Automating Boundary Filling in Cubical Agda

Maximilian Doré
Department of Computer Science, University of Oxford, United Kingdom

Evan Cavallo
Department of Computer Science and Engineering, University of Gothenburg, Sweden

Anders Mörtberg
Department of Mathematics, Stockholm University, Sweden

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 Agda` is an extension of `Agda` with computational support for HITs and univalence. A difficulty when working in `Cubical Agda` 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. We provide a solver for the contortion problem using a reformulation of contortions in terms of poset maps, while we solve Kan problems using constraint satisfaction programming. We have implemented our algorithms in an experimental Haskell solver that can be used to automatically solve goals presented by `Cubical Agda`. 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.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Computing methodologies → Theorem proving algorithms; Theory of computation → Constraint and logic programming; Theory of computation → Automated reasoning

Keywords and phrases Cubical Agda, Automated Reasoning, Constraint Satisfaction Programming

Digital Object Identifier 10.4230/LIPIcs.FSCD.2024.22

Supplementary Material *Software (Source Code)*: `https://github.com/maxdore/dedekind` archived at `swh:1:dir:eb5150105fe3dcbc370e5207f3dcb9de3c26619c`

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

Acknowledgements 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.

1 Introduction

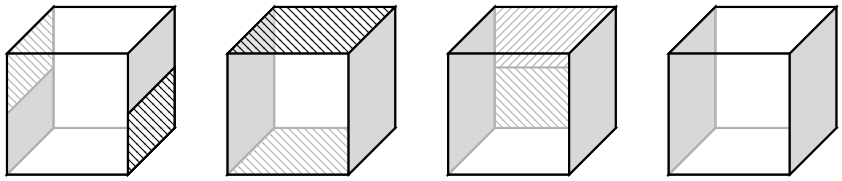
Homotopy Type Theory (HoTT) [35] adds new constructs to intensional dependent type theory [22] reflecting an interpretation of types as homotopy types of topological spaces. This allows homotopy theory to be developed *synthetically* inside HoTT; many classical results have

been reconstructed this way, such as the Hopf fibration [35], Blakers-Massey theorem [16], Seifert-van Kampen theorem [17], Atiyah-Hirzebruch and Serre spectral sequences [36], Hurewicz theorem [8], etc. However, as originally formulated, HoTT postulates both the univalence axiom [38] and the existence of HITs [35] without proper computational content – to rectify this, *cubical type theory* [9] replaces 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 [20], 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 [23]. A path in a type A connecting elements a and b can be thought of as a function $p\colon [\mathbf{0}, \mathbf{1}] \to A$ from the unit interval into the "space" A such that $p(\mathbf{0}) = a$ and $p(\mathbf{1}) = b$. 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 $[\mathbf{0}, \mathbf{1}] \to ([\mathbf{0}, \mathbf{1}] \to A)$ can be read as maps from the unit *square* (or *2-cube*) $[\mathbf{0}, \mathbf{1}]^2$ 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* [12], 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 gray 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.



(a) The theorem. **(b)** **(c)** **(d)**

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 gray 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, reparameterizing by a projection

[1] Because 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.

$[\mathbf{0},\mathbf{1}]^3 \to [\mathbf{0},\mathbf{1}]^2$. We cannot hope to derive the cube in Figure 1(a) only by contorting the gray or hatched 2-cube in isolation, of course. 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" $t \in [\mathbf{0},\mathbf{1}]$ 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 theory. 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.

We focus on the type theory of `Cubical Agda` [37], currently the most widely used cubical system. The `agda/cubical` [31] and `1lab` [30] libraries each contain ad-hoc collections of cubical reasoning combinators, providing examples on which we test our solver. However, some of the techniques developed in the paper also apply to other cubical type theories, in particular to cartesian cubical type theory [2, 1] as implemented in `redtt` [32] and `cooltt` [33]. A particular feature of `Cubical Agda` that contrasts with cartesian cubical systems is the richer language of contortions, which has proved interesting to study on its own. Like fully-featured cubical type theories, the intended denotational semantics of our language is in cubical sets over any cartesian cube category with connections, following Cohen et al. [9] or Orton and Pitts [24].

Contributions. This paper constitutes one of the first systematic studies of automated reasoning for cubical type theory. In it we

- formulate a minimal cubical language containing a class of "Dedekind" contortions and a Kan filling operator and precisely state the automation problems that we consider (§2),
- formulate an algorithm based on poset maps for solving problems using contortions (§3),
- formulate an algorithm based on constraint satisfaction programming for solving problems using Kan filling (§4), 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` (§5).

2 Boundary problems in cubical type theory

Cubical type theories are complex systems. Besides path types, one has the usual type formers of type theory – functions, products, inductive types, etc. – not to mention univalence and HITs. To make automation tractable, we restrict attention to a fragment including

[2] The 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.

only basic operations on cubical cells in a single type.[3] We also restrict attention to the Dedekind/distributive lattice fragment of `Cubical Agda` and do not rely on the De Morgan involution. In the terminology of `Cubical Agda`, we consider automation for homogeneous Kan filling (hfill) and connections ($\wedge$, $\vee$).

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 $[\mathbf{0}, \mathbf{1}]$; intuitively, a cell in a type A in n variables is a function $[\mathbf{0}, \mathbf{1}]^n \to A$. Our contexts are lists of cells each of which can have a specified *boundary*. For example, $p(i) : [\, i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto b \,]$ specifies a 1-dimensional cell p varying in $i \in [\mathbf{0}, \mathbf{1}]$ such that $p(\mathbf{0}) = a$ and $p(\mathbf{1}) = b$, i.e., a path from a to b. In general, a hypothesis has the form $q(\Psi) : [\, \phi \,]$ where Ψ is a list of variables and ϕ is a list of values at faces ($i = \mathbf{0}$ and $i = \mathbf{1}$ in the example above). A cell hypothesis is thus a judgmental analogue of a hypothesis of *extension type* à la Riehl and Shulman [26, §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 "$\Gamma \mid \Psi \vdash \mathbf{?} : [\, \phi \,]$". For example, if we want to prove that paths are invertible, then we could pose the boundary problem

$$a : [\,], b : [\,], p(i) : [\, i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto b \,] \mid j \vdash \mathbf{?} : [\, j = \mathbf{0} \mapsto b \mid j = \mathbf{1} \mapsto a \,] \tag{1}$$

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 $j \in [\mathbf{0}, \mathbf{1}]$ with fixed endpoints. We can leave the boundary of cells partially or completely unspecified, so we can formulate the same problem more compactly as

$$p(i) : [\,] \mid j \vdash \mathbf{?} : [\, j = \mathbf{0} \mapsto p(\mathbf{1}) \mid j = \mathbf{1} \mapsto p(\mathbf{0}) \,] \tag{2}$$

Now we assume a path p without naming its endpoints and seek a path from $p(\mathbf{1})$ to $p(\mathbf{0})$. The format extends gracefully to higher cells; the diagonal of a square can be obtained by posing

$$s(i, j) : [\,] \mid k \vdash \mathbf{?} : [\, k = \mathbf{0} \mapsto s(\mathbf{0}, \mathbf{0}) \mid k = \mathbf{1} \mapsto s(\mathbf{1}, \mathbf{1}) \,] \tag{3}$$

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 boundary problems more formally together with the two operations used to build solutions: contortions and Kan filling.

2.1 Boundary problems and contortion solving

The problem (3) has a simple solution: $\mathbf{?} := s(k, k)$. That is, we take the hypothesised 2-cube s and apply a reparameterisation $k \mapsto (k, k)$. We call such reparameterisations *contortions*. In addition to variables and the constants $\mathbf{0}$,$\mathbf{1}$, we follow Cohen et al. (CCHM) [9] and include binary operators $\wedge$ and $\vee$, conventionally called *connections* [5], in our language for contortions. We think of $\wedge$ as taking the *minimum* of two parameters and $\vee$ as taking the maximum; they obey the laws of a bounded distributive lattice with $\mathbf{0}$ and $\mathbf{1}$ as bottom and top element respectively. For example, given a cell context containing a path p, we can define a square whose value at the coordinate (j, k) is the value of p at their maximum:

$$p(i) : [\,] \mid j, k \vdash p(j \vee k) : \begin{bmatrix} j = \mathbf{0} \mapsto p(k) & \mid & k = \mathbf{0} \mapsto p(j) \\ j = \mathbf{1} \mapsto p(\mathbf{1}) & \mid & k = \mathbf{1} \mapsto p(\mathbf{1}) \end{bmatrix} \tag{4}$$

[3] This is similar to the fragments of type theory used to axiomatise higher structures such as weak ω-groupoids in e.g. [6, Appendix A] and [13].

Thus one way to solve a boundary problem is by applying a contortion to a hypothesised cell; searching for such solutions is the first automation problem we consider.

▶ Remark 1. Some cubical type theories, such as that of Angiuli et al. [2, 1], omit connections, while `Cubical Agda` also has the De Morgan involution of CCHM which sends $i \in [\mathbf{0}, \mathbf{1}]$ to $\mathbf{1} - i$. While removing connections would make it trivial to solve contortion goals, we would instead be more reliant on the even harder problem of using Kan filling to solve goals. On the other hand, the poset map representation we use for solving in §3 relies on the absence of the involution. Thus we choose the distributive lattice contortion theory as a happy medium.

Let us now introduce the formal language of boundary problems and solutions.

▶ **Definition 2.** *A **dimension context** Ψ is either a list of (unique) dimension variables $(i_1, \dots, i_n)$ or the **inconsistent context** $\bot$.*

We think of a dimension context with n variables as a unit n-cube, while $\bot$ is the empty space; note that the "empty" context $()$ is the unit 0-cube, which does have a unique point.

▶ **Definition 3.** *A **dimension term** $\Psi \vdash r$ dim over a dimension context Ψ is a term in the free bounded distributive lattice over Ψ. All terms are equal in the inconsistent dimension context. A dimension term r is **atomic** when it is a variable or endpoint, in which case we write $\Psi \vdash r$ atom. We write $\overline{e}$ for the opposite of an endpoint e, so $\overline{\mathbf{0}} \coloneqq \mathbf{1}$ and $\overline{\mathbf{1}} \coloneqq \mathbf{0}$.*

▶ **Definition 4.** *A **contortion** $\psi \colon \Psi' \rightsquigarrow \Psi$ when $\Psi = (i_1, \dots, i_n)$ is a list $\psi = (r_1, \dots, r_n)$ of dimension terms over Ψ'. There is a contortion $\psi \colon \Psi' \rightsquigarrow \bot$ only when $\Psi' = \bot$, in which case there is a unique one. A **substitution** $\psi \colon \Psi' \to \Psi$ is a contortion whose terms are atomic.*

A substitution defines an operation on terms in the usual way: given some kind of term t and a substitution $\psi \colon \Psi' \to \Psi$ 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. Only some of our syntactic sorts are closed under application of general contortions, namely dimension terms and contorted cells (Definition 9); for those sorts we write $t\langle\psi\rangle$ for application of a contortion.

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

▶ **Definition 5.** *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:*

$$(\Psi, i, \Psi')[i = e] \coloneqq \Psi, \Psi' \qquad \Psi[\overline{e} = e] \coloneqq \bot \qquad \Psi[r = e] \coloneqq \Psi, \text{ otherwise}$$

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

The *cell contexts* (Γ ctxt), *contorted boundaries* ($\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy), and *contorted cells* ($\Gamma \mid \Psi \vdash_{\mathrm{c}} t$ cell and $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$) are mutually inductively defined as follows.

▶ **Definition 6.** *The **cell contexts** Γ ctxt are inductively defined by rules*

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

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.

▶ **Definition 7.** *The **contorted boundaries** $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy are inductively defined by rules*

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

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 t is a contorted cell over $\Psi[r = e]$, such that each entry agrees with the previous entries when their constraints overlap.

▶ Remark 8. 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 5), while relaxing it is not particularly useful for practical boundary solving. We distinguish between substitutions and contortions to make this requirement sensible.

▶ **Definition 9.** *A **contorted cell** $\Gamma \mid \Psi \vdash_{\mathrm{c}} t$ cell is an application $t = a(\psi)$ of a variable $a(\Psi') : [\,\phi\,]$ in Γ to some $\psi \colon \Psi \rightsquigarrow \Psi'$. Equality of contorted cells is generated by the rule*

$$\frac{(a(\Psi') : [\,\phi\,]) \in \Gamma \qquad (r = e \mapsto t) \in \phi \qquad \psi \colon \Psi \rightsquigarrow \Psi' \qquad \Psi \vdash r\langle\psi\rangle = e\langle\psi\rangle \text{ dim}}{\Gamma \mid \Psi \vdash_{\mathrm{c}} a(\psi) = t\langle\psi\rangle \text{ cell}}$$

which is to say that a has the boundary assigned in the context. We write $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$ when t is a cell agreeing with ϕ, i.e., such that $t[r = e] = t'$ cell for each $(r = e \mapsto t') \in \phi$.

▶ Remark 10. We normalise a contorted cell by looking at its boundary, if it is specified. For example, in context (1) the cell $p(\mathbf{0})$ normalises to a. The number of steps necessary to normalise a contorted cell is bounded by the length of the context.

We can now state the first of the two problems we are interested in solving.

▶ **Problem 11** (Contortion)**.** *Given $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy, the problem Contortion(Γ, Ψ, ϕ) is to determine if there exists a contorted cell t such that $\Gamma \mid \Psi \vdash_{\mathrm{c}} t : [\,\phi\,]$.*

The problem Contortion(Γ, Ψ, ϕ) is decidable: there are finitely many cell variables in Γ, so we could try all possible contortions of each by brute force. On the other hand, it is certainly not easily solved efficiently. Even restricting to 1-dimensional goals, Contortion is NP-hard when the contortion language includes connections.

▶ **Proposition 12.** *Contortion(Γ, Ψ, ϕ) is NP-complete for Ψ with one variable.*

Proof. First, note that this problem is in NP: when the cardinality of Ψ is fixed, the normalisation necessary to verify a putative solution can be done in polynomial time. For completeness, we give a reduction from SAT. Suppose we have a Boolean CNF formula φ over $\vec{x} = x_1, \ldots, x_n$. 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 $\psi \colon () \rightsquigarrow (\vec{x}, \vec{y})$ 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

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

and consider the boundary problem $\Gamma_\varphi \mid i \vdash_{\mathrm{c}} ? : [\,i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto p(\mathbf{1}, \mathbf{0}, \mathbf{1})\,]$. Any $\psi \colon () \rightsquigarrow (\vec{x}, \vec{y})$ 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 $\Gamma_\varphi \mid i \vdash_{\mathrm{c}} q(\psi, i)$ cell. Conversely, any solution to the problem will be of the form $\Gamma_\varphi \mid i \vdash_{\mathrm{c}} q(\psi', r)$ cell for some $\psi' \colon i \rightsquigarrow (\vec{x}, \vec{y})$ and $i \vdash r$ dim, in which case $\psi'(\mathbf{1}) \colon () \rightsquigarrow (\vec{x}, \vec{y})$ induces a satisfying assignment for φ. ◀

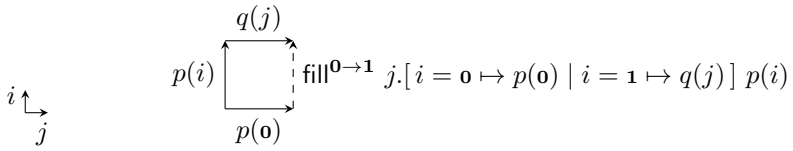
We expect that Contortion is significantly *more* complex in general and that the complexity is dominated by the dimensionality of the goal. The number of ways to contort an m-cube to fit an n-dimensional goal is $D(n)^m$ where $D(n)$ is the n-th *Dedekind number* [4, App. B]. While the exponential dependence on m is already problematic, the Dedekind numbers grow even more 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 [15, 19]. Thus, our focus in this paper is on heuristics that quickly yield solutions to boundary problems that appear in practice, rather than on worst-case asymptotics.

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 [20] 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 $\mathsf{fill}^{e\to r}\ j.[\,\phi\,]\ u$, 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 hcomp/hfill of [10], while being a special case of the more general hcom of [2, 1].

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



We write $(p \cdot q)(i)$ for the filler above. If we replace the target $\mathbf{1}$ of the fill with a variable, we get an interior for the depicted square. One says that this cell *fills* the open box formed by the solid paths, hence the name. In general, we have the following rules for building Kan cells:

▶ **Definition 13.** *The* ***Kan cells*** $\Gamma \mid \Psi \vdash t\ \mathsf{cell}$ *are inductively generated by the rules*

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

The ***Kan boundaries*** $\Gamma \mid \Psi \vdash \phi\ \mathsf{bdy}$ *are defined analogously to the contorted boundaries* $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi\ \mathsf{bdy}$*, and we write* $\Gamma \mid \Psi \vdash t : [\,\phi\,]$ *to mean that* t *is a Kan cell agreeing with* ϕ*.*

▶ Remark 14. Note that the $r = e \mapsto u$ constraint makes filling in direction $e \to e$ the identity function. This ensures that the face opposite the missing side in the filler agrees with the input u to the filling operator.

With this we can now state the second of the two problems we are interested in solving.

▶ **Problem 15** (KAN). *Given a Kan boundary* $\Gamma \mid \Psi \vdash \phi$ bdy*, the problem* $\textsc{Kan}(\Gamma, \Psi, \phi)$ *is to determine if there exists a Kan cell* t *such that* $\Gamma \mid \Psi \vdash t : [\,\phi\,]$.

For example, the problem (2) of inverting a path does not have a solution in CONTORTION but does have solutions in KAN, such as $\mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ i.[\,j = \mathbf{0} \mapsto p(i) \mid j = \mathbf{1} \mapsto p(\mathbf{0})\,]\ p(\mathbf{0})$.

▶ Remark 16. We claim that Kan is undecidable, but give only a proof sketch here. Although the main idea is simple, a fully precise proof would be combinatorially involved. We can encode the word problem for a finitely presented group G as an instance of KAN. Such a group can always be presented by a finite set of generators X and relations $ab = c$ where $a, b, c \in X$. Take the context Γ with a point $\star : [\,]$, a loop $\hat{a}(i) : [\,i = \mathbf{0} \mapsto \star \mid i = \mathbf{1} \mapsto \star\,]$ for each $a \in X$, and a square $s_{abc}(j,k) : [\,j = \mathbf{0} \mapsto \hat{a}(k) \mid j = \mathbf{1} \mapsto \star \mid k = \mathbf{0} \mapsto \hat{c}(j) \mid k = \mathbf{1} \mapsto \hat{b}(j)\,]$ for each $ab = c$. Note that each square s_{abc} expresses that the composite of the loops $\hat{a}$ and $\hat{b}$ is equal up to a path to the loop $\hat{c}$. Given arbitrary words w, v over X, we have loops $\Gamma \mid i \vdash \hat{w}, \hat{v} : [\,i = \mathbf{0} \mapsto \star \mid i = \mathbf{1} \mapsto \star\,]$ defined by concatenating generator loops. We claim that $w = v$ in G if and only if $\Gamma \mid i, \ell \vdash \mathbf{?} : [\,i = \mathbf{0} \mapsto \star \mid i = \mathbf{1} \mapsto \star \mid \ell = \mathbf{0} \mapsto \hat{w} \mid \ell = \mathbf{1} \mapsto \hat{v}\,]$ has a solution in KAN, i.e., if $\hat{w}$ and $\hat{v}$ are equal up to a path.

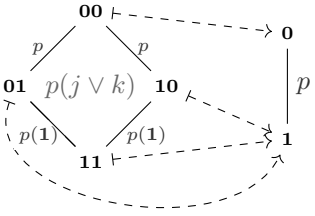
3 Finding contortions

The key to our approach for solving the CONTORTION problem is a characterisation of contortions as poset maps by Stone duality [18]. In §3.1, we introduce this characterisation along with *potential poset maps (PPMs)*, a lossy but space-saving representation of collections of poset maps. We use PPMs in §3.2 to develop an algorithm for solving CONTORTION.

3.1 Representing contortions with potential poset maps

Recall the example (4), where we contorted a path p into a square $p(i) : [\,] \mid j, k \vdash p(j \vee k)$ 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 particular, every contortion has a corresponding truth table, and in fact a contortion is uniquely determined by its truth table; for example, the contortion above is determined by the assignment $[\![-]\!] \colon \{\mathbf{0},\mathbf{1}\} \times \{\mathbf{0},\mathbf{1}\} \to \{\mathbf{0},\mathbf{1}\}$ defined by $[\![\mathbf{00}]\!] = \mathbf{0}$ and $[\![\mathbf{01}]\!] = [\![\mathbf{10}]\!] = [\![\mathbf{11}]\!] = \mathbf{1}$. In general, an n-term contortion in m variables gives a truth function $\{\mathbf{0},\mathbf{1}\}^m \to \{\mathbf{0},\mathbf{1}\}^n$.

Since a 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 $\mathbf{I}^m \to \mathbf{I}^n$, where $\mathbf{I}^k$ is the k-fold power of the poset $\mathbf{I} \coloneqq \{\mathbf{0} < \mathbf{1}\}$ with its product ordering. Conversely, any map of posets $\mathbf{I}^m \to \mathbf{I}^n$ 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:



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

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

With a PPM, we can represent a collection of contortions with very little data: representing all $D(m)^n$ poset maps $\mathbf{I}^m \to \mathbf{I}^n$ with the total PPM $x \mapsto \mathbf{I}^n$ for $x \in \mathbf{I}^m$ requires 2^m entries of 2^n values – the memory requirements are 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 $\sigma, \sigma' : \mathbf{I}^1 \to \mathbf{I}^2$ 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 UnfoldPPM(Σ). We update a PPM Σ to restrict its values at x to some set $vs \subseteq \Sigma(x)$ using UpdatePPM(Σ, 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 §5 for details.

3.2 An algorithm for gradually constructing contortions

We now use PPMs in Algorithm 1 to solve Contortion more efficiently than by brute force. Given a boundary problem $\Gamma \mid \Psi \vdash_{\mathrm{c}} ? : [\phi]$ and a cell $a(\Psi') : [\phi']$ in Γ, we search for a contortion $\psi\colon \Psi \rightsquigarrow \Psi'$ such that $a\langle\psi\rangle$ 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 initialize Σ to the total PPM on line 2. We then go through the faces of ϕ, each of which normalises to the form $b\langle\psi\rangle$ 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 $i = e \mapsto b\langle\psi\rangle$ 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 $\mathbf{I}^m_{i=e}$ on line 8. For each, we mark its values for retention only when $a\langle\sigma_{\vee\wedge}\rangle$ matches the face $b\langle\psi\rangle$. 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.

The main expense in Algorithm 1 is 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.

Algorithm 1 Constructing a contortion.

Input: $\Gamma \mid \Psi \vdash_{\mathrm{c}} \phi$ bdy and $a(\Psi') : [\,\phi'\,] \in \Gamma$. Let $m \coloneqq |\Psi|$ and $n \coloneqq |\Psi'|$.
Output: $\psi \colon \Psi \rightsquigarrow \Psi'$ s.t. $\Gamma \mid \Psi \vdash_{\mathrm{c}} a\langle\psi\rangle : [\,\phi\,]$ if such a ψ exists, `Unsolvable` otherwise

1: **procedure** CONTORT(Γ, Ψ, ϕ, a)
2: $\quad\Sigma \coloneqq \{x \mapsto \mathbf{I}^n \mid x \in \mathbf{I}^m\}$
3: $\quad$**for** $(i = e \mapsto b\langle\psi\rangle) \in \phi$ with $\psi \colon \Psi[i = e] \rightsquigarrow \Psi''$, in descending order of $|\Psi''|$ **do**
4: $\quad\quad$**if** $a = b$ **then**
5: $\quad\quad\quad\Theta \coloneqq \{x \mapsto \{\psi_{\mathbf{I}}(x)\} \mid x \in \mathbf{I}^m_{i=e}\}$
6: $\quad\quad$**else**
7: $\quad\quad\quad\Theta \coloneqq \{x \mapsto \emptyset \mid x \in \mathbf{I}^m_{i=e}\}$
8: $\quad\quad\quad$**for** $\sigma \in$ UNFOLDPPM$(\Sigma|_{\mathbf{I}^m_{i=e}})$ **do**
9: $\quad\quad\quad\quad$**if** $a\langle\sigma_{\vee\wedge}\rangle = b\langle\psi\rangle$ **then**
10: $\quad\quad\quad\quad\quad$**for** $x \in \mathbf{I}^m_{i=e}$ **do**
11: $\quad\quad\quad\quad\quad\quad\Theta(x) \coloneqq \Theta(x) \cup \{\sigma(x)\}$
12: $\quad\quad$**for** $x \in \mathbf{I}^m_{i=e}$ **do**
13: $\quad\quad\quad$UPDATEPPM$(\Sigma, x, \Theta(x))$
14: $\quad$**if** $\exists\sigma \in$ UNFOLDPPM(Σ) such that $\Gamma \mid \Psi \vdash_{\mathrm{c}} a\langle\sigma_{\vee\wedge}\rangle : [\,\phi\,]$ **then**
15: $\quad\quad$**return** $\sigma_{\vee\wedge}$
16: $\quad$**else**
17: $\quad\quad$**return** `Unsolvable`

▶ **Example 18** (Square to cube contortion)**.** Suppose that we are given the cell context $\Gamma \coloneqq a : [\,], s(i,j) : [\,i = \mathbf{0} \mapsto a \mid i = \mathbf{1} \mapsto a \mid j = \mathbf{0} \mapsto a \mid j = \mathbf{1} \mapsto a\,]$ 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:

$$\Gamma \mid i,j,k \vdash_{\mathrm{c}} \mathbf{?} : \begin{bmatrix} i = \mathbf{0} \mapsto s(j \wedge k, j \vee k) & \mid & j = \mathbf{0} \mapsto a & \mid & k = \mathbf{0} \mapsto a \\ i = \mathbf{1} \mapsto a & \mid & j = \mathbf{1} \mapsto a & \mid & k = \mathbf{1} \mapsto a \end{bmatrix}$$

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

On line 2, the total PPM $\Sigma : \mathbf{I}^3 \to \mathcal{P}(\mathbf{I}^2)$ is initialized with $x \mapsto \mathbf{I}^2$ for all $x \in \mathbf{I}^3$. We then go through all faces of the goal boundary and use them to restrict Σ, starting with the contortion of s at $i = \mathbf{0}$. Since s is also the cell that we are contorting, the subposet $\mathbf{I}^3_{i=\mathbf{0}}$ of the domain of Σ is mapped in a unique way to the elements of $\mathbf{I}^2$. The monotonicity restrictions on PPMs further restrict Σ, 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: $\Sigma(\mathbf{000}) = \{\mathbf{00}\}$, $\Sigma(\mathbf{001}) = \Sigma(\mathbf{010}) = \Sigma(\mathbf{011}) = \Sigma(\mathbf{100}) = \Sigma(\mathbf{101}) = \{\mathbf{01}\}$ and $\Sigma(\mathbf{110}) = \Sigma(\mathbf{111}) = \{\mathbf{11}\}$. Translating this poset map to a contortion gives rise to a solution for our boundary problem: $\Gamma \mid i,j,k \vdash_{\mathrm{c}} s(i \wedge j, i \vee j \vee k) : [\,\phi\,]$

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

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

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. We formulate the problem of finding a higher-dimensional filler which only uses contorted cells as a constraint satisfaction problem (CSP) [21, 34] in §4.1, 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 §4.2.

4.1 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, making it quickly infeasible to list all contortions. To rectify this, we again rely on PPMs. 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 §5 can then gradually narrow down the PPMs until it arrives at a solution.

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 19.** *Given a boundary* $\Gamma \mid \Psi \vdash \phi\ \mathsf{bdy}$ *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* $\mathrm{KANCSP}(\phi, Ope)$ *is given as follows:*

- $Var := \{X_{(i=e)} \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}, (i = e) \notin Ope\} \cup \{X_{(k=\mathbf{0})} \text{ if } (k = \mathbf{0}) \notin Ope\}$
- $D_{(i=e)} := \{(p, \Sigma) \mid p(\Psi') : [\,\ldots\,] \in \Gamma, \Sigma : \mathbf{I}^{|\Psi|} \to \mathcal{P}(\mathbf{I}^{|\Psi'|})\}$

- *and constraints for all* $\Psi \vdash i, j$ atom*,* $e, e' \in \{\mathbf{0}, \mathbf{1}\}$*:*

$$\Gamma \mid \Psi[i = e] \vdash_{\mathrm{c}} X_{(i=e)}[k = \mathbf{1}] = \phi[i = e]\ \mathsf{cell}\ \textit{if}\ (i, e)\ \textit{specified in}\ \phi$$
$$\Gamma \mid \Psi[i = e][j = e'] \vdash_{\mathrm{c}} X_{(i=e)}[j = e'] = X_{(j=e')}[i = e]\ \mathsf{cell}$$

The CSP contains a variable for any side of the boundary that is not left open, the domains contain all pairs of a cell p and PPM Σ contorting p into the right 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 $\textsc{KanCSP}(\phi, \mathit{Ope})$ is a solution to the boundary problem ϕ:

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

▶ **Example 20** (The Eckmann-Hilton cube)**.** We want to fill the cube from Figure 1(b), for which 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:

$$\Gamma \mid i, j, k \vdash \begin{bmatrix} i = \mathbf{0} \mapsto p(j,k) \mid & j = \mathbf{0} \mapsto q(i,k) \mid & k = \mathbf{0} \mapsto x \\ i = \mathbf{1} \mapsto p(j,k) \mid & j = \mathbf{1} \mapsto q(i,k) \mid & k = \mathbf{1} \mapsto x \end{bmatrix} \mathsf{bdy}$$

We try to solve KanCSP with no open sides. This CSP has 7 variables corresponding to sides i, j, k and a backside $l = \mathbf{0}$. After imposing the first set of constraints, the domains for the i and j sides are significantly reduced, e.g., $D_{(i=\mathbf{0})} = \{p\langle\Sigma\rangle\}$ for $\Sigma : \mathbf{I}^3 \to \mathcal{P}(\mathbf{I}^2)$ 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 $D_{(k=\mathbf{0})}$, $D_{(k=\mathbf{1})}$, and the back side $D_{(l=\mathbf{0})}$ still contain all contortions of x, p and q into three dimensions since the k sides of the goal boundary does 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:

$$\Gamma \mid i, j, k \vdash \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ l. \begin{bmatrix} i = \mathbf{0} \mapsto p(j, k \wedge l) & j = \mathbf{0} \mapsto q(i,k) & k = \mathbf{0} \mapsto x \\ i = \mathbf{1} \mapsto p(j, k \wedge l) & j = \mathbf{1} \mapsto q(i,k) & k = \mathbf{1} \mapsto p(j,l) \end{bmatrix} q(i,k)\ \mathsf{cell}$$

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).

4.2 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 can straightforwardly devise a procedure $\textsc{KanFill}(\Gamma, \Psi, \phi)$ 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.

Algorithm 2 Finding Kan cells.

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

1: **procedure** $\textsc{KanSolver}(\Gamma, \Psi, \phi, d)$
2: **if** $d = 0$ **then**
3: **return** `Unsolvable`
4: $t \leftarrow \textsc{KanFill}(\Gamma, \Psi, \phi) \cup \textsc{KanCube}(\Gamma, \Psi, \phi, d)$
5: **procedure** $\textsc{KanCube}(\Gamma, \Psi, \phi, d)$
6: $Ope \leftarrow \mathcal{P}(\{(i = e) \mid i \in \Psi, e \in \{\mathbf{0}, \mathbf{1}\}\} \cup \{(k = \mathbf{0})\})$
7: $\phi' \leftarrow \textsc{KanCSP}(\phi, Ope)$
8: **for** $(i = e) \in Ope$ **do**
9: $t \leftarrow \textsc{KanSolver}([\,\phi'[i = e]\,], d - 1)$
10: $\phi' \coloneqq [\,\phi' \mid i = e \mapsto t\,]$
11: **return** $\Gamma \mid \Psi \vdash \mathsf{fill}^{\mathbf{0}\to\mathbf{1}}\ k.[\,\phi' - (k = \mathbf{0})\,]\ (\phi'[k = \mathbf{0}]) : [\,\phi\,]$

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'.

The choices of solutions and open sides on lines 4, 6, 7 and 9 are made non-deterministically, which is done using the list monad in the solver discussed in §5. 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 $Ope = \emptyset$. 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 embed Algorithms 1 and 2 into a complete search procedure for Kan in Algorithm 3. Our Solver starts by trying to contort some cell of the cell context into the goal boundary. If this fails, we perform iterative deepening on the level of nested Kan cells.

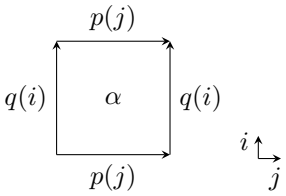
Algorithm 3 A solver for boundary problems.

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

1: **procedure** $\textsc{Solver}(\Gamma, \Psi, \phi)$
2: **for** $p \in \Gamma$ **do**
3: $t \leftarrow \textsc{Contort}(\Gamma, \Psi, \phi, p)$
4: **if** $t \neq$ `Unsolvable` **then**
5: **return** t
6: **for** $d \in \{1, \ldots\}$ **do**
7: $t \leftarrow \textsc{KanSolver}(\Gamma, \Psi, \phi, d)$
8: **if** $t \neq$ `Unsolvable` **then**
9: **return** t

▶ **Example 21** (Sq→Comp). To complete the proof of Eckmann-Hilton, we need to fill the cube from Figure 1(a) using Figure 1(b). We can do this already at a lower dimension – which means at greater generality: the cube from Figure 1(b) is captured with a square

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



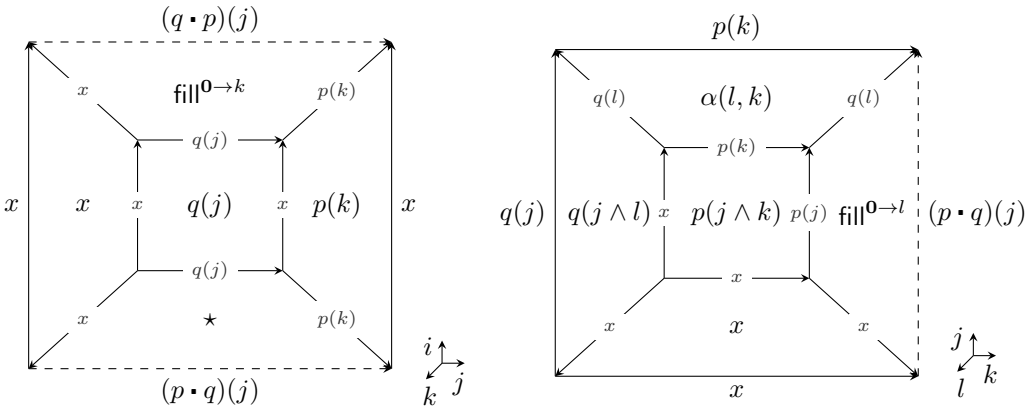
that we want to turn into a square with both path concatenations on opposite sides:

$$\Gamma \mid i,j \vdash \mathbf{?} : [\,i = \mathbf{0} \mapsto (p \cdot q)(j) \mid i = \mathbf{1} \mapsto (q \cdot p)(j) \mid j = \mathbf{0} \mapsto x \mid j = \mathbf{1} \mapsto x\,] \tag{5}$$

Incidentally, Γ is the list of generators of the HIT capturing the Torus in `agda/cubical`, while boundary (5) captures T^2, the definition of the torus in the HoTT book [35]. A solution to this problem thus induces a map from the cubical torus to the HoTT book torus.

We solve (5) 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 KanCSP 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 KanCSP 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.



5 A practical solver for Cubical Agda boundary problems

We have implemented the solver in `Haskell`,[4] 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 [25]. 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 work in progress.

[4] Source code and examples are available at: `https://github.com/maxdore/dedekind`

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 < 50ms). 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 20.

In `Cubical Agda`, the constant path at x of type $x \equiv x$ is expressed with λ-abstraction as $\lambda\ i \to x$. We can use the PathP type to describe higher-dimensional boundaries, e.g., PathP $(\lambda\ j \to x \equiv x)\ (\lambda\ i \to x)\ (\lambda\ i \to x)$ 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 ∼150ms:

```
EckmannHilton-Cube : PathP (λ i → q i ≡ q i) p p
EckmannHilton-Cube = λ i j k → hcomp (λ l → λ {
  (i = i0) → p j (k ∧ l) ; (j = i0) → q i k ; (k = i0) → x ;
  (i = i1) → p j (k ∧ l) ; (j = i1) → q i k ; (k = i1) → p j l }) (q i k)
```

The `Cubical Agda` primitive hcomp captures Kan fillers in direction $\mathbf{0} \to \mathbf{1}$. The solution to the boundary problem discussed in the Sq→Comp example is found in ∼15ms, 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 (λ j → q j ≡ q j) p p → p ∙ q ≡ q ∙ p
Sq→Comp α i j = hcomp (λ k → λ {
  (i = i0) → hcomp (λ l → λ {
    (j = i0) → x ; (k = i0) → q (j ∧ l) ; (j = i1) → α l k ;
    (k = i1) → hfill (λ m → λ { (j = i0) → x ; (j = i1) → q m }) (inS (p j)) l })
    (p (j ∧ k)) ;
  (i = i1) → hfill (λ l → λ { (j = i0) → x ; (j = i1) → p l }) (inS (q j)) k ;
  (j = i0) → x ; (j = i1) → p k })
  (q j)
```

The function hfill $\phi\ t\ i$ is used in `agda/cubical` to define fillers in direction $\mathbf{0} \to i$. 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 ∙ q ≡ q ∙ p
EckmannHilton = Sq→Comp p q EckmannHilton-Cube
```

The boundary problem posed by EckmannHilton can also be passed directly to our solver, however, it is not yet able to prove this problem 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 [28], 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.

6 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 `Cubical Agda`; the contortion theory should be extended to also include the De Morgan involution. 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 [7] which used path-induction and metaprogramming in `Agda`. The problem of deciding equality in the cofibration logic of cubical type theories has been studied by [27]. Among other things, they also establish complexity-related results, in particular, that the entailment problems of the cofibration languages of [1] and [9] are coNP-complete. Another line of related work where the relationship to our work needs to be better understood is higher-dimensional rewriting, in particular, those based on ∞-categories [14], operads [29], polygraphs [3] and associative n-categories [11].

References

1 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.

2 Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, pages 6:1–6:17, 2018. doi:10.4230/LIPIcs.CSL.2018.6.

3 Dimitri Ara, Albert Burroni, Yves Guiraud, Philippe Malbos, François Métayer, and Samuel Mimram. Polygraphs: From rewriting to higher categories. Submitted, 2023. arXiv:2312.00429.

4 Steve Awodey. Cartesian cubical model categories, 2023. arXiv:2305.00893.

5 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.

6 Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory*. PhD thesis, Université de Nice Sophia Antipolis, 2016.

7 Guillaume Brunerie. Computer-generated proofs for the monoidal structure of the smash product. *Homotopy Type Theory Electronic Seminar Talks*, November 2018. URL: https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest.html.

8 J. Daniel Christensen and Luis Scoccola. The Hurewicz theorem in Homotopy Type Theory. *Algebraic & Geometric Topology*, 23:2107–2140, 2023.

9 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 2015)*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.

10 Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 255–264. ACM, 2018. doi:10.1145/3209108.3209197.

11 Christoph Dorn. *Associative n-categories*. PhD thesis, University of Oxford, 2018.

12 Beno Eckmann and Peter J. Hilton. Group-like structures in general categories I: multiplications and comultiplications. *Mathematische Annalen*, 145:227–255, 1962.

13 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)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005124.

14 Eric Finster, David Reutter, Jamie Vicary, and Alex Rice. A type theory for strictly unital ∞-categories. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2022. doi:10.1145/3531130.3533363.

15 Lennart Van Hirtum, Patrick De Causmaecker, Jens Goemaere, Tobias Kenter, Heinrich Riebler, Michael Lass, and Christian Plessl. A computation of D(9) using FPGA Supercomputing, 2023. arXiv:2304.03039.

16 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 *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 565–574. Association for Computing Machinery, 2016. doi:10.1145/2933575.2934545.

17 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 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2016.22.

18 Peter Tennant Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

19 Christian Jäkel. A computation of the ninth Dedekind Number, 2023. arXiv:2304.00895.

20 Daniel M. Kan. Abstract homotopy. I. *Proceedings of the National Academy of Sciences of the United States of America*, 41(12):1092–1096, 1955.

21 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.

22 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.

23 Anders Mörtberg and Loïc Pujet. Cubical Synthetic Homotopy Theory. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 158–171, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373825.

24 Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science*, Volume 14, Issue 4, December 2018. doi:10.23638/LMCS-14(4:23)2018.

25 David Overton. Constraint Programming in Haskell. Melbourne Haskell Users Group, 2015. URL: https://de.slideshare.net/davidoverton/constraint-programming-in-haskell.

26 Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):116–193, 2017.

27 Matthew Z. Weaver Robert Rose and Dan R. Licata. Deciding entailment for cofibration languages. At the second international conference on *Homotopy Type Theory*, 2023. URL: https://hott.github.io/HoTT-2023/slides/rose.pdf.

28 Kristina Sojakova and G. A. Kavvos. Syllepsis in Homotopy Type Theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Association for Computing Machinery, 2022. doi:10.1145/3531130.3533347.

29 Cédric Ho Thanh, Pierre-Louis Curien, and Samuel Mimram. A sequent calculus for opetopes. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2019. doi:10.1109/LICS.2019.8785667.

30 The 1Lab Development Team. The 1Lab, 2023. URL: https://1lab.dev.

31 The Agda Community. Cubical Agda Library, October 2023. URL: https://github.com/agda/cubical.

32 The RedPRL Development Team. The `redtt` proof assistant, 2018. URL: https://github.com/RedPRL/redtt/.

33 The RedPRL Development Team. The `cooltt` proof assistant, 2021. URL: https://github.com/RedPRL/cooltt/.

34 Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993. doi:10.1016/C2013-0-07627-X.

35 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

36 Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory*. PhD thesis, Carnegie Mellon University, May 2018. URL: https://arxiv.org/abs/1808.10690.

37 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.

38 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.