimension term $\Psi \vdash r\ \mathsf{dim}$ takes some constant value $e \in \{\mathbf{0}, \mathbf{1}\}$. We write $\Psi[r{=}e]$ for this constrained context. Although presented in Figure 4.1 as a context constructor satisfying various equations, we can instead think of this as an *operation* on contexts defined as follows.

$$\Psi[e{=}e] := \Psi$$
$$\Psi[\overline{e}{=}e] := \emptyset$$
$$(\Psi, i, \Psi')[i{=}e] := \Psi, \Psi'$$

That is, the part of Ψ where $e = e$ is the whole of Ψ, the part of Ψ where $e = \overline{e}$ is the empty space, and the part of an $(n+1)$-cube Ψ where some variable i is e is an n-cube obtained by removing the i-axis from Ψ.

The *constraint substitution* $(r{=}e) \colon \Psi[r{=}e] \to \Psi$ is the inclusion of the constrained cube $\Psi[r{=}e]$ as the part of the cube Ψ where r evaluates to e. Like constrained cube contexts, concrete constraint substitutions always reduce to more primitive substitutions via the equations in Figure 4.1, which we can regard as the definition of an operation like so:

$$\begin{aligned}(e{=}e) &\coloneqq \mathbf{id}_\Psi \\ (\overline{e}{=}e) &\coloneqq !_\Psi \\ (i{=}e) &\coloneqq (i \mapsto e)\end{aligned}$$

Contortion interval terms: $\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim}$

$$\frac{e \in \{\mathbf{0}, \mathbf{1}\}}{\Psi \vdash_{\mathrm{c}} e\ \mathsf{dim}} \qquad \frac{\Psi\ \mathsf{dctxt}}{\Psi, i \vdash_{\mathrm{c}} i\ \mathsf{dim}} \qquad \frac{\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim}}{\Psi, i \vdash_{\mathrm{c}} r\ \mathsf{dim}} \qquad \frac{\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim} \qquad \xi \colon \Psi' \to \Psi}{\Psi' \vdash_{\mathrm{c}} r[\xi]\ \mathsf{dim}}$$

$$\frac{\Psi \vdash r\ \mathsf{dim}}{\Psi \vdash_{\mathrm{c}} \iota(r)\ \mathsf{dim}}$$

Equations:
The equations are the same as for the atomic interval term judgment $\Psi \vdash r\ \mathsf{dim}$ in Figure 4.4, with the additional equations for computing $\iota(-)$:

$$\iota(e) = e \qquad\qquad \iota(i) = i$$

Figure 4.6: Inference rules for contortion interval terms

In order to pass onto the next level of our syntax, we embed all atomic interval terms into contortion interval terms with the rules in Figure 4.6.

Finally, we have the *contortion* judgment $\sigma \colon \Psi' \nrightarrow \Psi$. The rules in Figure 4.7 constitute the *minimal* (or *cartesian*) contortion theory, which includes only substitutions: for any substitution ξ we have a corresponding contortion $\iota(\xi)$. Contortions capture maps between cubes, which suggests that we can assemble cubes and contortions into a category: call $\square$ the category which has objects cube contexts and morphisms the cartesian contortions. In Figure 4.8 we introduce several extensions to the minimal contortion theory. They all introduce additional morphisms between finite sets, and hence are wide supercategories of $\square$.

Contortions: $\psi \colon \Psi' \rightsquigarrow \Psi$

$$\frac{}{\mathbf{id}_\Psi \colon \Psi \rightsquigarrow \Psi} \qquad \frac{\psi \colon \Psi' \rightsquigarrow \Psi \qquad \psi' \colon \Psi'' \rightsquigarrow \Psi'}{\psi \circ \psi' \colon \Psi'' \rightsquigarrow \Psi} \qquad \frac{}{!_\Psi \colon \emptyset \rightsquigarrow \Psi}$$

$$\frac{\psi \colon \Psi' \rightsquigarrow \Psi \qquad \Psi' \vdash r\ \mathsf{dim}}{(\psi, i \mapsto r) \colon \Psi' \rightsquigarrow (\Psi, i)} \qquad \frac{}{\mathbf{s}_i \colon (\Psi, i) \rightsquigarrow \Psi}$$

Equations:
The equations are the same as for the substitution judgment $\xi \colon \Psi' \to \Psi$ in Figure 4.1, with the addition of equations for computing $\iota(-)$:

$$\iota(\mathbf{id}_\Psi) = \mathbf{id}_\Psi \qquad \iota(\xi \circ \xi') = \iota(\xi) \circ \iota(\xi') \qquad \iota(\xi, i \mapsto r) = (\iota(\xi), i \mapsto \iota(r)) \qquad \iota(\mathbf{s}_i) = \mathbf{s}_i$$

Figure 4.7: Inference rules for contortions

Disjunctive contortions:

$$\frac{\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim} \qquad \Psi \vdash_{\mathrm{c}} s\ \mathsf{dim}}{\Psi \vdash_{\mathrm{c}} r \vee s\ \mathsf{dim}}$$

$$r \vee \mathbf{0} = r \qquad r \vee \mathbf{1} = \mathbf{1} \qquad (r \vee s) \vee t = r \vee (s \vee t) \qquad r \vee s = s \vee r$$

Dedekind contortions:
In addition to the rules for disjunctive contortions:

$$\frac{\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim} \qquad \Psi \vdash_{\mathrm{c}} s\ \mathsf{dim}}{\Psi \vdash_{\mathrm{c}} r \wedge s\ \mathsf{dim}}$$

$$r \wedge \mathbf{0} = \mathbf{0} \qquad r \wedge \mathbf{1} = r \qquad (r \wedge s) \wedge t = r \wedge (s \wedge t) \qquad r \wedge s = s \wedge r$$

$$r \vee (r \wedge s) = r \qquad r \wedge (r \vee s) = r \qquad r \wedge (s \vee t) = (r \wedge s) \vee (r \wedge t)$$

De Morgan contortions:
In addition to the rules for Dedekind contortions:

$$\frac{\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim}}{\Psi \vdash_{\mathrm{c}} \neg r\ \mathsf{dim}}$$

$$\neg(r \wedge s) = \neg r \vee \neg s \qquad \neg\neg r = r$$

Figure 4.8: Examples of extensions to the contortion theory

The theory of *disjunctive* contortions introduces a connection $\vee$, define $\square_\vee$ to be the category of n-cubes and disjunctive contortions . The *Dedekind* contortion theory contains another connection $\wedge$ with both connections satisfying the laws of a distributive lattice; call the corresponding category $\square_{\vee\wedge}$. The term "Dedekind" was introduced by Awodey [3] since the number of contortions over a cube Ψ with n dimensions corresponds to the n-th Dedekind number, denoted $D(n)$ in the following. The *De Morgan* contortion theory gives rise to the largest class of morphisms by having both connections $\wedge$ and $\vee$ and a reversal operation $\neg$ satisfying the axioms of a De Morgan algebra, call the resulting category $\square_{\neg\vee\wedge}$.

Cubical sets based on $\square_{\neg\vee\wedge}$ were used by Cohen et al. [16] to give the first model of univalent type theory with higher inductive types and is at the core of Cubical Agda [78]. It was later realised by Orton and Pitts [66] that univalent type theory could also be modelled with cubical sets based on a category with fewer morphisms, in particular $\square_{\vee\wedge}$. While De Morgan and Dedekind contortion theories (as well as the minimal cartesian contortion theory consisting only of substitutions) have therefore been used to provide models of univalent type theory, attempts to show that the resulting cubical sets provide a model of homotopy types of spaces have so far been unsuccessful. Only for cubical sets based on $\square_\vee$ have Cavallo and Sattler [15] been able to show that they provide a model of homotopy types.

In summary, our framework captures most cubical type theories currently studied, except that of Bezem, Coquand, and Huber [6]. Their cubical type theory is substructural, i.e., does not admit all the substitutions we have stipulated. Since it does not form the basis of any current theorem prover, we have chosen to exclude it from our study of automated reasoning in CTT.

Buchholtz and Morehouse [12] study these (and many other) *cube categories* in more detail from a categorical point of view. Many questions concerning the use of the different cube categories in a model of cubical type theory remain open; in particular, so far it has not been possible to prove that cubical type theories based on larger cube categories are *conservative* in the sense that they do not allow for filling boundaries which could not be filled in less expressive theories. We will sidestep these issues in this chapter by developing our methods to work for all relevant notions of cubes.

Example 4.3. From the 2-cube (i, j), we have a disjunctive contortion

$$(i \mapsto i \vee j) \colon (i, j) \rightsquigarrow (i)$$

which we can think of as collapsing the unit 2-cube to a 1-cube by taking the maximum of both directions i and j. This means that if either i or j is set to $\mathbf{0}$, we regain the identity j- and i-map respectively, for instance we have $(i \mapsto i \vee j) \circ (j \mapsto \mathbf{0}) = \mathbf{id}_i : (i) \nrightarrow (i)$. If either dimension is set to $\mathbf{1}$, we obtain a map between 1-cubes which is constantly $\mathbf{1}$, for instance $(i \mapsto i \vee j) \circ (j \mapsto \mathbf{1}) = (i \mapsto \mathbf{1}) \circ \mathbf{s}_i : (i) \nrightarrow (i)$.

Note that by the rules in Figure 4.8, $(i \mapsto i \vee j) = (i \mapsto j \vee i) : (i, j) \nrightarrow (i)$. △

Example 4.4. In the Dedekind theory, we also have a conjunctive contortion

$$(i \mapsto i \wedge j) : (i, j) \to (i)$$

which can be thought of as mapping the 2-cube to a 1-cube which is the minimum of both axes i and j. This means that if either i or j is set to $\mathbf{0}$, we get a map between 1-cubes which is constantly $\mathbf{0}$ such as $(i \mapsto i \wedge j) \circ (j \mapsto \mathbf{0}) = (i \mapsto \mathbf{0}) \circ \mathbf{s}_i : (i) \nrightarrow (i)$. If one of both dimension variables is $\mathbf{1}$, we retain the identity map as we can detect, e.g., as follows:

$$(i \mapsto i \vee j) \circ (j \mapsto \mathbf{1}) = \mathbf{id}_i : (i) \nrightarrow (i)$$

△

Example 4.5. In the De Morgan theory, the reversal contortion

$$(i \mapsto \neg i) : (i) \to (i)$$

can be used to exchange the endpoints of a 1-cube, which we can detect by computing $(i \mapsto \neg i) \circ (i \mapsto \mathbf{0}) = (i \mapsto \mathbf{1}) : () \nrightarrow (i)$ and $(i \mapsto \neg i) \circ (i \mapsto \mathbf{1}) = (i \mapsto \mathbf{0}) : () \nrightarrow (i)$.

△

Remark 4.6. Note that unlike substitution, *not* all judgments are stable under contortion. For example, we cannot in general apply a contortion $\sigma : \Psi' \nrightarrow \Psi$ to an atomic interval term $\Psi \vdash r\ \mathsf{dim}$ to obtain an atomic interval term over Ψ'. This makes it easier to set up a syntax which works for a range of different contortion theories.

4.1.2 Term theory

The second layer of the theory is the *term theory* as depicted in Figure 4.9. Where the contortion theory concerns maps between unit cubes, the term theory concerns maps from unit cubes into an arbitrary space. We introduce the judgments of the term theory below.

Cell contexts A cell context, as defined by the judgment $\Gamma\ \mathsf{ctxt}$, is a representation of a finite cell complex in a space, that is a finite collection of images of unit cubes in a space along with information about how their boundaries attach to each other. In our type-theoretic presentation, we represent a cell complex as a context: a list of variables giving a name to each cell. Each variable is annotated with a cube context, which tells us what dimension of cube it is, and a *contortion boundary*, which explains how it attaches to the cells that come before it.

Cell contexts: $\Gamma\ \mathsf{ctxt}$

$$\frac{}{()\ \mathsf{ctxt}} \qquad \frac{\Gamma\ \mathsf{ctxt} \qquad \Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}}{\Gamma, p : (\Psi)[\,\phi\,]\ \mathsf{ctxt}}$$

Contortion terms: $\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm}$

$$\frac{}{\Gamma, p : (\Psi)[\,\phi\,] \mid \Psi \vdash_{\mathrm{c}} p : [\,\phi\,]} \qquad \frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm} \qquad \psi \colon \Psi' \nrightarrow \Psi}{\Gamma \mid \Psi' \vdash_{\mathrm{c}} t\langle\psi\rangle\ \mathsf{tm}}$$

$$t\langle \mathbf{id}_\Psi \rangle = t \qquad t\langle \psi \circ \psi' \rangle = t\langle\psi\rangle\langle\psi'\rangle \qquad t\langle \iota(\xi) \rangle = t[\xi]$$

Contortion boundaries: $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}$

$$\frac{}{\Gamma \mid \Psi \vdash_{\mathrm{c}} ()\ \mathsf{bdy}}$$

$$\frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy} \qquad \Psi \vdash r\ \mathsf{dim} \qquad e \in \{\mathbf{0}, \mathbf{1}\} \qquad \Gamma \mid \Psi[r{=}e] \vdash_{\mathrm{c}} t : [\,\phi[r{=}e]\,]}{\Gamma \mid \Psi \vdash_{\mathrm{c}} (\phi, r{=}e \mapsto t)\ \mathsf{bdy}}$$

Contortion terms with boundary: $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$
Introduction:

$$\frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm}}{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,]} \qquad \frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,] \qquad \Psi \vdash r\ \mathsf{dim} \qquad \Gamma \mid \Psi[r{=}e] \vdash_{\mathrm{c}} t[r{=}e] = t'\ \mathsf{tm}}{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi, r{=}e \mapsto t'\,]}$$

Projection:

$$\frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi, r{=}e \mapsto t'\,]}{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]} \qquad \frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi, e{=}e \mapsto t'\,]}{\Gamma \mid \Psi \vdash_{\mathrm{c}} t = t'\ \mathsf{tm}}$$

Figure 4.9: Inference rules for cell contexts and contortion terms and boundaries

Figure 4.10 gives one example of a cell context, broken up with one variable per line for readability.

$x : ()[]$,
$p : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto x \,]$,
$q : (j)[\, j{=}\mathbf{0} \mapsto x \mid j{=}\mathbf{1} \mapsto x \,]$,
$\alpha : (i, j)[\, i{=}\mathbf{0} \mapsto q \mid i{=}\mathbf{1} \mapsto q \mid j{=}\mathbf{0} \mapsto p \mid j{=}\mathbf{1} \mapsto p \,]$

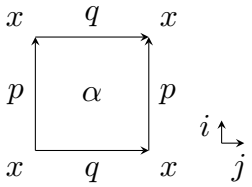


Figure 4.10: Example of a cell context representing a torus

The complex generated by this context has one base point x, i.e., a 0-cube with boundary [], and two 1-cubes p and q whose endpoints are both attached to x, respectively. This means that p and q are both loops around x. Each of these paths is abstracted over a different interval variable, which allows for directly fitting them into the square α, which is a 2-cube over the cube context i, j. We can think of α as being glued together along the p and q sides, which suggests that α is a combinatorial model of a particular topological space, namely the torus. Reasoning about the above cell context will allow us to derive properties of the homotopy type of the torus, which means that our language allows us to develop homotopy theory.

Contortion boundaries Contortion boundaries are defined by $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}$, a judgment which is parameterized by a cell context Γ and cube context Ψ; the subscript c distinguishes it from the corresponding judgment in the Kan theory to be introduced below. A boundary is a list of constraints—equations between a dimension term over Ψ and a constant—paired with *contortion terms*, which can refer to cells in Γ. This is where we first make use of constrained contexts: in an entry $r{=}e \mapsto t$ of a boundary $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}$, the term t must be well-formed in the cube context $\Psi[r{=}e]$, i.e., must be defined on the part of the cube where $r{=}e$. In Figure 4.10, for example, the entry $i{=}\mathbf{0} \mapsto q$ in the boundary of α is defined in the context $(i, j)[i{=}e] = (j)$.

The individual entries of a boundary must also form a coherent whole: entries should agree wherever they overlap. In the boundary of α in Figure 4.10, for example, it is important that the entries $i{=}\mathbf{0} \mapsto q$ and $j{=}\mathbf{0} \mapsto p$ match up on the part of the cube where both i and j are $\mathbf{0}$; in this case they are both equal to x.

We will write $\phi[i{=}e]$ for the $(i{=}e)$-th entry of a boundary ϕ in the following.

Contortion terms The terms which appear in contortion boundaries need not necessarily be variables, as they are in the example of Figure 4.10; in general, they are *contortion terms*, i.e., variables with some number of contortions applied. For

example, the cell context

$$\Gamma_0 \coloneqq \left\{ \begin{array}{l} p : (i)[\,], \\ q : (j)[\,j{=}\mathbf{0} \mapsto p\langle i \mapsto \mathbf{1}\rangle \mid j{=}\mathbf{1} \mapsto p\langle i \mapsto \mathbf{0}\rangle\,] \end{array} \right.$$

contains a line p without a specified boundary followed by a line q running from p's $\mathbf{1}$-endpoint to its $\mathbf{0}$-endpoint; we access the endpoints of p by applying the face contortions $i \mapsto \mathbf{1}$ and $i \mapsto \mathbf{0}$. As another example, the cell context

$$\Gamma_1 \coloneqq \left\{ \begin{array}{l} a : ()[\,], \\ s : (i,j)[\,i{=}\mathbf{0} \mapsto a\langle \mathbf{s}_j\rangle \mid i{=}\mathbf{1} \mapsto a\langle \mathbf{s}_j\rangle \mid j{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i\rangle \mid j{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle\,] \end{array} \right.$$

uses the degeneracy contortions to define a square s whose sides are all constantly a.

Contortion terms are defined by the judgment $\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm}$; it is also convenient to have an auxiliary judgment $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$ for contortion terms which agree with a given boundary. The variable rule tells us any variable from the cell context is a contortion term with its boundary. In the cell context Γ_0 from above, for example, we have the following:

$$\Gamma_0 \mid (j) \vdash_{\mathrm{c}} q : [\,j{=}\mathbf{0} \mapsto p\langle i \mapsto \mathbf{1}\rangle \mid j{=}\mathbf{1} \mapsto p\langle i \mapsto \mathbf{0}\rangle\,]$$

The equation $t\langle \iota(\xi)\rangle = t[\xi]$ tells us that when we apply a contortion that is actually a substitution to a term, the effect is the same as applying the substitution; we have for example $p\langle i \mapsto \mathbf{1}\rangle = p[i \mapsto \mathbf{1}]$. As a consequence of this—the stability of judgments under substitution, and the rules for terms with boundary—a variable will reduce when we apply a contortion that factors through an entry of its boundary. For example, if we apply the contortion $j \mapsto \mathbf{0}$ to the variable q above, we have the following equation.

$$\Gamma_0 \mid () \vdash_{\mathrm{c}} q\langle j \mapsto \mathbf{0}\rangle = p\langle i \mapsto \mathbf{1}\rangle\ \mathsf{tm}$$

As another example, in the context Γ_1 we have a sequence of equations as follows.

$$\Gamma_1 \mid () \vdash_{\mathrm{c}} s\langle i \mapsto \mathbf{0}, j \mapsto \mathbf{1}\rangle = a\langle \mathbf{s}_j \circ (j \mapsto \mathbf{1})\rangle = a\ \mathsf{tm}$$

Remark 4.7. A contortion or substitution applied to a variable need not reduce in general. In the example above $p\langle i \mapsto \mathbf{1}\rangle$ does not reduce, because the boundary of the variable has no entry for $i = \mathbf{1}$. In particular, it is not the case here that $p\langle i \mapsto \mathbf{1}\rangle = p$, as it might be tempting to assume from the fact that i does not literally appear in the syntax "p".

Remark 4.8. In order to compare terms for equality, it is convenient to always reduce a term if possible—i.e., to treat $p\langle i \mapsto \mathbf{1}\rangle$ as the normal form of $q\langle j \mapsto \mathbf{0}\rangle$, and a as the normal form of $s\langle i \mapsto \mathbf{0}, j \mapsto \mathbf{1}\rangle$ in the examples above. Note that we cannot have variables whose boundaries mutually refer to each other, which means that the reduction process is well-founded.

By always reducing terms along their boundary prescriptions we can represent any contortion term $\Psi \mid \Gamma \vdash_{\mathrm{c}} t\ \mathsf{tm}$ uniquely as $p\langle i_1 \mapsto r_1, \ldots, i_n \mapsto r_n\rangle$ for some variable $p : (i_1, \ldots, i_n)[\,\phi\,]$ in Γ, where each r_k is a contortion interval term $\Psi \vdash_{\mathrm{c}} r_k\ \mathsf{dim}$. The number of reductions required to normalise t is bound both by the length of Γ and by n.

Remark 4.9. The judgment $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$ means that the term t agrees with ϕ wherever ϕ is specified, even if we could establish a more specific description of the boundary of t. In particular, any term inhabits the completely unspecified boundary $[\ ]$; and in general, a term inhabits several (increasingly more specified) boundaries.

4.1.3 The Kan term theory

Kan terms: $\Gamma \mid \Psi \vdash_{\mathrm{k}} u\ \mathsf{tm}$

$$\frac{\Gamma \mid \Psi \vdash_{\mathrm{c}} t\ \mathsf{tm}}{\Gamma \mid \Psi \vdash_{\mathrm{k}} t^{\mathrm{c}}\ \mathsf{tm}} \qquad \frac{\Gamma \mid \Psi \vdash_{\mathrm{k}} u\ \mathsf{tm} \qquad \xi : \Psi' \to \Psi}{\Gamma \mid \Psi' \vdash_{\mathrm{k}} u[\xi]\ \mathsf{tm}}$$

$$\frac{e \in \{\mathbf{0}, \mathbf{1}\} \qquad \Psi \vdash r\ \mathsf{dim} \qquad \Gamma \mid \Psi, i \vdash_{\mathrm{k}} \phi\ \mathsf{bdy} \qquad \Gamma \mid \Psi \vdash_{\mathrm{k}} u : [\,\phi[i \mapsto e]\,]}{\Gamma \mid \Psi \vdash_{\mathrm{k}} \mathsf{fill}^{e \to r}\ [i.\phi]\ u : [\,\phi[i \mapsto r], r{=}e \mapsto u\,]}$$

Figure 4.11: Rules for the Kan layer. The rules for **Kan boundaries** ($\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$) and **Kan terms with boundary** ($\Gamma \mid \Psi \vdash_{\mathrm{k}} u : [\,\phi\,]$) are identical to the rules for contortion boundaries (Figure 4.7), but with Kan terms in place of contortion terms.

In the final layer of the theory, we have the *Kan term* judgment $\Gamma \mid \Psi \vdash_{\mathrm{k}} u\ \mathsf{tm}$ specified in Figure 4.11. The Kan terms consist of iterated applications of the Kan filling operator to contortion terms. As for contortion terms, we also have *Kan boundary* ($\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$) and *Kan term with boundary* ($\Gamma \mid \Psi \vdash_{\mathrm{k}} u : [\,\phi\,]$) judgments which have the same functions as their equivalents for contortion terms.

The operator $(-)^{\mathrm{c}}$ allows us to treat any contortion term as a Kan term. For example, using the example cell context Γ_0 from the previous section, we have the

following Kan term:

$$\Gamma_0 \mid () \vdash_{\mathrm{k}} p\langle i \mapsto \mathbf{0}\rangle^{\mathrm{c}}\ \mathsf{tm}$$

All other Kan terms arise from application of the Kan filling operator. The Kan filling operator takes three main arguments: a Kan boundary $\Gamma \mid \Psi, i \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$ which varies in a new interval variable i, a Kan term $\Gamma \mid \Psi \vdash_{\mathrm{k}} u : [\,\phi[i \mapsto e]\,]$ which agrees with the boundary ϕ with i substituted for e, and an atomic interval term r. The resulting term

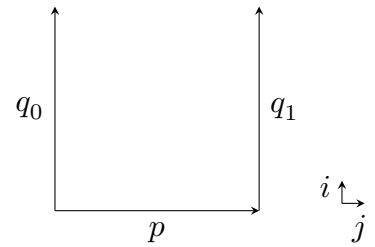
$$\Gamma \mid \Psi \vdash_{\mathrm{k}} \mathsf{fill}^{e \to r}\ [i.\phi]\ u : [\,\phi[i \mapsto r], r{=}e \mapsto u\,]$$

is a new Kan term which agrees with the boundary ϕ now with r substituted for i and with the original term u in the case that $r = e$. We can think of this operation as "adjusting" the term u to change its boundary from $\phi[i \mapsto e]$ to $\phi[i \mapsto r]$.

As an example, consider the context

$$\Gamma_2 = \begin{cases} w : ()[],\ x : ()[],\ y : ()[],\ z : ()[], \\ p : (j)[\,j{=}\mathbf{0} \mapsto x \mid j{=}\mathbf{1} \mapsto y\,], \\ q_0 : (i)[\,i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto w\,], \\ q_1 : (i)[\,i{=}\mathbf{0} \mapsto y \mid i{=}\mathbf{1} \mapsto z\,] \end{cases}$$

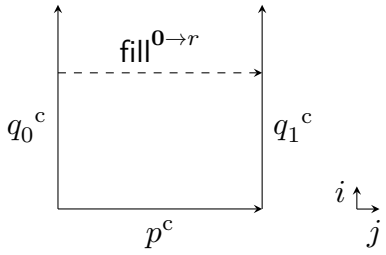
whose paths form three sides of a square (an "open cube") as shown below.



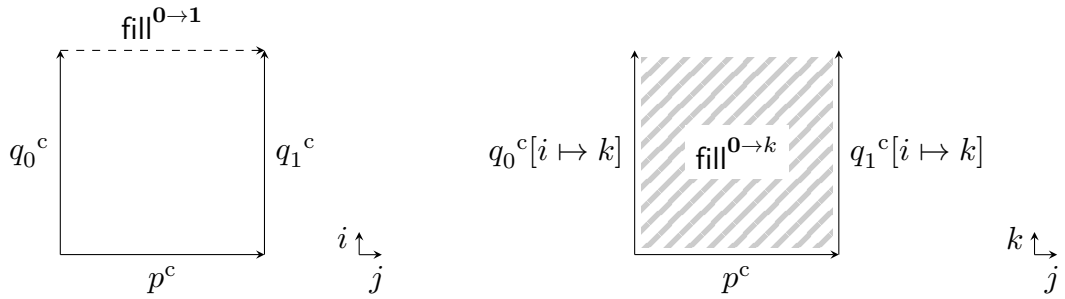
Then for any atomic interval term r, the Kan term

$$\Gamma_2 \mid j \vdash_{\mathrm{k}} \mathsf{fill}^{\mathbf{0} \to r}\ [i.[\,j{=}\mathbf{0} \mapsto {q_0}^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto {q_1}^{\mathrm{c}}\,]]\ p^{\mathrm{c}}\ \mathsf{tm}$$

is a path from $q_0[i \mapsto r]$ to $q_1[i \mapsto r]$ as shown in the diagram below.



Two important cases are where we take the destination r to be (a) the constant $\mathbf{1}$ or (b) a fresh variable k, shown to the left and right below respectively.



In the second case, the filling term is a Kan term in the cell context j, k, thus a two-dimensional cell; three of its faces are the input paths, while its $k = \mathbf{1}$ face is the filling term on the left.

We will often use a special case of the above construction with q_0 being a degenerate path of the left endpoint of p, and introduce special notation to capture this composition of two paths.

Definition 4.10. Given a cube context $\Psi, i, j \vdash k\ \mathsf{dim}$ and Kan terms

$$\Gamma \mid \Psi, j \vdash_{\mathrm{k}} t : [\, j{=}\mathbf{0} \mapsto x^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto y^{\mathrm{c}} \,]$$
$$\Gamma \mid \Psi, i \vdash_{\mathrm{k}} u : [\, i{=}\mathbf{0} \mapsto y^{\mathrm{c}} \mid j{=}\mathbf{0} \mapsto z^{\mathrm{c}} \,]$$

define the following abbreviations:

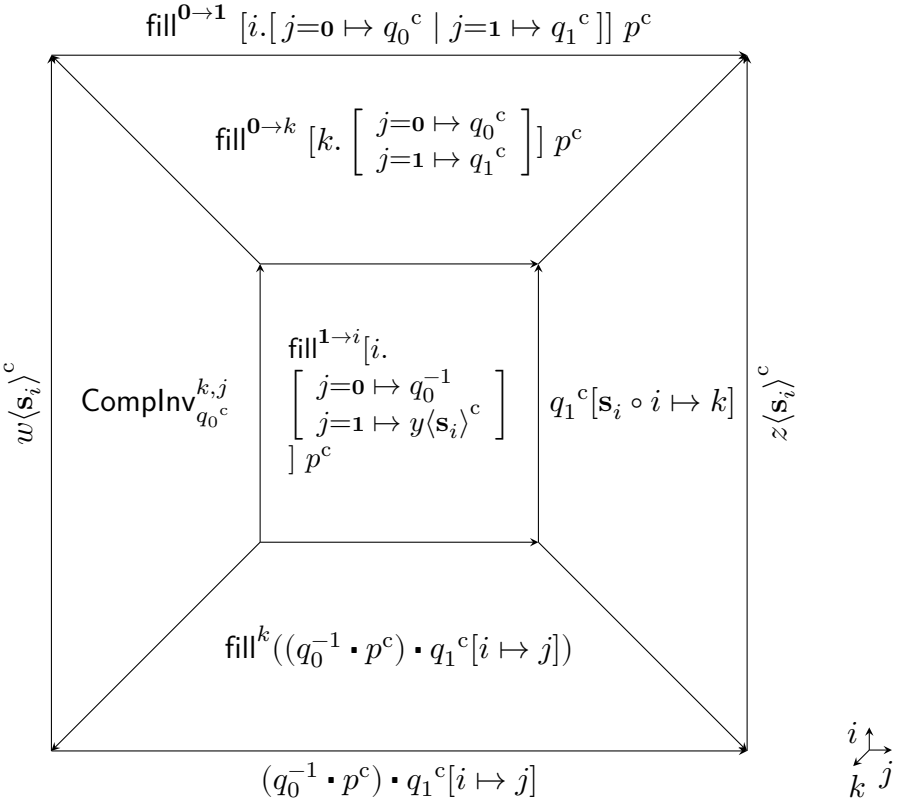
$$t \centerdot u := \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [i.[\, j{=}\mathbf{0} \mapsto x\langle \mathbf{s}_i \rangle^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto u \,]]\ t : [\, j{=}\mathbf{0} \mapsto x^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto z^{\mathrm{c}} \,]$$
$$\mathsf{fill}^{k}(t \centerdot u) := \mathsf{fill}^{\mathbf{0}\to k}\ [i.[\, j{=}\mathbf{0} \mapsto x\langle \mathbf{s}_i \rangle^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto u \,]]\ t : \begin{bmatrix} k{=}\mathbf{0} \mapsto t \\ k{=}\mathbf{1} \mapsto t \centerdot u \\ j{=}\mathbf{0} \mapsto x\langle \mathbf{s}_i \rangle^{\mathrm{c}} \\ j{=}\mathbf{1} \mapsto u \end{bmatrix}$$

Let us consider again the three paths from Γ_2. We can mimic the effect of the reversal contortion $\neg$ from the De Morgan contortion theory in any contortion theory using a Kan filling term. For example, we can construct a term with the reversed boundary of q_0 like this:

$$\Gamma_2 \mid j \vdash_{\mathrm{k}} \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [i.[\, j{=}\mathbf{0} \mapsto q_0{}^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto x\langle \mathbf{s}_j \rangle^{\mathrm{c}} \,]]\ x\langle \mathbf{s}_i \rangle^{\mathrm{c}} : [\, j{=}\mathbf{0} \mapsto w^{\mathrm{c}} \mid j{=}\mathbf{1} \mapsto x^{\mathrm{c}} \,]$$

Let us call this term q_0^{-1} in the following. We now have available two ways to construct a path from w to z, either using the iterated two-way path composition $(q_0^{-1} \centerdot p^{\mathrm{c}}) \centerdot q_1{}^{\mathrm{c}}[i \mapsto j]$, or by Kan filling a single open cube as we did above. To show in our

theory that these compositions are equivalent, we can construct a square with both paths on opposing sides. Such a square is depicted on the outside of the cube below, with the two- and three-way path compositions on the i sides and degenerate w- and z-paths on the other sides. On the inside of this square, we have depicted five squares which make up the sides of an open 3-cube. By giving appropriate terms for all the sides, we can use a Kan filling term for this open cube to obtain a proof that double two-way and single three-way path compositions are equivalent.



The i sides of the open cube can be constructed using the k-fillers for two- and three-way path composition, respectively. The $k{=}\mathbf{0}$ side of the cube, which is depicted on the back of the cube, then has $q_0^{-1} \cdot p^{\mathrm{c}}$ on the $i{=}\mathbf{0}$ side and p^{c} on the $i{=}\mathbf{1}$ side, which means we can use the Kan filler corresponding to two-way path composition, this time in the $\mathbf{1}{\to}i$ direction. As a result, the $j{=}\mathbf{1}$ side has ${q_0}^{\mathrm{c}}$ on both of its i faces such that we can use a degenerate q_0 square for this side.

The $j{=}\mathbf{0}$ side has q_0^{-1} on its $k{=}\mathbf{0}$ face and q_0 on its $i{=}\mathbf{1}$ side. We will show how to construct this square as $\mathsf{CompInv}^{k,j}_{{q_0}^{\mathrm{c}}}$ in Example 4.19.

The Kan filler of this open cube in the $\mathbf{0}{\to}\mathbf{1}$ direction gives us a term with the boundary of the outer square, which shows that both ways of composing three paths are equivalent. This proof has been done over the cartesian contortions, which means

that the construction can be carried out over any of the contortion theories introduced in Figure 4.8. Using a more expressive contortion theory would have reduced the complexity of the proof term as we could have constructed q_0^{-1} using a contortion, which would have allowed to construct side $j{=}\mathbf{0}$ of the open 3-cube as a contortion term.

Remark 4.11. In the remainder of this dissertation, we omit the annotation $(-)^{\mathrm{c}}$ that turns a contortion term into a filler term. The distinction between contortion terms and filler terms is nevertheless important to our approach. In particular, it is important that *contortions (unlike substitutions) are only ever applied to variables, never to filling terms*. This means that even in a contortion theory with connections, for example, terms looking like "$\mathsf{fill}^{\mathbf{0}\to r\wedge s}\ [i.\phi]\ u$" or "$\mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [i.j{=}r\wedge s\mapsto v]\ u$" can never appear. This allows us to treat proof search for Kan terms agnostic to the contortion theory without, we contend, a significant cost to the solver's power.

4.2 Boundary problems

With the syntax we introduced in the previous Section 4.1, we can now state the algorithmic problems that we will study in this and the following Chapter 5, which are all variations of the boundary problem. The most general and difficult problem is to find terms in the Kan theory.

Definition 4.12. Given $\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$, the problem KanCubicalCell(Γ, Ψ, ϕ) is to decide whether there is a term t such that $\Gamma \mid \Psi \vdash_{\mathrm{k}} t : [\,\phi\,]$.

We will study this problem in Section 4.3, where we give a semi-decision procedure which solves many practical instances of it. KanCubicalCell subsumes many undecidable decision problems, such as the word problem for finitely generated groups, and is consequently undecidable [19]. A clever enumeration of the terms is therefore the best we can hope for.

We construct our procedure based on the minimal cartesian contortion theory; all constructions in Section 4.3 directly carry over to all contortion theories introduced in Figure 4.8.

Using a richer contortion theory which contains more maps between cubes will speed up proof search since many boundaries can be filled directly with contortion terms and do not require the intricate construction of Kan filling terms. By increasing the power of the contortion theory, finding terms purely in the contortion theory

becomes more complex, which is why we also study the problem of finding terms purely in the contortion theory, which is a sub-problem of KanCubicalCell.

Definition 4.13. Given $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy, the problem CubicalCell(Γ, Ψ, ϕ) is to decide whether there is a term t such that $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$.
Call Cartesian the instance of CubicalCell based on the cartesian contortion theory; Disjunctive the problem for the disjunctive contortion theory; Dedekind the problem for the Dedekind contortion theory; and DeMorgan the problem for the De Morgan contortion theory.

Common to all contortion theories we study is that there are only finitely many ways to contort a variable, which means that all instances of CubicalCell are decidable. However, the complexity of the problem varies significantly between the different contortion theories.

When restricting our attention to 1-dimensional proof goals, i.e., instances of CubicalCell(Γ, Ψ, ϕ) where Ψ has only one variable, Cartesian is in P while Dedekind is NP-complete [19]. When treating Ψ, i.e., the dimension of the cube to be constructed, as the input size, the problem can furthermore seen to be coNP-hard with a reduction from the entailment problem of monotone Boolean formulas. In the remainder of this thesis, we mostly leave aside questions about complexity and focus on practical solvability of Contortion; in particular, we will study a more efficient way of solving Dedekind in Chapter 5.

4.3 Finding terms in the Kan theory

In the remainder of this chapter, we develop an algorithm for solving the problem KanCubicalCell. Cell contexts are very general and can be used to represent many mathematical objects such as algebraic structures or topological spaces. This entails that KanCubicalCell is undecidable [19]. The best we can hope for is therefore an enumeration which explores the search space in a sensible way.

Our semi-procedure is based on the heuristic that it is best to use a contortion term instead of a Kan term if possible since contortion terms provide the "simple" solutions to a boundary problem. We can decide if there exists a Kan filling term which has only contortion terms on its sides since there are only finitely many contortion terms in a given dimension. This decider will provide the basic building block for our algorithm as discussed in Section 4.3.1.

If we cannot find an open cube with only contortion terms on its sides, we will leave open increasingly more sides of the cube to be solved with Kan filling terms. These can either have the same dimension as the original boundary problem, or one dimension more, which means that we need to construct another open cube. In the first case, we can try out all possible fillers as there are only finitely many such fillers (and often only very few); a principled way to do this will be presented in Section 4.3.2. In the second case, we need to construct another open cube, which means that we can call the solver recursively on the boundary problem for the missing side. We combine all subprocedures and recursive calls in an algorithm in Section 4.3.3 and demonstrate the workings of the algorithm by showing how it derives a term evincing that path composition is associative.

For the underlying contortion theory we use the minimal, cartesian contortion theory from Figure 4.7. Since all contortion theories we study are extensions of the cartesian contortion theory, our algorithm is applicable to all contortion theories.

4.3.1 Finding Kan filling terms as open cubes

The adjacent sides of an open cube on which we can perform Kan filling need to have matching boundaries. This constraint suggests a recipe for constructing open cubes: we give the boundary constraints as constraints to a finite domain constraint solver. Constraint satisfaction programming (CSP) has been used to solve a rich variety of problems in artificial intelligence programming in the 1970s [51]; for a recent survey see Tsang [76]. We will introduce the basic setup of CSPs here; one can use any finite domain constraint solver to solve the problems in practice (in Section 5.2, we will present a implementation based on back-tracking).

Definition 4.14. A constraint satisfaction problem P is given by

- a set of variables Var,
- an assignment of domains to variables, i.e., a set D_X for each $X \in Var$, and
- a set of binary constraints, i.e., a set $C_{X,X'}$ for each $X, X' \in Var$ such that $c \subseteq D_X \times D_{X'}$ for $c \in C_{X,X'}$.

A solution S to P is a choice of one element of each domain such that all constraints are satisfied, i.e., a map $S : (X \in Var) \to D_X$ such that for all $X, X' \in Var$ and $c \in C_{X,X'}$ we have $c(S(X), S(X'))$.

Given a boundary, we will now specify a CSP to construct an open cube whose composition has the desired boundary. In many cases, the sides of the open cube are themselves results of Kan compositions and fillers. We could in principle construct these compositions and fillers by solving ever larger constraint satisfaction problems representing a collection of open cubes with matching boundaries. However, solving these larger CSPs requires a lot of memory. Instead, we will make choices for an open cube following the heuristic that it is best to use a contortion term or lower-dimensional Kan filling if possible, and then try to fill the remaining sides with Kan fillings of other open cubes. The advantage of our approach is that at any time, we are only solving a CSP for a single open cube.

The CSP for non-nested Kan filling terms is the following, where we say $(i{=}e) \in \phi$ if the side $(i{=}e)$ is specified in ϕ. Recall that we can access this term as $\phi[i{=}e]$.

Definition 4.15. Given a boundary $\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi\ \mathsf{bdy}$ and a fresh dimension $k \notin \Psi$, as well as a set of indices $\mathit{Ope} \subseteq \{(k{=}\mathbf{0})\} \cup \{(i{=}e) \mid i \in \Psi, e \in \{\mathbf{0},\mathbf{1}\}\}$, the constraint satisfaction problem $\textsc{Comp}\mathrm{CSP}(\phi, \mathit{Ope})$ is given as follows:

- $\mathit{Var} := \{X_{(k=\mathbf{0})} \text{ if } (k{=}\mathbf{0}) \notin \mathit{Ope}\} \cup \{X_{(i=e)} \mid i \in \Psi, e \in \{\mathbf{0},\mathbf{1}\}, (i{=}e) \notin \mathit{Ope}\}$
- with domains

$$D_{(i=e)} := \begin{cases} \{t \text{ s.t. } \Gamma \mid \Psi[i{=}e][k{=}\mathbf{1}] \vdash_{\mathrm{c}} t\langle k \mapsto \mathbf{1}\rangle = \phi[i{=}e]\ \mathsf{tm}\} \text{ if } (i{=}e) \in \phi \\ \{t \text{ s.t. } \Gamma \mid \Psi[i{=}e] \vdash_{\mathrm{c}} t\ \mathsf{tm}\} \text{ else} \end{cases}$$

- and constraints

$$\Gamma \mid \Psi[i{=}e][j{=}e'] \vdash_{\mathrm{c}} X_{(i=e)}\langle j \mapsto e'\rangle = X_{(j,e')}\langle i \mapsto e\rangle\ \mathsf{tm}$$

  for $\Psi \vdash i, j\ \mathsf{dim}$, $e, e' \in \{\mathbf{0},\mathbf{1}\}$.

The variables of the CSP correspond to the sides adjacent to the goal boundary and the back side of the open cube that we are trying to construct. The domains of the variables are restricted to contain only terms which match the goal boundary—for the back side $(k{=}\mathbf{0})$ we will not be able to restrict the search space as any term with the correct dimensions could be put on the back side of the cube. The constraints capture that adjacent sides need to have matching boundaries.

If we have $(i{=}e) \in \mathit{Ope}$ implies that $(i{=}e) \notin \phi$, i.e., any side left open is unspecified in the goal boundary, a solution to $\textsc{Comp}\mathrm{CSP}(\phi, \mathit{Ope})$ gives rise to a term

$$\Gamma \mid \Psi \vdash_{\mathrm{k}} \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [k.[\, i{=}e \mapsto t_{(i=e)} \text{ for } X_{(i=e)} \in \mathit{Var}]]\ t_{(k=\mathbf{0})} : [\, \phi\,],$$

which means we have obtained a solution to KanCubicalCell(Γ, Ψ, ϕ). Note that it does not matter whether we search for an open cube that we can compose in the $\mathbf{0}\to\mathbf{1}$, or the opposite direction—the two cubes are dual to each other.

Otherwise, the open cube constructed from a solution to the CSP has some sides unspecified which would need to be filled to give rise to a Kan filling term with the required boundary ϕ. In this case, we have to do more work to solve the boundary problem. We assume that each such $(i{=}e) \in \mathit{Ope} \cap \phi$ cannot be filled by a contortion term (otherwise it will not be left open in the final Algorithm 4.1). It can hence only be solved only with a Kan filling term, which has either the dimension of the goal boundary, which we consider in the next Section 4.3.2, or one dimension higher, which means that we have to construct another open cube. This can be done with another appropriate call to CompCSP as we will see in Section 4.3.3.

Example 4.16. Given a context with a single path:

$$\Gamma := x, y : ()[], p : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y \,]$$

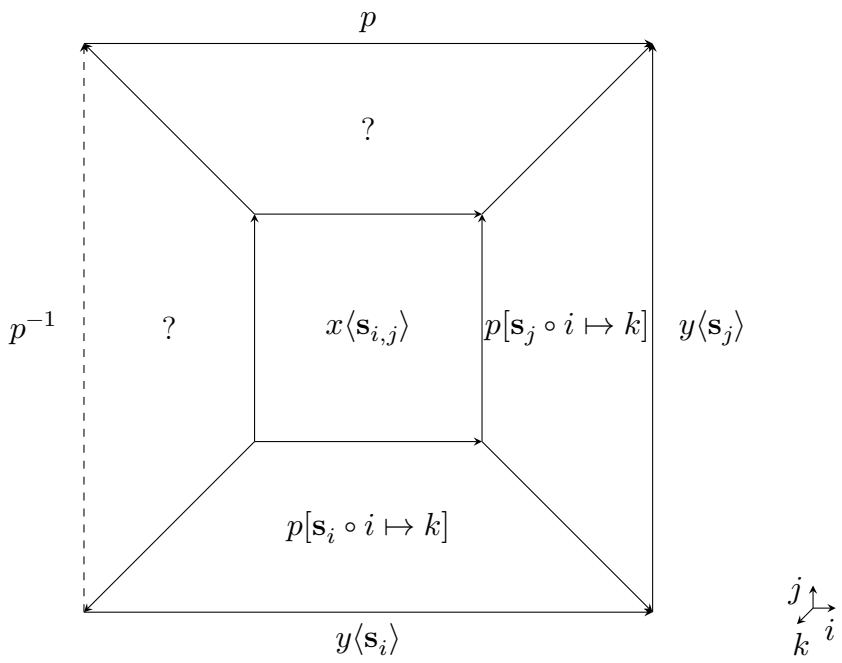
We want to construct a square witnessing that travelling along the inverse of p and then p is contractible. Recall that in a contortion theory without reversals, the inverse of p can be constructed as follows:

$$\Gamma_2 \mid j \vdash_{\mathrm{k}} \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [i.[\, j{=}\mathbf{0} \mapsto p \mid j{=}\mathbf{1} \mapsto x\langle \mathbf{s}_i \rangle \,]]\ x\langle \mathbf{s}_i \rangle : [\, j{=}\mathbf{0} \mapsto y \mid j{=}\mathbf{1} \mapsto x \,]$$

Let's call the above Kan filling term p^{-1} in the following. We now want to solve the following boundary problem:

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\, j{=}\mathbf{0} \mapsto y\langle \mathbf{s}_i \rangle \mid j{=}\mathbf{1} \mapsto p \mid i{=}\mathbf{0} \mapsto p^{-1} \mid i{=}\mathbf{1} \mapsto y\langle \mathbf{s}_j \rangle \,]\ \mathsf{bdy}$$

Since CompCSP attempts to insert only contortion terms, we cannot hope to construct a fully defined open cube yet—on side $(i{=}\mathbf{0})$, our boundary has a Kan filling term, and no contortion term has a Kan filling term on its boundary. Calling CompCSP with $\mathit{Ope} = \{(i{=}\mathbf{0}), (j{=}\mathbf{1})\}$ yields several solutions, one of which we have depicted below:



The back side of the open cube is a constant x square. On the (j=$\mathbf{0}$) and (i=$\mathbf{1}$) sides, we travel with a degenerate p square outwards in direction k. △

4.3.2 Fitting Kan filling terms into sides of an open cube

In this section we assume that we have an open cube stemming from a solution to CompCSP defined in Definition 4.15 which has sides missing for which we still have to find a term. We want to see if we can find solutions for these sides using a lower-dimensional Kan filling term. Such filling terms can come about either because (a) the goal boundary has a Kan term on one side and we use the corresponding higher-dimensional filler, or (b) because two adjacent sides of the cube have the same filler in the directions pointing at each other. We will compute all combinations in which fillers can be used in a cube in the next Section 4.3.3; for now, we will assume that we are given which sides of the cube we should fill in which directions.

The boundary ψ in this CSP is usually not the boundary of a boundary problem, but rather a cube that has been partially constructed and that has some sides still missing.

In the following, we write $[\,\psi[i{=}e]\,]$ for the boundary of the (i=e)-th side of ψ, if (i=e) $\in \psi$, then $[\,\psi[i{=}e]\,]$ is simply the boundary of $\psi[i{=}e]$. Otherwise, we look at adjacent sides of (i=e) which are specified in ψ to derive what $[\,\psi[i{=}e]\,]$ is.

Definition 4.17. Given a boundary $\Gamma \mid \Psi \vdash_{\mathrm{k}} \psi$ bdy and a pair $(Fil, Fil_{\to})$ where

- $Fil \subseteq \{(i{=}e) \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}, (i{=}e) \notin \psi\}$ and
- $Fil_{\to} : ((i{=}e) \in Fil) \to \{(r{=}e') \mid r \in \Psi[i{=}e], e' \in \{\mathbf{0}, \mathbf{1}\}, (r{=}e') \notin [\,\psi[i{=}e]\,]\}$.

Fil gives the sides of the cube ψ that we want to fill with a Kan filling term, and $Fil_{\to}$ their directions.

The constraint satisfaction problem $\textsc{FillCSP}(\psi, Fil, Fil_{\to})$ is given as follows:

- $Var := \{X_{(i=e),(j=e')} \mid \begin{array}{l}(i{=}e) \in Fil, j \in \Psi[i{=}e], e' \in \{\mathbf{0}, \mathbf{1}\},\\ (j{=}e') \notin ([\,\psi[i{=}e]\,], Fil_{\to}(i{=}e))\end{array}\}$
- with domains
  $D_{(i=e),(j=e')} = \{t \text{ s.t. } \begin{array}{l}\Gamma \mid \Psi[i{=}e][j{=}e'][k{=}e''] \vdash_{\mathrm{c}} t\langle k \mapsto e''\rangle = [\,\psi[i{=}e]\,][j \mapsto e']\\ \text{for all } (k{=}e'') \in [\,\psi[i{=}e]\,]\end{array}\}$
- and constraints
  1. $\Gamma \mid \Psi[i{=}e][j{=}e'][k{=}e''] \vdash_{\mathrm{c}} X_{(i=e),(j=e')}\langle k \mapsto e''\rangle = X_{(i=e),(k=e'')}\langle j \mapsto e'\rangle$ tm
     for $X_{(i=e),(j=e')}, X_{(i=e),(k=e'')} \in Var$, $j \neq k$
  2. $\Gamma \mid \Psi[i{=}e][j{=}e'] \vdash_{\mathrm{c}} X_{(i=e),(k=e'')} = X_{(j,e),(k=e'')}$ tm
     for $(i{=}e), (j{=}e') \in Fil$ such that $Fil_{\to}(i{=}e) = (j{=}e')$ and $Fil_{\to}(j{=}e') = (i{=}e)$, and $X_{(i=e),(k=e'')}, X_{(j,e),(k=e'')} \in Var$.
     If we have $(k{=}e'') \in \psi[i{=}e]$, the constraint applies for $[\,\psi[i{=}e]\,][k \mapsto e'']$ instead of the variable $X_{(i=e),(k=e'')}$. The same is done for $(j{=}e')$.

The variables of the CSP represent all *the sides of the sides of the cube* that we want to construct Kan filling terms for. The domains are restricted according to the sides already present in ψ. We impose two kinds of constraints. With 1., we ensure that all sides of the Kan filling term for side $(i{=}e)$ have matching boundaries. With 2., we ensure that adjacent Kan filling terms have matching boundaries. We do this by comparing all relevant sides with each other.

In the following, we will need to assume a term which has the goal boundary ϕ and fit it into the cube ψ to make sure that the newly constructed sides match with the goal boundary. For this, we write T^{ϕ} for a putative term with boundary ϕ.

Example 4.18. Given again $\Gamma = x, y : ()[], p : (i)[\,i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y\,]$, we want to mimic the $\wedge$ connection using only cartesian contortions. This means want to construct a square witnessing that travelling along the constant x path and then p

on both ways from the bottom left to the top right edge is contractible, i.e., we face the following boundary problem ϕ:

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\, j{=}\mathbf{0} \mapsto x\langle \mathbf{s}_i \rangle \mid j{=}\mathbf{1} \mapsto p \mid i{=}\mathbf{0} \mapsto x\langle \mathbf{s}_j \rangle \mid i{=}\mathbf{1} \mapsto p\langle i \mapsto j \rangle \,] \;\mathsf{bdy}$$

Calling CompCSP introduced in Section 4.3.1 with $Ope = \{(i{=}\mathbf{1}), (j{=}\mathbf{1})\}$ yields one solution with the constant x square for all three sides we wanted to fill with a contortion. To fill the remaining sides, we call FillCSP for the boundary

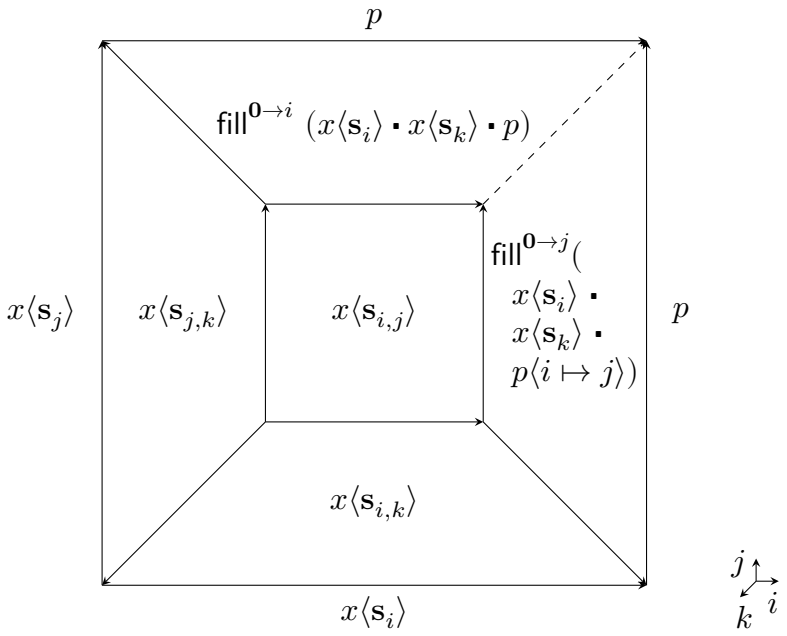
$$[\, j{=}\mathbf{0} \mapsto x\langle \mathbf{s}_{i,k} \rangle \mid i{=}\mathbf{0} \mapsto x\langle \mathbf{s}_{j,k} \rangle \mid k{=}\mathbf{0} \mapsto x\langle \mathbf{s}_{i,j} \rangle \mid k{=}\mathbf{1} \mapsto T^{\phi} \,]$$

to obtain terms for the remaining sides. This means we set $Fil = Ope$, the only possible filling directions are $Fil_{\rightarrow}(j{=}\mathbf{1}) = (i{=}\mathbf{0})$ and $Fil_{\rightarrow}(i{=}\mathbf{1}) = (j{=}\mathbf{0})$.

This means we are trying to find Kan filling terms for sides $(j{=}\mathbf{1})$ and $(i{=}\mathbf{1})$ which are pointing towards each other. The resulting FillCSP has 0 variables since all 1-dimensional paths in the cube are already specified, with the exception of the common boundary of $(j{=}\mathbf{1})$ and $(i{=}\mathbf{1})$. We still have to check some constraints, however: for 2., we need to check, e.g.:

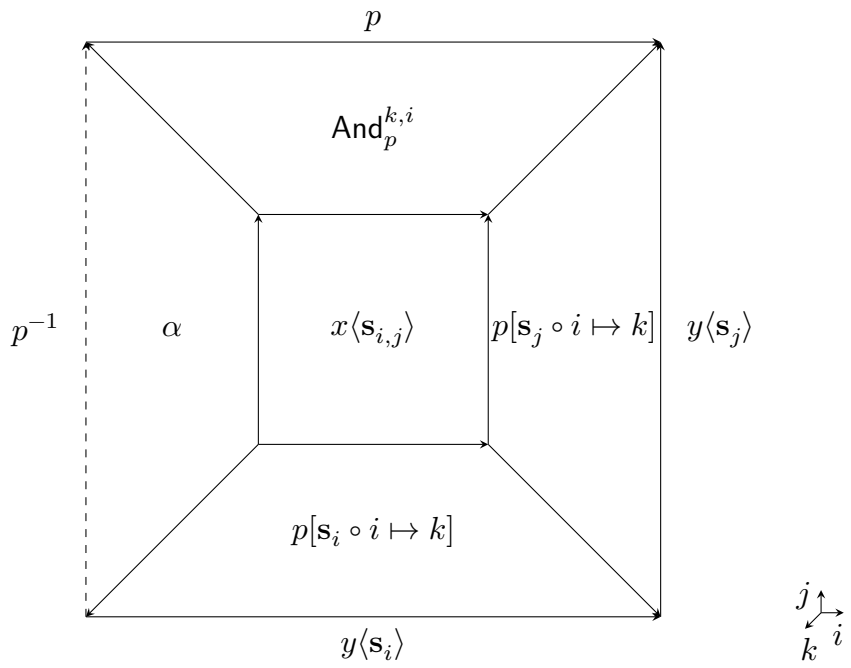
$$[\, \phi[i{=}\mathbf{1}] \,][k \mapsto \mathbf{1}] = [\, \phi[j{=}\mathbf{1}] \,][k \mapsto \mathbf{1}]$$

Since all corresponding sides of the squares do indeed match, we can insert Kan filling terms for the missing sides, leading to the following open cube after removing the temporarily assumed term T^{ϕ}:



Taking the Kan composition over k in direction $\mathbf{0} \to \mathbf{1}$ will therefore yield a term with the same boundary as $p\langle i \mapsto i \wedge j\rangle$ in the Dedekind contortion theory. In the following, we will refer to this composition as $\mathsf{And}_p^{i,j}$ for a variable p. △

Example 4.19. Continuing Example 4.16, we try to construct a term for the outer boundary of the cube below. Using CompCSP, we had already filled three sides of the cube with contortions. Moreover, we can use the term $\mathsf{And}_p^{k,i}$ from Example 4.18 to fill the $(j{=}\mathbf{1})$ side:



We can fill side $(i{=}\mathbf{0})$ in direction $\mathbf{0}{\to}k$ since we have a Kan filling term on the $(k{=}\mathbf{1})$-side of this square. The constraints in FillCSP verify that the 1-dimensional sides of the square at $(i{=}\mathbf{0})$ agree with the goal boundary at $(i{=}\mathbf{0})$, which means we can indeed use the Kan filling term

$$\alpha := \mathsf{fill}^{\mathbf{0}\to k}\ [i.[\, i{=}\mathbf{0} \mapsto p[i \mapsto j] \mid i{=}\mathbf{1} \mapsto x\langle \mathbf{s}_j\rangle\,]]\ x\langle \mathbf{s}_i\rangle$$

to complete our missing side. The Kan filling term of the open 3-cube in direction $\mathbf{0}{\to}\mathbf{1}$ yields a term with for the goal boundary, call this term $\mathsf{CompInv}_p^{i,j}$. △

Example 4.20. Given a context

$$\Gamma := \begin{cases} x, y, z : ()[], \\ p : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y\,], \\ q : (i)[\, i{=}\mathbf{0} \mapsto y \mid i{=}\mathbf{1} \mapsto z\,] \end{cases}$$

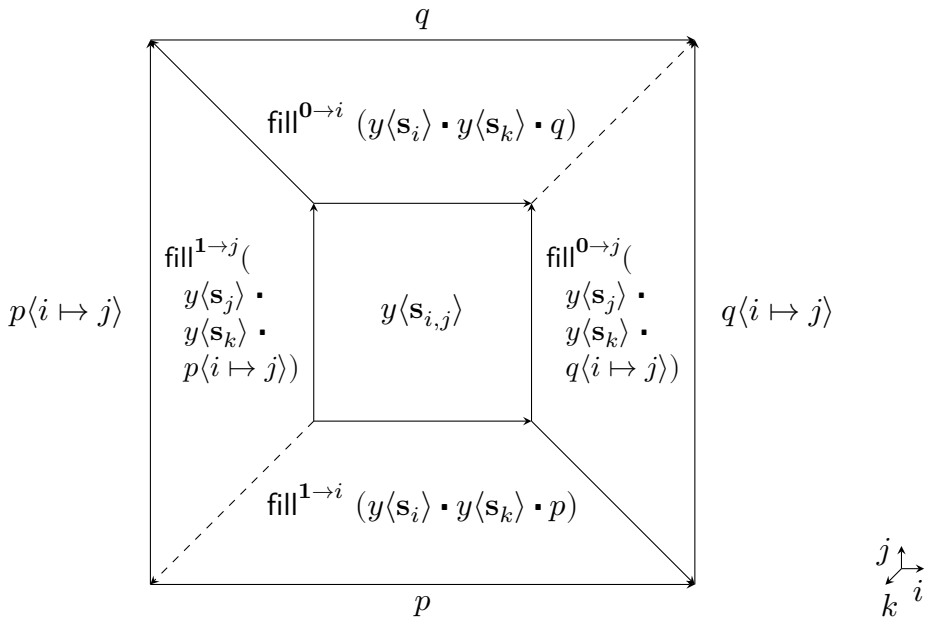
we want to construct a square witnessing that travelling along p and q in both ways from the bottom left to the top right edge is contractible, i.e., a term with boundary ϕ given by:

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\, j{=}\mathbf{0} \mapsto p \mid j{=}\mathbf{1} \mapsto q \mid i{=}\mathbf{0} \mapsto p\langle i \mapsto j\rangle \mid i{=}\mathbf{1} \mapsto q\langle i \mapsto j\rangle \,] \ \mathsf{bdy}$$

Calling the CompCSP solver of the previous section with all sides but the back side open, i.e., $Ope = \{(i{=}\mathbf{0}), (i{=}\mathbf{1}), (j{=}\mathbf{0}), (j{=}\mathbf{1})\}$, will yield as one solution $X_{(k=\mathbf{0})} = y\langle \mathbf{s}_{i,j} \rangle$. In order to fill the four sides of the open cube, we call FillCSP for the cube $[\, k{=}\mathbf{0} \mapsto y\langle \mathbf{s}_{i,j} \rangle \mid k{=}\mathbf{1} \mapsto T^{\phi} \,]$. We set $Fil = Ope$ and filling directions $Fil_{\rightarrow}$ as follows:

$$(j{=}\mathbf{0}) \mapsto (i{=}\mathbf{1}), (j{=}\mathbf{1}) \mapsto (i{=}\mathbf{0}), (i{=}\mathbf{0}) \mapsto (j{=}\mathbf{1}), (i{=}\mathbf{1}) \mapsto (j{=}\mathbf{0})$$

This means that FillCSP has four variables: $X_{(j=\mathbf{0}),(i=\mathbf{1})}$, $X_{(i=\mathbf{1}),(j=\mathbf{0})}$, $X_{(j=\mathbf{1}),(i=\mathbf{0})}$ and $X_{(i=\mathbf{0}),(j=\mathbf{1})}$. The domains for these variables are restricted to agree with the back square $y\langle \mathbf{s}_{i,j} \rangle$ and the front square corresponding to the goal boundary T^{ϕ}, which means that all domains only contain a constant 1-dimensional y-path. The first set of constraints is vacuously satisfied since any side of the cube has only a single path that needs to be constructed. The second set of constraints ensures that $X_{(j=\mathbf{0}),(i=\mathbf{1})}$ agrees with $X_{(i=\mathbf{1}),(j=\mathbf{0})}$, and $X_{(j=\mathbf{1}),(i=\mathbf{0})}$ with $X_{(i=\mathbf{0}),(j=\mathbf{1})}$. Since this is the case, we can return the only result available in our domains.



We can therefore take the Kan filler of the above cube in the k direction from $\mathbf{0}$ to $\mathbf{1}$ to solve the boundary problem ϕ. In the following we will refer to this term as $\mathsf{SqComp}^{j,i}_{p,q}$ for any such p and q.

Note that the discussed boundary problem is trivial in a globular setting, where the corresponding statement amounts to reflexivity of the composition of p and q. Since our type theory is inspired by cubical sets, a relatively weak model of spaces, we have to do more work to fill this boundary—but this work can be carried out automatically by our solver. △

4.3.3 An algorithm for finding Kan terms

With the defined constraint satisfaction problems we can specify a semi-decision procedure for the general boundary problem. The base cases, where a contortion term or a Kan filling term with the same dimension as the original boundary problem constitutes a solution, can be decided trivially since there are only finitely many such contortions and Kan filling terms. The difficult part of KanCubicalCell is to check whether there is a Kan term of a higher-dimensional open cube which gives a term with the required boundary. We focus on this problem with our algorithm; a practical algorithm such as the one presented in Chapter 5 will naturally first check the base cases before invoking the more complicated proof search.

The basic working principle of our Kan composition solver is as follows:

- First, construct an open cube which has as many sides filled with contortion terms as possible;
- then try to use as many lower-dimensional Kan filling terms for the other sides;
- finally call the solver recursively on the remaining sides.

We use iterative deepening on the level of Kan filling terms as well as backtracking to make sure that our procedure at some point tries to use a Kan filling term even if a side could have been filled with a contortion. This way our procedure can be seen to be a complete search procedure for the semi-decidable problem KanCubicalCell.

We have to call the CSPs with appropriate parameters Ope and $(Fil, Fil_{\to})$. Since there is no ad-hoc way to see which sides need to be left open or which can be filled with lower-dimensional Kan terms, we have to try out all possibilities for these parameters.

For the open sides passed as a parameter to CompCSP, we define

$$\textsc{Open}(\Psi) := \mathcal{P}(\{(i{=}e) \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}\} \cup \{(k{=}\mathbf{0})\})$$

In our solver discussed in Section 5.2, we start with *Ope* being the empty set to search for a single Kan filling term, and then increase the number of open side. We also need to make sure that all open sides are listed in all possible permutations—the order in which we try to fill the open sides matters, since unspecified sides might be filled with Kan filling terms which impact the boundary of other open sides.

Lower-dimensional Kan terms can be inserted into a side $(i{=}e)$ of a cube ψ either if $\psi[i{=}e]$ has a Kan filling term on its side, or if ψ has two adjacent sides open and the Kan filling terms point at each other. We refer the interested reader to the exact computation of all such possibilities in the cube solver discussed in Section 5.2; in the following algorithm we assume that $\textsc{Fillers}(\psi)$ generates all possible pairs $(Fil, Fil_{\rightarrow})$ that might give rise Kan filling terms fitted into ψ.

Equipped with these helper functions we can define our algorithm for finding higher-dimensional Kan filling terms.

Algorithm 4.1 Finding Kan compositions

Input: $\Gamma \mid \Psi \vdash_{\mathrm{k}} \phi$ bdy
Output: $\Gamma \mid \Psi \vdash_{\mathrm{k}} \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ [k.\psi]\ u : [\,\phi\,]$ where $k \notin \Psi$, if such a term exists

```
1: procedure KanSolver(Γ, Ψ, φ)
2:     for d ∈ 1, ... do
3:         t ← BoundKanSolver(Γ, Ψ, φ, d)
4:         if t ≠ Unsolved then
5:             return t
6: procedure BoundKanSolver(Γ, Ψ, φ, d)
7:     if d = 0 then
8:         return Unsolved
9:     else
10:        for Ope ← Open(Ψ) do
11:            for ψ ← CompCSP(φ, Ope) do
12:                for (Fil, Fil→) ← Fillers(ψ, k=1 ↦ T^φ) do
13:                    ψ := ψ ∪ FillCSP((ψ, k=1 ↦ T^φ), Fil, Fil→)
14:                    for (i=e) ∈ Ope − Fil do
15:                        t ← BoundKanSolver([ ψ[i=e] ], d − 1)
16:                        ψ := [ ψ, i=e ↦ t ]
17:                return Γ | Ψ ⊢k fill^{0→1} [k.ψ − (k=0)] ψ[k=0] : [ φ ]
18:        return Unsolved
```

The algorithm works by iteratively deepening the maximal depth of the Kan composition on line 2. This means we proceed with a sort of breadth-first search to prevent extending out one side of the cube indefinitely. We write $\leftarrow$ for non-deterministically choosing an element from a set of elements, such as on line 3 (in the implementation discussed in Chapter 5, this non-determinism is realised with the list monad).

The main procedure starts on line 6. If the maximal depth d is reached, the construction will fail and return `Unsolved` to the main procedure on line 6. Otherwise, we call the CompCSP solver with any possible combination of open sides as computed with the Open on line 10.

After having computed a candidate open cube with CompSolver called on line 11, we first try to insert as many lower-dimensional Kan filling terms as possible by trying out all possibilities offered by Fillers with the loop on line 12. Afterwards, we call the solver recursively on line 15 on the remaining sides.

Since any side of a cube is allowed to be the result of a Kan composition or filler at some point (even if it could be filled with a contortion term), we will eventually find any nested Kan composition. This means that KanSolver is semi-complete, i.e., if a boundary problem is solvable, the algorithm will eventually find a solution.

We demonstrate the workings of the algorithm by showing how it derives a proof of associativity of path composition.

Example 4.21. Given a context

$$\Gamma := \begin{cases} w, x, y, z : ()[], \\ p : (i)[\, i{=}\mathbf{0} \mapsto w \mid i{=}\mathbf{1} \mapsto x \,], \\ q : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y \,], \\ r : (i)[\, i{=}\mathbf{0} \mapsto y \mid i{=}\mathbf{1} \mapsto z \,] \end{cases}$$

we want to show that path composition from Definition 4.10 is associative. This amounts to solving the following boundary problem:

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\, j{=}\mathbf{0} \mapsto (p \cdot q) \cdot r \mid j{=}\mathbf{1} \mapsto p \cdot (q \cdot r) \mid i{=}\mathbf{0} \mapsto w\langle \mathbf{s}_j \rangle \mid i{=}\mathbf{1} \mapsto z\langle \mathbf{s}_j \rangle \,] \ \mathsf{bdy}$$

The solver finds that almost no sides of the open cube corresponding to this boundary can be filled with a contortion: the bottom and top sides have a Kan composition on one side and are hence best filled using the corresponding fillers. This means that the back and right side of the cube will also have a path composition on their boundary. Hence only the left side can be filled with a contortion, namely the square which is constant w. Therefore the solver has constructed the following cube after invoking CompCSP and FillCSP once:

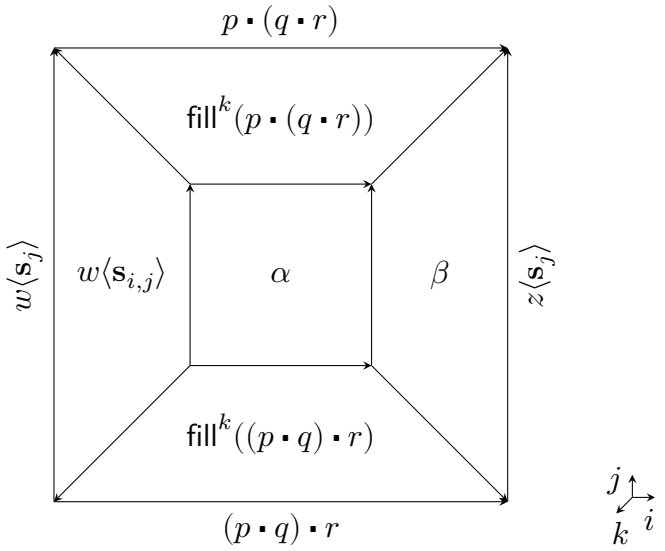
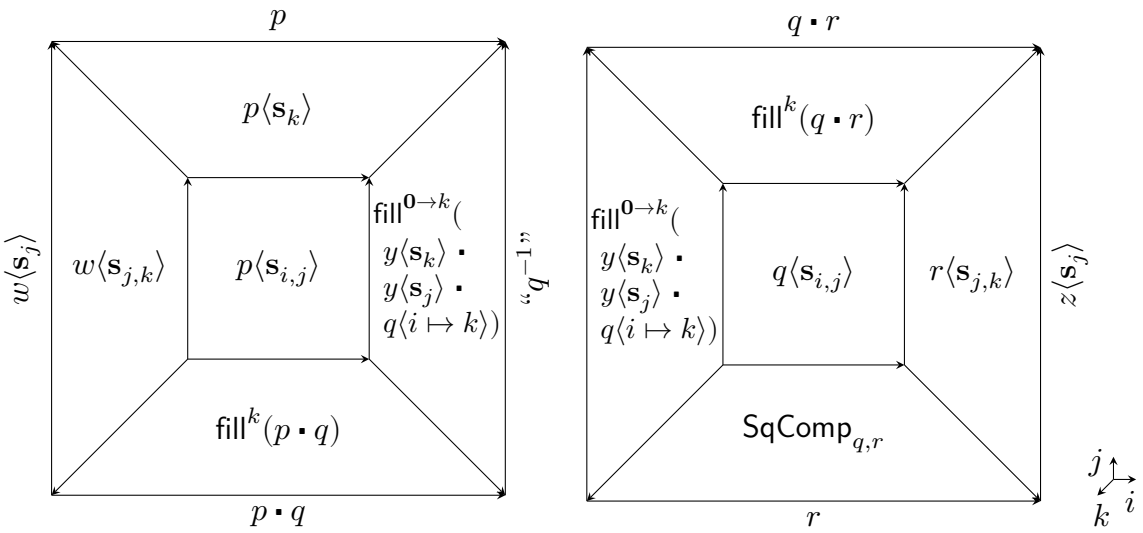


Figure 4.12: Proving associativity of path composition

The solver is called recursively on the sides ($i{=}\mathbf{1}$) and ($k{=}\mathbf{0}$); we will construct these as α and β in the following.

The square α has an open side at $i{=}\mathbf{1}$ as we do not know what the common boundary of α and β should be. The solver hence constructs a constraint satisfaction problem with the right-hand side free. Since the goal boundary of this right side is $[\, j{=}\mathbf{0} \mapsto z \mid j{=}\mathbf{1} \mapsto y \,]$, it cannot possibly be solved with a contortion. Calling CompCSP with open sides $j{=}\mathbf{0}$ and $i{=}\mathbf{1}$ will find that three sides of the cube can be filled with contortion terms. Since the goal boundary on side $i{=}\mathbf{1}$ is unspecified, FillCSP can fill this side as a lower-dimensional Kan term; the remaining side $j{=}\mathbf{0}$ is naturally filled by the square filler for path composition. In summary, the solver produces the Kan filling term of the open cube depicted in Figure 4.13a to solve α.

For the left-hand side β it remains to construct a square with the boundary shown in Figure 4.13b. Not all sides of this open cube can be filled with contortions or lower-dimensional Kan filling terms; the depth of the proof search has to be set to 3 to find that the bottom square can be filled with $\mathsf{SqComp}^{k,i}_{q,r}$, which we constructed in Example 4.20.



(a) Filler for α (b) Filler for β

Figure 4.13: Kan compositions for filling back and left side of associativity proof

In order to prove associativity of path composition, the solver has hence constructed four open cubes, glued together appropriately. We can summarise the construction in Figure 4.14 as follows. The front left cube with ϕ on the front corresponds to the original boundary problem we wanted to solve. The cube in the back gives rise to the term α, and the cube on the right to β. When constructing β, the solver had to find another Kan filler, namely the term for SqComp, which is depicted on the bottom.

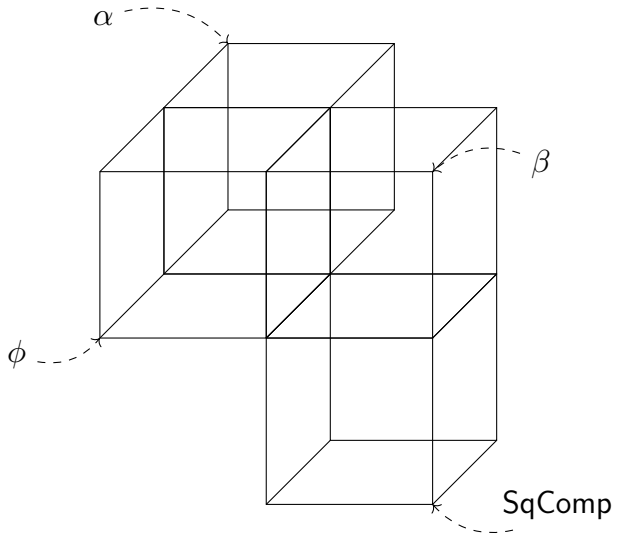


Figure 4.14: The space constructed to prove associativity of path composition

△

Chapter 5

Dedekind cubes & a practical cube solver

With Algorithm 4.1 from Section 4.3.3 we have available a method to solve the boundary problem for Kan terms over an arbitrary contortion theory. In all examples discussed in Section 4.3, we have only used contortions present in the minimal, cartesian contortion theory such that all constructed terms are available in all contortion theories introduced in Figure 4.8. Using a richer contortion theory has the advantage that we can solve more boundary problems in the contortion theory already, which means that need to construct fewer intricate Kan filling terms. However, the number of contortions grows more quickly in a richer contortion theory, which makes it infeasible to simply list the domains for higher-dimensional boundary problems—for instance, the number of Dedekind contortions [3], which are formed using connections $\wedge$ and $\vee$, are the Dedekind numbers, which grow super-exponentially.

In this chapter, we study a solution to this dilemma using an equivalent characterisation of Dedekind contortions as certain poset maps, which allow for a concise representation of a collection of Dedekind contortions. This allows us to use our algorithm for this expressive contortion theory also in higher dimensions. In Section 5.1, we will introduce the equivalence between Dedekind contortions and poset maps; define potential poset maps as a memory efficient but lossy representation of Dedekind contortions; and present an algorithm to search for Dedekind contortions. We then discuss how the alternative characterisation of Dedekind contortions plays out when using Algorithm 4.1.

Afterwards, we introduce a practical solver, written in Haskell, which implements the results from this and the previous Chapter 4. The implementation of the solver and documentation can be found online[1]. In Section 5.2, we discuss how the syntax

[1] `https://zenodo.org/doi/10.5281/zenodo.10966924`

from Section 4.1 gives rise to proof terms in Cubical Agda—since all contortion theories we study are subsets of the De Morgan contortion theory, they can all be used to resolve proof goals in Cubical Agda. We highlight the power of our Haskell solver by proving automatically the Eckmann-Hilton argument, a classical result from higher category theory. We derive the argument in a cubical fashion which is due to Tom Jack and Axel Ljungström. This constitutes, to the knowledge of the author, the first automatic proof of the argument.

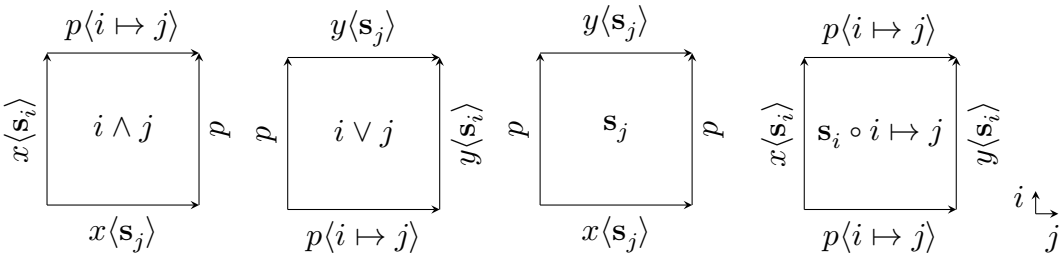
Afterwards, we compare how well different contortion theories work in our solver. We test the De Morgan contortions [16] which form the basis of Cubical Agda; the Dedekind contortions [3] with and without the refinements developed in this chapter; the disjunctive contortions [15]; the conjunctive contortions (which only have the connection $\wedge$); and the cartesian contortions [2] which are used in the theorem prover redtt[2]. Since cubical type theory is a research field which is in flux and it is unclear which formulation of it is the best, we hope to provide another angle from which we can compare the different proposals.

5.1 An alternative characterisation of Dedekind contortions

Recall that in the Dedekind contortion theory, we have available both connections $\wedge$ and $\vee$ (as well as all substitutions). Given the following context

$$\Gamma := \begin{cases} x : ()[\,], \\ y : ()[\,], \\ p : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y\,], \end{cases}$$

we can contort p into a square using the $\wedge$ and $\vee$ connections and i- and j-degeneracies to obtain the following squares:



[2] `https://redprl.org/#redtt`

Including the two contortion terms $p\langle \mathbf{s}_i \circ \mathbf{s}_j \circ i \mapsto \mathbf{0}\rangle$ and $p\langle \mathbf{s}_i \circ \mathbf{s}_j \circ i \mapsto \mathbf{1}\rangle$ (which reduce to normal forms $x\langle \mathbf{s}_i \circ \mathbf{s}_j\rangle$ and $y\langle \mathbf{s}_i \circ \mathbf{s}_j\rangle$, respectively), this gives us six ways in which we can turn a 1-dimensional path into a square. In general, there are $D(n)^m$ many contortions from the n-cube to the m-cube. $D(n)$ grows extremely quickly, for instance, there are $D(6) = 7,828,354$ many ways to turn a 1-cube into a 6-dimensional cube. Only recently, $D(9)$ could be computed [38, 45]. Naïvely trying out all Dedekind contortions is hence infeasible for boundary problems in higher dimensions. To represent and find Dedekind contortions, we will utilise an alternative characterisation of Dedekind contortions as truth-tables of monotone Boolean formulas. These can be defined as certain poset maps as we will see in Section 5.1.1. This perspective gives geometric intuition for Dedekind contortions and allows us to construct them gradually. This way, we can represent and navigate the search space of DEDEKIND more efficiently with the algorithm presented in Section 5.1.2. Moreover, the poset map perspective makes the solver for KANCUBICALCELL from Chapter 3 more memory efficient since the domains for the constraint satisfaction problems can be captured concisely. We will demonstrate this with an example in Section 5.1.3.

Remark 5.1. We are not aware of a characterisation of the De Morgan contortions which would allow for a similar treatment of this even more expressive theory. A different contortion theory that one could consider would be a *Boolean* contortion theory in which the connections $\wedge$ and $\vee$ and the reversal $\neg$ obey the laws of a Boolean algebra. This contortion theory has an alternative characterisation in terms of truth-tables which allows to compactly represent a collection of Boolean contortions. There is no cubical type theory based on the Boolean cube category however, which is why we do not study this contortion theory in this dissertation.

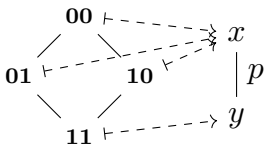
5.1.1 Dedekind contortions as poset maps

Dedekind contortions are precisely terms in the free distributive lattice over n generators, which means that a Dedekind contortion $\psi \colon \Psi \twoheadrightarrow i$ can be regarded as a propositional formula with only positive literals over the set of variables Ψ. This entails that we have a characterisation of ψ as the truth-table of ψ read as a propositional formula. Since ψ only contains positive literals, its truth-table satisfies a monotonicity property: we cannot make ψ false by setting variables to true. By imposing the product order on the possible assignments to each variable, we can regard the truth-table as a monotonic map between posets.

Let us return to our context Γ from above. Treating the contortion $i \wedge j$ as a propositional formula, its truth-table has four rows, where each row assigns either $\mathbf{0}$ or $\mathbf{1}$ to i and j and evaluates $i \wedge j$ correspondingly:

i	j	$i \wedge j$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

The assignments to i and j can be partially ordered using the product order on $\mathbf{I} := \{\mathbf{0} < \mathbf{1}\}$ with itself, and we can view evaluation of the formula as a map to the poset $\mathbf{I}$. The truth-table hence induces a map from the four-element poset $\mathbf{I} \times \mathbf{I}$ to the two-element poset $\mathbf{I}$, which is monotonic since we have only positive literals in any Dedekind contortion read as a propositional formula. We have depicted the corresponding Hasse diagrams, as well as the action of the map corresponding to $i \wedge j$ below:



By filling in the definition of p into the target poset, we can retrace how the boundary of the $\wedge$-square comes about: going from $\mathbf{00}$ to $\mathbf{01}$ corresponds to the $i{=}\mathbf{0}$ side of $p\langle i \wedge j\rangle$, which remains at x, i.e., is the degenerate x path. When passing from $\mathbf{01}$ to $\mathbf{11}$, our assignment switches from x to y along p. Therefore the boundary of $p\langle i \wedge j\rangle$ at its $j{=}\mathbf{1}$ side is p. The poset map perspective hence provides geometric intuition for the boundary of the square $p\langle i \mapsto i \wedge j\rangle$.

The poset map perspective on Dedekind contortions—and its geometric intuition—generalises to contortions of higher-dimensional terms: given a Dedekind contortion $\psi \colon \Psi' \twoheadrightarrow \Psi$, we can regard ψ as $|\ \Psi\ |$ propositional formulas over variables Ψ', which in turn can be understood as a poset map between the $|\ \Psi'\ |$- and the $|\ \Psi\ |$-fold product of $\mathbf{I}$. For example, given the cell context

$$\Gamma_1 := \begin{cases} a : ()[\,], \\ s : (i,j)[\, i{=}\mathbf{0} \mapsto a\langle \mathbf{s}_j\rangle \mid i{=}\mathbf{1} \mapsto a\langle \mathbf{s}_j\rangle \mid j{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i\rangle \mid j{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle\,] \end{cases}$$

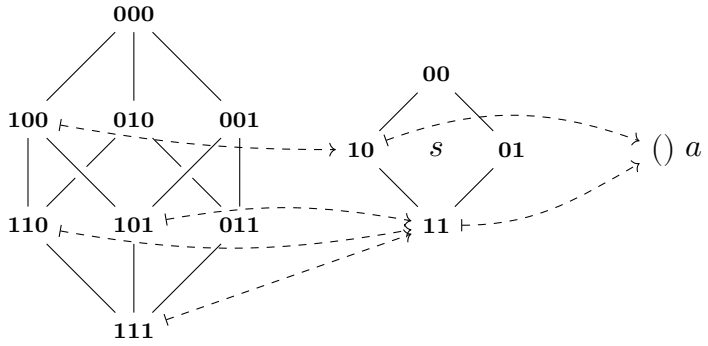
we can turn s into a cube with the following contortion:

$$\Gamma_1 \mid i, j, k \vdash_{\mathrm{c}} s\langle i \mapsto i \vee k, j \mapsto (i \wedge j) \vee k\rangle \ \mathsf{tm} : \left[\begin{array}{l} i{=}\mathbf{0} \mapsto s\langle i \mapsto k, j \mapsto k\rangle \\ i{=}\mathbf{1} \mapsto a\langle \mathbf{s}_j\rangle\langle \mathbf{s}_k\rangle \\ j{=}\mathbf{0} \mapsto s\langle i \mapsto i \vee k, j \mapsto k\rangle \\ j{=}\mathbf{1} \mapsto s\langle i \mapsto i \vee k, j \mapsto i \vee k\rangle \\ k{=}\mathbf{0} \mapsto s\langle i \mapsto i, j \mapsto i \wedge j\rangle \\ k{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle\langle \mathbf{s}_j\rangle \end{array}\right]$$

Recall that the $i{=}\mathbf{1}$-th face of this contortion term is $s\langle i \mapsto \mathbf{1}, j \mapsto j \vee k\rangle$, which reduces to $a\langle \mathbf{s}_j\rangle\langle j \mapsto j \vee k\rangle = a\langle \mathbf{s}_j\rangle\langle \mathbf{s}_k\rangle$ (similar for the $k = \mathbf{1}$ face). In order to make sense of the boundary of this contortion term, we can inspect the truth-tables of $i \vee k$ and $(i \wedge j) \vee k$:

i	j	k	$i \vee k$	$(i \wedge j) \vee k$
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**1**
0	**1**	**0**	**0**	**0**
0	**1**	**1**	**1**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**1**	**1**
1	**1**	**0**	**1**	**1**
1	**1**	**1**	**1**	**1**

If i is $\mathbf{1}$, the elements are sent to either $\mathbf{10}$ or $\mathbf{11}$, which means that only the $i{=}\mathbf{1}$ face of s is relevant for this side of the cube, which is a degenerate a square. The relevant parts of the poset maps inducing the $i{=}\mathbf{1}$ face of the above contortion are hence those from $(\mathbf{1}--)$ to $(\mathbf{1}-)$, and from $(\mathbf{1}-)$ to $()$ as depicted below:



All four elements of $i{=}\mathbf{1}$ are sent to a, which means that this side of the cube is a constant a square.

We now make precise the correspondence between Dedekind contortions and poset maps. The equivalence can be viewed as an instance of Stone duality [46] and a more

abstract version it has been proved by Wraith [83]. To the author's knowledge, this is the first concrete proof in the literature of the statement.

We show that the category $\square_{\vee\wedge}$ as introduced in Section 4.1.1 is equivalent to the category $\square_{\mathbf{I}}$, which is a full subcategory of the category of posets and monotonic maps as follows:

Definition 5.2. The category $\square_{\mathbf{I}}$ is defined with:

- Objects are $\mathbf{I}^n$ for $n \geq 0$, where $\mathbf{I} \coloneqq \{\mathbf{0} < \mathbf{1}\}$.

  We denote elements of $\mathbf{I}^n$ with $(e_1 \dots e_n)$ where $e_i \in \{\mathbf{0}, \mathbf{1}\}$ for $1 \leq i \leq n$.

  For an element $x = (e_1 \dots e_n)$ write $x_i \coloneqq e_i$.

- Morphisms $\mathbf{I}^m \to \mathbf{I}^n$ are maps monotonic with respect to $<$.

Given some context $\Psi \coloneqq j_1, \dots, j_m$, we can treat an element $x = (e_1 \dots e_m) \in \mathbf{I}^m$ as an assignment of each j_k to e_k. Given some Dedekind contortion interval term $\Psi \vdash_{\mathrm{c}} r\ \mathsf{dim}$ (i.e., an element of the free distributive lattice over Ψ), we can evaluate r at x to either $\mathbf{0}$ or $\mathbf{1}$. We will denote this evaluation as $[\![r]\!]_x$.

Proposition 5.3. $\square_{\vee\wedge}$ and $\square_{\mathbf{I}}$ are equivalent.

Proof. The functor $F : \square_{\vee\wedge} \to \square_{\mathbf{I}}$ sends Ψ to $\mathbf{I}^{|\Psi|}$ and

$$\psi = (i_1 \mapsto r_1, \dots, i_{|\Psi|} \mapsto r_{|\Psi|}) : \Psi' \to \Psi$$

to $\sigma : \mathbf{I}^{|\Psi'|} \to \mathbf{I}^{|\Psi|}$ defined by $\sigma(x)_i = [\![r_i]\!]_x$ for all $x \in \mathbf{I}^{|\Psi'|}$.

The functor $G : \square_{\mathbf{I}} \to \square_{\vee\wedge}$ sends $\mathbf{I}^n$ to $\{1, \dots, n\}$ and $\sigma : \mathbf{I}^m \to \mathbf{I}^n$ to the contortion $(i_1 \mapsto r_1, \dots, i_n \mapsto r_n)$ where

$$r_i = \bigvee \bigwedge_{\{x \mid \sigma(x)_i = \mathbf{1}\}} \{j \mid x_j = \mathbf{1}\}.$$

Note that FG is just the identity functor; to see that we have a natural isomorphism to GF observe that Ψ is isomorphic to $\{1, \dots, \mid \Psi \mid\}$. □

Using this equivalence allows us to switch between characterisations of Dedekind contortions as poset maps and as tuples of formulas in the free distributive lattice. We will in the following denote $\psi_{\mathbf{I}}$ for the poset map induced by the Dedekind contortion ψ, and $\sigma_{\vee\wedge}$ for the Dedekind contortion prescribed by the poset map σ.

5.1.2 Finding Dedekind contortions with potential poset maps

In this section we use the poset map perspective to develop an algorithm solving DEDEKIND more efficiently. We will try to contort a given cell p into a boundary ϕ by gradually restricting a poset map according to the faces of ϕ. In many cases, our algorithm will find a solution quickly: if p appears as a contorted term in one of the faces of ϕ, we can reduce the search space significantly before performing any expensive operations. In the worst case, our algorithm has a runtime of the order of the previous Dedekind number, already offering an improvement over brute-force search.

At the heart of our algorithm is a memory-efficient representation of a collection of poset maps, called a potential poset map, which allows for representing many poset maps at once. A potential poset map keeps track of all the values that elements of the domain of the poset map could take while making sure that there are no superfluous values that cannot be part of any poset map as they would otherwise violate monotonicity.

Definition 5.4. A *potential poset map* (*ppm*) is a map $\Sigma : \mathbf{I}^m \to \mathcal{P}(\mathbf{I}^n)$ such that $\forall x \leq y$:

- $\forall u \in \Sigma(y) : \exists v \in \Sigma(x) : v \leq u$.
- $\forall v \in \Sigma(x) : \exists u \in \Sigma(y) : v \leq u$.

The two conditions ensure that any possible value in a ppm gives rise to at least one poset map. Suppose that we send x to some $v \in \Sigma(x)$. Then we can choose for any element $z \leq x$ an element $u \in \Sigma(z)$ satisfying monotonicity with $u \leq v$; and we can choose for any $y \geq x$ an element $w \in \Sigma(y)$ satisfying monotonicity with $v \leq w$. Iterating this consideration entails that a ppm Σ gives rise to at least one ppm iff $\Sigma(x) \neq \emptyset$ for all x.

Potential poset maps are useful since they capture a collection of poset maps with little data: representing all D_m^n poset maps $\mathbf{I}^m \to \mathbf{I}^n$ with the total ppm $x \mapsto \mathbf{I}^n$ for $x \in \mathbf{I}^m$ requires 2^m entries of 2^n values—the memory requirements are independent of the Dedekind numbers and grow only exponentially with the dimensions.

Algorithm 5.1 Update potential poset map

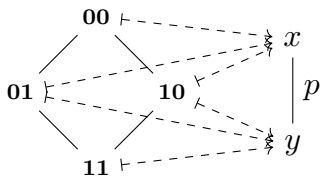
Input: $\Sigma : \mathbf{I}^m \to \mathcal{P}(\mathbf{I}^n)$ ppm, $x \in \mathbf{I}^m$, $vs \subseteq \Sigma(x)$
Output: Updated ppm $\Sigma' : \mathbf{I}^m \to \mathbf{I}^n$ with $\Sigma(x) = vs$.

```
procedure UpdatePPM(Σ, x, vs))
    for y ∈ I^m do
        if x = y then
            Σ'(y) := vs
        else if y ≤ x then
            Σ'(y) := {u | u ∈ Σ(y) such that ∃v ∈ vs : u ≤ v}
        else if x ≤ y then
            Σ'(y) := {u | u ∈ Σ(y) such that ∃v ∈ vs : v ≤ u}
    return Σ'
```

When updating a ppm we have to make sure that superfluous elements are eliminated and the properties of a ppm are maintained, for which we can use Algorithm 5.1. The algorithm takes a ppm and a restriction of values vs at an element x and yields an updated ppm. It works by looping through all elements in the domain of the ppm and restricting their values to contain only elements which satisfy monotonicity with respect to at least one element of vs. The algorithm has a worst-case runtime of $\mathcal{O}(2^m 2^n)$.

We can unfold a ppm Σ into the set of poset maps that it represents with a procedure UnfoldPPM(Σ). The procedure can be implemented by non-deterministically choosing any possible value v for an element x and propagating this choice through the ppm with UpdatePPM($\Sigma, x, \{v\}$). Since for a ppm $\Sigma : \mathbf{I}^m \to \mathbf{I}^n$ we have $\mathcal{O}(D_m^n)$ many poset maps, the runtime of UnfoldPPM(Σ) is determined by the Dedekind numbers.

Example 5.5. Given again context $\Gamma = x, y : ()[], p : (i)[\, i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y\,]$ and consider the potential poset map $\mathbf{0} \mapsto \{\mathbf{0}\}$, $\mathbf{10} \mapsto \{\mathbf{0}, \mathbf{1}\}$, $\mathbf{01} \mapsto \{\mathbf{0}, \mathbf{1}\}$ and $\mathbf{11} \mapsto \{\mathbf{1}\}$. The ppm can be unfolded into four poset maps as we can choose the values for $\mathbf{01}$ and $\mathbf{10}$ independently. The resulting squares correspond to the four possible ways in which p can be stretched into a square. To understand the action of the ppm on p, we can illustrate it by filling in the definition of p into the image of the ppm as follows.



All contortions represented by the ppm have in common that the bottom left corner is x and the top right corner is y, which we can read off from the potential poset map as it has unique values for $\mathbf{00}$ and $\mathbf{11}$. △

We now give an algorithm to construct a contortion for a given boundary using the notion of a potential poset map. The algorithm starts with a total ppm Σ and goes through the faces of the goal boundary to restrict Σ until it only contains valid contortions. It follows the heuristic that faces of the boundary which stem from contorting higher-dimensional cells restrict the search space the most.

In the following, we are given a context Γ containing a variable p and try to contort p into some boundary ϕ over Ψ. We will write $\mathbf{I}^n_{i=e} \coloneqq \{x \in \mathbf{I}^n \mid x_i = e\}$, which is a subposet of $\mathbf{I}^n$.

Algorithm 5.2 Constructing a Dedekind contortion

Input: $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy with $p : (\Psi')[\,\phi'\,] \in \Gamma$
Output: $\psi \colon \Psi \nrightarrow \Psi'$ s.t. $\Gamma \mid \Psi \vdash_{\mathrm{c}} p\langle\psi\rangle : [\,\phi\,]$ if such a ψ exists, `Unsolvable` otherwise

```
1:  procedure DedekindContortion(φ)
2:      m := | Ψ |
3:      n := | Ψ' |
4:      Σ := {x ↦ I^n | x ∈ I^m}
5:      for q⟨ψ : Ψ[i=e] ↛ Ψ'⟩ := φ[i=e] with i ∈ Ψ and | Ψ' | descending do
6:          if p = q then
7:              Θ := {x ↦ {ψ_I(x)} | x ∈ I^m_{i=e}}
8:          else
9:              Θ := {x ↦ ∅ | x ∈ I^m_{i=e}}
10:             for σ ∈ UnfoldPPM(Σ|_{I^m_{i=e}}) do
11:                 if φ[i=e] = p⟨σ_{∨∧}⟩ then
12:                     for x ∈ I^m_{i=e} do
13:                         Θ(x) := Θ(x) ∪ {σ(x)}
14:         for x ∈ I^m_{i=e} do
15:             UpdatePPM(Σ, x, Θ(x))
16:     if ∃σ ∈ UnfoldPPM(Σ) such that Γ | Ψ ⊢_c p⟨σ_{∨∧}⟩ : [φ] then
17:         return σ_{∨∧}
18:     else
19:         return Unsolvable
```

Consider Algorithm 5.2. We first generate the total ppm Σ on line 4. We then go through each face $q\langle\psi\rangle$ of the goal boundary ϕ and use it to restrict Σ. Crucially, we order the boundary faces by the dimension of the contorted variable.

Given a face $q\langle\psi\rangle$ of ϕ, we proceed as follows: if the face is a contortion of cell p, we give unique values for all elements of the subposet using ψ on line 7. Otherwise, we generate all possible faces that could be generated from Σ at the subposet and only keep values that contributed to some valid face. We finally propagate our findings through the whole poset map Σ on line 15.

After restricting Σ according to all faces of ϕ, we generate a valid solution and return it on line 16. This check ensures that Algorithm 5.2 is correct since it only returns a valid contortion. Note that we have to do this step as there might be multiple solutions to the problem, and since a ppm captures all solutions at once, we have to pick a single solution.

Moreover, Algorithm 5.2 is complete: suppose that p can be contorted with ψ to solve the goal boundary, and consider the corresponding poset map $\psi_{\mathbf{I}} : \mathbf{I}^m \to \mathbf{I}^n$. For any $x \in \mathbf{I}^m$, the algorithm will keep $\psi_{\mathbf{I}}(x)$ in Σ, by the operations either on line 7 or line 13.

The most expensive step in Algorithm 5.2 is unfolding all poset maps of the subposet on line 10. In the worst case, we have D^n_{m-1} many poset maps that we need to check, and as we are doing this for all $2m$ faces of ϕ, we have a factor of $2mD^n_{m-1}$ in the runtime of this algorithm. This constitutes an improvement over naïvely checking all D^n_m poset maps as the Dedekind numbers grow quicker than the runtime of any other operation in our algorithm.

In many practical cases, the cell to be contorted appears in the boundary, which means the search space significantly reduces before any ppm is unfolded, as demonstrated in Example 5.6. This means that we can compute many contortions that would have been impossible to find by brute-force.

Example 5.6. Given again context

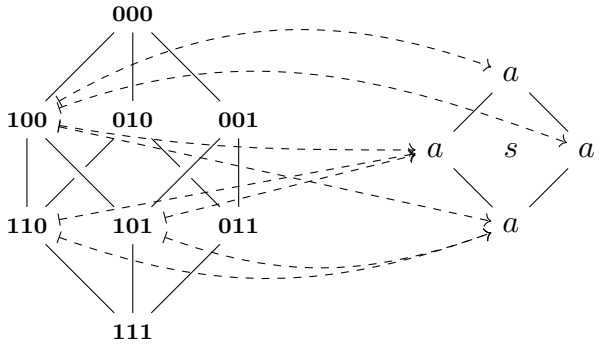
$$\Gamma_1 := \left\{ \begin{array}{l} a : ()[\,], \\ s : (i,j)[\, i{=}\mathbf{0} \mapsto a\langle \mathbf{s}_j\rangle \mid i{=}\mathbf{1} \mapsto a\langle \mathbf{s}_j\rangle \mid j{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i\rangle \mid j{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle \,] \end{array} \right.$$

we want to find a contortion $\psi\colon i,j,k \rightsquigarrow i,j$ such that $s\langle\psi\rangle$ has the following boundary:

$$\Gamma \mid i,j,k \vdash_{\mathrm{c}} s\langle\psi\rangle \ \mathsf{tm} : \left[\begin{array}{l} i{=}\mathbf{0} \mapsto s\langle i \mapsto j \wedge k, j \mapsto j \vee k\rangle \\ i{=}\mathbf{1} \mapsto a\langle \mathbf{s}_j\rangle\langle \mathbf{s}_k\rangle \\ j{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i\rangle\langle \mathbf{s}_k\rangle \\ j{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle\langle \mathbf{s}_k\rangle \\ k{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i\rangle\langle \mathbf{s}_j\rangle \\ k{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i\rangle\langle \mathbf{s}_j\rangle \end{array} \right]$$

Note that constructing such a contortion is a difficult instance of DEDEKIND: since we only have a single 0-cell in the context, we cannot restrict the search space by tracing where the points of the cube need to go. Instead, we have to look at the maximal faces to restrict the search space.

We run Algorithm 5.2 on this problem. On line 4, the total ppm $\Sigma : \mathbf{I}^3 \to \mathcal{P}(\mathbf{I}^2)$ is initialized with $x \mapsto \mathbf{I}^2$ for all $x \in \mathbf{I}^3$. We then go through all faces of the goal boundary and use them to restrict Σ. The face of the boundary with the largest original dimension is a contortion of s. Since s is also the cell that we are contorting, the subposet $\mathbf{I}^3_{1=\mathbf{0}}$ of the domain of Σ is mapped in a unique way to the elements of $\mathbf{I}^2$. The monotonicity restrictions on ppms further restrict Σ: since **011** is mapped to the bottom element **11**, also **111** will need to be mapped to **11**. The elements **101** and **110** lie below elements which are mapped to **01**, which means there are only two possible values $\{\mathbf{01}, \mathbf{11}\}$ for each of these elements. Only the element **100** has the full poset $\mathbf{I}^2$ as potential values. In summary, updating the ppm on line 15 after the first restriction of Σ leads to the following potential poset map, were elements which have no outgoing arrows have a unique value:



In the next iteration of restricting Σ, we only have degenerate a faces left. For instance, using the k=**1**-th face we restrict Σ as follows: the only choice left for $\mathbf{I}^3_{3=\mathbf{1}}$ is whether element **101** is sent to **01** or to **11**. Sending **101** to **11** would yield a contortion of s which does not normalise to $a\langle \mathbf{s}_i \rangle \langle \mathbf{s}_j \rangle$; we therefore discard this choice and assign to **101** the singleton value **01**.

Even though unfolding a ppm is slow in general, calls to UNFOLDPPM are quick in this case as there are only 10 poset maps in the ppm Σ remaining after restricting Σ according to the first face of ϕ. This contrasts with having to check $D_3^2 = 20^2 = 400$ poset maps with a brute-force search.

The increase in speed gets apparent for a larger goal: a 6-dimensional analogue of the above proof goal can be found by unfolding less than 16,000 poset maps. A brute-

force search would have to find a solution in a search space with $D_6^2 = 7,828,354^2 = 61,283,126,349,316$ contortions. △

5.1.3 Potential poset maps in the Kan solver

Potential poset maps allow us to capture a collection of Dedekind contortions with little data. This is particularly useful for the constraint satisfaction problems introduced in Section 4.3, since we can represent domains for the CSPs without enumerating all Dedekind contortions.

To incorporate ppms, we have to change the CSPs from Definitions 4.15 and 4.17 such that their domains are not sets of terms, but sets of pairs (p, Σ) where $p : (\Psi')[\,\phi'\,]$ and $\Sigma : \mathbf{I}^{|\Psi|} \to \mathbf{I}^{|\Psi'|}$. When imposing a boundary constraint on two variables X and X', the relevant ppms have to be unfolded and compared. For details, we refer the interested reader to the implementation of the solver discussed in Section 5.2.1; we simply demonstrate with an example how potential poset maps play out in the Kan solver.

Example 5.7. Take the boundary problem from Example 4.20, where we were given a context

$$\Gamma := \begin{cases} x, y, z : ()[\,], \\ p : (i)[\,i{=}\mathbf{0} \mapsto x \mid i{=}\mathbf{1} \mapsto y\,], \\ q : (i)[\,i{=}\mathbf{0} \mapsto y \mid i{=}\mathbf{1} \mapsto z\,] \end{cases}$$

and tried to derive a term with boundary ϕ given by

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\,j{=}\mathbf{0} \mapsto p \mid j{=}\mathbf{1} \mapsto q \mid i{=}\mathbf{0} \mapsto p\langle i \mapsto j\rangle \mid i{=}\mathbf{1} \mapsto q\langle i \mapsto j\rangle\,] \ \mathsf{bdy}.$$

We now show how this boundary problem is solved by Algorithm 4.1 based on ppms representing Dedekind contortions. The procedure will first try to construct a solution for ϕ as a Kan filling term of an open cube consisting only of contortions, i.e., it looks for a solution of CompCSP$(\phi, \emptyset)$. This CSP has 5 variables with the following initial domains, where we write $t\langle\Sigma\rangle$ for the pair (t, Σ):

$$D_{(j=\mathbf{0})} = D_{(i=\mathbf{0})} = \{p\langle{}^{\mathbf{00}\mapsto\mathbf{0}}_{\mathbf{10}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{01}\mapsto\mathbf{0}}_{\mathbf{11}\mapsto\mathbf{1}}\rangle\},$$

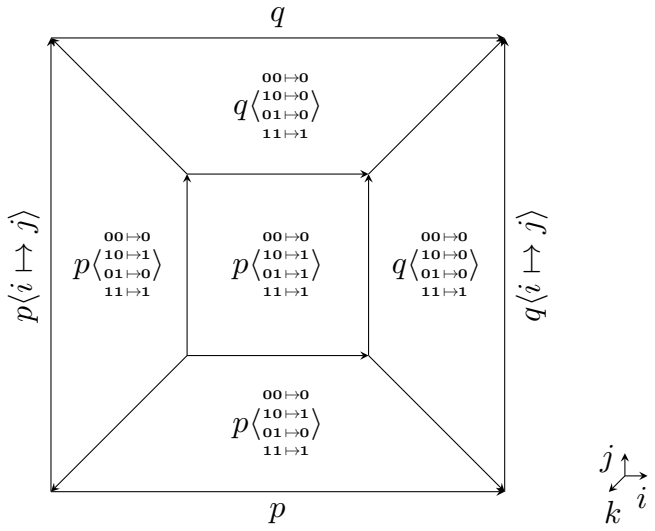
$$D_{(j=\mathbf{1})} = D_{(i=\mathbf{1})} = \{q\langle{}^{\mathbf{00}\mapsto\mathbf{0}}_{\mathbf{10}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{01}\mapsto\mathbf{0}}_{\mathbf{11}\mapsto\mathbf{1}}\rangle\},$$

$$D_{(k=\mathbf{0})} = \{x\langle{}^{\mathbf{00}\mapsto()}_{\mathbf{10}\mapsto()}_{\mathbf{01}\mapsto()}_{\mathbf{11}\mapsto()}\rangle, y\langle{}^{\mathbf{00}\mapsto()}_{\mathbf{10}\mapsto()}_{\mathbf{01}\mapsto()}_{\mathbf{11}\mapsto()}\rangle, z\langle{}^{\mathbf{00}\mapsto()}_{\mathbf{10}\mapsto()}_{\mathbf{01}\mapsto()}_{\mathbf{11}\mapsto()}\rangle, p\langle{}^{\mathbf{00}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{10}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{01}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{11}\mapsto\mathbf{0},\,\mathbf{1}}\rangle, q\langle{}^{\mathbf{00}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{10}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{01}\mapsto\mathbf{0},\,\mathbf{1}}_{\mathbf{11}\mapsto\mathbf{0},\,\mathbf{1}}\rangle\}.$$

Note that the domain for the back of cube $D_{(k=\mathbf{0})}$—which is completely unconstrained by the goal boundary—contains only five entries. Each of the entries for

p and q represents all six ways in which the respective path can be contorted into a square (in fact, the two entries for p and q subsume the degenerate x, y and z contortions, which could be omitted in an optimised implementation).

A backtracking procedure will find the only solution to this CSP: when checking the constraint $X_{(j=\mathbf{0})}\langle i \mapsto \mathbf{1}\rangle = X_{(i=\mathbf{1})}\langle j \mapsto \mathbf{0}\rangle$, the solver finds that this can only be satisfied if these sides of the box share the boundary $y\langle \mathbf{s}_k \rangle$, the domain $D_{(j,\mathbf{0})}$ is hence restricted to only contain $p\langle \substack{00\mapsto 0\\10\mapsto 1\\01\mapsto 0\\11\mapsto 1} \rangle$. Propagating this change and satisfying all other constraints then leads to the following solution:



In order to translate this solution back to the algebraic characterisation of Dedekind contortion, we have to apply the transformation from Proposition 5.3 and replace the indices by the variable names bound in each respective term. For instance, side $(j{=}\mathbf{0})$ of the above cube is spanned by the interval variables i and k, the contortion corresponding to the poset map applied to p is $\mathbf{s}_k \colon (i,k) \twoheadrightarrow (i)$. △

The solver from Algorithm 4.1 can therefore effectively be combined with Dedekind contortions regarded as poset maps. Representing the domains for the constraint satisfaction problems using potential poset maps requires little memory, and the constraint solver can gradually narrow down the ppms until it arrives at a solution. In contrast to the solution based on the cartesian contortion theory from Example 4.20, solutions over the Dedekind contortion theory are often significantly more concise and require fewer Kan filling terms.

5.2 A practical cube solver

We now put to test our procedures for solving the general boundary problem. We present an experimental solver implemented in Haskell in Section 5.2.1 and benchmark the solver on a set of boundary problems in Section 5.2.2. We focus our analysis on two aspects. First, we test how the simple heuristic of using contortions where possible and constructing nested Kan filling terms as separate CSPs works in practice. Second, we compare how well the solver works for different contortion theories.

5.2.1 An experimental solver

We have implemented Algorithms 4.1 and 5.2 in Haskell to provide a solver for boundary problems in cubical type theory; its source code can be found online[3]. The implementation is based on the back-tracking constraint solver of Overton [67] and we use de Bruijn levels for dimension variables.

The user provides a cube context and boundary problem over some contortion theory. The solver then attempts to find a term with this boundary in the respective contortion theory and, if that fails, tries to construct a term in the Kan theory. In the current implementation of the solver, the user has to encode the boundary problem manually in the language of the solver. The solver can output terms in the syntax of Cubical Agda; the output then needs to be copied into the relevant place in the Agda file. In order to use lemmas, the user has to explicitly extend the context with the terms instantiated appropriately. This allows for also deriving automatically all homogeneous compositions constructed in Chapter 3.

A proper integration into Cubical Agda is work in progress and we hope to distribute a tactic via the Cubical library soon. Once the tactic is implemented, the user can decide whether the terms generated by the solver (which can be lengthy, as we see below) should be an explicit part of the formalisation, or kept hidden from the user. Eventually, we also want to combine our solver with automated reasoning capabilities of Agda, in particular with Agsy [52].

As discussed in Section 3.2, Cubical Agda offers a primitive term for homogeneous composition, hcomp, which acts very similar to our fill, but is limited to compositions in direction $\mathbf{0}\to\mathbf{1}$. Our Algorithm 4.1 looks precisely for Kan terms in this direction such that we can simply translate all Kan filling terms into the corresponding hcomp terms. On the level of contortion terms, the translation is also straightforward since Cubical Agda's syntax is less verbose, e.g., weakening is implicit.

[3] `https://zenodo.org/doi/10.5281/zenodo.10966924`

In order to prove in Cubical Agda that path composition as defined on page 70 is associative, we need to construct a term of type $(p \cdot q) \cdot r \equiv p \cdot (q \cdot r)$. After abstracting over two interval variables i and j, the goal has the following boundary:

$$
\begin{aligned}
i = \mathsf{i0} &\vdash ((p \cdot q) \cdot r)\ j \\
i = \mathsf{i1} &\vdash (p \cdot (q \cdot r))\ j \\
j = \mathsf{i0} &\vdash w \\
j = \mathsf{i1} &\vdash z
\end{aligned}
$$

In our syntax, this corresponds to the boundary discussed in Example 4.21:

$$\Gamma \mid i, j \vdash_{\mathrm{k}} [\, i{=}\mathbf{0} \mapsto (p \cdot q) \cdot r \mid i{=}\mathbf{1} \mapsto p \cdot (q \cdot r) \mid j{=}\mathbf{0} \mapsto w\langle \mathbf{s}_i \rangle \mid j{=}\mathbf{1} \mapsto z\langle \mathbf{s}_i \rangle \,] \ \mathsf{bdy}$$

Note that our syntax is more explicit than Agda's: we have to explicitly introduce the degeneracies of w and z. After encoding this boundary in the solver, we can let it run over any contortion theory. Using the Dedekind contortion theory represented with potential poset maps, the solver needs about 2 seconds to produce the following:

```
assoc : (p · q) · r ≡ p · (q · r)
assoc i j = hcomp (λ k →
  λ { (j = i0) → w
    ; (j = i1) → hcomp (λ l →
                  λ { (i = i0) → hcomp (λ m →
                                  λ { (l = i0) → q k
                                    ; (l = i1) → r (k ∧ m)
                                    ; (k = i0) → q l
                                    ; (k = i1) → r (l ∧ m) }) (q (l ∨ k))
                    ; (k = i0) → hcomp (λ m →
                                  λ { (i = i0) → q (l ∧ m)
                                    ; (i = i1) → x
                                    ; (l = i0) → x }) (q i0)
                    ; (k = i1) → r l }) (q k) })
  (hcomp (λ l → λ { (i = i1) → p j
                  ; (j = i0) → w
                  ; (j = i1) → hcomp (λ m → λ { (i = i0) → q (l ∧ m)
                                              ; (i = i1) → x
                                              ; (l = i0) → x
                                              }) (q i0) }) (p j))
```

We can also use the solver to derive the Eckmann-Hilton argument, which establishes that two-dimensional path composition is commutative. In contrast to a purely algebraic proof, showing this argument in a proof-relevant setting is complicated by the fact that higher coherences have to be dealt with. Following the presentation in [77, Theorem 2.1.6], we give the Eckmann-Hilton argument for elements of the second *loop space*, which can be captured in Cubical Agda as the following type:

```
Ω : Σ[ A ∈ Type ] A → Σ[ A ∈ Type ] A
Ω (_ , a) = ((a ≡ a) , refl)
```

The loop space collects all the loops around a given base point $a : A$. Since we always have the refl path around a, we can iterate the loop space by considering loops around refl in the next iteration. To prove commutativity of path composition between elements of the second loop space over a given (A, a), we need to show for $p, q :$ fst $(\Omega\ (\Omega\ A))$ that we have a term of type $p \cdot q \equiv q \cdot p$.

We first establish a lemma about path composition: any square can be turned into a square with constant sides in one dimension, and composed paths on the sides in the other dimension. For our purposes, we are interested in the case where we have two paths on opposing sides. We can quickly derive the following lemma in the Dedekind contortion theory using our solver:

```
Sq→Comp : (p q : x ≡ x) → PathP (λ j → q j ≡ q j) p p → p · q ≡ q · p
Sq→Comp p q α i j = hcomp (λ k → λ {
    (j = i0) → x
  ; (j = i1) → hcomp (λ l → λ {
        (i = i0) → α (k ∧ l) l
      ; (i = i1) → α l (k ∧ l)
      ; (k = i1) → α l l
      }) x
  }) (hcomp (λ m → λ {
    (i = i0) → p (m ∧ j)
  ; (i = i1) → q (m ∧ j)
  ; (j = i0) → x
  }) x)
```

This lemma entails that in order to prove commutativity of composition of p and q, it suffices to provide a filler for a 3-dimensional cube which has p and q on opposing

sides in two dimensions as follows:

$$\Gamma \mid i, j, k \vdash_{\mathrm{k}} \left[\begin{array}{l} i{=}\mathbf{0} \mapsto p \\ i{=}\mathbf{1} \mapsto p \\ j{=}\mathbf{0} \mapsto q \\ j{=}\mathbf{1} \mapsto q \\ k{=}\mathbf{0} \mapsto a\langle \mathbf{s}_i, \mathbf{s}_j \rangle \\ k{=}\mathbf{1} \mapsto a\langle \mathbf{s}_i, \mathbf{s}_j \rangle \end{array}\right] \mathsf{bdy}$$

This boundary problem can be solved using a Kan filling term over the Dedekind contortion theory; the term looks as follows in Cubical Agda:

```
EckmannHilton-Cube : PathP (λ i → q i ≡ q i) p p
EckmannHilton-Cube i j k =
  hcomp (λ l → λ { (i = i0) → p j (k ∧ l)
                 ; (i = i1) → p j (k ∧ l)
                 ; (j = i0) → q i k
                 ; (j = i1) → q i k
                 ; (k = i0) → a
                 ; (k = i1) → p j l }) (q i k)
```

This proof term is difficult for a human to derive, but can be found quickly by our solver—since the term has no nested Kan terms, the solver only has to solve a single constraint satisfaction problem. Using the two lemmas, we can establish the original formulation of Eckmann-Hilton:

```
EckmannHilton : p ∙ q ≡ q ∙ p
EckmannHilton = Sq→Comp p q EckmannHilton-Cube
```

In conclusion, the solver can often directly find solutions for 1-dimensional boundary problems. For higher-dimensional boundary problems, it is often expedient to decompose Kan filling terms in the goal boundary and derive lemmas guiding the solver. In principle, the solver can also find directly a term for EckmannHilton without requiring the user to come up with the lemma Sq→Comp, but the current implementation of our solver has not been able to do this within 10 minutes.

5.2.2 Benchmarking the solver

As discussed in Section 5.1, more expressive contortion theories have the trade-off that they need fewer Kan terms to fill some boundary, but the number of contortions

grows quicker, making each constraint solving problem in Algorithm 4.1 more difficult. It is hence expedient to study how this trade-off pans out in practice.

We have curated a set of 14 exemplary boundary problems formulated over the cartesian contortion theory. Since all other contortion theories are extensions thereof, we can use the solver with any contortion theory to solve these problems. We have run the solver on a standard MacBook with M1 processor with a timeout of 60 seconds, the number of problems solved and the average time needed for the successful runs can be found in Table 5.1.

Problem	Cartesian	Conj	Disj	Dede	Dede (ppm)	DeMorgan
Meet	5ms	2ms	1ms	4ms	2ms	7ms
Join	5ms	2ms	29ms	2ms	2ms	5ms
SqComp	108ms	9ms	8ms	1ms	1ms	3ms
Right unit	0ms	1ms	0ms	1ms	1ms	2ms
Left unit	3ms	1ms	6ms	1ms	1ms	2ms
Associativity α	1ms	16ms	5ms	29ms	24ms	2ms
Associativity β	43ms	22ms	1s	3ms	3ms	9ms
Associativity	17s	545ms	✗	387ms	338ms	98ms
Associativity'	✗	365ms	✗	18ms	14ms	96ms
Associativty $\vee$	✗	✗	✗	44ms	41ms	874ms
Associativity $\wedge$	✗	12s	✗	17s	16s	✗
Loop associativity	✗	39s	✗	✗	✗	✗
EH square	✗	131ms	✗	✗	953ms	✗
EH direct	✗	✗	✗	✗	✗	✗
# Solved	8	12	7	11	12	10

Table 5.1: Performance of contortion theories

As we would expect, the cartesian contortion theory solved only 8 out of the 14 boundary problems, since the solver has to construct many Kan terms for problems which can be solved with contortions in other theories.

We have also studied the contortion theory consisting only of the conjunctive contortion, called Conj in Table 5.1. Interestingly, it could solve 12 out of the 14 problems. However, this is likely due to a bias in the benchmarking suite, as the disjunctive theory only solved 7 of the problems. Since both theories are dual to each other, we expect that both theories work equally well on a larger set of examples.

For the Dedekind contortion theory, we benchmarked both the naïve implementation listing all contortions, and an implementation based on potential poset maps. The solver based on ppms was consistently quicker than the naïve Dedekind solver,

and could consequently solve more problems. Both Dedekind solvers were also superior to the DeMorgan solver since listing all DeMorgan contortions explicitly takes a lot of time.

We can therefore conclude that the refinement of the solver based on the poset map characterisation of Dedekind contortions means an improvement over a solver simply listing all contortions. In practice, it is worthwhile to also run the solver with different contortion theories, as contortion theories have biases towards certain problems. The simpler contortion theories based on a single connection can resolve fewer boundary problems, but these sometimes very quickly.

More work is necessary to enlarge the benchmarking suite. Once the solver is part of Cubical Agda, it will likely be applied to more problems, which will suggest how to improve its workings and heuristics. This will also provide stronger evidence for which contortion theory can be automated most easily.

Chapter 6

Conclusions

In the preceding chapters we have seen different ways in which we can perform computations with, and establish properties of, topological spaces in cubical type theory. Before we reflect on the outcomes of this thesis, it might be worthwhile to recap how it came about. In the beginning of my DPhil stood the wonderment that while dependent type theory is a programming language, it is rarely used as such in mathematical formalisations. Following Samson's suggestion, I looked at topological data analyis since it is a mathematical area concerned with both mathematical algorithms and topological spaces, and hence a natural candidate for formalisation in Cubical Agda. I started out trying to formalise the central result of Nanda [62], who develops discrete Morse theory for simplicial complexes in a way that the homotopy type of the Morse reduct can be recovered. Even after considerable effort and time, I had not found a fruitful way to reason about simplicial complexes of arbitrary dimension in Cubical Agda and paused this project. During my formalisation efforts, I had become intimately familiar with Cubical Agda and its reasoning principle of homogeneous composition and wondered how one could apply it systematically. A prototype deriving homogeneous compositions based on constraint satisfaction programming was quickly developed, and Evan and Anders gave me precious impetus to carry this project further after a workshop talk. After completing a joint project with them on Dedekind cubes, I returned to formalising discrete Morse theory, this time focusing on the simplified result for graphs. With now plentiful experience with Cubical Agda under my belt, I could bring this project to completion in due course; and subsequently wrote up this thesis.

In the following, I will summarise the findings from these projects—both what worked well, and where the chosen approaches and tools showed deficiencies. I will outline how the presented projects could be carried further and close with an opin-

ionated view on where the journey of formalising mathematics in type theory may lead us.

Contributions & benefits. We have seen how formalisation in dependent type theory offers a novel perspective on mathematics. Mathematicians primarily think of mathematical objects in terms of *specifications*, while computer scientists think about how to *compute* the objects in terms of concrete representations. In type theory, we can fruitfully combine both viewpoints.

This makes it possible to implement complex mathematical algorithms in a way that they provably conform to the mathematical theory. We have laid the groundwork for a tool for topological data analysis which is correct by construction. We have seen how reasoning about homotopy types in Cubical Agda can be applied to a new domain, namely discrete Morse theory. We can directly talk about homotopy types, a structure that is otherwise intricate to set up and reason about.

Using the novel logic for homotopy types offered by cubical type theory can sometimes be tricky, but we have seen that many reasoning steps can be automated. We have devised a general method for finding cells in a cubical set, which amounts to automating equational reasoning in cubical type theory. We hope that our cube solver will provide a powerful tool to advance mathematics developed in a synthetic fashion. At the moment, progress in this field is lagging behind mathematics pursued with traditional methods, e.g., deriving the Syllepsis, a coherence property of the Eckmann-Hilton argument which we showed in Section 5.2, took considerable effort in homotopy type theory [72]. Using an automated tool which can reason with these coherence conditions offers the chance to accelerate synthetic mathematics.

We have shed light on computational aspects of different variations of cubical type theory, possibly contributing to the convergence of the field towards one particular version of cubical type theory.

Learnings & limitations. We have seen that following a correct by construction approach for mathematical functions is quite laborious—even a simple function such as the powerset program developed in Chapter 2 required to write hundreds of lines of code to make sure that the program lines up with the mathematical definition of the powerset. This is in part due to our chosen theorem prover Agda which has limited support for tactics, in contrast to tools like Coq or Lean. While we have devised a general method to compute with dependent pair types, we need to have a clearer picture on how to compose objects and programs to scale our approach

to more intricate mathematical objects and functions. This is a challenge faced by correct by construction programming more generally [8].

While using Lean or Coq would have made some of the proofs developed in this thesis shorter, these tools also have their shortcomings as discussed in Section 2.3 (and they would not have been suitable for the developments in Chapter 3). Combining the strengths of different theorem provers for a single project would be very useful but is difficult to do in practice. While there is considerable work on increasing the interoperability of different theorem provers, formalising mathematics fundamentally requires the monolithical development of mathematics, which means a considerable coordination and maintenance effort.

While we have been able to carry out all work we wanted in Cubical Agda, there are a variety of technical peculiarities making its use more tricky and laborious than necessary. In particular the management of h-levels and the interaction with boundary problems could be improved. Future theorem provers based on cubical type theory will likely alleviate these issues.

We have seen that Kan filling is a powerful reasoning principle which can be automated in many cases. To some extent, this can hide that cubical type theory is built on a comparatively weak model of homotopy types, but not fully. Alternative approaches to higher-dimensional type theories such as those of Finster, Rice, and Vicary [24], Finster et al. [23] and Thanh, Curien, and Mimram [75] are based on stricter models. As a consequence, many statements that we had to prove explicitly hold by judgmental equality in these theories. More work is necessary to understand the interplay of these different proposals and proliferate the exchange of useful concepts and methods. In general, proof-relevant higher-dimensional equality is a subject of considerable complexity, and its widespread application remains impeded by a lack of understanding of the subject matter.

Open questions & future work. We want to close with a number of questions that remain open, as well as some future work that was not prevented by a lack of understanding, but rather research time.

We will need to formalise discrete Morse theory for more complex topological spaces to turn the results developed in Chapter 3 into a practical tool for topological data analysis. For some such spaces this should be possible relatively straightforwardly, e.g., it should be possible to extend our approach to 2-dimensional cubical complexes, as used by Robins, Wood, and Sheppard [70] to analyse grayscale images.

It is unclear how best to reason about and manage complexes of an arbitrary dimension. More work is necessary to make synthetic methods workable in the context of applied topology.

As discussed in Chapter 4, Doré, Cavallo, and Mörtberg [19] present results about the complexity theory of Dedekind, namely that it is NP-complete for 1-dimensional proof goals. More work is necessary to understand the complexity of the different versions of CubicalCell, in particular with respect to different input parameters. The additional expressivity of, e.g., Dedekind comes with the cost of an increase in complexity, and it might turn out that some contortion theory provides a particularly good trade-off between expressivity and complexity. We need to gather further evidence, both theoretical and practical, to have a clearer view on the efficacy of the different contortion theories.

Our Algorithm 4.1 presents a first attempt at automatically solving boundary problems. It will become apparent how it can be tweaked and improved when using the solver for more examples. Besides improving the algorithms itself—e.g., we could try to unfold potential poset maps more cleverly when using the solver with the Dedekind contortion theory—it is worthwhile to try out different heuristics from ours. The implementation in Haskell could be optimised to provide a faster solver which resolves more goals. Integrating it properly into Cubical Agda is work in progress and we hope to distribute the solver as a standard part of the Cubical Agda library in due course.

Combinatorially, Kan filling is the most complicated reasoning principle introduced by cubical type theory, but it is a long way from being the only one. It will be useful to generalise our solver to also work with heterogeneous equalities, which means that we have to consider homotopical structure over more than one type. Moreover, it would be very useful to have an automatic method to reason with transports, which manifest itself in Cubical Agda as `transp` (we have not discussed `transp` in this thesis, but have tacitly used it whenever applying `subst`). More automation for cubical type theory would make its use significantly easier.

Outlook. With the rise of interactive theorem provers, many of which are based on dependent type theory, the relevance of computations in mathematics will increase. Since we can both reason and program in the same language, our view on mathematics changes. We can engage more directly with the mathematical objects under question and can design algorithms alongside the mathematical theory. We hope that this thesis has provided some indications how this change will pan out in practice.

Moreover, we hope to have stressed the importance of foundational work on theorem provers—also to researchers who are only interested in theorem provers from a user perspective. While it remains a challenge to logically reconstruct and utilise proof-relevant equality, it is a worthwhile endeavour. Only if we pay attention to details will we be able to devise a well-understood and well-working theorem prover; and once we have understood its underlying logic—however complicated—automating and streamlining it will make its use easier.

Index of Agda symbols

++-dec, 28
++-ordered, 21
Bool, 13
Bool_1, 45
Discrete, 13
FacePoset, 68
Fin, 13
Graph, 45
H^1-two≅$\mathbb{Z}$, 60
Iso, 16
Listable, listable, 17
List, 13
Listη, 24
Matching, 48
MorseProgram, 59
MorseTheorem, 59
Powerset, 31
SimplicialComplex, 67
$\text{S}^1 \vee \text{S}^1 \equiv |\text{Bool}_1|$, 45
$\text{S}^1 \equiv |\top_1|$, 45
Type, 12
V-isHSet, 45
[]⊆$_l$-all, 27
_++_, 13
acyclic, 48
cell, 67
coHomGr, 41
compCrit, 62
compGrad, 62
crit?, 62
critIsHSet, 47
crit, 47
crit_1, 49
dec, 17
emptySet, 18
end, 47
exp-μ, 61
exp_1?, 64
exp_1, 61
insertL-head, 28
insertL-tail, 28
insertL, 26
insert, 26
isContr, 15
isCritIsHProp, 47
isCrit, 47
isEven, 22
isGroupoid, 15
isHProp, 15
isHSet, 15
isHSet→SquareP, 43
isoToPath, 16
isoToReal, 45
line-acy, 51
line-isContr, 60
line-μ, 51

line, 46
line$_1$?, 63
line$_1$, 46
linv, 57
listBool, 18
listFamily, 30
listProp, 20
listΣ, 20
list$\mathbb{N}$, 19
map, 13
mp, 47
ordered, 18
powerset-comp, 29
powerset-insert, 26
powerset-ordered, 30
powerset-sound, 27
powerset, 25
rinv, 53
set-isHSet, 20
setBool, 18
setEven, 22
setFin, 19
setOrder, 30
setProp, 20
set, 18
setΣ, 21
set$\mathbb{N}$-ordered, 19
set$\mathbb{N}$, 19
sinkUniq, 49
sinkUniq^c, 49
toList$\equiv$, toSet$\equiv$, 22
two-μ-acy, 50
two-μ, 50
two, 46
two$_1$?, 63
two$_1$, 46
x$\sqsubset$Lpowerset, 26
$\neg$, 13
$\times$G, 41
Δ, 67
ι, 52
ϕ, 52
$_v$, 47
ℓ-suc, 12
$\mathbb{N}$, 13
$\mathbb{Z}$Group, 41
$\Rightarrow^m$?, 47
$\Rightarrow^m$IsHProp, 47
$\Rightarrow^m$, 47
$\Rightarrow^c$?, 62
$\Rightarrow^c$IsHProp, 47
$\rightsquigarrow$IsHProp, 49
$\rightsquigarrow$sink, 48
$\rightsquigarrow$, 47
$\in$-isHProp, 23
$\in$, 23
$\in_l$++-inj$_1$, 29
$\in_l$++-inj$_2$, 30
$\in_l$-singl-extract, 27
$|_\Rightarrow_\ni_|_1$, 44
$|_|$, 44
$|_|_0$, 44
$|_|\rightsquigarrow$, 52
$||\rightsquigarrow$Contr, 53
$\cong$, 41
$\equiv$, 15
$\not\equiv$, 15
$\ll$, 30
$\succcurlyeq$, 67
$\subseteq$-antisym, 24
$\subseteq$-dec, 24
$\subseteq$-isHProp, 24

⊆-step, 25
⊆[]→≡[], 29
⊆, 24
⊆$_l$-insert, 28
⊆$_l$-weaken, 28
⊎, 13
⊏, ⊏-irreflexive, ⊏-trans, ⊏-prop, 17
⊏-tails, 23
⊏LS, 26
⊏L, 21
⊏→⊉, 30
⊤*, 13
⊤Group, 41
⊤, 13
⊤$_1$, 44
⊤$_1$≡Mtwo, 50
⊤≡|⊥$_1$|, 44
⊥*, 13
⊥, 13
⊥$_1$, 44
$\mathcal{P}$-closed, 31
$\mathcal{P}$, 31
[], 13
_::_, 13
false, 13
inl, 13
inr, 13
no, 13
suc, 13
true, 13
tt*, 13
tt, 13
yes, 13
zero, 13
fst, 12
snd, 12

Bibliography

[1] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. "Computer algebra meets automated theorem proving: Integrating Maple and PVS". In: *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Springer. 2001, pp. 27–42. DOI: 10.1007/3-540-44755-5_4.

[2] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities". In: *27th EACSL Annual Conference on Computer Science Logic (CSL)*. 2018, 6:1–6:17. DOI: 10.4230/LIPIcs.CSL.2018.6.

[3] Steve Awodey. "Recent Work in Homotopy Type Theory: Modal, Algebraic, Synthetic, and Cubical". In: *Unified Foundations of Mathematics and Computation* (2018). Available at https://ncatlab.org/homotopytypetheory/files/awodeyMURI18.pdf.

[4] Steve Awodey and Michael A Warren. "Homotopy theoretic models of identity types". In: *Mathematical proceedings of the cambridge philosophical society*. Vol. 146. 1. Cambridge University Press. 2009, pp. 45–55. DOI: 10.1017/S0305004108001783.

[5] Rafarl Ayala, Luis M. Fernández, Antonio Quintero, and Jose Antonio Vilches. "A Note on the Pure Morse Complex of a Graph". In: *Topology and its Applications* 155.17 (2008), pp. 2084–2089. DOI: 10.1016/j.topol.2007.04.023.

[6] Marc Bezem, Thierry Coquand, and Simon Huber. "A Model of Type Theory in Cubical Sets". In: *19th International Conference on Types for Proofs and Programs (TYPES)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26.

Leibniz International Proceedings in Informatics (LIPIcs). 2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: 10.4230/LIPIcs.TYPES.2013.107.

[7] Bryan Birch and Peter Swinnerton-Dyer. "Notes on Elliptic Curves (II)". In: *Journal für die reine und angewandte Mathematik* 1965.218 (1965), pp. 79–108. DOI: doi:10.1515/crll.1965.218.79.

[8] Tabea Bordis, Tobias Runge, Alexander Kittelmann, and Ina Schaefer. "Correctness-By-Construction: an Overview of the Corc Ecosystem". In: *ACM SIGAda Ada Letters* 42.2 (Apr. 2023), pp. 75–78. DOI: 10.1145/3591335.3591343.

[9] Guillaume Brunerie. "On the homotopy groups of spheres in homotopy type theory". PhD thesis. Université Nice Sophia Antipolis, 2016. URL: https://arxiv.org/abs/1606.05916.

[10] Guillaume Brunerie, Axel Ljungström, and Anders Mörtberg. "Synthetic Integral Cohomology in Cubical Agda". In: *30th EACSL Annual Conference on Computer Science Logic (CSL)*. Ed. by Florin Manea and Alex Simpson. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). 2022, 11:1–11:19. DOI: 10.4230/LIPIcs.CSL.2022.11.

[11] Ulrik Buchholtz and Kuen-Bang Hou (Favonia). "Cellular cohomology in homotopy type theory". In: *Proceedings of the 33rd ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2018, pp. 521–529.

[12] Ulrik Buchholtz and Edward Morehouse. "Varieties of Cubical Sets". In: *Proceedings of the 16th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS)*. 2017, pp. 77–92. DOI: 10.1007/978-3-319-57418-9_5.

[13] Gunnar Carlsson and Mikael Vejdemo-Johansson. *Topological Data Analysis with Applications*. Cambridge University Press, 2021. DOI: 10.1017/9781108975704.

[14] Evan Cavallo. "Synthetic cohomology in homotopy type theory". MA thesis. Carnegie Mellon University, 2015. URL: https://ncatlab.org/nlab/files/Cavallo-CohomologyInHoTT.pdf.

[15] Evan Cavallo and Christian Sattler. *Relative elegance and cartesian cubes with one connection*. 2023. arXiv: 2211.14801.

[16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). 2018, 5:1–5:34. DOI: 10.4230/LIPIcs.TYPES.2015.5.

[17] Haskell B Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pp. 584–590.

[18] Edsger Wybe Dijkstra. *A discipline of programming*. Automatic Computation. Prentice-Hall, 1976.

[19] Maximilian Doré, Evan Cavallo, and Anders Mörtberg. *Automating Boundary Filling in Cubical Agda*. 2024. arXiv: 2402.12169.

[20] Christoph Dorn. "Associative n-categories". PhD thesis. University of Oxford, 2018. URL: https://ora.ox.ac.uk/objects/uuid:9f9d8668-a48a-4b30-9883-b11e4edbfc54.

[21] Xavier Dousson, Julio Rubio, Francis Sergeraert, and Yvon Siret. *The Kenzo program*. Available at http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/. 1998.

[22] Herbert Edelsbrunner and John Harer. *Computational topology: an introduction*. American Mathematical Society, 2010.

[23] Eric Finster, David Reutter, Jamie Vicary, and Alex Rice. "A type theory for strictly unital ∞-categories". In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2022, pp. 1–12. DOI: 10.1145/3531130.3533363.

[24] Eric Finster, Alex Rice, and Jamie Vicary. *A Type Theory for Strictly Associative Infinity Categories*. 2021. arXiv: 2109.01513.

[25] Robin Forman. "A User's Guide To Discrete Morse Theory." In: *Séminaire Lotharingien de Combinatoire* 48 (2002), B48c–35.

[26] Robin Forman. "Morse Theory for Cell Complexes". In: *Advances in Mathematics* 134.1 (1998), pp. 90–145. DOI: 10.1006/aima.1997.1650.

[27] Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. “Finite Sets in Homotopy Type Theory”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. Association for Computing Machinery, 2018, pp. 201–214. DOI: 10.1145/3167085.

[28] Robert W Ghrist. *Elementary applied topology*. Vol. 1. CreateSpace, 2014.

[29] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. “Proof certificates for algebra and their application to automatic geometry theorem proving”. In: *7th International Workshop on Automated Deduction in Geometry (ADG)*. Springer. 2011, pp. 42–59.

[30] David Gries. *The Science of Programming*. Monographs in Computer Science. Springer, 1987. DOI: 10.1007/978-1-4612-5983-1.

[31] Allen Hatcher. *Algebraic topology*. Cambridge University Press, 2001.

[32] Michael Hedberg. “A Coherence Theorem for Martin-Löf’s Type Theory”. In: *Journal of Functional Programming* 8.4 (1998), pp. 413–436. DOI: 10.1017/S0956796898003153.

[33] Lukas Heidemanna, Nick Hu, and Jamie Vicary. *homotopy.io*. Available at https://homotopy.io. 2018.

[34] Jónathan Heras, Thierry Coquand, Anders Mörtberg, and Vincent Siles. “Computing Persistent Homology Within Coq/SSReflect”. In: *ACM Transactions on Computational Logic (TOCL)* 14.4 (2013), pp. 1–16. DOI: 10.1145/2528929.

[35] Jónathan Heras, María Poza, Maxime Dénès, and Laurence Rideau. “Incidence Simplicial Matrices Formalized in Coq/SSReflect”. In: *International Conference on Intelligent Computer Mathematics*. Ed. by James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe. Springer, 2011, pp. 30–44. DOI: 10.1007/978-3-642-22673-1_3.

[36] Jónathan Heras, María Poza, and Julio Rubio. “Verifying an algorithm computing Discrete Vector Fields for digital imaging”. In: *International Conference on Intelligent Computer Mathematics*. Springer. 2012, pp. 216–230. DOI: 10.1007/978-3-642-31374-5_15.

[37] Hugo Herbelin. “A Dependently-Typed Construction of Semi-Simplicial Types”. In: *Mathematical Structures in Computer Science* 25.5 (2015), pp. 1116–1131.

[38] Lennart Van Hirtum, Patrick De Causmaecker, Jens Goemaere, Tobias Kenter, Heinrich Riebler, Michael Lass, and Christian Plessl. *A computation of D(9) using FPGA Supercomputing*. 2023. arXiv: 2304.03039.

[39] Martin Hofmann. “Conservativity of equality reflection over intensional type theory”. In: *International Workshop on Types for Proofs and Programs*. Springer. 1995, pp. 153–164.

[40] Martin Hofmann and Thomas Streicher. “The groupoid interpretation of type theory”. In: *Twenty Five Years of Constructive Type Theory*. Vol. 36. Oxford University Press, 1998, pp. 83–112. DOI: 10.1093/oso/9780198501275.003.0008.

[41] Karsten Homann and Jacques Calmet. “Combining theorem proving and symbolic mathematical computing”. In: *Second International Conference on Integrating Symbolic Mathematical Computation and Artificial Intelligence (AISMC-2)*. Springer. 1995, pp. 18–29.

[42] William Alvin Howard. “The formulae-as-types notion of construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan. Academic Press, 1980.

[43] Simon Huber. “Canonicity for Cubical Type Theory”. In: *Journal of Automated Reasoning* 63.2 (2018). DOI: 10.1007/s10817-018-9469-1.

[44] The MathWorks Inc. *MATLAB*. Available at https://www.mathworks.com. 1984.

[45] Christian Jäkel. *A computation of the ninth Dedekind Number*. 2023. arXiv: 2304.00895.

[46] Peter Tennant Johnstone. *Stone Spaces*. Vol. 3. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.

[47] Michael Joswig and Marc E Pfetsch. “Computing Optimal Morse Matchings”. In: *SIAM Journal on Discrete Mathematics* 20.1 (2006), pp. 11–25.

[48] Daniel M. Kan. “Abstract Homotopy. I”. In: *Proceedings of the National Academy of Sciences of the United States of America* 41.12 (1955), pp. 1092–1096.

[49] Nicolai Kraus. “On the Role of Semisimplicial Types”. In: *21st International Conference on Types for Proofs and Programs TYPES* (2018). Available at `https://nicolaikraus.github.io/docs/on_semisimplicial_types.pdf`.

[50] Nicolai Kraus and Jakob von Raumer. “Coherence via Well-Foundedness: Taming Set-Quotients in Homotopy Type Theory”. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Association for Computing Machinery, 2020, pp. 662–675. DOI: `10.1145/3373718.3394800`.

[51] Jean-Louis Laurière. “A Language and a Program for Stating and Solving Combinatorial Problems”. In: *Artificial Intelligence* 10.1 (1978), pp. 29–127. DOI: `10.1016/0004-3702(78)90029-2`.

[52] Fredrik Lindblad and Marcin Benke. “A Tool for Automated Theorem Proving in Agda”. In: *International Workshop on Types for Proofs and Programs (TYPES)*. Dec. 2004, pp. 154–169. DOI: `10.1007/11617990_10`.

[53] Axel Ljungström and Anders Mörtberg. “Formalizing $\pi 4(S3) \cong Z/2Z$ and Computing a Brunerie Number in Cubical Agda”. In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2023, pp. 1–13. DOI: `10.1109/LICS56636.2023.10175833`.

[54] Maplesoft, a division of Waterloo Maple Inc.. *Maple*. Available at `https://www.maplesoft.com`. 1982.

[55] Per Martin-Löf. “An Intuitionistic Theory of Types: Predicative Part”. In: *Logic Colloquium ’73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: `10.1016/S0049-237X(08)71945-1`.

[56] Per Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *Studies in Logic and the Foundations of Mathematics* 104 (1982), pp. 153–175.

[57] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[58] Conor McBride. “A polynomial testing principle”. 2012. URL: `https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf`.

[59] Konstantin Mischaikow and Vidit Nanda. “Morse Theory for Filtrations and Efficient Computation of Persistent Homology”. In: *Discrete & Computational Geometry* 50.2 (2013), pp. 330–353. DOI: 10.1007/s00454-013-9529-6.

[60] Anders Mörtberg and Loïc Pujet. “Cubical Synthetic Homotopy Theory”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 158–171. DOI: 10.1145/3372885.3373825.

[61] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. “The Lean theorem prover (system description)”. In: *International Conference on Automated Deduction (CADE)*. Springer. 2015, pp. 378–388.

[62] Vidit Nanda. “Discrete Morse Theory and Localization”. In: *Journal of Pure and Applied Algebra* 223.2 (2019), pp. 459–488. DOI: 10.1016/j.jpaa.2018.04.001.

[63] Vidit Nanda, Dai Tamaki, and Kohei Tanaka. “Discrete Morse Theory and Classifying Spaces”. In: *Advances in Mathematics* 340 (2018), pp. 723–790. DOI: 10.1016/J.AIM.2018.10.016.

[64] Monica Nicolau, Arnold J Levine, and Gunnar Carlsson. “Topology based data analysis identifies a subgroup of breast cancers with a unique mutational profile and excellent survival”. In: *Proceedings of the National Academy of Sciences* 108.17 (2011), pp. 7265–7270.

[65] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology and Göteborg University, 2007. URL: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[66] Ian Orton and Andrew M. Pitts. “Axioms for Modelling Cubical Type Theory in a Topos”. In: *Logical Methods in Computer Science* Volume 14, Issue 4 (Dec. 2018). DOI: 10.23638/LMCS-14(4:23)2018.

[67] David Overton. *Constraint Programming in Haskell*. Melbourne Haskell Users Group, Available at https://de.slideshare.net/davidoverton/constraint-programming-in-haskell. 2015.

[68] María Poza, César Domínguez, Jónathan Heras, and Julio Rubio. “A Certified Reduction Strategy for Homological Image Processing”. In: *ACM Transactions on Computational Logic (TOCL)* 15.3 (2014), pp. 1–23.

[69] George Raptis. “Homotopy Theory of Posets”. In: *Homology, homotopy and applications* 12.2 (2010), pp. 211–230. DOI: 10.4310/HHA.2010.v12.n2.a7.

[70] Vanessa Robins, Peter John Wood, and Adrian P Sheppard. “Theory and Algorithms for Constructing Discrete Morse Complexes From Grayscale Digital Images”. In: *IEEE Transactions on pattern analysis and machine intelligence* 33.8 (2011), pp. 1646–1658.

[71] Ana Romero and Francis Sergeraert. “Discrete Vector Fields and Fundamental Algebraic Topology”. Available at https://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Vector-Fields.pdf. 2012.

[72] Kristina Sojakova and G. A. Kavvos. “Syllepsis in Homotopy Type Theory”. In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Association for Computing Machinery, 2022. DOI: 10.1145/3531130.3533347.

[73] Jonathan Sterling and Carlo Angiuli. “Normalization for Cubical Type Theory”. In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Association for Computing Machinery, 2021. DOI: 10.1109/LICS52264.2021.9470719.

[74] Andrew W. Swan. “On the Nielsen-Schreier Theorem in Homotopy Type Theory”. In: *Logical Methods in Computer Science* 18.1 (2022). DOI: 10.46298/LMCS-18(1:18)2022.

[75] Cédric Ho Thanh, Pierre-Louis Curien, and Samuel Mimram. “A Sequent Calculus for Opetopes”. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2019, pp. 1–12. DOI: 10.1109/LICS.2019.8785667.

[76] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993. DOI: 10.1016/C2013-0-07627-X.

[77] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: Available at `https://homotopytypetheory.org/book`, 2013.

[78] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A dependently typed programming language with univalence and higher inductive types". In: *Journal of Functional Programming* 31 (2021), e8. DOI: `10.1017/S0956796821000034`.

[79] Vladimir Voevodsky. *A very short note on the homotopy λ-calculus*. Available at `https://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf`. 2006.

[80] Vladimir Voevodsky. "Univalent Foundations Project". Available at `https://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf`. 2010.

[81] David Wärn. *Path spaces of pushouts*. 2024. arXiv: `2402.12339`.

[82] John Henry Constantine Whitehead. "Simplicial Spaces, Nuclei and M-Groups". In: *Proceedings of the London Mathematical Society* 45.1 (1939), pp. 243–327.

[83] Gavin C. Wraith. "Using the generic interval". In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 34.4 (1993), pp. 259–266.