



AUTOMATING BOUNDARY FILLING IN CUBICAL TYPE THEORIES

Downloaded from: <https://research.chalmers.se>, 2026-07-18 00:37 UTC

Citation for the original published paper (version of record):

Doré, M., Cavallo, E., Mörtberg, A. (2026). AUTOMATING BOUNDARY FILLING IN CUBICAL TYPE THEORIES. Logical Methods in Computer Science, 22(2).
[http://dx.doi.org/10.46298/LMCS-22\(2:28\)2026](http://dx.doi.org/10.46298/LMCS-22(2:28)2026)

N.B. When citing this work, cite the original published paper.

AUTOMATING BOUNDARY FILLING IN CUBICAL TYPE THEORIES

MAXIMILIAN DORÉ ^a, EVAN CAVALLO ^b, AND ANDERS MÖRTBERG ^c

^a Department of Computer Science, University of Oxford, United Kingdom
e-mail address: maximilian.dore@cs.ox.ac.uk

^b Department of Computer Science and Engineering, University of Gothenburg and Chalmers
University of Technology, Sweden
e-mail address: evan.cavallo@gu.se

^c Department of Mathematics, Stockholm University, Sweden
e-mail address: anders.mortberg@math.su.se

ABSTRACT. When working in a proof assistant, automation is key to discharging routine proof goals such as equations between algebraic expressions. Homotopy type theory allows the user to reason about higher structures, such as topological spaces, using higher inductive types (HITs) and univalence. Cubical type theory provides computational support for HITs and univalence. A difficulty when working in cubical type theory is dealing with the complex combinatorics of higher structures, an infinite-dimensional generalisation of equational reasoning. To solve these higher-dimensional equations consists in constructing cubes with specified boundaries.

We develop a simplified cubical language in which we isolate and study two automation problems: contortion solving, where we attempt to “contort” a cube to fit a given boundary, and the more general Kan solving, where we search for solutions that involve pasting multiple cubes together. Both problems are difficult in the general case—Kan solving is even undecidable—so we focus on heuristics that perform well on practical examples. Our language encompasses different variations of cubical type theory which differ in their “contortion theory”, i.e., the class of contortions they support. We provide a solver for the contortion problem for the most complex contortion theories currently being researched, namely the Dedekind and De Morgan contortions, by utilising a reformulation of contortions in terms of poset maps. We solve Kan problems using constraint satisfaction programming, which is applicable independently of the underlying contortion theory. We have implemented our algorithms in an experimental Haskell solver that can be used to automatically solve many goals a user of cubical type theory might face. We illustrate this with a case study establishing the Eckmann-Hilton theorem using our solver, as well as various benchmarks—providing the ground for further study of proof automation in cubical type theories.

Key words and phrases: Cubical Type Theory, Automated Reasoning, Constraint Satisfaction Programming.

* This paper is an extended version of *Automating Boundary Filling in Cubical Agda* [DCM24].

Maximilian Doré was supported by COST Action EuroProofNet, supported by COST (European Cooperation in Science and Technology, www.cost.eu). Evan Cavallo was supported by the Knut and Alice Wallenberg Foundation through the Foundation’s program for mathematics. Anders Mörtberg was supported by the Swedish Research Council (Vetenskapsrådet) under Grant No. 2019-04545.

1. INTRODUCTION

Homotopy type theory (HoTT) [Uni13] adds new constructs to intensional dependent type theory [ML75] reflecting an interpretation of types as homotopy types of topological spaces. This allows homotopy theory to be developed *synthetically* inside HoTT; many classical results have been reconstructed this way, such as the Hopf fibration [Uni13], Blakers-Massey theorem [HFLL16], Seifert-van Kampen theorem [HS16], Atiyah-Hirzebruch and Serre spectral sequences [vD18], Hurewicz theorem [CS23], etc. However, as originally formulated, HoTT postulates both the univalence axiom [Voe10] and the existence of HITs [Uni13] without proper computational content—to rectify this, *cubical type theories* [CCHM18, AFH18] replace the identity type with a primitive *path* type, yielding a computationally well-behaved theory which validates the axioms of HoTT.

Inspired by Daniel Kan’s cubical sets [Kan55], cubical type theory represents elements of iterated identity types as higher-dimensional cubes. Synthetic homotopy theory in cubical type theory thereby attains a particular “cubical” flavour [MP20]. A path in a type A connecting elements a and b can be thought of as a function $p: [0, 1] \rightarrow A$ from the unit interval into the “space” A such that $p(0) = a$ and $p(1) = b$. Paths play the role of equalities in the theory, and operations on paths encode familiar laws of equality: reflexivity is a constant path, transitivity is concatenation of paths, and symmetry is following a path in reverse.

Paths can also be studied in their own right. In particular, we can consider equalities *between* paths in A , which as functions $[0, 1] \rightarrow ([0, 1] \rightarrow A)$ can be read as maps from the unit *square* (or *2-cube*) $[0, 1]^2$ to A ; iterating, we find ourselves considering n -cubes in A . Algebraic laws such as the associativity of path concatenation or identity laws are represented as squares with certain boundaries.

For instance, a foundational result in algebraic topology is the *Eckmann-Hilton argument* [EH62], which states that concatenation of 2-spheres, i.e., 2-cubes with constant boundaries, is commutative up to a path. As a path between 2-cubes, the theorem is a 3-cube as shown in Figure 1(a): on the left we have a grey 2-cube concatenated with a hatched 2-cube, on the right they are concatenated in the opposite order, and the interior is the path between them.

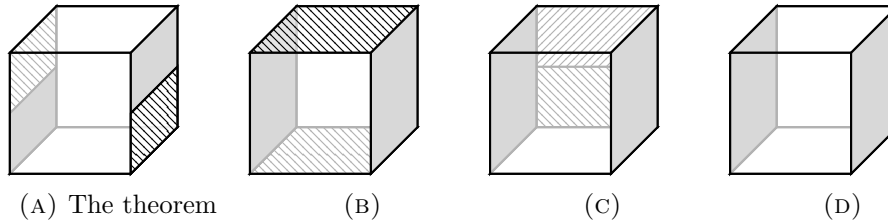


FIGURE 1. A cubical Eckmann-Hilton argument in four steps.

In cubical type theory, we can construct such an interior by starting from some 3-cube we know can be filled, then deforming its boundary via certain basic operations until it has the desired form. It can be more intuitive to work backwards: deform the “goal” boundary until we reach a boundary we can fill. Figure 1 shows one solution: we (b) shift the copies of the hatched 2-cube to the top and bottom faces, (c) further shift them both to the back face, whereupon they face each other in opposite directions, and then (d) cancel the concatenation of the hatched 2-cube with its inverse. The boundary in (d) can be filled immediately by the constant homotopy—i.e., reflexive equality—from the grey 2-cube to itself.

This example illustrates the two main principles we use to build cubes in type theory, which we call contortion and Kan filling.¹ To *contort* a cube is to reparameterise it, stretching it into a higher dimension or projecting a face. For example, we fill the cube (d) by taking the gray 2-cube and stretching it into a degenerate 3-cube, reparameterising by a projection $[0, 1]^3 \rightarrow [0, 1]^2$. Different cubical type theories come equipped with different kinds of reparameterisations, which we call their “contortion theories”. None of the contortion theories we will consider in this paper allow us to derive the cube in Figure 1(a) only by contorting the gray or hatched 2-cube in isolation, however. Thus the role of *Kan filling*, which lets us modify a cube by a continuous deformation of its boundary. Each of the reductions (a) to (b) to (c) to (d) above is an instance of Kan filling.² Kan filling admits a second geometric reading: it states that for every *open box*, i.e., the boundary of a cube with one face unspecified, there is a lid for the box for which an interior (“filler”) exists. The two readings agree because a continuous deformation of the boundary of an n -cube over “time” $t \in [0, 1]$ can also be seen as all but two opposing faces of an $(n + 1)$ -cube; the cube to be deformed fits into one of the missing faces, and the lid produced by box filling is then the deformed cube.

Reasoning with contortions and Kan fillings can pose a challenge when formalising mathematics or computer science in cubical type theories. It is the essence of standalone theorems such as Eckmann-Hilton, but cubical puzzles also often appear as routine lemmas in more complex proofs. One may need to relate one arrangement of concatenations and inverses of paths to another, for example; such coherence conditions often appear in definitions by pattern-matching on HITs. Just as it is difficult to anticipate all types of equations between algebraic expressions that one might need in a large formalisation project, it is infeasible to enumerate every routine cubical lemma in a standard library. The purpose of this paper is to instead devise an algorithm which can automatically prove such lemmas as needed.

Different underlying contortion theories for cubical types theories have been considered, ranging from simple contortion theories which form the basis of the redtt proof assistant [The18] to more expressive contortion theories which provide the basis of the Cubical Agda proof assistant [VMA19]. Our language supports all contortion theories currently under study and is thereby applicable to all variations of cubical type theory in the literature. In the paper we will, in particular, focus on the more complex contortion theories as Cubical Agda is currently the most widely used cubical system. With `agda/cubical` [The23b] and `11ab` [The23a] there are extensive libraries for Cubical Agda which contain ad-hoc collections of cubical reasoning combinators that we aim to automate with our solver.

Contributions: The work presented in this paper constitutes one of the first systematic studies of automated reasoning for cubical type theories. In it we

- formulate a general cubical language which contains a Kan filling operator and is parameterised over the class of “contortions” the language possesses, which allows us to precisely state important classes of automation problems in cubical type theories (§2),
- map out the computational complexity landscape for different cubical type theories and show Kan filling undecidable for all theories under investigation (§3),

¹Because we only reason within individual types in this paper, we encounter only so-called *homogeneous* Kan filling. General Kan filling also incorporates *transport* (or *coercion*) between different indices of a dependent type family, but we leave this aspect to future work.

²The fact that a concatenation of a 2-cube with its inverse can be deformed away, which we use in the step (c) to (d), is a lemma that can itself be proven with contortion and Kan filling.

- formulate an algorithm based on poset maps for solving problems with the “Dedekind” and “De Morgan” contortions, thereby making these computationally hard problems more tractable (§4),
- formulate an algorithm based on constraint satisfaction programming for solving problems using Kan filling (§5), and
- provide a practical Haskell implementation of our algorithms and exhibit its effectiveness on a selection of theorems and lemmas taken from libraries for Cubical Agda (§6).

This paper extends [DCM24], which only considered Dedekind contortions, by generalising our framework to work with any kind contortion theory that is considered in the literature (cartesian [AFH18, ABC⁺21, ACC⁺26], disjunctive [CS25], Dedekind [Awo26] and De Morgan [CCHM18, VMA19]). We give additional complexity results for these newly arisen problems and extend our efficient representation of Dedekind contortions to De Morgan contortions. Furthermore, we spell out the proof that finding Kan filling is undecidable, which was only sketched in [DCM24]. Making precise the reduction from the word problem for groups to Kan filling requires a considerable amount of care. Lastly, we have rectified an error in the definition of our language given in [DCM24]; see Remark 2.14 below.

2. FILLING CUBES IN CUBICAL TYPE THEORIES

Cubical type theories are complex systems. Besides path types, one has the usual type formers of type theory—dependent functions, products, inductive types, etc.—not to mention univalence and HITs. To make automation tractable, we restrict attention to a fragment including only basic operations on cubical cells in a single type.³

Rather than use path *types* to encode cubical cells, as one does in a fully-featured cubical type theory, we take cells as a primitive notion. A cell is a term parameterised by one or more *dimension variables*, which we think of as ranging in the interval $[0, 1]$; intuitively, a cell of type A in n variables is a function $[0, 1]^n \rightarrow A$. *Contexts* are lists of cells each of which can have a specified *boundary*. For example, an entry $p(i) : [i = 0 \mapsto a \mid i = 1 \mapsto b]$ specifies a 1-dimensional cell p varying in $i \in [0, 1]$ such that $p(0) = a$ and $p(1) = b$, i.e., a path from a to b . In general, an entry in a context has the form $q(\Psi) : [\phi]$ where Ψ is a list of variables and ϕ is a list of values at faces ($i = 0$ and $i = 1$ in the example above). A cell hypothesis is thus a judgemental analogue of a hypothesis of *extension type* à la Riehl and Shulman [RS17, §2.2].

The problems we aim to solve are *boundary problems*: given a context of cells Γ , a list of dimension variables Ψ , and a boundary ϕ , can we use the cells in Γ to build a cell varying in Ψ with boundary ϕ ? We write such a problem as “ $\Gamma \mid \Psi \vdash ? : [\phi]$ ”. For example, if we want to prove that paths are invertible, then we could pose the boundary problem

$$a : [], b : [], p(i) : [i = 0 \mapsto a \mid i = 1 \mapsto b] \mid j \vdash ? : [j = 0 \mapsto b \mid j = 1 \mapsto a] \quad (2.1)$$

Here Γ has three cells: points a and b , and a path p . Our goal is a path from b to a , written as a function of $j \in [0, 1]$ with fixed endpoints. We allow leaving the boundary of cells partially or completely unspecified, so that we can formulate the same problem more compactly as

$$p(i) : [] \mid j \vdash ? : [j = 0 \mapsto p(1) \mid j = 1 \mapsto p(0)] \quad (2.2)$$

³This is similar to the fragments of type theory used to axiomatise higher structures such as weak ω -groupoids in e.g. [Bru16, Appendix A] and [FM17].

This variant assumes a path p without naming its endpoints and seeks a path from $p(\mathbf{1})$ to $p(\mathbf{0})$. The format extends gracefully to higher cells; for example, the diagonal of a square can be requested by posing

$$s(i, j) : [] \mid k \vdash ? : [k = \mathbf{0} \mapsto s(\mathbf{0}, \mathbf{0}) \mid k = \mathbf{1} \mapsto s(\mathbf{1}, \mathbf{1})] \quad (2.3)$$

Here we assume a 2-dimensional cell s with unspecified boundary and seek a path from $s(\mathbf{0}, \mathbf{0})$ to $s(\mathbf{1}, \mathbf{1})$. In the remaining section, we introduce two ways to produce *solutions* to boundary problems: contortions and Kan filling.

2.1. Contorting cubes. Intuitively, the problem (2.3) has a simple solution: $? := s(k, k)$. That is, we take the hypothesised 2-cube s and apply a reparameterisation $k \mapsto (k, k)$. We call such reparameterisations *contortions*. Different cubical type theories offer different kinds of contortions. The only contortions of cartesian cubical type theory of Angiuli et al. [AFH18, ABC⁺21] are variables and the constants $\mathbf{0}, \mathbf{1}$, whereas the theory of Cohen et al. (CCHM) [CCHM18] includes binary operators $\vee$ and $\wedge$, conventionally called *connections* [BH81], as well as a unary *involution* operator $\sim$. We think of $\vee$ as taking the *maximum* of two parameters and $\wedge$ as taking the minimum, whereas $\sim$ is thought of as negation sending $i \in [0, 1]$ to $1 - i$. For example, given a cell context containing a path p , the operator $\vee$ can be used to define a square whose value at coordinate (j, k) is the value of p at the maximum of j and k :

$$p(i) : [] \mid j, k \vdash p(j \vee k) : \left[\begin{array}{l|l} j = \mathbf{0} \mapsto p(k) & k = \mathbf{0} \mapsto p(j) \\ j = \mathbf{1} \mapsto p(\mathbf{1}) & k = \mathbf{1} \mapsto p(\mathbf{1}) \end{array} \right] \quad (2.4)$$

We will study both the cartesian and CCHM theories in the following, as well as two theories which lie in between the two in terms of expressiveness. If we remove the involution operation of CCHM, leaving $\vee$ and $\wedge$, we have a distributive lattice which we call the *Dedekind* contortion theory. Removing moreover one of the connections yields the *disjunctive* contortion theory, which is used by Cavallo and Sattler [CS25]. Choosing a more expressive contortion theory naturally means more problems can be solved by contortion. For example, the path inversion problem (2.2) above is immediately solved with an involution:

$$p(i) : [] \mid j \vdash p(\sim i) : [j = \mathbf{0} \mapsto p(\mathbf{1}) \mid j = \mathbf{1} \mapsto p(\mathbf{0})]$$

Without an involution, this problem instead requires Kan filling (which will be introduced in §2.2). On the other hand, adding more contortions makes contortion solving more complex.⁴ There is hence a trade-off for which class of contortions are allowed for proof search.

We now formally introduce the language of boundary problems, starting with the fragment needed to formulate solutions by contortion.

Definition 2.1. A **dimension context** Ψ is either a list of (unique) dimension variables $(i_1, \dots, i_n)$ or the **inconsistent context** $\perp$.

We think of a dimension context with n variables as a topological unit n -cube, each axis being labelled with one variable, while $\perp$ is the empty space; note that the “empty” context $()$ is the unit 0-cube, which does have a unique point. We write Ψ, i for the extension of Ψ by a fresh variable i , which is $(i_1, \dots, i_n, i)$ when $\Psi = (i_1, \dots, i_n)$ and $\perp$ when $\Psi = \perp$.

⁴It is also unclear whether cubical type theories with more complex contortion theories admit semantics in standard homotopy types; see discussion in [CS25, ACC⁺26].

Definition 2.2. The **dimension terms** $\Psi \vdash r \text{ dim}$ over a dimension context Ψ are generated by variables in Ψ , the constants $\mathbf{0}$ and $\mathbf{1}$, the unary operator $\sim$, and binary operators $\vee$ and $\wedge$, subject to the equations

$$\begin{aligned} \sim \mathbf{0} = \mathbf{1} \quad \sim \mathbf{1} = \mathbf{0} \quad \sim \sim x = x \quad r \vee s = s \vee r \quad r \vee \mathbf{0} = r \quad r \vee \mathbf{1} = \mathbf{1} \quad r \vee r = r \\ r \wedge (s \vee t) = (r \wedge s) \vee (r \wedge t) \quad \sim(r \vee s) = \sim r \wedge \sim s \quad \sim(r \wedge s) = \sim r \vee \sim s \end{aligned}$$

When Ψ is the inconsistent context $\perp$, we consider all dimension terms to be equal.

In other words, a dimension term over $\Psi = (i_1, \dots, i_n)$ is an element of the free De Morgan algebra on n variables. We call terms in the full language **De Morgan** dimension terms. We also consider several sublanguages of **De Morgan**: we say that a dimension term is

- **cartesian** if it is an element of Ψ or $\mathbf{0}$ or $\mathbf{1}$.
- **disjunctive** if it mentions only variables, $\mathbf{0}$, $\mathbf{1}$, and $\vee$, i.e., is an element of the free bounded semilattice with join $\vee$ over Ψ .
- **Dedekind** if it mentions only variables, $\mathbf{0}$, $\mathbf{1}$, and $\vee$ and $\wedge$, i.e., is an element of the free bounded distributive lattice over Ψ .

The cartesian dimension terms play a special role in the definition of our theory. In this context we will instead call them **atomic** terms for emphasis; we use the judgement $\Psi \vdash r \text{ atom}$ to denote atomic dimension terms.

Definition 2.3. We write $\bar{e}$ for the opposite of an endpoint e , so $\bar{\mathbf{0}} := \mathbf{1}$ and $\bar{\mathbf{1}} := \mathbf{0}$.

Note that any cartesian term is also a disjunctive term, etc., as the language for dimension terms gets increasingly more expressive from cartesian to De Morgan.

Definition 2.4. A **contortion** $\psi: \Psi' \rightsquigarrow \Psi$ when $\Psi = (i_1, \dots, i_n)$ is a list

$$\psi = (i_1 \mapsto r_1, \dots, i_n \mapsto r_n)$$

consisting of a dimension term $\Psi' \vdash r_k \text{ dim}$ for each $i_k \in \Psi$. When $\Psi = \perp$, there is a contortion $\psi: \Psi' \rightsquigarrow \Psi$ only when $\Psi' = \perp$, in which case there is a unique one. A contortion is called **cartesian/disjunctive/Dedekind/De Morgan** if all of its dimension terms are cartesian/disjunctive/Dedekind/De Morgan.

A **substitution** $\psi: \Psi' \rightarrow \Psi$ is a contortion whose terms are atomic.

Remark 2.5. When Ψ is clear from context, we will write $(r_1, \dots, r_n)$ for a substitution as shorthand for $(i_1 \mapsto r_1, \dots, i_n \mapsto r_n)$. We also write, e.g., $(i \mapsto r): \Psi \rightarrow (\Psi, i)$ as shorthand for $(i_1 \mapsto i_1, \dots, i_n \mapsto i_n, i \mapsto r)$ when $\Psi = (i_1, \dots, i_n)$.

A contortion $\psi: \Psi' \rightsquigarrow \Psi$ can be seen as a continuous mapping from the domain cube to the codomain cube. For example, $(i \mapsto i, j \mapsto \mathbf{0}): (i) \rightsquigarrow (i, j)$ is the inclusion of a 1-dimensional face of the 2-cube, specifically the face where the second coordinate is $\mathbf{0}$. Another example is, $(i \mapsto i): (i, j) \rightsquigarrow (i)$, which maps a 2-cube to a 1-cube by collapsing the second coordinate.

We will need the following operation on dimension contexts to define boundaries.

Definition 2.6. When Ψ is a dimension context, r is an atomic dimension term, and e is an endpoint, we define the **constrained** dimension context $\Psi[r = e]$ by cases:

$$(\Psi, i, \Psi')[i = e] := \Psi, \Psi' \quad \Psi[\bar{e} = e] := \perp \quad \Psi[r = e] := \Psi, \text{ otherwise}$$

We have a **constraining substitution** $(r = e): \Psi[r = e] \rightarrow \Psi$ that sends r to e if r is a variable, is the unique substitution from $\perp$ when r is $\bar{e}$, and the identity substitution otherwise.

For example, $(j = \mathbf{0}): (i, j)[j = \mathbf{0}] \rightarrow (i, j)$ is the inclusion $(i \mapsto i, j \mapsto \mathbf{0}): (i) \rightsquigarrow (i, j)$ of the face where $j = \mathbf{0}$ into the 2-cube (i, j) .

The *cell contexts* $(\Gamma \text{ ctxt})$, *contorted boundaries* $(\Gamma \mid \Psi \parallel \Psi' \vdash_c \phi \text{ bdy})$, and *contorted cells* $(\Gamma \mid \Psi \vdash_c t \text{ cell and } \Gamma \mid \Psi \vdash_c t : [\phi])$ are mutually inductively defined as follows. The subscript c on $\vdash_c$ indicate that these judgements concern contortions and we leave the contortion theory implicit as the judgements are the same for all theories. Substitutions act on each of these judgements in the usual way: given some kind of term t and a substitution $\psi: \Psi' \rightarrow \Psi$ where $\Psi = (i_1, \dots, i_n)$ and $\psi = (r_1, \dots, r_n)$, we write $t[\psi]$ for the result of replacing each i_k by r_k in t . General contortions act only on some of our syntactic sorts, namely dimension terms and contorted cells (Definition 2.10); for those sorts we write $t\langle\psi\rangle$ for application of a contortion. As above, we say a context/boundary/term is **cartesian/disjunctive/Dedekind/De Morgan** when it only mentions contortion operations from that sublanguage.

Definition 2.7. The **cell contexts** $\Gamma \text{ ctxt}$ are inductively defined by the rules

$$\frac{}{() \text{ ctxt}} \quad \frac{\Gamma \text{ ctxt} \quad \Gamma \mid \Psi \parallel () \vdash_c \phi \text{ bdy}}{(\Gamma, a(\Psi) : [\phi]) \text{ ctxt}}$$

where in the second rule, a is a fresh variable name standing for a cell over dimension variables Ψ and with boundary ϕ .

That is, a cell context is a list of variables each paired with a dimension context and boundary over that context; the boundary for one variable may mention preceding variables. The list of inputs to a boundary problem, such as $a : []$, $b : []$, $p(i) : [i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto b]$ from (2.1), is a cell context.

Definition 2.8. The **contorted boundaries** $\Gamma \mid \Psi \parallel \Psi' \vdash_c \phi \text{ bdy}$ are inductively defined by the rules

$$\frac{\Gamma \mid \Psi \parallel \Psi' \vdash_c () \text{ bdy}}{\Gamma \mid \Psi \parallel () \vdash_c \phi \text{ bdy}} \quad \frac{\Psi \vdash r \text{ atom} \quad e \in \{0, 1\} \quad \Gamma \mid \Psi[r = e], \Psi' \vdash_c t : [\phi[r = e]]}{\Gamma \mid \Psi \parallel \Psi' \vdash_c (\phi \mid r = e \mapsto t) \text{ bdy}}$$

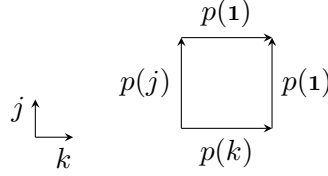
Here $\phi[r = e]$ is the application of the constraining substitution $(r = e)$ to the boundary ϕ .

A contorted boundary is thus a list of entries $r = e \mapsto t$, where each t is a contorted cell over $\Psi[r = e]$, Ψ' , such that each entry agrees with the previous entries when their constraints overlap. The constraints $r = e$ can only refer to variables in Ψ , while the constrained terms t can also refer to variables in Ψ' . We will only use the cases where Ψ' is empty or a singleton, the latter being used in the definition of Kan cells (Definition 2.12). We write $\Gamma \mid \Psi \vdash_c \phi \text{ bdy}$ as shorthand for $\Gamma \mid \Psi \parallel () \vdash_c \phi \text{ bdy}$, and use implicitly that any $\Gamma \mid \Psi \parallel \Psi' \vdash_c \phi \text{ bdy}$ can be viewed as a $\Gamma \mid \Psi, \Psi' \vdash_c \phi \text{ bdy}$ (but not vice versa).

In (2.4), for example, we saw the contorted boundary

$$p(i) : [] \mid j, k \vdash_c (j = \mathbf{0} \mapsto p(k) \mid k = \mathbf{0} \mapsto p(j) \mid j = \mathbf{1} \mapsto p(1) \mid k = \mathbf{1} \mapsto p(1)) \text{ bdy} \quad (2.5)$$

which we picture as the following unfilled square:



The compatibility condition in the formation rule ensures that we can only form this boundary when the edges of the squares actually line up on the intersections; for example, when we add the final face $k = 1 \mapsto p(1)$, we must check that $p(k) = p(1)$ when both $j = 0$ and $k = 1$.

Remark 2.9. The requirement that the term r in a constraint $r = e$ is atomic is absent in Cubical Agda. Imposing it simplifies the constrained context operation (Definition 2.6), while relaxing it is not particularly useful for practical boundary solving. We distinguish between substitutions and contortions to make this requirement sensible.

Finally, we can introduce the main protagonists of our contortion theory: the cells that stem from contorting some generating cell of the cell context.

Definition 2.10. A **contorted cell** $\Gamma \mid \Psi \vdash_c t \text{ cell}$ is a contortion ψ applied to a variable a from the cell context:

$$\frac{(a(\Psi') : [\phi]) \in \Gamma \quad \psi : \Psi \rightsquigarrow \Psi'}{\Gamma \mid \Psi \vdash_c a(\psi) \text{ cell}}$$

We write a instead of $a()$ when $\Psi' = ()$. Equality of contorted cells is generated by the rule

$$\frac{(a(\Psi') : [\phi]) \in \Gamma \quad (r = e \mapsto t) \in \phi \quad \psi : \Psi \rightsquigarrow \Psi' \quad \Psi \vdash r\langle\psi\rangle = e\langle\psi\rangle \text{ dim}}{\Gamma \mid \Psi \vdash_c a(\psi) = t\langle\psi\rangle \text{ cell}}$$

which is to say that $a(\psi)$ has exactly the boundary assigned to it by the context. Additionally, all contorted cells are considered equal in the inconsistent dimension context $\perp$.

We write $\Gamma \mid \Psi \vdash_c t : [\phi]$ when t is a cell agreeing with ϕ , i.e., such that $t[r = e] = t' \text{ cell}$ for each $(r = e \mapsto t') \in \phi$. The two rules above can then be restated as the single rule

$$\frac{(a(\Psi') : [\phi]) \in \Gamma \quad \psi : \Psi \rightsquigarrow \Psi'}{\Gamma \mid \Psi \vdash_c a(\psi) : [\phi\langle\psi\rangle]}$$

For example, (2.4) applies the contortion $(j \vee k) : (j, k) \rightsquigarrow (i)$ to a 1-cell $p(i)$ to define a contorted cell $p(i) : [\] \mid j, k \vdash_c p(j \vee k) \text{ cell}$ matching the boundary (2.5).

Remark 2.11. For ease of reading, we have described the action of substitutions and contortions above as a meta-operation on raw terms, replacing dimension variables by terms in the expected way. Formally, however, we shall consider the theory to come with a calculus of explicit substitutions as described by Abadi et al. [ACCL91]. That is, for example, the action of substitutions on terms is given by a term former

$$\frac{\Gamma \mid \Psi \vdash_c t \text{ cell} \quad \psi : \Psi' \rightarrow \Psi}{\Gamma \mid \Psi' \vdash_c t[\psi] \text{ cell}}$$

whose behaviour is specified by equations such as $a(\psi)[\psi'] = a(\psi\psi')$. We make use of this formal description in §3.2, where we prove some results by induction on syntax.

2.2. Kan filling. Paths in spaces can be *concatenated*: if there is a path from a to b and a path from b to c , then there is a path from a to c . Concatenation generalises to higher cells; for example, we can attach several surfaces at their boundaries to form a new surface. Kan [Kan55] devised a single property that encompasses all of these operations in the context of cubical sets. In cubical type theory, it is embodied by the *Kan filling* (sometimes *Kan composition*) operator.

We write an application of the filling operator as $\text{fill}^{e \rightarrow r} j.[\phi] u$, where u is a cell, $j.[\phi]$ is a boundary varying in a dimension variable j , e is an endpoint, and r is an atomic dimension term. For the operator to be well-formed u must have boundary $\phi[j \mapsto e]$, while the resulting cell has boundary $\phi[j \mapsto r]$; thus we think of fill as deforming the boundary of u from $\phi[j \mapsto e]$ to $\phi[j \mapsto r]$. The fact that we fill to a dimension term r means that the operation unifies $\text{hcomp}/\text{hfill}$ of [CHM18], while being a special case of the more general hcom of [AFH18, ABC⁺21].

As an example, consider the context $p(i) : [\] , q(j) : [j = 0 \mapsto p(1)]$ with p and q such that the 1 -endpoint of p lines up with the 0 -endpoint of q . Suppose we want to concatenate them and produce a cell with boundary $[i = 0 \mapsto p(0) \mid i = 1 \mapsto q(1)]$. Observe that the boundary $[i = 0 \mapsto p(0) \mid i = 1 \mapsto q(j)]$ varying in j is the boundary of p at “time” $j = 0$ and of our desired concatenation at “time” $j = 1$. Thus, deforming p with fill can give us our goal:

$$\begin{array}{c}
 \begin{array}{c} i \uparrow \\ \downarrow j \end{array} \quad \begin{array}{c} q(j) \\ \boxed{} \\ p(0) \end{array} \quad \text{fill}^{0 \rightarrow 1} j.[i = 0 \mapsto p(0) \mid i = 1 \mapsto q(j)] p(i)
 \end{array}$$

We write $(p \bullet q)(i)$ for the filler above. If we replace the target 1 of the fill with a variable, we get a 2-dimensional cell serving as an interior for the depicted square. One says that this cell *fills* the open box formed by the solid paths, hence the name. We define a *Kan cell* to be one formed by iterated applications of the filling operator to contorted cells.

Definition 2.12. The **Kan cells** $\Gamma \mid \Psi \vdash t \text{ cell}$ are inductively generated by the rules

$$\frac{\Gamma \mid \Psi \vdash_c t \text{ cell}}{\Gamma \mid \Psi \vdash t \text{ cell}}$$

$$\frac{e \in \{0, 1\} \quad \Psi \vdash r \text{ atom} \quad \Gamma \mid \Psi \parallel i \vdash \phi \text{ bdy} \quad \Gamma \mid \Psi \vdash u : [\phi[i \mapsto e]]}{\Gamma \mid \Psi \vdash \text{fill}^{e \rightarrow r} i.[\phi] u : [\phi[i \mapsto r], r = e \mapsto u]}$$

where, as in Definition 2.10, we write $\Gamma \mid \Psi \vdash t : [\phi]$ as shorthand to mean that $\Gamma \mid \Psi \vdash t \text{ cell}$ and t agrees with $\Gamma \mid \Psi \vdash \phi \text{ bdy}$. All Kan cells are considered equal in the inconsistent dimension context $\perp$. The **Kan boundaries** $\Gamma \mid \Psi \parallel \Psi' \vdash \phi \text{ bdy}$ are defined analogously to the contorted boundaries $\Gamma \mid \Psi \parallel \Psi' \vdash_c \phi \text{ bdy}$.

Remark 2.13. Note that the $r = e \mapsto u$ constraint makes filling in direction $e \rightarrow e$ the identity function. This ensures that the face opposite the missing side in the filler agrees with the input u to the filling operator.

Remark 2.14. In [DCM24], the rule we gave for the fill constructor had the premise $\Gamma \mid \Psi, i \vdash \phi \text{ bdy}$ in place of $\Gamma \mid \Psi \parallel i \vdash \phi \text{ bdy}$. This is incorrect: it would allow us for every $\Gamma \mid \Psi \vdash t, u$ cell to construct a path

$$\Gamma \mid \Psi, i \vdash \text{fill}^{0 \rightarrow i} j. [j = \mathbf{1} \mapsto u] \ t : [i = \mathbf{0} \mapsto t \mid i = \mathbf{1} \mapsto u]$$

between them. Fortunately, our results and implementation from [DCM24] did not actually make use of this error.

3. COMPLEXITY OF CONTORTION SOLVING AND UNDECIDABILITY OF KAN FILLING

After having specified what solutions to boundary problems look like, we will now classify the different kinds of problems that we study in this paper. We will first look at the complexities of contortion solving for the different contortion theories that we introduced, and then show that the problem of finding Kan fillers is undecidable.

3.1. Complexity of contortion solving. Let us formally introduce the contortion problem for the different contortion theories that we study.

Problem 3.1 (CARTESIAN/DISJUNCTIVE/DEDEKIND/DEMORGAN). Given $\Gamma \mid \Psi \vdash_c \phi \text{ bdy}$,

- the problem CARTESIAN(Γ, Ψ, ϕ) is to determine if there exists a cartesian contortion $\psi: \Psi \rightsquigarrow \Psi'$ such that $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$ for some variable $a(\Psi') : [\phi']$ in Γ .
- the problem DISJUNCTIVE(Γ, Ψ, ϕ) is to determine if there exists a disjunctive contortion $\psi: \Psi \rightsquigarrow \Psi'$ such that $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$ for some variable $a(\Psi') : [\phi']$ in Γ .
- the problem DEDEKIND(Γ, Ψ, ϕ) is to determine if there exists a Dedekind contortion $\psi: \Psi \rightsquigarrow \Psi'$ such that $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$ for some variable $a(\Psi') : [\phi']$ in Γ .
- the problem DEMORGAN(Γ, Ψ, ϕ) is to determine if there exists a De Morgan contortion $\psi: \Psi \rightsquigarrow \Psi'$ such that $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$ for some variable $a(\Psi') : [\phi']$ in Γ .

All four problems are decidable: there are finitely many cell variables in Γ and all contortion theories that we consider are finite, so we can try all possible contortions of each cell variable by brute-force.

Moreover, we can efficiently recognise a solution if we are given one. For this, we need to decide equality between contorted cells, for which we need to normalise a contorted cell by looking at its boundary, if it is specified. For example, in context (2.1) the cell $p(\mathbf{0})$ normalises to a . The number of such normalisation steps is bounded by the length of the context.

Proposition 3.2. CARTESIAN(Γ, Ψ, ϕ), DISJUNCTIVE(Γ, Ψ, ϕ), DEDEKIND(Γ, Ψ, ϕ) and DEMORGAN(Γ, Ψ, ϕ) are in NP as functions of Γ, Ψ , and ϕ .

Proof. For any contortion theory, we can determine in polynomial time if a given variable $a(\Psi') : [\phi']$ and contortion $\psi: \Psi \rightsquigarrow \Psi'$ form a solution to a boundary problem, i.e., that $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$, by normalising $a(\psi)$ and ϕ and comparing. Note that the complexity of our check increases polynomially with the number of cells in Γ ; the dimension variables in Ψ ; and the specified faces of ϕ . $\square$

Boundary problems have multiple parameters; for more detailed analysis of the complexity of the different contortion problems we will in the following fix some of them. In our contortion solver in §4, we will primarily study boundary problems over some small fixed cell context Γ , and try to contort a given cell into ever higher dimensions. It hence makes

sense to study the contortion problems only with respect to the number of variables in Ψ , which in turn determines the size of the goal boundary ϕ .

Proposition 3.3. *For any Γ , $\text{CARTESIAN}(\Gamma, \Psi, \phi)$ is in P as a function of Ψ and ϕ .*

Proof. For any $a(\Psi') : [\phi']$ in Γ , there are $(n+2)^m$ many cartesian contortions for $m := |\Psi'|$ and $n = |\Psi|$. Since we treat the cell context (and therefore also the dimension m of each of its cells) as constant, there are polynomially many contortions that we need to check. $\square$

If we include disjunctions into our dimension terms, we have $(2^n + 1)^m$ contortions of an m -dimensional cell into n dimensions, which means brute-force is not polynomial for DISJUNCTIVE , even for fixed cell contexts. However, enumerating all DISJUNCTIVE contortions can still feasibly be done also in higher dimensions—in contrast to the Dedekind and De Morgan theories, whose sizes explode with growing n . In the case of DEDEKIND , the number of ways to contort an m -cube to fit an n -dimensional goal is $D(n)^m$ where $D(n)$ is the n -th *Dedekind number* [Awo26, App. B]. The Dedekind numbers grow extremely quickly: there are $D(6) = 7\,828\,354$ many ways to contort a 1-cube into a 6-dimensional cube; the 42-digit $D(9)$ was only recently computed using supercomputing [VHDCG⁺24, Jäk23]. The problems DEDEKIND and DEMORGAN thus seem to be computationally very hard, and our focus in §4 will be on heuristics that quickly yield solutions to boundary problems that appear in practice, rather than on worst-case asymptotics. The following results give some indication of this difficulty.

Proposition 3.4. *There exist Γ for which $\text{DEDEKIND}(\Gamma, \Psi, \phi)$ is coNP -hard as a function of Ψ and ϕ .*

Proof. We give a reduction from the entailment problem for monotone Boolean formulas, which is equivalent to the equivalence problem for monotone Boolean formulas, which is known to be coNP -complete [Rei03, Theorem 15].

Given two monotone formulas φ_0 and φ_1 over variables $\vec{x} = x_1, \dots, x_n$, we want to decide whether $\varphi_0 \models \varphi_1$. Note that we can treat each φ_i as a Dedekind dimension term $\vec{x} \vdash \varphi_i$ **dim** by reading $\perp$ and $\top$ as **0** and **1** and disjunction and conjunction as $\vee$ and $\wedge$. We claim that $\varphi_0 \models \varphi_1$ iff the following boundary problem is solvable:

$$s(j) : [] \mid l, \vec{x} \vdash_c ? : [l = \mathbf{0} \mapsto s(\varphi_0) \mid l = \mathbf{1} \mapsto s(\varphi_1)].$$

Suppose $\varphi_0 \models \varphi_1$. Then we can define a Dedekind contortion $\psi : (l, \vec{x}) \rightsquigarrow (j)$ as $\psi(l, \vec{x}) = (\varphi_0(\vec{x}) \vee (l \wedge \varphi_1(\vec{x})))$. The contorted cell $s(\psi)$ then solves the boundary problem.

Conversely, if $\varphi_0 \not\models \varphi_1$, then there is an assignment $\vec{e}$ to the variables $\vec{x}$ such that $\varphi_0(\vec{e}) = \mathbf{1}$ but $\varphi_1(\vec{e}) = \mathbf{0}$. But then any contortion $\psi : (l, \vec{x}) \rightsquigarrow (j)$ which would contort s into the goal would be non-monotone, with $\psi(\mathbf{0}, \vec{e}) = \varphi_0(\vec{e}) > \varphi_1(\vec{e}) = \psi(\mathbf{1}, \vec{e})$, which is impossible with a Dedekind contortion. $\square$

Another perspective on contortion problems is given by considering not the cell context as constant, but instead the dimension of the goal boundary. Even restricting to 1-dimensional goals, contortion problems are NP -hard as soon as the contortion language includes two connections, which underlines that the contortion problems DEDEKIND and DEMORGAN are very complex.

Proposition 3.5. *For Ψ containing at least one variable, $\text{DEDEKIND}(\Gamma, \Psi, \phi)$ is NP -complete as a function of Ψ and ϕ .*

Proof. Membership in NP follows from Proposition 3.2. For completeness, we give a reduction from SAT. Suppose we have a Boolean CNF formula φ over $\vec{x} = x_1, \dots, x_n$. Replace each $\neg x_i$ in φ by a variable y_i to obtain a dimension term r in variables $\vec{x}, \vec{y}$. Then φ is satisfiable if and only if there is $\psi: () \rightsquigarrow (\vec{x}, \vec{y})$ such that $r\langle\psi\rangle = \mathbf{1}$ and $(x_k \wedge y_k)\langle\psi\rangle = \mathbf{0}$ and $(x_k \vee y_k)\langle\psi\rangle = \mathbf{1}$ for each k . Take Γ_φ to be the context

$$a : [], p(z, j_0, j_1) : [], q(\vec{x}, \vec{y}, i) : [i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto p(r, \bigvee_k (x_k \wedge y_k), \bigwedge_k (x_k \vee y_k))]$$

and consider the boundary problem $\Gamma_\varphi \mid i \vdash_c ? : [i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto p(\mathbf{1}, \mathbf{0}, \mathbf{1})]$. Any $\psi: () \rightsquigarrow (\vec{x}, \vec{y})$ such that $r\langle\psi\rangle = \mathbf{1}$ and $(x_k \wedge y_k)\langle\psi\rangle = \mathbf{0}$ and $(x_k \vee y_k)\langle\psi\rangle = \mathbf{1}$ for each k yields a solution $\Gamma_\varphi \mid i \vdash_c q(\psi, i)$ cell. Conversely, any solution to the problem will be of the form $\Gamma_\varphi \mid i \vdash_c q(\psi', r)$ cell for some $\psi': i \rightsquigarrow (\vec{x}, \vec{y})$ and $i \vdash r$ dim, in which case $\psi'(\mathbf{1}): () \rightsquigarrow (\vec{x}, \vec{y})$ induces a satisfying assignment for φ . $\square$

The same reduction also works to establish that $\text{DEMORGAN}(\Gamma, \Psi, \phi)$ is NP-hard. In fact, the proof can be slightly simplified as the negated variables do not have to be replaced first.

Corollary 3.6. *For Ψ with at least one variable, $\text{DEMORGAN}(\Gamma, \Psi, \phi)$ is NP-complete as a function of Ψ and ϕ .*

In summary, contortion problems are, even if decidable, not necessarily tractable for the more complicated contortion theories that we consider.

3.2. Undecidability of Kan solving. The Kan filling problem has—in contrast to the contortion problems—an infinite search space, and we will in the following establish that it is undecidable. This result is independent of which underlying contortion theory one considers. Let us formally introduce the problem of finding Kan cells.

Problem 3.7 (KAN). Given a Kan boundary $\Gamma \mid \Psi \vdash \phi$ bdy, the problem $\text{KAN}(\Gamma, \Psi, \phi)$ is to determine if there exists a Kan cell t such that $\Gamma \mid \Psi \vdash t : [\phi]$.

For example, the problem (2.2) of inverting a path does not have a solution in DEDEKIND but does have solutions in KAN , such as $\text{fill}^{0 \rightarrow 1} i. [j = \mathbf{0} \mapsto p(i) \mid j = \mathbf{1} \mapsto p(\mathbf{0})] p(\mathbf{0})$.

Unlike contortion solving, deciding whether a Kan solution to a problem exists is not only difficult but actually impossible in general. Intuitively, Kan solving is a higher-dimensional generalisation of a more familiar undecidable problem: the word problem for a finitely presented group. This is the problem of deciding, for finite sets X of generators and R of equations, whether two words on X are equal in the free group on X modulo R . In Kan solving, the context Γ can be thought of as a collection of generators in which each $(n + 1)$ -dimensional cell serves as an “equation” between n -dimensional cells, while Kan filling generalises the multiplication and inverse operations available in a group.

We now make this precise by giving a reduction from the word problem for a given finitely presented group to Kan solving over a corresponding context. This argument applies to Kan solving relative to any of the sublanguages of contortions (cartesian, disjunctive, Dedekind, De Morgan) we have introduced. As a side effect, we will get to see some more complex constructions in the cubical type theory we have defined.

A group presentation $\langle X \mid R \rangle$ consists of a finite set X , the *generators*, and a finite set R of equations of the form $w = 1$ where w is a word on X , i.e., a finite list of the form $x_0^{\alpha_0}, \dots, x_k^{\alpha_k}$ where each x_i is in X and each α_i is -1 or 1 . The group G presented by $\langle X \mid R \rangle$ is the free group on X modulo the equations in R ; given words w, v over X , we write $w \equiv_G v$

to mean that they represent equal elements of G . Before beginning the reduction, we first show that we can assume the equations in R are of a more restricted form.

Definition 3.8. Say a finite presentation $\langle X|R \rangle$ of a group G is *convenient* when

- X is closed under inverses;
- every equation in R is of the form $abc^{-1} = 1$ for some $a, b, c \in X$.

Proposition 3.9. *Let G be a finitely presented group. Then G has a convenient presentation.*

Proof. Suppose that G is presented by a finite set of generators X and a finite set of equations $y_{i,0}^{\alpha_{i,0}}, \dots, y_{i,k_i}^{\alpha_{i,k_i}} = 1$ for $0 \leq i < n$, where for each i we have $k_i \in \mathbb{N}$ and then $y_{i,j} \in X$ and $\alpha_{i,j} \in \{-1, 1\}$ for $0 \leq j \leq k_i$.

For each $0 \leq i < n$ and $0 \leq j \leq k_i + 1$, define $z_{i,j} := y_{i,0}^{\alpha_{i,0}}, \dots, y_{i,j-1}^{\alpha_{i,j-1}} \in G$. Then G is presented by the set of generators $X \cup \{z_{i,j} \mid 0 \leq i < n, 0 \leq j \leq k_i\}$ and equations

$$\begin{aligned} z_{i,0}z_{i,0} &= z_{i,0} && \text{for } 0 \leq i < m \\ z_{i,j}y_{i,j} &= z_{i,j+1} && \text{for } 0 \leq i < m \text{ and } 0 \leq j \leq k_i \text{ with } \alpha_{i,j+1} = 1 \\ z_{i,j+1}y_{i,j} &= z_{i,j} && \text{for } 0 \leq i < m \text{ and } 0 \leq j \leq k_i \text{ with } \alpha_{i,j+1} = -1 \\ z_{i,k_i}z_{i,k_i} &= z_{i,k_i} && \text{for } 0 \leq i < m \end{aligned}$$

Note that the first and last equations encode that $z_{i,0} = 1$ and $z_{i,k_i} = 1$ for $0 \leq i < m$ respectively. $\square$

We encode a (conveniently) finitely presented group $\langle X|R \rangle$ as a context with a single point $\star$. Each generator is encoded as a 1-cell, namely a path from $\star$ to itself, and each equation as a 2-cell.

Definition 3.10. In this section, we use $\partial i \mapsto t$ as a shorthand for the pair of boundary entries $i = 0 \mapsto t \mid i = 1 \mapsto t$.

Definition 3.11. Given a convenient presentation $\langle X|R \rangle$, define the context $\ulcorner X|R \urcorner$ ctxt to consist of

- a point $\star : []$,
- a loop $\hat{a}(i) : [\partial i \mapsto \star]$ for each $a \in X$,
- a square

$$s_{a,b,c}(j, k) : [k = 0 \mapsto \hat{a}(j) \mid k = 1 \mapsto \hat{c}(j) \mid j = 0 \mapsto \star \mid j = 1 \mapsto \hat{b}(k)]$$

for each equation $abc^{-1} = 1$ in R :

$$\begin{array}{ccc} & \hat{b}(k) & \\ & \boxed{\phantom{s_{a,b,c}(j,k)}} & \\ \hat{a}(j) \uparrow & & \downarrow \hat{c}(j) \\ j \uparrow & & \downarrow \\ k \rightarrow & \star & \end{array}$$

Any word on X can then be encoded as a path from $\star$ to $\star$ in the context $\ulcorner X|R \urcorner$ ctxt.

Definition 3.12. Let $\langle X|R \rangle$ be a convenient presentation of a group, $a \in X$ be a generator, and $\ulcorner X|R \urcorner \mid i \vdash t : [\partial i \mapsto \star]$ be a cell. For $e \in \{0, 1\}$, define cells

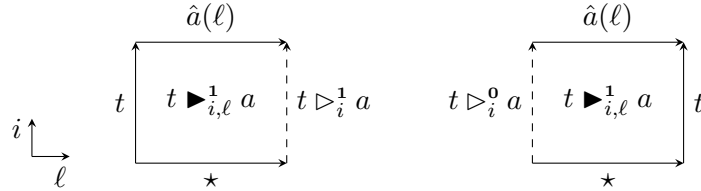
$$\ulcorner X|R \urcorner \mid i \vdash t \triangleright_{i,e}^e a : [\partial i \mapsto \star]$$

$$\ulcorner X|R \urcorner \mid i, \ell \vdash t \blacktriangleright_{i,\ell}^e a : [i = 0 \mapsto \star \mid i = 1 \mapsto \hat{a}(\ell) \mid \ell = 0 \mapsto t \mid \ell = 1 \mapsto t \triangleright_i^e a]$$

by

$$\begin{aligned} t \blacktriangleright_{i,\ell}^e a &:= \text{fill}^{\bar{e} \rightarrow \ell} j. [i = \mathbf{0} \mapsto \star \mid i = \mathbf{1} \mapsto \hat{a}(j)] t \\ t \triangleright_i^e a &:= t \blacktriangleright_{i,e}^e a \end{aligned}$$

as pictured below.



Definition 3.13. Let $\langle X|R \rangle$ be a finite presentation of a group. For each word w on X , define a cell $\ulcorner X|R^\top \mid i \vdash \ulcorner w^\top(i) : [\partial i \mapsto \star]$ by recursion on w as follows.

$$\begin{aligned} \ulcorner \epsilon^\top(i) &= \star \\ \ulcorner wa^\top(i) &= \ulcorner w^\top(i) \triangleright_i^1 a \\ \ulcorner wa^{-1\top}(i) &= \ulcorner w^\top(i) \triangleright_i^0 a \end{aligned}$$

Now we show that when two words represent the same element of the presented group, their encodings as paths are related by a 2-cell. First, we prove a lemma corresponding to cancellation of inverses.

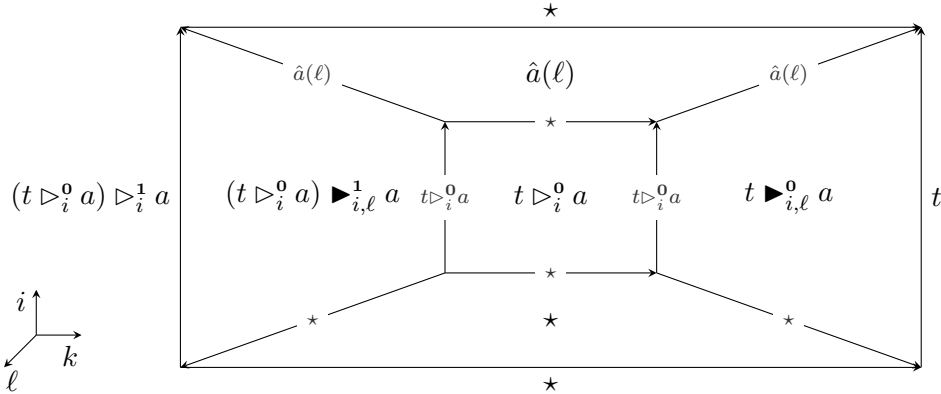
Definition 3.14. Let $\langle X|R \rangle$ be a convenient presentation of a group, $a \in X$ be a generator, and $\ulcorner X|R^\top \mid i \vdash t : [\partial i \mapsto \star]$ be a cell. For $e \in \{\mathbf{0}, \mathbf{1}\}$, define the cell

$$\ulcorner X|R^\top \mid i, k \vdash \text{cancel}_{i,k}^e(t, a) : [\partial i \mapsto \star \mid k = \mathbf{0} \mapsto (t \triangleright_i^e a) \triangleright_i^{\bar{e}} a \mid k = \mathbf{1} \mapsto t]$$

as follows.

$$\text{cancel}_{i,k}^e(t, a) := \text{fill}^{e \rightarrow \bar{e}} \ell. \left[\begin{array}{l} i = \mathbf{0} \mapsto \star \\ i = \mathbf{1} \mapsto \hat{a}(\ell) \\ k = \mathbf{0} \mapsto (t \triangleright_i^e a) \blacktriangleright_{i,\ell}^{\bar{e}} a \\ k = \mathbf{1} \mapsto t \blacktriangleright_{i,\ell}^e a \end{array} \right] (t \triangleright_i^e a)$$

In the case $e = \mathbf{0}$, this is the front face of the filler for the open cube pictured below.



Next we construct cells corresponding to the equations in R . For this we will make use of the following auxiliary construction.

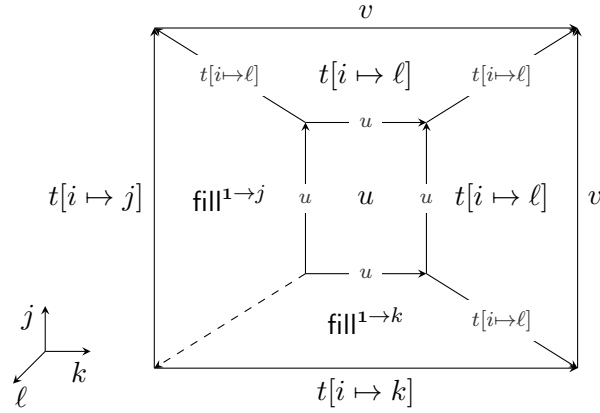
Definition 3.15 (Pseudo- $\vee$). Given a cell $\Gamma \mid i \vdash t : [i = \mathbf{0} \mapsto u \mid i = \mathbf{1} \mapsto v]$, define the cell

$$\Gamma \mid j, k \vdash \text{cnx}_{\vee}(i.t)(j, k) : \left[\begin{array}{c|c} j = \mathbf{0} \mapsto t[i \mapsto k] & j = \mathbf{1} \mapsto v \\ k = \mathbf{0} \mapsto t[i \mapsto j] & k = \mathbf{1} \mapsto v \end{array} \right]$$

to be

$$\text{fill}^{0 \rightarrow 1} \ell. \left[\begin{array}{c} j = \mathbf{0} \mapsto \text{fill}^{1 \rightarrow k} m. [\ell = \mathbf{0} \mapsto u \mid \ell = \mathbf{1} \mapsto t[i \mapsto m]] \quad t[i \mapsto \ell] \\ k = \mathbf{0} \mapsto \text{fill}^{1 \rightarrow j} m. [\ell = \mathbf{0} \mapsto u \mid \ell = \mathbf{1} \mapsto t[i \mapsto m]] \quad t[i \mapsto \ell] \\ j = \mathbf{1} \mapsto t[i \mapsto \ell] \\ k = \mathbf{1} \mapsto t[i \mapsto \ell] \end{array} \right] u$$

which is the front face of the filler for the open cube pictured below.



Note that if we are working relative to the disjunctive, Dedekind, or De Morgan contortion theory, then there is a much simpler construction of a cell with the same boundary when t is a contorted cell, namely $\text{cnx}_{\vee}(i.t)(j, k) := t\langle i \mapsto j \vee k \rangle$ (cf. (2.4)).

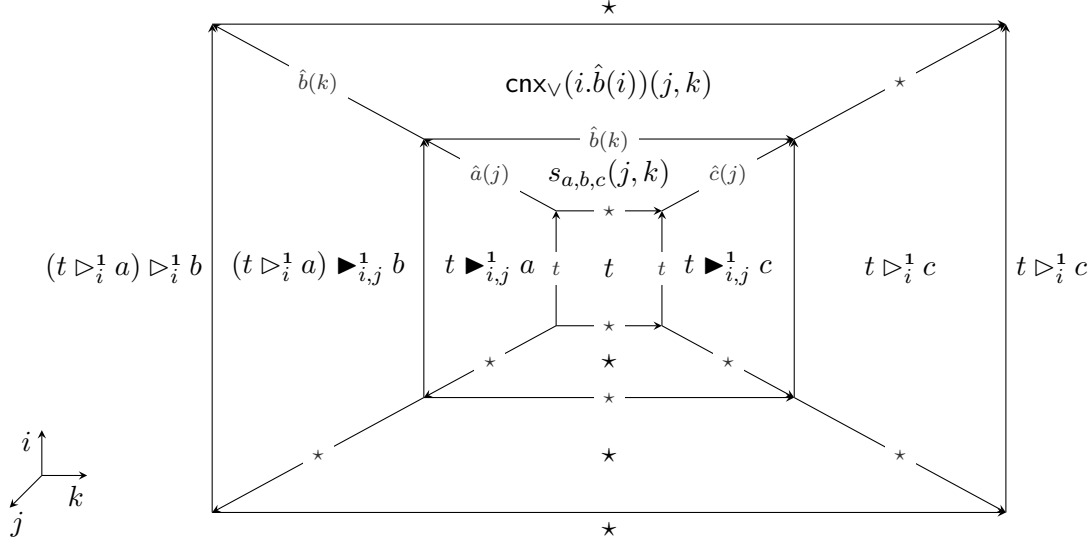
Definition 3.16. Let $\langle X|R \rangle$ be a convenient presentation of a group, $abc^{-1} = 1$ be an equation in R , and $\ulcorner X|R \urcorner \mid i \vdash t : [\partial i \mapsto \star]$ be a cell. Define the cell

$$\ulcorner X|R \urcorner \mid i, k \vdash \text{rew}_{i,k}^{a,b,c}(t) : [\partial i \mapsto \star \mid k = \mathbf{0} \mapsto (t \triangleright_i^1 a) \triangleright_i^1 b \mid k = \mathbf{1} \mapsto t \triangleright_i^1 c]$$

to be

$$\text{fill}^{0 \rightarrow 1} j. \left[\begin{array}{c} i = \mathbf{0} \mapsto \star \\ i = \mathbf{1} \mapsto \text{cnx}_{\vee}(i.\hat{b}(i))(j, k) \\ k = \mathbf{0} \mapsto (t \triangleright_i^1 a) \blacktriangleright_{i,j}^1 b \\ k = \mathbf{1} \mapsto t \triangleright_i^1 c \end{array} \right] \left(\text{fill}^{0 \rightarrow 1} j. \left[\begin{array}{c} i = \mathbf{0} \mapsto \star \\ i = \mathbf{1} \mapsto s_{a,b,c}(j, k) \\ k = \mathbf{0} \mapsto t \blacktriangleright_{i,j}^1 a \\ k = \mathbf{1} \mapsto t \blacktriangleright_{i,j}^1 c \end{array} \right] t \right)$$

which is the iterated composite pictured below.



Combining the previous constructions, we can obtain a cell from any equation in the presented group.

Proposition 3.17. *Let $\langle X|R \rangle$ be a convenient presentation of a group G . For any pair of words v, w on X such that $v \equiv_G w$, there exists a cell*

$$\ulcorner X | R \urcorner \mid i, k \vdash t_{v,w} : [\partial i \mapsto \star \mid k = \mathbf{0} \mapsto \ulcorner v \urcorner(i) \mid k = \mathbf{1} \mapsto \ulcorner w \urcorner(i)] \quad (3.1)$$

Proof. The relation $\equiv_G$ on words on X is generated by the clauses

- (1) $wab \equiv_G wc$ whenever $abc^{-1} = 1$ is an equation in R .
- (2) $waa^{-1} \equiv_G w$ and $wa^{-1}a \equiv_G w$ for a word w and $a \in X$;
- (3) $va \equiv_G wa$ and $va^{-1} \equiv_G wa^{-1}$ for all words v, w with $v \equiv_G w$ and $a \in X$.
- (4) $w \equiv_G w$ for all words w ;
- (5) $w \equiv_G v$ for all words v, w with $v \equiv_G w$;
- (6) $u \equiv_G w$ for all words u, v, w with $u \equiv_G v$ and $v \equiv_G w$.

It thus suffices to show that we have operations on cells of the form (3.1) corresponding to each clause. Clause (1) is covered by Definition 3.16 and (2) is covered by Definition 3.14. Clause (3) corresponds to applying $(-) \triangleright_i^1 a$ or $(-) \triangleright_i^0 a$. Clauses (4), (5), and (6) correspond to reflexivity, symmetry, and transitivity of 2-dimensional paths respectively. For the latter two, given $t_{u,v}$ and $t_{v,w}$ as in (3.1), we have

$$t_{w,v} := \text{fill}^{0 \rightarrow 1} j. \left[\begin{array}{l} \partial i \mapsto \star \\ k = \mathbf{0} \mapsto t_{v,w}[k \mapsto j] \\ k = \mathbf{1} \mapsto \ulcorner v \urcorner(i) \end{array} \right] \ulcorner v \urcorner(i)$$

and

$$t_{u,w} := \text{fill}^{0 \rightarrow 1} j. \left[\begin{array}{l} \partial i \mapsto \star \\ k = \mathbf{0} \mapsto \ulcorner u \urcorner(i) \\ k = \mathbf{1} \mapsto t_{v,w}[k \mapsto j] \end{array} \right] t_{u,v}$$

respectively. □

Now we define an inverse mapping, sending 1-cells over $\ulcorner X|R \urcorner$ to elements of the presented group and 2-cells to equations between them. Again, these definitions make sense for any of the contortion theories we consider; we rely only of the fact that any dimension term in the unit context $()$ is equal to either $\mathbf{0}$ or $\mathbf{1}$. For each definition, we go by structural induction on the Kan cell term formers; as mentioned in Remark 2.11, we treat the syntax here as coming with explicit substitutions, which simplifies the reasoning.

Definition 3.18. Fix a convenient presentation $\langle X|R \rangle$ of a group G . For each Kan cell $\ulcorner X|R \urcorner \mid \Psi \vdash t$ cell, we define a family of elements $\llbracket t \rrbracket_\psi \in G$ for each substitution $\psi: (i) \rightarrow \Psi$. We go by structural induction on t as follows.

- Define $\llbracket t[\psi'] \rrbracket_\psi := \llbracket t \rrbracket_{\psi'\psi}$.
- Define $\llbracket \star \rrbracket_\psi := 1$.
- Define $\llbracket \hat{a}(\psi') \rrbracket_\psi := g_a(\psi'\psi(\mathbf{0}))^{-1}g_a(\psi'\psi(\mathbf{1}))$, where $g_a(e)$ is defined for $e \in \{\mathbf{0}, \mathbf{1}\}$ by $g_a(e) = a^e$, i.e.,

$$\begin{aligned} g_a(\mathbf{0}) &:= 1 \\ g_a(\mathbf{1}) &:= a \end{aligned}$$

Note that this assignment respects cell equality: we have $\llbracket \hat{a}(\mathbf{0}) \rrbracket_\psi = \llbracket \hat{a}(\mathbf{1}) \rrbracket_\psi = \llbracket \star \rrbracket_\psi$.

- Define $\llbracket s_{a,b,c}(\psi') \rrbracket_\psi := g_{a,c}(\psi'\psi(\mathbf{0}))^{-1}g_{a,c}(\psi'\psi(\mathbf{1}))$, where $g_{a,c}(e, e')$ is defined for $e, e' \in \{0, 1\}$ by

$$\begin{aligned} g_{a,c}(\mathbf{00}) &:= 1 & g_{a,c}(\mathbf{10}) &:= a \\ g_{a,c}(\mathbf{01}) &:= 1 & g_{a,c}(\mathbf{11}) &:= c \end{aligned}$$

Again, we can check that $\llbracket s_{a,b,c}(r, \mathbf{0}) \rrbracket_\psi = \llbracket \hat{a}(r) \rrbracket_\psi$, $\llbracket s_{a,b,c}(r, \mathbf{1}) \rrbracket_\psi = \llbracket \hat{c}(r) \rrbracket_\psi$, $\llbracket s_{a,b,c}(\mathbf{0}, r) \rrbracket_\psi = \llbracket \star \rrbracket_\psi$, and $\llbracket s_{a,b,c}(\mathbf{1}, r) \rrbracket_\psi = \llbracket \hat{b}(r) \rrbracket_\psi$ as required by the equational theory.

- We define $\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi$ as follows. First, for $\psi': (i) \rightarrow (\Psi, \ell)$, say that ϕ is satisfied at ψ' if we have some $(s = e' \mapsto t) \in \phi$ with $s[\psi'] = e'$; in this case, we write $\llbracket \phi \rrbracket_{\psi'}$ to mean $\llbracket t \rrbracket_{\psi'[s=e']}$. Note that if there are multiple applicable clauses in ϕ , this value is independent of the choice. In general, define

$$\llbracket \phi \rrbracket_{\psi'}^* := \begin{cases} \llbracket \phi \rrbracket_{\psi'}, & \text{if } \phi \text{ is satisfied at } \psi' \\ 1, & \text{otherwise} \end{cases}$$

We now divide into two cases.

- If ϕ is satisfied at ψ , then set $\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi := \llbracket \phi \rrbracket_{(\psi, r[\psi])}$.
- Otherwise, set

$$\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi := (\llbracket \phi \rrbracket_{(\psi(\mathbf{0}), i)}^*)^{e-r[\psi(\mathbf{0})]} \llbracket u \rrbracket_\psi (\llbracket \phi \rrbracket_{(\psi(\mathbf{1}), i)}^*)^{r[\psi(\mathbf{1})]-e}$$

Here, for $e' \in \{\mathbf{0}, \mathbf{1}\}$, $\psi[i = e']: () \rightarrow \Psi[i = e']$ is the induced substitution between constrained contexts and thus we have $(\psi[i = e'], i): (i) \rightarrow (\Psi[i = e'], \ell)$.

Once more, we check that the assignment respects cell equality: we have $\llbracket \text{fill}^{e \rightarrow e} \ell.[\phi] u \rrbracket_\psi = \llbracket u \rrbracket_\psi$ and $\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi = \llbracket t \rrbracket_{(\psi, r)} = \llbracket t[i \mapsto r] \rrbracket_\psi$ whenever $(e' = e' \mapsto t) \in \phi$.

Lemma 3.19. Fix a convenient presentation $\langle X|R \rangle$ of a group G . For each word w on X , we have $\llbracket \ulcorner w \urcorner(i) \rrbracket_{(i)} = w$ in G .

Proof. By calculation using the definition of $\ulcorner w \urcorner(i)$. □

Lemma 3.20. Fix a convenient presentation $\langle X|R \rangle$ of a group G . For each Kan cell $\ulcorner X|R \urcorner \mid \Psi \vdash t$ cell, if $\psi: (i) \rightarrow \Psi$ is a constant substitution (i.e., either $(\mathbf{0})$ or $(\mathbf{1})$), then $\llbracket t \rrbracket_\psi = 1$.

Proof. The cases where t is $\star$, $\hat{a}(\psi')$, or $s_{a,b,c}(\psi')$ are immediate from the definition, and the case where t is a substitution application is direct by induction hypothesis. For $\text{fill}^{e \rightarrow r} \ell.[\phi] u$, we have two cases.

- If ϕ is satisfied at ψ , then $\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi = \llbracket \phi \rrbracket_{(\psi, r[\psi])}$. Since ψ is constant, so is $(\psi, r[\psi])$, and thus the result follows by induction hypothesis.
- Otherwise, we have

$$\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi = (\llbracket \phi \rrbracket_{(\psi(\mathbf{0}), i)}^*)^{e-r[\psi(\mathbf{0})]} \llbracket u \rrbracket_\psi (\llbracket \phi \rrbracket_{(\psi(\mathbf{1}), i)}^*)^{r[\psi(\mathbf{1})]-e}$$

By induction hypothesis, we have $\llbracket u \rrbracket_\psi = 1$. Moreover, we have $\psi(\mathbf{0}) = \psi(\mathbf{1})$, so

$$\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi = (\llbracket \phi \rrbracket_{(\psi(\mathbf{0}), i)}^*)^{e-r[\psi(\mathbf{0})]} (\llbracket \phi \rrbracket_{(\psi(\mathbf{0}), i)}^*)^{r[\psi(\mathbf{0})]-e} = 1. \quad \square$$

For 2-cells, we intuitively want to show that any cell

$$\lceil X | R \rceil \mid j, k \vdash t : [j = \mathbf{0} \mapsto u_0 \mid j = \mathbf{1} \mapsto u_1 \mid k = \mathbf{0} \mapsto v_0 \mid k = \mathbf{1} \mapsto v_1]$$

induces an equality between the group elements $\llbracket u_0 \rrbracket_{(k)} \llbracket v_1 \rrbracket_{(j)}$ and $\llbracket v_0 \rrbracket_{(j)} \llbracket u_1 \rrbracket_{(k)}$. To make the induction go through, we prove a more general statement where instead of the boundary of t , we consider any “quadrilateral” of 1-cells tracing out a closed loop inside a 2-cell.

Lemma 3.21. *For a convenient presentation $\langle X | R \rangle$ of a group G , a term $\lceil X | R \rceil \mid \Psi \vdash t \text{ cell}$, a substitution $\psi: (j, k) \rightarrow \Psi$, and a quadruple of substitutions $\delta_{\mathbf{0}\bullet}, \delta_{\mathbf{1}\bullet}, \delta_{\bullet\mathbf{0}}, \delta_{\bullet\mathbf{1}}: (i) \rightarrow (j, k)$ such that*

$$\psi\delta_{\mathbf{0}\bullet}(\mathbf{0}) = \psi\delta_{\bullet\mathbf{0}}(\mathbf{0}) \quad \psi\delta_{\mathbf{0}\bullet}(\mathbf{1}) = \psi\delta_{\bullet\mathbf{1}}(\mathbf{0}) \quad \psi\delta_{\mathbf{1}\bullet}(\mathbf{0}) = \psi\delta_{\bullet\mathbf{0}}(\mathbf{1}) \quad \psi\delta_{\mathbf{1}\bullet}(\mathbf{1}) = \psi\delta_{\bullet\mathbf{1}}(\mathbf{1})$$

we have $\llbracket t \rrbracket_{\psi\delta_{\bullet\mathbf{0}}} \llbracket t \rrbracket_{\psi\delta_{\mathbf{1}\bullet}} = \llbracket t \rrbracket_{\psi\delta_{\mathbf{0}\bullet}} \llbracket t \rrbracket_{\psi\delta_{\bullet\mathbf{1}}}$.

Proof. In the situation of the statement, we abbreviate $\delta_{ee'} := \delta_{e\bullet}(e') = \delta_{\bullet e'}(e)$ for $e, e' \in \{\mathbf{0}, \mathbf{1}\}$. We go by structural induction on t as follows.

- For $t[\psi']$, we have

$$\begin{aligned} \llbracket t[\psi'] \rrbracket_{\psi\delta_{\bullet\mathbf{0}}} \llbracket t[\psi'] \rrbracket_{\psi\delta_{\mathbf{1}\bullet}} &= \llbracket t \rrbracket_{\psi'\psi\delta_{\bullet\mathbf{0}}} \llbracket t \rrbracket_{\psi'\psi\delta_{\mathbf{1}\bullet}} \\ &= \llbracket t \rrbracket_{\psi'\psi\delta_{\mathbf{0}\bullet}} \llbracket t \rrbracket_{\psi'\psi\delta_{\bullet\mathbf{1}}} = \llbracket t[\psi'] \rrbracket_{\psi\delta_{\mathbf{0}\bullet}} \llbracket t[\psi'] \rrbracket_{\psi\delta_{\bullet\mathbf{1}}} \end{aligned}$$

where the middle step is by induction hypothesis.

- For $\star$, we have $\llbracket \star \rrbracket_{\psi\delta_{\bullet\mathbf{0}}} \llbracket \star \rrbracket_{\psi\delta_{\mathbf{1}\bullet}} = 1 \cdot 1 = \llbracket \star \rrbracket_{\psi\delta_{\mathbf{0}\bullet}} \llbracket \star \rrbracket_{\psi\delta_{\bullet\mathbf{1}}}$.
- For $\hat{a}(\psi')$, we have

$$\begin{aligned} \llbracket \hat{a}(\psi') \rrbracket_{\psi\delta_{\bullet\mathbf{0}}} \llbracket \hat{a}(\psi') \rrbracket_{\psi\delta_{\mathbf{1}\bullet}} &= g_a(\psi'\psi\delta_{\mathbf{0}\mathbf{0}})^{-1} g_a(\psi'\psi\delta_{\mathbf{1}\mathbf{0}}) g_a(\psi'\psi\delta_{\mathbf{1}\mathbf{0}})^{-1} g_a(\psi'\psi\delta_{\mathbf{1}\mathbf{1}}) \\ &= g_a(\psi'\psi\delta_{\mathbf{0}\mathbf{0}})^{-1} g_a(\psi'\psi\delta_{\mathbf{1}\mathbf{1}}) \\ &= g_a(\psi'\psi\delta_{\mathbf{0}\mathbf{0}})^{-1} g_a(\psi'\psi\delta_{\mathbf{0}\mathbf{1}}) g_a(\psi'\psi\delta_{\mathbf{0}\mathbf{1}})^{-1} g_a(\psi'\psi\delta_{\mathbf{1}\mathbf{1}}) \\ &= \llbracket \hat{a}(\psi') \rrbracket_{\psi\delta_{\mathbf{0}\bullet}} \llbracket \hat{a}(\psi') \rrbracket_{\psi\delta_{\bullet\mathbf{1}}}. \end{aligned}$$

- For $\llbracket s_{a,b,c}(\psi') \rrbracket_\psi$, the same argument applies as for $\hat{a}(\psi')$.
- For $\llbracket \text{fill}^{e \rightarrow r} \ell.[\phi] u \rrbracket_\psi$, we proceed as follows. For simplicity we restrict our attention to the case $e = \mathbf{0}$; a symmetric argument applies for $e = \mathbf{1}$.

First, we use our induction hypothesis to prove the following: for all $\delta: (i) \rightarrow (j, k)$, we have

$$\llbracket \text{fill}^{\mathbf{0} \rightarrow r} \ell.[\phi] u \rrbracket_{\psi\delta} = (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{0}), i)}^*)^{-r[\psi\delta(\mathbf{0})]} \llbracket u \rrbracket_{\psi\delta} (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{1}), i)}^*)^{r[\psi\delta(\mathbf{1})]}.$$

If ϕ is not satisfied at $\psi\delta$ then this is true by definition. Suppose then that ϕ is satisfied at $\psi\delta$, i.e., there is $(s = e' \mapsto t) \in \phi$ with $s[\psi\delta] = e'$. Define substitutions $\delta'_{\bullet\bullet}, \delta'_{1\bullet}, \delta'_{\bullet\bullet}, \delta'_{\bullet 1} : (i) \rightarrow (i, \ell)$ by

$$\begin{aligned} \delta'_{\bullet\bullet} &:= (i, \delta) & \delta'_{1\bullet} &:= (i, r[\psi\delta]) \\ \delta'_{\bullet\bullet} &:= \begin{cases} (\mathbf{0}, \mathbf{0}) & \text{if } r[\psi\delta(\mathbf{0})] = \mathbf{0} \\ (\mathbf{0}, i) & \text{if } r[\psi\delta(\mathbf{0})] = \mathbf{1} \end{cases} & \delta'_{\bullet 1} &:= \begin{cases} (\mathbf{1}, \mathbf{0}) & \text{if } r[\psi\delta(\mathbf{1})] = \mathbf{0} \\ (\mathbf{1}, i) & \text{if } r[\psi\delta(\mathbf{1})] = \mathbf{1} \end{cases} \end{aligned}$$

By induction hypothesis applied to the cell $\ulcorner X|R^\top \mid \Psi[s = e'], \ell \vdash t \text{ cell}$, the substitution $((\psi\delta)[s = e'], \ell) : (i, \ell) \rightarrow (\Psi[s = e'], \ell)$, and the four substitutions $\delta'_{\bullet\bullet}, \delta'_{1\bullet}, \delta'_{\bullet\bullet}, \delta'_{\bullet 1}$ just defined, we have

$$\llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet\bullet}} \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{1\bullet}} = \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet\bullet}} \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet 1}}. \quad (3.2)$$

Calculating, we have

$$\begin{aligned} \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet\bullet}} &= \llbracket t \rrbracket_{((\psi\delta)[s=e'], \mathbf{0})} = \llbracket t[\ell \mapsto \mathbf{0}] \rrbracket_{(\psi\delta)[s=e']} = \llbracket u \rrbracket_{(\psi\delta)[s=e']} = \llbracket u \rrbracket_{\psi\delta} \\ \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{1\bullet}} &= \llbracket t \rrbracket_{((\psi\delta)[s=e'], r[\psi\delta])} = \llbracket \phi \rrbracket_{(\psi\delta, r[\psi\delta])} = \llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta} \\ \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet\bullet}} &= (\llbracket t \rrbracket_{((\psi\delta)[s=e'](\mathbf{0}), i)})^{r[\psi\delta(\mathbf{0})]} = (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{0}), i)})^{r[\psi\delta(\mathbf{0})]} \\ \llbracket t \rrbracket_{((\psi\delta)[s=e'], \ell) \delta'_{\bullet 1}} &= (\llbracket t \rrbracket_{((\psi\delta)[s=e'](\mathbf{1}), i)})^{r[\psi\delta(\mathbf{1})]} = (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{1}), i)})^{r[\psi\delta(\mathbf{1})]} \end{aligned}$$

where in the last two rows we use case analysis on $r[\psi\delta(\mathbf{0})]$ and $r[\psi\delta(\mathbf{1})]$ and Lemma 3.20. Rearranging (3.2), we thus have

$$\llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta} = (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{0}), i)})^{-r[\psi\delta(\mathbf{0})]} \llbracket u \rrbracket_{\psi\delta} (\llbracket \phi \rrbracket_{(\psi\delta(\mathbf{1}), i)})^{r[\psi\delta(\mathbf{1})]}$$

as desired.

Returning to the main claim, we now have

$$\begin{aligned} &\llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta_{\bullet\bullet}} \llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta_{1\bullet}} \\ &= (\llbracket \phi \rrbracket_{(\psi\delta_{\bullet\bullet}, i)})^{-r[\psi\delta_{\bullet\bullet}]} \llbracket u \rrbracket_{\psi\delta_{\bullet\bullet}} \llbracket u \rrbracket_{\psi\delta_{1\bullet}} (\llbracket \phi \rrbracket_{(\psi\delta_{1\bullet}, i)})^{r[\psi\delta_{1\bullet}]} \\ &= (\llbracket \phi \rrbracket_{(\psi\delta_{\bullet\bullet}, i)})^{-r[\psi\delta_{\bullet\bullet}]} \llbracket u \rrbracket_{\psi\delta_{\bullet\bullet}} \llbracket u \rrbracket_{\psi\delta_{\bullet 1}} (\llbracket \phi \rrbracket_{(\psi\delta_{1\bullet}, i)})^{r[\psi\delta_{1\bullet}]} \\ &= \llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta_{\bullet\bullet}} \llbracket \text{fill}^{0 \rightarrow r} \ell. [\phi] u \rrbracket_{\psi\delta_{\bullet 1}} \end{aligned}$$

as required. $\square$

Corollary 3.22. *Let $\langle X|R \rangle$ be a convenient presentation of a group G . For any pair of words v, w on X there exists a cell*

$$\ulcorner X|R^\top \mid i, k \vdash t : [\partial i \mapsto \star \mid k = \mathbf{0} \mapsto \ulcorner v^\top(i) \mid k = \mathbf{1} \mapsto \ulcorner w^\top(i) \rceil]$$

if and only if $v \equiv_G w$.

Proof. One direction was already proven in Proposition 3.17. For the converse, suppose we have such a cell t . By applying Lemma 3.21 with the $\delta_{\bullet\bullet} = (\mathbf{0}, i)$, $\delta_{1\bullet} = (\mathbf{1}, i)$, $\delta_{\bullet\bullet} = (i, \mathbf{0})$, and $\delta_{\bullet 1} = (i, \mathbf{1})$, we get that $\llbracket \ulcorner v^\top(i) \rrbracket_{(i)} \llbracket \star \rrbracket_{(i)} = \llbracket \star \rrbracket_{(i)} \llbracket \ulcorner w^\top(i) \rrbracket_{(i)}$. By Lemma 3.20 and Lemma 3.19, this means that $v \equiv_G w$. $\square$

Theorem 3.23. *KAN is undecidable. More specifically, there are contexts Γ for which there is no algorithmic decision procedure for the problem $\Gamma \mid i, k \vdash ? : [\phi]$ uniformly in Kan boundaries $\Gamma \mid i, k \vdash \phi \text{ bdy}$.*

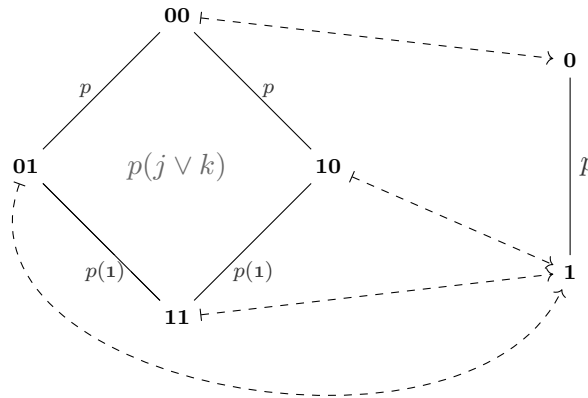
Proof. This follows from Corollary 3.22 and the fact that there are finitely presented groups with undecidable word problem; see, e.g., Collins [Col86] for an example of the latter. $\square$

4. FINDING DEDEKIND AND DE MORGAN CONTORTIONS

We have seen that CARTESIAN and, to a lesser extent, DISJUNCTIVE are problems that can be feasibly solved by enumeration. In contrast, the search space for DEDEKIND and DEMORGAN quickly explodes, which make a brute-force approach infeasible even when solving boundary problems in lower dimensions. We hence need to explore the search space for Dedekind and De Morgan contortions more carefully. In §4.1, we will see how an alternative characterisation based on a Stone-type duality [Joh86], specifically Birkhoff’s duality between finite distributive lattices and finite posets [Bir37], gives rise to a lossy but space-saving representation of collections of Dedekind contortions. We use this representation in §4.2 to develop an algorithm for solving DEDEKIND. We also have a duality between De Morgan algebras and finite poset maps, which allows us to adapt our space-saving representation of Dedekind contortions to also represent collections of De Morgan contortions (§4.3).

4.1. Representing Dedekind contortions with potential poset maps. Recall the example (2.4), where we contorted a path p into a square using a Dedekind contortion $p(i) : [\] \mid j, k \vdash p(j \vee k)$ cell. We can think of $\vee$ as logical disjunction—if either j or k is $\mathbf{1}$, the contortion evaluates to $\mathbf{1}$. Similarly, we can treat the connection $\wedge$ as logical conjunction, which means that we can view any contortion as a tuple of propositional formulas. In fact a Dedekind contortion is uniquely determined by its truth table; for example, the contortion above is determined by the assignment $\llbracket - \rrbracket : \{\mathbf{0}, \mathbf{1}\} \times \{\mathbf{0}, \mathbf{1}\} \rightarrow \{\mathbf{0}, \mathbf{1}\}$ defined by $\llbracket \mathbf{00} \rrbracket = \mathbf{0}$ and $\llbracket \mathbf{01} \rrbracket = \llbracket \mathbf{10} \rrbracket = \llbracket \mathbf{11} \rrbracket = \mathbf{1}$. In general, an n -term Dedekind contortion in m variables gives a truth function $\{\mathbf{0}, \mathbf{1}\}^m \rightarrow \{\mathbf{0}, \mathbf{1}\}^n$.

Since a Dedekind contortion ψ contains no negations, its truth function is *monotone*—we cannot make ψ false by setting more variables to true. Thus the truth function induced by ψ is in fact a map of posets $\mathbf{I}^m \rightarrow \mathbf{I}^n$, where $\mathbf{I}^k$ is the k -fold power of the poset $\mathbf{I} := \{\mathbf{0} < \mathbf{1}\}$ with its product ordering. Conversely, any map of posets $\mathbf{I}^m \rightarrow \mathbf{I}^n$ determines a unique n -term contortion in m variables. For example, we can depict the poset map corresponding to $j \vee k$ as an assignment between the posets $\mathbf{I}^2$ and $\mathbf{I}^1$, which we draw as a Hasse diagram below.



We can read off the boundary of $p(j \vee k)$ by looking at the action of the poset map: the edge from $\mathbf{00}$ to $\mathbf{01}$ is sent to the edge from $\mathbf{0}$ to $\mathbf{1}$ in the target, so the $j = \mathbf{0}$ side of $p(j \vee k)$ is $p(k)$. Between $\mathbf{01}$ and $\mathbf{11}$, we stay at $\mathbf{1}$, so the boundary at $j = \mathbf{1}$ is constantly $p(\mathbf{1})$.

We will in the following freely switch between regarding Dedekind contortions as tuples of propositional formulas and as poset maps. We write $\psi_{\mathbf{I}}$ for the poset map induced by the contortion ψ and $\sigma_{\vee\wedge}$ for the contortion prescribed by the poset map σ . The poset map perspective on contortions does not only give geometric intuition for the boundary of a contorted cell, but also allows for concisely representing a collection of contortions: by assigning a *set* of values $Y \subseteq \mathbf{I}^n$ to each element of $\mathbf{I}^m$, we can at once represent several contortions. The monotonicity constraint on poset maps entails that only some such assignments are meaningful; we call these *potential poset maps*.

Definition 4.1. A **potential poset map (PPM)** is a map $\Sigma: \mathbf{I}^m \rightarrow \mathcal{P}(\mathbf{I}^n)$ s.t. $\forall x \leq y$:

$$\forall u \in \Sigma(y). \exists v \in \Sigma(x). v \leq u \quad \text{and} \quad \forall v \in \Sigma(x). \exists u \in \Sigma(y). v \leq u$$

Readers familiar with bisimulation might intuit PPMs as order-bisimulating, multivalued relations satisfying a certain back-and-forth condition.

With a PPM, we can represent a collection of contortions with very little data: representing all $D(m)^n$ poset maps $\mathbf{I}^m \rightarrow \mathbf{I}^n$ 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 therefore independent of the Dedekind numbers and grow “only” exponentially in m and n . This comes with the trade-off that PPMs are a lossy representation of sets of poset maps. For example, any PPM containing the two poset maps $\sigma, \sigma': \mathbf{I}^1 \rightarrow \mathbf{I}^2$ defined by $\sigma(\mathbf{0}) = (\mathbf{00}), \sigma(\mathbf{1}) = (\mathbf{10})$ and $\sigma'(\mathbf{0}) = (\mathbf{01}), \sigma'(\mathbf{1}) = (\mathbf{11})$ also contains the diagonal map sending $\mathbf{0} \mapsto (\mathbf{00})$ and $\mathbf{1} \mapsto (\mathbf{11})$.

In the following, we unfold a PPM Σ into the set of poset maps it contains with $\text{UNFOLDPPM}(\Sigma)$. We update a PPM Σ to restrict its values at x to some set $vs \subseteq \Sigma(x)$ using $\text{UPDATEPPM}(\Sigma, x, vs)$, thereby obtaining a new PPM Σ' with $\Sigma'(x) = vs$. Due to space constraints we refer to the source code of the solver discussed in §6 for details.

4.2. An algorithm for gradually constructing Dedekind contortions. We now use PPMs in Algorithm 1 to solve DEDEKIND more efficiently than by brute-force. Given a boundary problem $\Gamma \mid \Psi \vdash_c ? : [\phi]$ and a cell $a(\Psi') : [\phi']$ in Γ , we search for a contortion $\psi: \Psi \rightsquigarrow \Psi'$ such that $a(\psi)$ has boundary ϕ by gradually restricting a PPM to be compatible with the faces of ϕ . If a contortion of a appears as one of the faces of ϕ , we can even reduce the search space significantly before performing any expensive operations.

We first initialise Σ to the total PPM on line 2. We then go through the faces of ϕ , each of which normalises to the form $b(\psi)$ for some variable b and contortion ψ , and use them to restrict Σ . Crucially, we order the boundary faces by descending dimensionality of the contorted variable on line 3, as contortions of higher-dimensional variables constrain the search space more. Given a face $i = e \mapsto b(\psi)$ of ϕ , we proceed as follows: if b is in fact a , we can constrain Σ to maps that agree with ψ where $i = e$ on line 5. Otherwise, we iterate through the poset maps σ contained in the restriction of Σ to $\mathbf{I}_{i=e}^m$ on line 8. For each, we mark its values for retention only when $a(\sigma_{\vee\wedge})$ matches the face $b(\psi)$. Finally, we propagate our findings to Σ on line 13. After restricting Σ according to all faces of ϕ , we unfold Σ and brute-force search the results for a valid solution to return. Note that not all poset maps in Σ need be solutions, as a PPM is a lossy representation of a set of maps. The algorithm is complete: if a can be contorted by some ψ to solve the goal boundary, then it keeps $\psi_{\mathbf{I}}(x)$ in $\Sigma(x)$ for all $x \in \mathbf{I}^m$ in each iteration of the main loop, whether in line 5 or line 11.

Algorithm 1 Constructing a Dedekind contortion

Input: $\Gamma \mid \Psi \vdash_c \phi$ bdy and $a(\Psi') : [\phi'] \in \Gamma$. Let $m := |\Psi|$ and $n := |\Psi'|$.
Output: $\psi : \Psi \rightsquigarrow \Psi'$ s.t. $\Gamma \mid \Psi \vdash_c a(\psi) : [\phi]$ if such a ψ exists, **Unsolvable** otherwise

```

1: procedure DEDEKINDCONTORT( $\Gamma, \Psi, \phi, a$ )
2:    $\Sigma := \{x \mapsto \mathbf{I}^n \mid x \in \mathbf{I}^m\}$ 
3:   for  $(i = e \mapsto b(\psi)) \in \phi$  with  $\psi : \Psi[i = e] \rightsquigarrow \Psi''$ , in descending order of  $|\Psi''|$  do
4:     if  $a = b$  then
5:        $\Theta := \{x \mapsto \{\psi_{\mathbf{I}}(x)\} \mid x \in \mathbf{I}_{i=e}^m\}$ 
6:     else
7:        $\Theta := \{x \mapsto \emptyset \mid x \in \mathbf{I}_{i=e}^m\}$ 
8:       for  $\sigma \in \text{UNFOLDPPM}(\Sigma|_{\mathbf{I}_{i=e}^m})$  do
9:         if  $a(\sigma_{\vee \wedge}) = b(\psi)$  then
10:          for  $x \in \mathbf{I}_{i=e}^m$  do
11:             $\Theta(x) := \Theta(x) \cup \{\sigma(x)\}$ 
12:       for  $x \in \mathbf{I}_{i=e}^m$  do
13:          $\text{UPDATEPPM}(\Sigma, x, \Theta(x))$ 
14:   if  $\exists \sigma \in \text{UNFOLDPPM}(\Sigma)$  such that  $\Gamma \mid \Psi \vdash_c a(\sigma_{\vee \wedge}) : [\phi]$  then
15:     return  $\sigma_{\vee \wedge}$ 
16:   else
17:     return Unsolvable

```

The main computational effort in Algorithm 1 consists unfolding all poset maps from a subposet on line 8. For an unconstrained PPM, we have to check $D(m-1)^n$ poset maps, and as we are doing this for up to $2m$ faces of ϕ , we are unfolding $2m \cdot D(m-1)^n$ poset maps in the worst case. In many boundary problems, the cell to be contorted appears in the boundary, which means the search space significantly shrinks before any PPM is unfolded. This allows us to compute many contortions that would have been impossible to find by naïve brute-force.

Example 4.2 (Square to cube contortion). Suppose that we are given the cell context $\Gamma := a : [\]$, $s(i, j) : [i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto a \mid j = \mathbf{0} \mapsto a \mid j = \mathbf{1} \mapsto a]$ and want to contort the square s to match the following 3-cube boundary, which has a contortion of s on one face and squares which are constantly a otherwise:

$$\Gamma \mid i, j, k \vdash_c ? : \left[\begin{array}{c|c|c} i = \mathbf{0} \mapsto s(j \wedge k, j \vee k) & j = \mathbf{0} \mapsto a & k = \mathbf{0} \mapsto a \\ i = \mathbf{1} \mapsto a & j = \mathbf{1} \mapsto a & k = \mathbf{1} \mapsto a \end{array} \right]$$

This is a difficult instance of DEDEKIND because most faces of the goal are contortions of a 0-cell, which can be obtained in many ways. To construct $\psi : (i, j, k) \rightsquigarrow (i, j)$ such that $s(\psi)$ has boundary ϕ , we search for the equivalent poset map $\mathbf{I}^3 \rightarrow \mathbf{I}^2$ using 1.

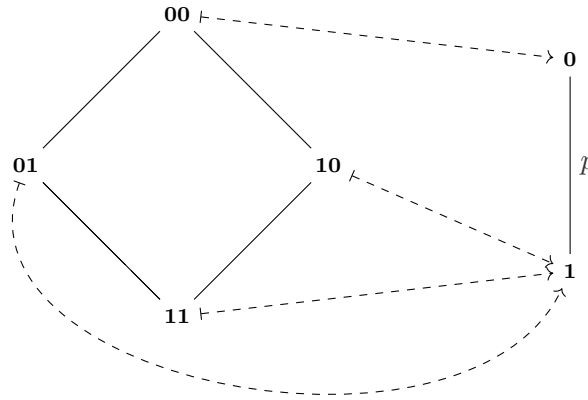
On line 2, the total PPM $\Sigma : \mathbf{I}^3 \rightarrow \mathcal{P}(\mathbf{I}^2)$ is initialised with $x \mapsto \mathbf{I}^2$ for all $x \in \mathbf{I}^3$. We then go through all faces of the goal boundary and use them to restrict Σ , starting with the contortion of s at $i = \mathbf{0}$. Since s is also the cell that we are contorting, the subposet $\mathbf{I}_{i=\mathbf{0}}^3$ 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 , which only contains 10 poset maps after this first restriction. In the next iteration of the outer loop, we only have degenerate a faces left in the goal boundary. Going through each face further restricts Σ , as most induced poset maps give rise to a contortion of s which is not the constant a square. Afterwards, Σ comprises a

single poset map: $\Sigma(000) = \{00\}$, $\Sigma(001) = \Sigma(010) = \Sigma(011) = \Sigma(100) = \Sigma(101) = \{01\}$ and $\Sigma(110) = \Sigma(111) = \{11\}$. Translating this poset map to a contortion gives rise to a solution for our boundary problem: $\Gamma \mid i, j, k \vdash_c s(i \wedge j, i \vee j \vee k) : [\phi]$

Our algorithm finds this solution quickly since the search space is restricted to only 10 possible contortions after looking at the first face of ϕ . This contrasts with brute-force search, where we would have to check $D(3)^2 = 400$ contortions. 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 16000 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.

4.3. De Morgan contortions as poset maps. The most expressive contortion theory that we consider are the De Morgan contortions which can be formed with both $\wedge$, $\vee$ as well as a unary operator $\sim$ which captures reversal of paths. In fact, the number of De Morgan contortions grows with the even Dedekind numbers, i.e., there are $D(2m)$ many ways to contort a 1-cube into an m -dimensional cube using a De Morgan contortion. These combinatorics suggest a connection with Dedekind formulas, and indeed any De Morgan contortion over m variables corresponds to a monotone boolean function in $2m$ variables [MA14, Theorem 3.2] [GWW03]. Intuitively, we can regard a variable j and its inverse $\sim j$ separately since, e.g., $j \vee \sim j$ is in normal form and does not reduce to 1 . We can hence reuse our potential poset maps to also represent De Morgan contortions, and thereby obtain a space-efficient representation for this comprehensive contortion theory. Our construction is reminiscent of the proof of coNP-hardness of equivalence between monotone Boolean formulas using a reduction from the tautology problem [Rei03].

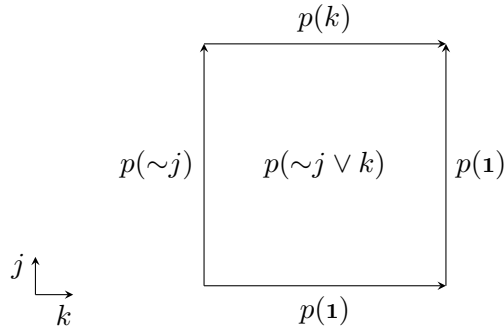
Consider again the cell context from above, but we now contort p into another 1-dimensional path using a De Morgan contortion: $p(i) : [] \mid j \vdash p(j \vee \sim j)$ cell. The poset map corresponding to $j \vee \sim j$ has two variables, one for j and one for $\sim j$, and captures that we take the disjunction of both literals.



Of interest to us is the antichain consisting of 01 and 10 , which are both sent to 1 , which captures that both j and $\sim j$ are present in the contortion. Note that we cannot read off the boundary of the contorted term directly from the poset map anymore ($p(j \vee \sim j)$ is a 1-dimensional path after all), but that we have to focus on the part of the poset map that corresponds to a “consistent” assignment of truth values to the variables. In this case, these are precisely 01 and 10 , inspecting their values under the poset map allows us to compute

the boundary of the contorted term, i.e., $p(j \vee \sim j) : [j = \mathbf{0} \mapsto p(\mathbf{1}) \mid j = \mathbf{1} \mapsto p(\mathbf{1})]$. We hence have another solution to a boundary problem that could have been also solved simply with $p(\mathbf{1})$.

The power of De Morgan contortions comes from the fact that we can directly reverse paths as discussed in §2.1, which gives us a rich language also for higher contortions. For example, the contortion $p(i) : [] \mid j, k \vdash p(\sim j \vee k)$ cell corresponds to a square where between the bottom left and top right corner we once travel along the inverse p and p , and once constantly stay at $p(\mathbf{1})$.



We can intuit the poset map $\sigma : \mathbf{I}^4 \rightarrow \mathbf{I}^1$ corresponding to the contortion $\sim j \vee k$ as follows. An element of the domain $x \in \mathbf{I}^4$ has four indices, where x_1 stands for j , x_2 for k , x_3 for $\sim j$ and x_4 for $\sim k$. The antichain which determines σ is consequently 0010 and 0100 , corresponding to $\sim j$ and k , respectively.

The total potential poset map $\Sigma : \mathbf{I}^4 \rightarrow \mathbf{I}^1$ is again a space-efficient representation of all $D(4) = 168$ De Morgan contortions of p into a square, and we can restrict a potential poset map corresponding to De Morgan contortions similarly to Algorithm 1 to gradually construct a contortion involving reversals.

By constructing De Morgan contortions in this way, we can find compact solutions to lower-dimensional boundary problems for theories which support reversals (such as Cubical Agda). However, this approach is not very practical for higher-dimensional boundary problems as the space of possible contortions grows too quickly.

5. FINDING KAN FILLERS

We now turn to KAN and develop an algorithm for solving general boundary problems. Recall that a Kan cell is of the form $\text{fill}^{e \rightarrow r} i.[\phi] u$, where ϕ and u constitute an “open box” which is filled in direction $e \rightarrow r$. Searching for such fillers requires a different approach depending on whether r is a dimension variable or an endpoint. In the former case, $\text{fill}^{e \rightarrow j} i.[\phi] u$ has the same dimension as ϕ and has $\text{fill}^{e \rightarrow \bar{e}} i.[\phi] u$ as its $j = \bar{e}$ face. This means that it is easy to recognise if a boundary problem can be solved by a filler $e \rightarrow j$: we simply have to check if some face of the goal boundary is an $e \rightarrow \bar{e}$ filler. We hence call the filler in direction $e \rightarrow j$ the “natural filler” for a goal boundary which has $\text{fill}^{e \rightarrow \bar{e}} i.[\phi] u$ at side $j = \bar{e}$.

In contrast, determining when we have to introduce $e \rightarrow \bar{e}$ fillers is difficult. We focus our attention on fillers in direction $\mathbf{0} \rightarrow \mathbf{1}$, since such a filler can be constructed if and only if we can construct a filler in the converse direction. Note that a cell $\text{fill}^{\mathbf{0} \rightarrow \mathbf{1}} i.[\phi] u$ is of one dimension less than the open box spanned by ϕ and u —put differently, to solve a given goal boundary by a $\mathbf{0} \rightarrow \mathbf{1}$ filler, we need to first construct a higher-dimensional cube. We hence

call fillers in direction $\mathbf{0} \rightarrow \mathbf{1}$ “higher-dimensional” fillers. Searching for such cells is difficult because the goal boundary only partially constrains the faces of ϕ , while u could be any cell of the correct dimension. In particular, we could again use higher-dimensional fillers as faces for ϕ or u , leading to infinite search spaces that we have to carefully navigate with heuristics.

Remark 5.1. Some presentations of cubical type theories [CHM18] use separate primitives depending on whether we are filling towards a dimension variable or an endpoint, where usually only the former is called “filling”, while the latter is called “composition” (`hfill` and `hcomp` in Cubical Agda, respectively). To simplify our theory, we have chosen a more general notion of Kan cells which can be used both for filling and composition, but for the purpose of proof search it might be helpful of thinking of these two as different operations.

In our solver we follow the principle that when solving a boundary problem, it’s best to use contorted cells and natural fillers if possible, and only construct higher-dimensional fillers if necessary. The more powerful our contortion theory is, the more boundary problems can be solved without Kan filling, but in general we will often have to construct Kan fillings even when using the most powerful De Morgan contortion theory. For example, our contortions do not allow us to concatenate several paths, such that we always have to resort to Kan filling as soon as a boundary problem cannot be solved using a single cell. Since more powerful contortion theories mean that the search space for contortions is larger, it is sometimes expedient to choose a simpler contortion theory for a given boundary problem. We will hence treat the underlying contortion theory as a parameter to our solver for Kan fillings, which means the results in this section are applicable to all contortion theories.

We formulate the problem of finding a higher-dimensional filler which only uses contorted cells as a constraint satisfaction problem (CSP) [Lau78, Tsa93] in §5.1, which allows us to employ finite domain constraint solvers for this sub-problem of KAN. By carefully calling this solver, we then give a complete search procedure for KAN in §5.2.

5.1. Kan filling as a constraint satisfaction problem. When constructing a higher-dimensional cell for a goal boundary, the sides of the open box that we fill need to match up. This suggests a recipe for constructing fillers: we formulate the search problem as a CSP. In this section, we focus on the problem where all the sides of the filler are contortions. Since there are only finitely many contortions into a given dimension, we can use a finite domain constraint solver to solve this CSP. Still, the number of contortions grows very quickly in the case of the Dedekind and De Morgan contortion theories, making it quickly infeasible to list all contortions.

To rectify this, we may again use PPMs to represent a collection of contortions. We will write $\text{Conts}(\Psi, \Psi')$ for the set of all contortions of an n -dimensional cell into m dimensions, where $n = |\Psi'|$ and $m = |\Psi|$. In the case of Dedekind contortions, this set can be represented by the total PPM $\Sigma : \mathbf{I}^m \rightarrow \mathcal{P}(\mathbf{I}^n)$, for De Morgan contortions we could use $\Sigma : \mathbf{I}^{2m} \rightarrow \mathcal{P}(\mathbf{I}^n)$. By representing a collection of contortions with a PPM, we can quickly construct our CSP with little memory requirements; a solver such as the one discussed in §6 can then gradually narrow down the PPMs until it arrives at a solution. For cartesian and disjunctive contortions it is expedient to simply list all formulas.

Recall that a CSP is given by a set of variables Var ; an assignment of domains to Var , i.e., a set D_X for each $X \in Var$; and a set of constraints $C \subseteq D_X \times D_{X'}$ for $X, X' \in Var$.

A solution is a choice of one element of each domain, i.e., $t_X \in D_X$ for all $X \in Var$, s.t., all constraints are satisfied, i.e., $C(t_X, t_{X'})$ for all C, X, X' .

We now state the CSP for filling boundaries via Kan fillers that have only contortions as sides.

Definition 5.2. Given a boundary $\Gamma \mid \Psi \vdash \phi \text{ bdy}$ and a fresh dimension $k \notin \Psi$, as well as a set of indices $Ope \subseteq \{(k = \mathbf{0})\} \cup \{(i = e) \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}\}$, the CSP $\text{KANCSP}(\phi, Ope)$ is given as follows:

- $Var := \{X_{(i=e)} \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}, (i = e) \notin Ope\} \cup \{X_{(k=\mathbf{0})} \text{ if } (k = \mathbf{0}) \notin Ope\}$
- $D_{(i=e)} := \{(p, \text{Conts}(\Psi, \Psi')) \mid p(\Psi') : [\phi'] \in \Gamma\}$
- and constraints for all $\Psi \vdash i, j \text{ atom}, e, e' \in \{\mathbf{0}, \mathbf{1}\}$:

$$\Gamma \mid \Psi[i = e] \vdash_c X_{(i=e)}[k = \mathbf{1}] = \phi[i = e] \text{ cell if } (i, e) \text{ specified in } \phi$$

$$\Gamma \mid \Psi[i = e][j = e'] \vdash_c X_{(i=e)}[j = e'] = X_{(j=e')}[i = e] \text{ cell}$$

The CSP contains a variable for any side of the boundary that is not left open, the domains contain pairs representing all contortions of a cell p into the needed dimension. The first set of constraints ensures that all sides agree with the goal boundary, while the second set of constraints makes sure that all sides have mutually matching boundaries.

If Ope contains only sides which are unspecified in ϕ , a solution $\text{KANCSP}(\phi, Ope)$ is a solution to the boundary problem ϕ :

$$\Gamma \mid \Psi \vdash \text{fill}^{\mathbf{0} \rightarrow \mathbf{1}} k.[i = e \mapsto t_{(i,e)} \text{ for } i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}, (i = e) \notin Ope] t_{(k,\mathbf{0})} : [\phi]$$

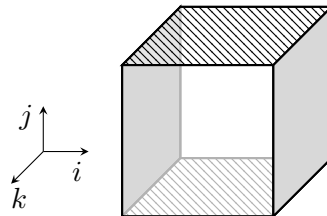
When calling the solver, one has to carefully consider which underlying contortion theory one should choose. For higher-dimensional problems, the De Morgan contortions are often too unwieldy since the domain of the poset maps grows with the even Dedekind numbers.

Take for example the Eckmann-Hilton argument, the cubical version of which we introduced in §1. If we were to solve the cube presented in Figure 1(b) using De Morgan contortions, we would have to consider of the order of $D(6)^2 = 7828354^2$ possible contortions. Since the cube in the example can be constructed without reversals, it is more expedient to instead use the Dedekind contortion theory, leading to a different filler than that depicted in Figure 1(c) and Figure 1(d).

Example 5.3 (The Eckmann-Hilton cube). Using Dedekind contortions, we want to fill the cube from Figure 1(b), where we are given a cell context Γ with a point $x : []$ and two squares $p(i, j)$ and $q(i, j)$ with boundaries $[i = \mathbf{0} \mapsto x \mid i = \mathbf{1} \mapsto x \mid j = \mathbf{0} \mapsto x \mid j = \mathbf{1} \mapsto x]$, and which are assembled into:

$$\Gamma \mid i, j, k \vdash \left[\begin{array}{l|l|l} i = \mathbf{0} \mapsto p(j, k) & j = \mathbf{0} \mapsto q(i, k) & k = \mathbf{0} \mapsto x \\ i = \mathbf{1} \mapsto p(j, k) & j = \mathbf{1} \mapsto q(i, k) & k = \mathbf{1} \mapsto x \end{array} \right] \text{ bdy}$$

Our boundary problem hence corresponds to the cube from Figure 1(b), where the gray 2-cubes of the i sides are filled with p , the j sides with q , and the k sides are constantly x (note also that all corner points of the cube are judgmentally equal to x):



We try to solve KANCSP with no open sides. This CSP has 7 variables corresponding to sides i, j, k and a backside $l = \mathbf{0}$. If we can construct all 7 cubes in a compatible way, we have solved the boundary problem as we can then return a filler in direction $\mathbf{0} \rightarrow \mathbf{1}$. Note in particular that the dimension ℓ is only part of the filler term, while the cube we are filling is three-dimensional.

After imposing the first set of constraints, the domains for the i and j sides are significantly reduced, e.g., $D_{(i=0)} = \{p(\Sigma)\}$ for $\Sigma : \mathbf{I}^3 \rightarrow \mathcal{P}(\mathbf{I}^2)$ is given by:

$$\begin{array}{llll} \mathbf{000} \mapsto \{\mathbf{00}\} & \mathbf{001} \mapsto \{\mathbf{00}\} & \mathbf{010} \mapsto \{\mathbf{00}, \mathbf{01}\} & \mathbf{011} \mapsto \{\mathbf{01}\} \\ \mathbf{100} \mapsto \{\mathbf{00}, \mathbf{10}\} & \mathbf{101} \mapsto \{\mathbf{10}\} & \mathbf{110} \mapsto \{\mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}\} & \mathbf{111} \mapsto \{\mathbf{11}\} \end{array}$$

The PPM Σ gives rise to 9 contortions of p , which contrasts with $D(3)^2 = 400$ total contortions of p . The domains for $D_{(k=0)}$, $D_{(k=1)}$, and the back side $D_{(l=0)}$ still contain all contortions of x, p and q into three dimensions since the k sides of the goal boundary do not give any indication which contortion could be used for this side of the filler.

The second set of constraints ensures that all sides of the Kan filler have matching boundaries, after which we find a solution to KANCSP that gives rise to the following filler:

$$\Gamma \mid i, j, k \vdash \text{fill}^{\mathbf{0} \rightarrow \mathbf{1}} l. \left[\begin{array}{lll} i = \mathbf{0} \mapsto p(j, k \wedge l) & j = \mathbf{0} \mapsto q(i, k) & k = \mathbf{0} \mapsto x \\ i = \mathbf{1} \mapsto p(j, k \wedge l) & j = \mathbf{1} \mapsto q(i, k) & k = \mathbf{1} \mapsto p(j, l) \end{array} \right] q(i, k) \text{ cell}$$

This filler captures the argument sketched in Figure 1, albeit in a single step: the p sides are mapped to the $k = \mathbf{1}$ side such that they cancel out as in Figure 1(c), while the q sides are constantly mapped to the backside of the filler, which is the cube from Figure 1(d).

5.2. A solver for Kan. We now give an algorithm to construct fillers of open cubes which might have fillers on their faces, and not only contorted terms as in KANCSP. We also make use of a procedure $\text{KANFILL}(\Gamma, \Psi, \phi)$ which produces fillers with the same dimension as ϕ : we check for any face of ϕ if it gives rise to a natural filler.

The difficult part of KAN is the construction of higher-dimensional fillers, which might possibly have fillers on their sides. We introduce a variable d to iteratively deepen the level of such nested fillers, which effects a sort-of “breadth-first” search for nested fillers.

Given a goal boundary ϕ , we search for solutions either by natural fillers or by higher-dimensional fillers constructed with KANCUBE on line 4. In KANCUBE , we first select a set of sides that are left open on line 6 and then pick a solution to the corresponding KANCSP on line 7, which will fill all sides not left open with contorted cells. Finally, we call KANsolver recursively on the open sides on line 9, where $[\phi'[i = e]]$ denotes the boundary at $i = e$ induced by the faces already present in ϕ' .

Algorithm 2 Finding Kan cells

Input: $\Gamma \mid \Psi \vdash \phi$ bdy, depth variable d **Output:** $\Gamma \mid \Psi \vdash t : [\phi]$, if $\text{KAN}(\Gamma, \Psi, \phi)$ solvable with $\leq d$ nested Kan fillers

```

1: procedure KANSOLVER( $\Gamma, \Psi, \phi, d$ )
2:   if  $d = 0$  then
3:     return Unsolvable
4:    $t \leftarrow \text{KANFILL}(\Gamma, \Psi, \phi) \cup \text{KANCUBE}(\Gamma, \Psi, \phi, d)$ 
5: procedure KANCUBE( $\Gamma, \Psi, \phi, d$ )
6:    $\text{Ope} \leftarrow \mathcal{P}(\{(i = e) \mid i \in \Psi, e \in \{0, 1\}\} \cup \{(k = 0)\})$ 
7:    $\phi' \leftarrow \text{KANCSP}(\phi, \text{Ope})$ 
8:   for  $(i = e) \in \text{Ope}$  do
9:      $t \leftarrow \text{KANSOLVER}([\phi'[i = e]], d - 1)$ 
10:     $\phi' := [\phi' \mid i = e \mapsto t]$ 
11:  return  $\Gamma \mid \Psi \vdash \text{fill}^{0 \rightarrow 1} k. [\phi' - (k = 0)] (\phi'[k = 0]) : [\phi]$ 

```

The choices of solutions and open sides on lines 4, 6, 7 and 9 are non-deterministic, which is implemented using the list monad in the solver discussed in §6. In practice, the performance of the algorithm depends heavily on the choices we make at this point. In our implementation, we first try to solve KANCSP with $\text{Ope} = \emptyset$. If contortions are not enough to construct all sides, it is useful to first use natural fillers which are induced by the goal boundary. In addition, it is expedient to incrementally increase the number of open sides solutions of KANCSP, e.g., using the depth-parameter d .

We now devise a complete search procedure for KAN with Algorithm 3. The SOLVER starts by trying to contort some cell of the context into the goal boundary. If this fails, we perform iterative deepening on the level of nested Kan cells constructed by Algorithm 2. Again, the contortion theory is a parameter to our solver, which means that CONTORT will call the solver for Dedekind and De Morgan contortions introduced in §4, or simply look for a cartesian or disjunctive contortion by brute-force.

Algorithm 3 A solver for boundary problems

Input: $\Gamma \mid \Psi \vdash \phi$ bdy**Output:** $\Gamma \mid \Psi \vdash t : [\phi]$, if $\text{KAN}(\Gamma, \Psi, \phi)$ is solvable

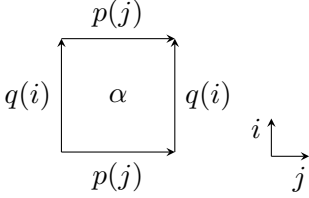
```

1: procedure SOLVER( $\Gamma, \Psi, \phi$ )
2:   for  $p \in \Gamma$  do
3:      $t \leftarrow \text{CONTORT}(\Gamma, \Psi, \phi, p)$ 
4:     if  $t \neq \text{Unsolvable}$  then
5:       return  $t$ 
6:   for  $d \in \{1, \dots\}$  do
7:      $t \leftarrow \text{KANSOLVER}(\Gamma, \Psi, \phi, d)$ 
8:     if  $t \neq \text{Unsolvable}$  then
9:       return  $t$ 

```

Example 5.4 ($\text{Sq} \rightarrow \text{Comp}$). To complete the proof of Eckmann-Hilton, we need to fill the cube from Figure 1(a) using Figure 1(b). This problem can be solved directly using Dedekind contortions, but finding the four-dimensional filler is relatively involved (but can be done using the solver presented in §6). Instead, it is easier to consider a lower-dimensional—and

hence more general—version of the boundary problem. Concretely, the cube from Figure 1(b) is captured with a square

$$\Gamma := \left\{ \begin{array}{l} x : [], \\ p(i) : [i = \mathbf{0} \mapsto x \mid i = \mathbf{1} \mapsto x], \\ q(i) : [i = \mathbf{0} \mapsto x \mid i = \mathbf{1} \mapsto x], \\ \alpha(i, j) : \left[\begin{array}{l|l} i = \mathbf{0} \mapsto p(j) & j = \mathbf{0} \mapsto q(i) \\ i = \mathbf{1} \mapsto p(j) & j = \mathbf{1} \mapsto q(i) \end{array} \right] \end{array} \right.$$


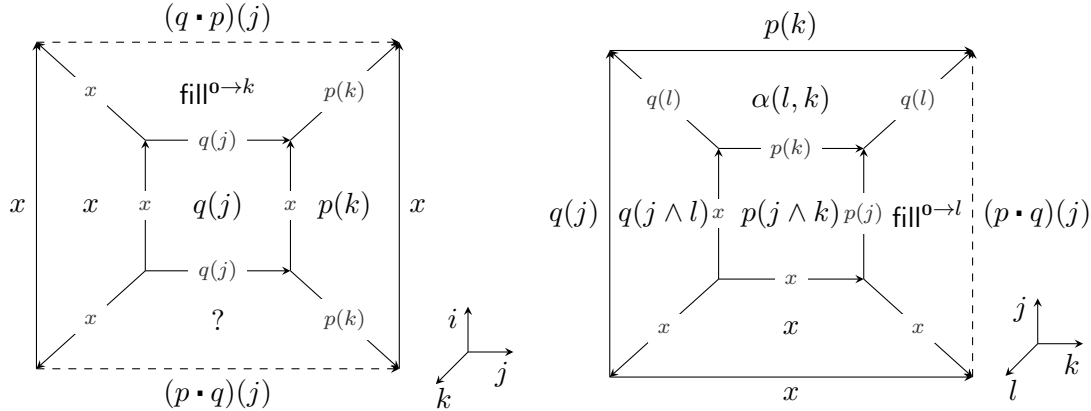
that we want to turn into a square with both path concatenations on opposite sides:

$$\Gamma \mid i, j \vdash ? : [i = \mathbf{0} \mapsto (p \cdot q)(j) \mid i = \mathbf{1} \mapsto (q \cdot p)(j) \mid j = \mathbf{0} \mapsto x \mid j = \mathbf{1} \mapsto x] \quad (5.1)$$

Incidentally, Γ is the list of generators of the HIT capturing the [Torus](#) in `agda/cubical`, while boundary (5.1) captures T^2 , the definition of the torus in the HoTT book [Uni13]. A solution to this problem thus induces a map from the cubical torus to the HoTT book torus.

We solve (5.1) using Algorithm 3. After seeing that we cannot solve this goal with a contortion, the algorithm at some point reaches depth $d = 3$ and solves KANCSPP with open sides $Ope = \{(i = \mathbf{0}), (i = \mathbf{1})\}$. A solution to this CSP has the constant x square for $j = \mathbf{0}$, $p(k)$ for $j = \mathbf{1}$ and $q(j)$ for $k = \mathbf{0}$ as depicted in the left cube below.

When calling KANSOLVER recursively on the two missing sides, we find with KANFILL that the $i = \mathbf{1}$ side can be solved with the natural filler for $q \cdot p$. To fill side $i = \mathbf{0}$, we again have to construct an open cube. One solution of KANCSPP for this open cube is depicted on the right below. The $k = \mathbf{1}$ side is filled by the natural filler for $p \cdot q$. The other sides can be filled with contortions, where side $j = \mathbf{1}$ makes use of α .



While the Dedekind contortions are quite powerful and were an apt contortion theory to prove Eckmann-Hilton, it can often be useful to have reversal $\sim$ available, in particular for lower-dimensional proof goals where the even faster blowup of the number of De Morgan contortions is not so severe.

Example 5.5 (Associativity of path concatenation). Given a context Γ

$$p(i) : [], q(i) : [i = \mathbf{0} \mapsto p(\mathbf{1})], r(i) : [i = \mathbf{0} \mapsto q(\mathbf{1})]$$

For the remaining side $?_1$ we are left to construct a square with the boundary

$$\Gamma \mid j, k \vdash [j = \mathbf{0} \mapsto r(k) \mid j = \mathbf{1} \mapsto (q \cdot r)(k) \mid k = \mathbf{0} \mapsto q(\sim j) \mid k = \mathbf{1} \mapsto r(\mathbf{1})] \text{ bdy}$$

which can be done in much the same spirit as $?_0$ was solved.

In summary, having the powerful De Morgan theory at hand makes the proof of associativity of path concatenation relatively straightforward, while already in the case of Dedekind contortions we would have had to come up with additional fillers wherever $\sim$ was used, let alone the cartesian theory where the solver has to come up with quite involved nested fillers.

6. A PRACTICAL SOLVER FOR CUBICAL AGDA BOUNDARY PROBLEMS

We have implemented the solver in Haskell,⁵ providing the first experimental solver for boundary problems coming from Cubical Agda. The implementation of KANCSP is based on a monadic solver for finite domain constraint satisfaction problems [Ove15]. The user inputs problems in a `.cube` file which contains a cell context and boundary problems over that context. If the solver finds a solution, it is printed in Cubical Agda syntax so that it can be copied and pasted into proof goals. Proper integration into Cubical Agda that allows the solver to be called as a tactic from Agda is left to future work.

We have curated a small benchmarking suite of boundary problems, many of which are from the `agda/cubical` library. The problems are common proof obligations, such as associativity of path concatenation, rearrangements of sides of cubes, etc. On a standard laptop, all problems are quickly solved (often in $< 50\text{ms}$). This means that the solver is fast enough to fit seamlessly into a formalisation workflow and can be used as a tactic for solving routine proof goals. It can also solve some more complex goals such as Example 5.3.

In Cubical Agda, the constant path at x of type $x \equiv x$ is expressed with λ -abstraction as $\lambda i \rightarrow x$. We can use the `PathP` type to describe higher-dimensional boundaries, e.g., `PathP` $(\lambda j \rightarrow x \equiv x) (\lambda i \rightarrow x) (\lambda i \rightarrow x)$ is the boundary of a square with reflexive paths on its sides. Given two such squares p and q , The Eckmann-Hilton cube is derived in $\sim 150\text{ms}$:

```
EckmannHilton-Cube : PathP ( $\lambda i \rightarrow q \ i \equiv q \ i$ ) p p
EckmannHilton-Cube =  $\lambda i \ j \ k \rightarrow \text{hcomp} (\lambda l \rightarrow \lambda \{$ 
  ( $i = \mathbf{i0}$ )  $\rightarrow p \ j \ (k \wedge l)$  ; ( $j = \mathbf{i0}$ )  $\rightarrow q \ i \ k$  ; ( $k = \mathbf{i0}$ )  $\rightarrow x$  ;
  ( $i = \mathbf{i1}$ )  $\rightarrow p \ j \ (k \wedge l)$  ; ( $j = \mathbf{i1}$ )  $\rightarrow q \ i \ k$  ; ( $k = \mathbf{i1}$ )  $\rightarrow p \ j \ l \}$ ) ( $q \ i \ k$ )
```

The Cubical Agda primitive `hcomp` captures Kan fillers in direction $\mathbf{0} \rightarrow \mathbf{1}$. The solution to the boundary problem discussed in the `Sq→Comp` example is found in $\sim 15\text{ms}$, its translation into Cubical Agda looks as follows (manually compressed to not use too much space in the paper; the actual pretty-printed output is more readable):

```
Sq→Comp : PathP ( $\lambda j \rightarrow q \ j \equiv q \ j$ ) p p  $\rightarrow p \cdot q \equiv q \cdot p$ 
Sq→Comp  $\alpha \ i \ j = \text{hcomp} (\lambda k \rightarrow \lambda \{$ 
  ( $i = \mathbf{i0}$ )  $\rightarrow \text{hcomp} (\lambda l \rightarrow \lambda \{$ 
    ( $j = \mathbf{i0}$ )  $\rightarrow x$  ; ( $k = \mathbf{i0}$ )  $\rightarrow q \ (j \wedge l)$  ; ( $j = \mathbf{i1}$ )  $\rightarrow \alpha \ l \ k$  ;
    ( $k = \mathbf{i1}$ )  $\rightarrow \text{hfill} (\lambda m \rightarrow \lambda \{ (j = \mathbf{i0}) \rightarrow x ; (j = \mathbf{i1}) \rightarrow q \ m \}) (\text{inS} (p \ j)) \ l \}$ 
  )
```

⁵We have implemented a solver which is parametric over all contortion theories (<https://github.com/maxdore/cubetac>) as well as a solver specialised to the Dedekind contortions which comes with an interface to Cubical Agda which was used to generate the code in this paper (<https://github.com/maxdore/dedekind>).

$$\begin{aligned}
& (p \ (j \wedge k)) ; \\
& (i = \mathbf{i1}) \rightarrow \mathbf{hfill} \ (\lambda \ l \rightarrow \lambda \ \{ (j = \mathbf{i0}) \rightarrow x ; (j = \mathbf{i1}) \rightarrow p \ l \}) \ (\mathbf{inS} \ (q \ j)) \ k ; \\
& (j = \mathbf{i0}) \rightarrow x ; (j = \mathbf{i1}) \rightarrow p \ k \} \\
& (q \ j)
\end{aligned}$$

The function `hfill` $\phi \ t \ i$ is used in `agda/cubical` to define fillers in direction $\mathbf{o} \rightarrow i$. The term t has to be embedded into the cube structure using `inS`, which is inserted automatically by the Cubical Agda syntax pretty-printer of the solver.

Using these two automatically constructed proofs, we can readily establish by hand the classical formulation of the Eckmann-Hilton argument in terms of path concatenations:

`EckmannHilton` : $p \cdot q \equiv q \cdot p$
`EckmannHilton` = `Sq→Comp` $p \ q$ `EckmannHilton-Cube`

The boundary problem posed by `EckmannHilton` can also be passed directly to our solver as a single problem instance (without requiring a manual decomposition into `Sq→Comp` and `EckmannHilton-Cube`), and our search strategy should in principle yield a solution to this boundary problem. However, our solver is not yet able to find such a solution within 100s. We have also curated some further boundary problems which cannot be solved at the moment, these include a 7-dimensional analogue of the **Square to cube contortion** example and the syllepsis [SK22], which establishes a higher coherence property of the Eckmann-Hilton proof.

In summary, while there is room to make the solver more performant, it can quickly prove technical lemmas for us that would be tedious to prove by hand, taking significant proof burden from a user of Cubical Agda. Furthermore, some deeper results of synthetic homotopy theory, like the Eckmann-Hilton argument, can also be proved if the statement is phrased carefully.

7. FUTURE AND RELATED WORK

There are many ways in which our work can be extended: the performance of the solver can be improved by exploring other heuristics and refinements of the algorithms; the solver should be properly integrated into theorem provers such as Cubical Agda and `redtt`. The solver could be extended to problems involving multiple types and functions and to use cubical type theory’s *transport* primitive.

Early work on proof automation in HoTT is Brunerie’s work on computer-generated proofs for the monoidal structure of smash products [Bru18] which used path-induction and metaprogramming in Agda. [Grz23] generates visual presentations of Cubical Agda proof terms. The problem of deciding equality in the cofibration logic of cubical type theories has been studied by [RL25]. Among other things, they also establish complexity-related results, in particular, that the entailment problems of the cofibration languages of [ABC⁺21] and [CCHM18] are coNP-complete. Another line of related work is on higher-dimensional algebraic rewriting, in particular, on ∞ -categories [FRVR22], operads [TCM19], polygraphs [ABG⁺25] and associative n -categories [Dor18]. For the latter, the tool `homotopy.io` [RV21] gives a graphical user interface for constructing cells based on a higher-dimensional generalisation of string diagrams. Recently, there has been work on automatically constructing coherences for globular theories which are “weak” in the sense of having, e.g., unitality and associativity of path concatenation hold not definitionally, but only up to some computational witness (similar to cubical type theory). [BMO⁺25b] have devised

a “naturality construction” which gives rise to several such coherences, for instance, they derive associativity of path concatenation in their setting, akin to our Example 5.5. Relatedly, [BMO⁺25a] devise a general recipe for proving (higher versions of) the Eckmann-Hilton argument, which we established for 2-loops in Example 5.3. Our approach differs to theirs however in that we start with a (possibly unsolvable) boundary problem and try to construct a coherence for it, while [BMO⁺25b, BMO⁺25a] construct a class of coherences for some given construction.

More work is necessary to understand the precise relationship between geometric accounts of higher-dimensional paths, due to Kan and at the heart of cubical type theories, and theories based on algebraic rewriting, such as associative n -categories. It would be highly beneficial if methods and ideas could more easily be exchanged between both approaches, e.g., to devise a semi-automated proof search which allows the user to interact with higher-dimensional cells as pioneered by `homotopy.io`, but also resolves some problems using an automatic search similar to the present system. Besides, given that the search space for higher-dimensional paths is generally vast, it could be expedient to employ recent advances in machine learning for proof search.

ACKNOWLEDGEMENT

We are grateful to Axel Ljungström for discussions about the solver and to him and Tom Jack for cubical versions of Eckmann-Hilton and syllepsis for us to test it with; and to both the FSCD and LMCS reviewers for their careful reading and constructive feedback.

REFERENCES

- [ABC⁺21] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science*, 31(4):424–468, 2021. doi:10.1017/S0960129521000347.
- [ABG⁺25] Dimitri Ara, Albert Burroni, Yves Guiraud, Philippe Malbos, François Métayer, and Samuel Mimram. *Polygraphs: From Rewriting to Higher Categories*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2025. doi:10.1017/9781009498968.
- [ACC⁺26] Steve Awodey, Evan Cavallo, Thierry Coquand, Emily Riehl, and Christian Sattler. The equivariant model structure on cartesian cubical sets. *Advances in Mathematics*, 495:110965, 2026. doi:10.1016/j.aim.2026.110965.
- [ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacque Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.
- [AFH18] 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 '18)*, volume 119 of *LIPICs*, pages 6:1–6:17, 2018. doi:10.4230/LIPICs.CSL.2018.6.
- [Awo26] Steve Awodey. *Cartesian cubical model categories*. Lecture Notes in Mathematics. Springer, 2026. doi:10.1007/978-3-032-08730-0.
- [BH81] Ronald Brown and Philip J. Higgins. On the algebra of cubes. *Journal of Pure and Applied Algebra*, 21(3):233–260, 1981. doi:10.1016/0022-4049(81)90018-9.
- [Bir37] Garrett Birkhoff. Rings of sets. *Duke Mathematical Journal*, 3(3):443–454, 1937. doi:10.1215/S0012-7094-37-00334-X.
- [BMO⁺25a] Thibaut Benjamin, Ioannis Markakis, Wilfred Offord, Chiara Sarti, and Jamie Vicary. Beyond Eckmann-Hilton: Commutativity in higher categories, 2025. arXiv:2501.16465.
- [BMO⁺25b] Thibaut Benjamin, Ioannis Markakis, Wilfred Offord, Chiara Sarti, and Jamie Vicary. Naturality for higher-dimensional path types. In *40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '25)*. ACM, 2025.

- [Bru16] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory*. PhD thesis, Université de Nice Sophia Antipolis, 2016.
- [Bru18] Guillaume Brunerie. Computer-generated proofs for the monoidal structure of the smash product. *Homotopy Type Theory Electronic Seminar Talks*, 2018. URL: <https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest.html>.
- [CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES '15)*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.
- [CHM18] Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*, pages 255–264. ACM, 2018. doi:10.1145/3209108.3209197.
- [Col86] Donald J. Collins. A simple presentation of a group with unsolvable word problem. *Illinois Journal of Mathematics*, 30(2):230–234, 1986. doi:10.1215/ijm/1256044631.
- [CS23] J. Daniel Christensen and Luis Scoccola. The Hurewicz theorem in Homotopy Type Theory. *Algebraic & Geometric Topology*, 23:2107–2140, 2023.
- [CS25] Evan Cavallo and Christian Sattler. Relative elegance and cartesian cubes with one connection. *Canadian Journal of Mathematics*, pages 1–64, 2025. doi:10.4153/S0008414X25101466.
- [DCM24] Maximilian Doré, Evan Cavallo, and Anders Mörtberg. Automating Boundary Filling in Cubical Agda. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction (FSCD '24)*, volume 299 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl, 2024. doi:10.4230/LIPIcs.FSCD.2024.22.
- [Dor18] Christoph Dorn. *Associative n -categories*. PhD thesis, University of Oxford, 2018.
- [EH62] Beno Eckmann and Peter J. Hilton. Group-like structures in general categories I: multiplications and comultiplications. *Mathematische Annalen*, 145:227–255, 1962.
- [FM17] Eric Finster and Samuel Mimram. A type-theoretical definition of weak ω -categories. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS' 17)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005124.
- [FRVR22] Eric Finster, David Reutter, Jamie Vicary, and Alex Rice. A type theory for strictly unital ∞ -categories. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22)*, pages 1–12, 2022. doi:10.1145/3531130.3533363.
- [Grz23] Marcin Grzybowski. cubeviz2, 2023. URL: <https://github.com/marcinjangrzybowski/cubeViz2>.
- [GWW03] Mai Gehrke, Carol L. Walker, and Elbert A. Walker. Normal forms and truth tables for fuzzy logics. *Fuzzy Sets and Systems*, 138(1):25–51, 2003. doi:10.1016/S0165-0114(02)00566-3.
- [HFLL16] Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. A Mechanization of the Blakers-Massey Connectivity Theorem in Homotopy Type Theory. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS' 16)*, pages 565–574. ACM, 2016. doi:10.1145/2933575.2934545.
- [HS16] Kuen-Bang Hou (Favonia) and Michael Shulman. The Seifert-van Kampen Theorem in Homotopy Type Theory. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL '16)*, volume 62 of *LIPIcs*, pages 22:1–22:16. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.CSL.2016.22.
- [Jäk23] Christian Jäkel. A computation of the ninth dedekind number. *Journal of Computational Algebra*, 6-7:100006, 2023. doi:10.1016/j.jaca.2023.100006.
- [Joh86] Peter Tennant Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.
- [Kan55] Daniel M. Kan. Abstract homotopy. I. *Proceedings of the National Academy of Sciences of the United States of America*, 41(12):1092–1096, 1955.
- [Lau78] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978. doi:10.1016/0004-3702(78)90029-2.
- [MA14] Yu.M. Movsisyan and V.A. Aslanyan. A functional completeness theorem for de morgan functions. *Discrete Applied Mathematics*, 162:1–16, 2014. doi:10.1016/j.dam.2013.08.006.

- [ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975. doi:10.1016/S0049-237X(08)71945-1.
- [MP20] Anders Mörtberg and Loïc Pujet. Cubical Synthetic Homotopy Theory. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*, pages 158–171. ACM, 2020. doi:10.1145/3372885.3373825.
- [Ove15] David Overton. Constraint Programming in Haskell. Melbourne Haskell Users Group, 2015. URL: <https://de.slideshare.net/davidoverton/constraint-programming-in-haskell>.
- [Rei03] Steffen Reith. On the complexity of some equivalence problems for propositional calculi. In Branislav Rován and Peter Vojtáš, editors, *Mathematical Foundations of Computer Science 2003*, volume 2747 of *Lecture Notes in Computer Science*, pages 632–641. Springer, 2003. doi:10.1007/978-3-540-45138-9_57.
- [RL25] Robert Rose and Daniel R. Licata. Complexity of Cubical Cofibration Logics I: coNP-Complete Examples. In Rasmus Ejlers Møgelberg and Benno van den Berg, editors, *30th International Conference on Types for Proofs and Programs (TYPES '24)*, volume 336 of *LIPICs*, pages 9:1–9:21. Schloss Dagstuhl, 2025. doi:10.4230/LIPICs.TYPES.2024.9.
- [RS17] Emily Riehl and Michael Shulman. A type theory for synthetic ∞ -categories. *Higher Structures*, 1(1):116–193, 2017.
- [RV21] David Reutter and Jamie Vicary. High-level methods for homotopy construction in associative n-categories. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '19)*. IEEE Press, 2021. doi:10.5555/3470152.3470214.
- [SK22] Kristina Sojakova and G. A. Kavvos. Syllepsis in Homotopy Type Theory. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS' 22)*. ACM, 2022. doi:10.1145/3531130.3533347.
- [TCM19] Cédric Ho Thanh, Pierre-Louis Curien, and Samuel Mimram. A sequent calculus for opetopes. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '19)*, pages 1–12, 2019. doi:10.1109/LICS.2019.8785667.
- [The18] The RedPRL Development Team. The redtt proof assistant, 2018. URL: <https://github.com/RedPRL/redtt/>.
- [The23a] The 1Lab Development Team. The 1Lab, 2023. URL: <https://1lab.dev>.
- [The23b] The Agda Community. Cubical Agda Library, 2023. URL: <https://github.com/agda/cubical>.
- [Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993. doi:10.1016/C2013-0-07627-X.
- [Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.
- [vD18] Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory*. PhD thesis, Carnegie Mellon University, 2018.
- [VHDCG⁺24] Lennart Van Hirtum, Patrick De Causmaecker, Jens Goemaere, Tobias Kenter, Heinrich Riebler, Michael Lass, and Christian Plessl. A computation of the ninth dedekind number using FPGA supercomputing. *ACM Trans. Reconfigurable Technol. Syst.*, 17(3), 2024. doi:10.1145/3674147.
- [VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a Dependently Typed Programming Language With Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341691.
- [Voe10] Vladimir Voevodsky. The equivalence axiom and univalent models of type theory, February 2010. Notes from a talk at Carnegie Mellon University, available at http://www.math.ias.edu/vladimir/files/CMU_talk.pdf.