Thesis Proposal

Higher Inductive Types and Parametricity in Cubical Type Theory

Evan Cavallo

June 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Keywords: cubical type theory, higher inductive types, internal parametricity, parametricity, homotopy type theory, cubical sets

Abstract

Cubical type theory is a novel extension of dependent type theory with a form of equality called a path. Path equality enjoys extensionality properties missing from traditional treatments of equality, and it is proof-relevant: paths are data, and there can be many different paths between the same pair of objects. In particular, any isomorphism of types gives rise to a path between them; as everything respects equality, this means that constructions can be transferred between isomorphic objects just as in informal mathematical practice.

By providing a sensible equality, cubical type theory also enables sensible quotient types. However, the proof-relevant setting also motivates a more general notion of quotient known as a higher inductive type. The idea is to regard a quotient as an inductive definition where the generating operations may construct paths as well as ordinary elements. From this starting point, several directions of generalization are possible, just as with ordinary inductive types: there are indexed higher inductive types, higher inductive-inductive types, and so on. Originally proposed for homotopy type theory, a progenitor of cubical type theory, instances of higher inductive types have been used to great effect, but their general theory is still emerging. In this proposal, I describe a bare-bones schema for indexed higher inductive types, complete with a computational interpretation. I propose to extend this work to a fully-featured schema including inductive-inductive types and to implement it as part of a cubical proof assistant.

Working with proof-relevant equality has its own challenges, and effectively reasoning with higher inductive types is still a murky business. I propose that parametricity is a useful tool for this purpose, and describe an extension of cubical type theory with internal parametricity primitives. I propose to establish a connection between the extended theory and ordinary cubical type theory, to use it to simplify presently difficult proofs, and to implement the parametric extension.

Contents

1 Introduction

Martin-Löf's dependent type theory [49, 50] is a powerful language for expressing constructive mathematics, one that takes seriously the principle that *proofs are programs*. It is particularly well-suited as a foundation for formal theorem proving: such storied proof assistants as Nuprl [26], Coq [60], and Agda [53] are based on closely related theories. However, its treatment of equality is fraught with complications.

A central tension is between the external *judgmental equality*, which is used to determine well-typedness, and *identity types*, the internal equality with which one states and proves theorems. In Martin-Löf type theory, these two notions of equality are tightly coupled—they coincide on closed terms—but are subject to conflicting demands. In a formalism, it is convenient to include only a decidable fragment of judgmental equality, enabling algorithmic type-checking. Identity types, on the other hand, are the subject of theorems, so ought to behave as "mathematically" as possible. For example, functions should be equal when they are equal pointwise, and equality of types should be somehow semantic rather than syntactic. But these semantic equalities are undecidable; so one must make a choice. Nuprl, for example, opts to work with the full, undecidable judgmental equality, while Coq and Agda sacrifice function extensionality and related principles.

Cubical type theory [25, 4] is a new solution to the problem of equality. An extension of Martin-Löf type theory, cubical type theory introduces a second form of judgmental equality, called a *path*, that plays the role of mathematical equality in place of identity types. By decoupling the mathematical equality from the existing judgmental "exact" equality, the difficulties inherent in Martin-Löf type theory are neutralized. Function extensionality for paths, for one, is a simple consequence of their judgmental structure. The separation of exact and mathematical equality also enables a *proof-relevant* treatment of the latter: a path is a piece of data, and there can be many paths between a pair of terms. This allows for a novel view of equality of types that validates Voevodsky's *univalence axiom* [66]: an equality between types is an equivalence (coherent isomorphism) between them. Univalence formalizes the familiar informal mathematical practice of treating isomorphic types as equal.

A new treatment of equality demands an attendant reassessment of *quotient types*. Quotients are ubiquitous in mathematics, but they present the same issues as functions for practical formalization: to quotient by undecidable relations, one must accept undecidable equality. (Accordingly, Agda and Coq lack quotient types, while Nuprl includes them.) Here, too, cubical type theory resolves the tension. In the richer setting of proof-relevant equality, however, orthodox quotient types are just one piece of a wilder vista of *higher inductive types*. In short, it matters not only *what* terms one equates, but *how* one equates them.

The higher inductive approach treats the paths introduced by a quotient type as constructors of an inductive definition. Intuitively, equalities are higher-dimensional elements of a type, so can be inductively generated just as ordinary elements are. Quotient types, then, inherit all the potential for generalization enjoyed by inductive types: indexed inductive types [29, 31], inductive-inductive types [52], inductive-recursive types [32], and so on. These have potential applications not only to formalization of results in programming [2] and type theory [1], but also to *homotopy theory*, the general study of higher-dimensional structure ([62, 36, 64], to list a few). However, much of the work in this area is speculative, using axiomatic extensions of type theory with instances of "higher inductive types" without precisely defining the general concept or constructing models to verify consistency. Only recently have researchers begun to address this lacuna [7, 33, 30, 41, 48].

Proof-relevant equality adds new expressiveness to type theory, but it also creates new obligations. As higher inductive constructions are stacked, for example, a programmer is forced to reason with two- and higher-dimensional objects. Such difficulties often arise in proofs of apparently innocuous properties. One notorious example is the *smash product*, a binary type constructor used in homotopy theory. Proving that this operator is associative is already a formidable task; proving that the associator is properly "1-coherent" even more so. Partial proofs have been given by van Doorn and Brunerie, but both contain gaps—for reasons of technical rather than conceptual complexity [64, 18]. For problems like these, the field of programming languages suggests a proof technique: Reynolds' relational parametricity [56]. In brief, parametricity uses a relational interpretation of type theory to prove meta-theoretic "uniformity" properties of terms, which could in this case be used to see that certain maps must be equivalences.

For formalized theorem proving, we would like to have access to these properties *inside of* type theory. Bernardy, Jansson, and Paterson observe that dependent type theory is powerful enough to internally express the consequences of parametricity at a given type [13]. Bernardy and Moulin take this a step farther, developing an *internally parametric type theory* [11, 12, 14]. Internally parametric type theory has a close relationship with cubical type theory, as both use a judgmental notion of relatedness based on *dimension variables*. Moreover, it provides a proof-relevant notion of parametricity, essential for proving results about proof-relevant equality.

Thesis. *The program of higher inductive types can be realized in cubical type theory, providing quotient types appropriate for proof-relevant equality. Internal parametricity is a tool well-suited to analyzing these types.*

I propose to flesh out the theory and practice of higher inductive types in cubical type theory. On the theory side, I intend to expand the universe of precisely-understood higher inductive types, building on my work with Robert Harper on ordinary and indexed higher inductive types [22] to encompass higher inductive-inductive types. On the practice side, I plan to develop parametricity techniques for cubical type theory, with a focus on internal parametricity. To complete the current picture of internal parametricity, I intend to examine the degree to which results in type theory with parametricity primitives can be transferred to ordinary type theory, a question that is yet unstudied. I plan to implement this work as an extension to the cubical proof assistant `redtt` [61].

2 Background

2.1 Cubical type theory

Cubical type theory extends Martin-Löf type theory with a new form of equality called a *path*. I will first summarize the variant of cubical type theory presented by Angiuli et al. in [5], on which my work is based, then discuss alternatives. Like Martin-Löf type theory (as presented in [50]), cubical type theory is designed around four main judgments: A is a type, A and B are equal types, M is an element of A, and M and N are equal elements of A. In cubical type theory, however, each of these judgments is parameterized by a context $\Psi = (x_1, \ldots, x_n)$ of *dimension variables*.

$$A \ \mathsf{type}\ [\Psi] \qquad A = B \ \mathsf{type}\ [\Psi] \qquad M \in A\ [\Psi] \qquad M = N \in A\ [\Psi]$$

As usual, these have open equivalents parameterized by a context of ordinary variables Γ: we write $\Gamma \vdash A \ \mathsf{type}\ [\Psi]$ and so on. (Note that the dimension context is the outermost: the ordinary context, types, and terms above can all refer to variables in Ψ.) A dimension variable is intuitively thought of as varying in the unit interval $[0,1]$ of the real line. Accordingly, there are two *dimension constants* $0, 1$ which can be substituted for a dimension variable: if we have $A \ \mathsf{type}\ [x]$, we also have $A\langle 0/x\rangle \ \mathsf{type}\ [\cdot]$ and $A\langle 1/x\rangle \ \mathsf{type}\ [\cdot]$. Under the topological intuition, we think of A as a *path* between the types $A\langle 0/x\rangle$ and $A\langle 1/x\rangle$; logically, as a witness to the equality of $A\langle 0/x\rangle$ and $A\langle 1/x\rangle$.

$$\begin{tikzcd} 0 \arrow[dr, "x"] & & & A\langle 0/x\rangle \arrow[dr, "A"] & \\ & 1 & \longmapsto & & A\langle 1/x\rangle \end{tikzcd}$$

In the same way, we can speak of paths between elements within a given type. When a type or element depends on a context of n dimension variables, we think of it as an *n-dimensional cube* relating its various substitution instances. The higher-dimensional cubes make it possible, for example, to speak of paths between paths, that is, of proofs that two ways of equating a pair of terms are themselves equal. Semantically, a type is defined by explaining what its values are in each dimension context: a type is an assemblage of n-dimensional cubes. This can be seen as a higher-dimensional variation on setoid semantics, where a type is defined by a collection of elements and an equivalence relation upon them.

The ordinary type formers maintain their familiar rules uniformly in the dimension context. Function types, for example, satisfy the following (among other) rules.

$$\frac{a : A \vdash N \in B\ [\Psi]}{\lambda a.N \in (a{:}A) \to B\ [\Psi]} \qquad \frac{N \in (a{:}A) \to B\ [\Psi] \qquad M \in A\ [\Psi]}{N M \in B[M/a]\ [\Psi]}$$

The judgmental concept of path is internalized by a *path type*. These resemble function types which abstract over dimension variables, but the endpoints of the path are also fixed by the type. In the general case, they are *dependent* function types: the type can also depend on the dimension variable.

$$\frac{A\ \mathsf{type}\ [\Psi, x] \qquad P \in A\ [\Psi, x]}{\lambda^{\mathbb{I}} x.P \in \mathsf{Path}_{x.A}(P\langle 0/x\rangle, P\langle 1/x\rangle)\ [\Psi]}$$

In particular, for any A type $[\Psi]$ and element $M \in A$ $[\Psi]$, there is a reflexive path $\lambda^{\mathbb{I}}\_.A \in \mathsf{Path}_{\_.A}(M, M)$ $[\Psi]$ given by the constant function. (We will henceforth abbreviate $\mathsf{Path}_{\_.A}(M, N)$ as $\mathsf{Path}_A(M, N)$.) A path can be applied at any dimension term, which can be a dimension variable or one of the constants $0, 1$.

$$\frac{Q \in \mathsf{Path}_{x.A}(M, N)\ [\Psi] \qquad r \in \Psi \cup \{0, 1\}}{Q@r \in A\langle r/x\rangle\ [\Psi]}$$

Extensionality principles for path equality follow straightforwardly from the fact that all rules apply uniformly in the dimension context: a path of pairs has the form of a pair of paths, a path of functions the form of a function of paths. More explicitly, if we have functions $F_0, F_1 \in (a{:}A) \to B$ that are path-equal pointwise, as witnessed by a term $H \in (a{:}A) \to \mathsf{Path}_B(Fa, Ga)$, then by merely rearranging the order of binders we obtain $\lambda^{\mathbb{I}} x.\lambda a.Ha@x \in \mathsf{Path}_{(a:A)\to B}(F, G)$.

2.1.1 Kan operations

In and of itself, the addition of the dimension context merely endows each type with some kind of infinite-dimensional reflexive relation. To enforce that this is an *equality* relation, cubical type theory requires each type to support two operations: *coercion* and *(homogeneous) composition*. The former ensures that all constructions respect path equality, while the latter is a higher-dimensional generalization of algebraic laws such as symmetry and transitivity. These are inspired by the *Kan condition* of algebraic topology [40], and are therefore known as *(uniform) Kan operations.*

Coercion, or coe, takes a term that inhabits one point on a line of types and converts it into a term at any other point, as shown below. The second rule ensures that transporting a term from one point to the same point is a no-op.

$$\frac{A\ \mathsf{type}\ [\Psi, x] \qquad M \in A\ [\Psi]}{\mathsf{coe}^{r\rightsquigarrow s}_{x.A}(M) \in A\langle r'/x\rangle\ [\Psi]} \qquad \frac{A\ \mathsf{type}\ [\Psi, x] \qquad M \in A\ [\Psi]}{\mathsf{coe}^{r\rightsquigarrow r}_{x.A}(M) = M \in A\langle r/x\rangle\ [\Psi]}$$

These rules suffice to show that $\mathsf{coe}^{0\rightsquigarrow 1}_{x.A}(-)$ is an *equivalence*: a map that has a left and a right inverse, where the inverse conditions hold up to path equality [62, §4.3]. In other words, any path of types $x.A$ gives rise to an equivalence $A\langle 0/x\rangle \simeq A\langle 1/x\rangle$. Coercion computes by case analysis on $x.A$. For example, a coercion across a product type turns into a product of coercions in the component types, as shown below.

$$\mathsf{coe}^{r\rightsquigarrow s}_{x.A\times B}(M) \longmapsto \langle \mathsf{coe}^{r\rightsquigarrow s}_{x.A}(\mathsf{fst}(M)), \mathsf{coe}^{r\rightsquigarrow s}_{x.B}(\mathsf{snd}(M))\rangle$$

The homogeneous composition operator, called hcom, adjusts the *boundary* of a term by a collection of paths. This operation is used to explain how coe computes when the type argument is a path type. To see why it is necessary, consider the case of $\mathsf{coe}^{0\rightsquigarrow 1}_{y.\mathsf{Path}_{x.A}(M_0,M_1)}(-)$ where A does not depend on y; this function has the following type.

$$\mathsf{Path}_{x.A}(M_0\langle 0/y\rangle, M_1\langle 0/y\rangle) \to \mathsf{Path}_{x.A}(M_0\langle 1/y\rangle, M_1\langle 1/y\rangle)$$

In words, it takes a path P and adjusts the endpoints of P by the two paths $y.M_0$ and $y.M_1$. Pictorially, we have the following.

$$\begin{tikzcd} {} \arrow[r, "x"] \arrow[d, "y"'] & {} & \cdot \arrow[r, "P@x"] \arrow[d, "M_0"'] & \cdot \arrow[d, "M_1"] \\ {} & & \cdot \arrow[r, dashed, "\mathsf{coe}^{0\rightsquigarrow 1}_{y.\mathsf{Path}_{x.A}(M_0,M_1)}(P)@x"'] & \cdot \end{tikzcd}$$

Intuitively, the output of this operation should be "$(\lambda^{\mathbb{I}}y.M_0)^{-1} \cdot P \cdot (\lambda^{\mathbb{I}}y.M_1)$", where $^{-1}$ is some kind of inversion of paths and $\cdot$ some kind of concatenation: to get from $M_0\langle 1/y\rangle$ to $M_1\langle 1/y\rangle$, we follow M_0 backwards, then P forwards, and finally M_1 forwards. In order to define coercion at path type, every type must support this kind of operation; the hcom operation is introduced for that purpose. Per the above, we get symmetry and transitivity of paths as a special case. To begin with an example, the solution to the "composition problem" described above is provided by the following hcom term.

$$\mathsf{hcom}^{0\rightsquigarrow 1}_{A}(P@x; x = 0 \hookrightarrow y.M_0, x = 1 \hookrightarrow y.M_1)$$

Here, the *cap* $P@x$, which has type A, is adjusted on its $x = 0$ and $x = 1$ faces by the paths $y.M_0$ and $y.M_1$ (also of type A). Like coercion, composition is defined by cases on the type A. In order to implement *composition* at path type, a further generalization is necessary: hcom is permitted to take any number of adjustment faces as arguments. The general operator is written as $\mathsf{hcom}^{r\rightsquigarrow s}_{A}(M; \overrightarrow{\xi_i \hookrightarrow y.N_i})$, where each ξ_i is an equation on dimension terms and $y.N_i$ is its accompanying adjustment path. These adjustments are required to agree where their respective equations overlap. I refer to [5] for the typing rules for hcom; we will only need an intuitive understanding of its function.

In addition to Path types, Angiuli et al. [5] define coercion and composition: for all the standard type formers of Martin-Löf type theory: functions, products, basic inductive types such as bool, and universes. In addition, it is possible to define a type former (G in [15], Glue in [25], V in [5, 3]) which creates a path between two types from an equivalence between them, providing an inverse to the path-to-equivalence map provided by coercion. Cubical type theory therefore validates Voevodsky's univalence axiom [66], which asserts that the type $\mathsf{Path}_{\mathcal{U}}(A, B)$ of paths between elements A, B of the universe $\mathcal{U}$ is equivalent to the type $A \simeq B$ of equivalences between them. This is a powerful tool: it gives the programmer access to the fact that all constructions in type theory are invariant under equivalence.

2.1.2 Constructivity and canonicity

Cubical type theory originally grew out of attempts to give a constructive semantics for *homotopy type theory* (HoTT) [62], an axiomatic extension of Martin-Löf type theory with the univalence axiom and certain higher inductive types (to be discussed momentarily). Bezem, Coquand, and Huber gave the first candidate constructive model in the category of cubical sets [15].[1] If one squints, this model can be read type-theoretically as an extension of Martin-Löf type theory with *substructural* (specifically, affine) dimension variables. Later work shifted to using structural dimension variables, which are simpler to present type-theoretically and appeared necessary to treat higher inductive types. Cohen, Coquand, Huber, and Mörtberg presented a cubical type theory with structural dimension terms carrying a De Morgan algebra structure: minimum, maximum, and negation operations on the interval [25]. This theory came with the first explicit proof of univalence in a constructive model. Angiuli, Favonia, and Harper developed a second univalent cubical type theory, this one based on structural dimension variables but without the additional De Morgan structure [5]. I take the latter, known as *cartesian cubical type theory*, as the basis of my proposal.

The constructivity of cubical type theory can be expressed—in one way, at least—by the following *canonicity* theorem, here formulated in terms of a type of booleans.

Proposition. *There is an algorithm that, for any closed* $M \in \mathsf{bool}\ [\Psi]$*, produces evidence either that* $M = \mathsf{true} \in \mathsf{bool}\ [\Psi]$ *or that* $M = \mathsf{false} \in \mathsf{bool}\ [\Psi]$*.*

In other words, any boolean term can be *evaluated* either to true or to false. In proving boolean canonicity, one in fact establishes a collection of canonical forms of each type: for bool, these are true and false, while for functions they are λ-abstractions, and so on. Canonicity for the Cohen et al. theory was established by Huber [37]; in the Angiuli et al. theory, it is built into the definitions of the types in the style of [50].

The approach of Angiuli et al. is to define the judgments of type theory—type and term equality—in terms of computation, so that the closed elements of bool are *by definition* those for which (in a certain sense) the canonicity property holds. This computational semantics of the judgments can serve as a canonicity proof for any collection of inference rules it models, as long as the equality of the formal system is closed under reduction in the computation system.

The starting point for a computational semantics is an untyped programming language, together with a specification of *canonical types* and *canonical elements*. In cubical type theory, the main complication is that one must consider the canonical values at each context Ψ of dimension variables. The specification of canonical types and elements is given by the following data.

[1]That this model indeed validates univalence was later shown in [17]; whether it supports higher inductive types remains unknown.

Definition. A *candidate cubical type system* is a four-place relation $\tau(\Psi, V_0, V_1, \varphi)$ ranging over dimension contexts Ψ, values V_0, V_1 with dimension variables limited to Ψ, and binary relations φ which themselves range over values W_0, W_1 with variables limited to Ψ.

We read an instance $\tau(\Psi, V_0, V_1, \varphi)$ of this relation as saying that V_0 and V_1 are equal canonical types in context Ψ, and that the equal canonical values of these types in context Ψ are given by the relation φ. Two terms A_0, A_1 (not necessarily values) are said to be *equal types* when they "coherently evaluate" to equal canonical types. In brief, this condition requires that not only that A_0 and A_1 evaluate to equal canonical types, but that this property is preserved when that evaluation is interleaved with dimension substitutions. Similarly, two terms in a type are equal when they coherently evaluate to equal canonical elements of that type.

2.2 Higher inductive types

Higher inductive types, also originally introduced in the context of HoTT, play the role of quotient types for the proof-relevant equality represented by paths. The central idea is that generating equalities can be treated as higher-dimensional constructors of an inductive definition. As a contrived example, consider a definition of the integers that takes two copies of the natural numbers, one for positive and one for negative integers, and identifies the negative and positive zero. In cubical type theory, such a definition might be written in the following form.

$$\begin{array}{l}
\mathsf{data\ int\ where} \\
\mid \mathsf{pos}(n : \mathsf{nat}) \in \mathsf{int} \\
\mid \mathsf{neg}(n : \mathsf{nat}) \in \mathsf{int} \\
\mid \mathsf{seg}(x : \mathbb{I}) \in \mathsf{int}\ [x = 0 \hookrightarrow \mathsf{neg}(\mathsf{zero}) \mid x = 1 \hookrightarrow \mathsf{pos}(\mathsf{zero})]
\end{array}$$

The equation between $\mathsf{neg}(\mathsf{zero})$ and $\mathsf{pos}(\mathsf{zero})$ is prescribed by a third constructor seg, which takes a dimension variable x as a parameter. Unlike an ordinary constructor, the seg constructor also specifies a *boundary*: its $x = 0$ face is attached to $\mathsf{neg}(\mathsf{zero})$ and its $x = 1$ face to $\mathsf{pos}(\mathsf{zero})$. We can visualize the resulting type as shown below.

$$\begin{tikzcd}
\mathsf{neg}(2) & \mathsf{neg}(1) & \mathsf{neg}(0) \arrow[d, "\mathsf{seg}(x)"] & \downarrow x & \\
& & \mathsf{pos}(0) & \mathsf{pos}(1) & \mathsf{pos}(2)
\end{tikzcd}$$

This is the basic pattern of a higher inductive type used as an ordinary quotient. That the type is inductively generated is expressed by an elimination principle, which I will return to momentarily. Generalizing the situation above, we can write down a general operator that quotients a type A by a binary relation $R \in A \times A \to \mathsf{type}$ as a *parameterized higher inductive type.*

$A : \mathsf{type}, R : A \to A \to \mathsf{type} \vdash \mathsf{data\ quo\ where}$
$\mid \mathsf{pt}(a : A) \in \mathsf{quo}$
$\mid \mathsf{rel}(a_0 : A, a_1 : A, u : R\langle a_0, a_1\rangle, x : \mathbb{I}) \in \mathsf{quo}\ [x = 0 \hookrightarrow \mathsf{pt}(a_0) \mid x = 1 \hookrightarrow \mathsf{pt}(a_1)]$

Here, we are constructing a path $\mathsf{rel}(a_0, a_1, u, i)$ between any pair of elements $a_0, a_1 : A$ for which there is some $u : R\langle a_0, a_1\rangle$.

However, this apparently general example does not completely satisfy our desire for quotients in the setting of proof-relevant equality. To see why, consider the following common use case: quotienting a type by the total relation to make all its elements indistinguishable. The type constructor $\langle - \rangle$ defined by $\langle A\rangle := \mathsf{quo}(A, \lambda\_.\mathsf{unit})$, where unit is the type with a single element $*$, would seem to serve this purpose. Consider, however, the case of $\langle \mathsf{bool}\rangle$. We can sketch this type as the following picture, where arrows indicate paths created by the rel constructor.

$$\begin{tikzcd} \mathsf{pt}(\mathsf{true}) \arrow[loop left] \arrow[r, bend left] & \mathsf{pt}(\mathsf{false}) \arrow[loop right] \arrow[l, bend left] \end{tikzcd}$$

This type has non-trivial path structure; for example, one can prove that the left loop $\lambda^{\mathbb{I}}x.\mathsf{rel}(\mathsf{true}, \mathsf{true}, *, x)$ is distinct from the reflexive loop $\lambda^{\mathbb{I}}x.\mathsf{true}$. Although every pair of *zero-dimensional* elements is connected by a path, this is not true for the one-dimensional elements: we can prove $(a, b{:}A) \to \mathsf{Path}_{\langle A\rangle}(\mathsf{pt}(a), \mathsf{pt}(b))$, but not $(c, d{:}\langle A\rangle) \to \mathsf{Path}_{\langle A\rangle}(c, d)$.

To get the behavior we are after, a more sophisticated form of higher inductive type is required[2]: one with a *recursive* path constructor. The following type operator is called the *truncation* or (-1)-*truncation* [62, §3.7].

$A : \mathsf{type} \vdash \mathsf{data\ trunc\ where}$
$\mid \mathsf{pt}(a : A) \in \mathsf{trunc}$
$\mid \mathsf{squash}(c : \mathsf{trunc}, d : \mathsf{trunc}, x : \mathbb{I}) \in \mathsf{trunc}\ [x = 0 \hookrightarrow c \mid x = 1 \hookrightarrow d]$

Unlike the rel constructor of the quotient type, which only creates paths between pt terms, the squash constructor of the truncations adds a new path between any pair of elements *in the type being constructed*. Because these constructors are uniform in dimension context, they can themselves be applied to paths to identify them. It can be shown that $\mathsf{trunc}(A) \simeq \mathsf{unit}$ for any inhabited A, so this operator indeed "collapses" a type's structure.

As a more exotic example, consider the *torus*, a simple surface that appears in algebraic topology. It is notable for us because it involves a *two-dimensional* or *square*

[2]Strictly speaking, this is not true: a type constructor with the desired properties can be encoded with some difficulty using quo and a natural number type [46, 63]. But such encodings are inefficient and satisfy fewer useful computation rules, and there are limits to their expressivity [48, §9].

constructor. To describe it concisely, we borrow some convenient notation from the `redtt` cubical proof assistant: we group faces with the same boundary together, and use $\partial[x_1, \ldots, x_n]$ to stand for the set of faces $\{(x_i = \varepsilon) \mid 1 \leq i \leq n, \varepsilon \in \{0,1\}\}$.

```
data torus where
| base ∈ torus
| loopa($x : \mathbb{I}$) ∈ torus [$\partial[x] \hookrightarrow$ base]
| loopb($y : \mathbb{I}$) ∈ torus [$\partial[y] \hookrightarrow$ base]
| surf($x : \mathbb{I}, y : \mathbb{I}$) ∈ torus [$\partial[x] \hookrightarrow$ loopb($y$), $\partial[y] \hookrightarrow$ loopa($x$)]
```

This data declaration, which describes a cellular construction of the torus familiar to topologists, can be visualized as follows.

$$\begin{tikzcd}
\cdot \arrow[r, "x"] \arrow[d, "y"'] & {} & \mathsf{base} \arrow[rr, "\mathsf{loopa}(x)"] \arrow[dd, "\mathsf{loopb}(y)"'] & & \mathsf{base} \arrow[dd, "\mathsf{loopb}(y)"] \\
{} & & & \mathsf{surf}(x,y) & \\
& & \mathsf{base} \arrow[rr, "\mathsf{loopa}(x)"'] & & \mathsf{base}
\end{tikzcd}$$

These simple examples demonstrate the essential features desirable in a general schema for higher inductive types: higher-dimensional constructors with specified boundaries that can mention previously declared constructors as well as recursive arguments. In the " HoTT Book" [62], many more examples are presented—including some based on indexed inductive and inductive-inductive types—but a general framework is only given as a sketch (in §6.19). Nor are they included in Voevodsky's initial (non-constructive) model of univalent type theory in simplicial sets [43]. Only gradually has the theory caught up with the variety of higher inductive types used in the informal type theory of the HoTT Book and (via postulates) in formalization projects [19, 8, 65].

Robert Harper and I have developed a schema for higher inductive types in (univalent) cubical type theory and proven canonicity by way of a computational semantics [20, 22]. We designed the schema to be fairly simple while including most examples in common use, such as those described above. The schema also includes indexed inductive types, which are non-trivial to implement in higher-dimensional type theory for much the same reason that higher constructors are. In the remainder of this section, I summarize our schema and its semantics. As part of my proposed work, which I describe in Section 3.1, I plan to extend the theory with inductive-inductive types, as well as more mundane features that we originally excluded for sake of expedience.

2.2.1 A schema for indexed higher inductive types

In this section, I summarize the schema and computational semantics that I developed with Robert Harper in [22]. In short, a declaration of an indexed higher inductive type takes the following form.

$$\begin{array}{l} \mathsf{data}\ \mathsf{X}(\Delta)\ \mathsf{where} \\ \mid \cdots \\ \mid \mathsf{intro}_\ell(\gamma : \Gamma, \theta : \Theta, \overline{x}) \in \mathsf{X}(\overline{I})\ [\overrightarrow{\xi_i \hookrightarrow \mathrm{M}_i}] \\ \mid \cdots \end{array}$$

An *indexed inductive type* is a family of types that is simultaneously generated by constructors that introduce terms at specific indices [29, 31]; here, the declared family is indexed by a variable context Δ, and the constructor labelled ℓ constructs an element at index $\overline{I} \in \Delta$ (which may depend on the arguments to the constructor). Indices are distinct from the *parameters* shown before a turnstile $\vdash$ in the above examples; an inductive type is defined uniformly in its parameters, so we will leave them implicit here.

The arguments to each constructor are divided into three groups: *non-recursive arguments*, *recursive arguments*, and *dimensions*. The first of these is specified by a context (i.e., a telescope) $\gamma : \Gamma$ of ordinary types. The second is specified by an *argument context* $\theta : \Theta$. This is a list of *argument types* A, which are drawn from a small type theory on the following grammar.

$$\mathrm{A} ::= \mathsf{X}(\overline{I}) \mid (a{:}A) \to \mathrm{A}$$

In words, an argument type can either be some index of the family being defined, or a function from some existing type into an argument type. This grammar enforces the familiar *strict positivity condition* on the recursive arguments to an inductive constructor. Finally, we have the novel list $\overline{x}$ of dimension arguments; a constructor with n dimension arguments defines an n-cube in the type.

The major addition to the ordinary shape of an inductive declaration is the list $\overrightarrow{\xi_i \hookrightarrow \mathrm{M}_i}$ of *boundary terms*, which specify how a higher-dimensional constructor should reduce when particular equations on its dimension arguments hold. Each ξ_i is a *constraint*: an equation $r = s$, where r, s can each be a variable in $\overline{x}$ or one of the constants $0, 1$. Each constraint has a corresponding *argument term*, which may mention the variables in γ and θ. Argument terms are the elements that inhabit argument types and are drawn from the following grammar.

$$\mathrm{M} ::= \mathsf{intro}_\ell(\overline{M}, \overline{\mathrm{M}}, \overline{r}) \mid \mathsf{hcom}^{r \rightsquigarrow s}_{\overline{I}}(\mathrm{M}; \overrightarrow{\xi_i \hookrightarrow x.\mathrm{M}}) \mid \mathsf{coe}^{r \rightsquigarrow s}_{x.\overline{I}}(\mathrm{M}) \mid \lambda a.\mathrm{M} \mid \mathrm{M}M$$

An argument term can thus be a constructor term (which must be declared before the constructor being specified), a composition term in some index of the type being defined, a coercion between indices, or a function abstraction or application. The type theory of argument types and terms consists of two judgments $\Delta \rhd \mathrm{A}\ \mathsf{atype}\ [\Psi]$

and $\Delta \triangleright \mathcal{K}; \Theta \vdash \text{M} : \text{A}\ [\Psi]$ and their equational equivalents; both are parameterized by indexing context Δ, while the argument terms are also parameterized by a telescope $\mathcal{K}$ of the preceding constructor declarations and an argument context Θ.

Each inductive type declaration gives rise to a corresponding elimination principle. As an example, the quotient type $\mathsf{quo}(A, R)$ will satisfy the following rule.

$$\frac{\begin{array}{c} d : \mathsf{quo}(A,R) \vdash C\ \mathsf{type}\ [\Psi] \qquad M \in \mathsf{quo}(A,R)\ [\Psi] \\ a : A \vdash P \in C[\mathsf{pt}(a)/d]\ [\Psi] \qquad a_0, a_1 : A, u : Rab \vdash T \in C[\mathsf{rel}(a_0,a_1,u,x)/d]\ [\Psi, x] \\ a_0, a_1 : A, u : Rab \vdash T\langle 0/x\rangle = P[a_0/a] \in C[\mathsf{pt}(a_0)/d]\ [\Psi] \\ a_0, a_1 : A, u : Rab \vdash T\langle 1/x\rangle = P[a_1/a] \in C[\mathsf{pt}(a_1)/d]\ [\Psi] \end{array}}{\mathsf{quo\text{-}elim}_{d.C}(M; a.P, a.b.u.x.T) \in C[M/d]\ [\Psi]}$$

The first four premises resemble those of an ordinary inductive type: the eliminator takes a motive $d.C$, a term M in the inductive type, and two cases $a.P$ and $a.b.u.x.T$ explaining how to behave on pt and rel constructor values. The final two premises ensure that the eliminator respects the equations imposed on the rel constructor by its boundary conditions: the behavior of the eliminator applied to $\mathsf{rel}(a_0, a_1, u, 0)$ should match its behavior on $\mathsf{pt}(a_0)$, because the former reduces to the latter. The generation of these boundary conditions is the most complicated component of the schema—it requires describing the action of the eliminator on each form of boundary term—so I refer to [20] for the general case.

2.2.2 Canonicity for higher inductive types

A central goal of our work is to give a computational interpretation of higher inductive types, which for us means giving an operational semantics that realizes a canonicity theorem. Recall that to prove canonicity for any particular type, it is necessary to assign a collection of canonical values at *every* type. Thus, the central conceptual problem is to determine the canonical values of an inductive type. For an ordinary inductive type, this is fairly simple: the canonical values are the constructor terms. However, the situation is more complicated for higher inductive types because of the need to support the Kan operations.

Consider, for example, the case of the quotient $\mathsf{quo}(A, R)$. If we have four points $M, N, P, Q \in A$ which are related to each other by terms $U \in R\langle M, P\rangle$, $V \in R\langle M, N\rangle$, and $W \in R\langle N, Q\rangle$, we can assemble the corresponding rel terms into the following composition problem.

$$\begin{tikzcd}
{} \arrow[r, "x"] \arrow[d, "y"'] & {} & \mathsf{pt}(M) \arrow[rr, "\mathsf{rel}(M,N,V,x)"] \arrow[dd, "\mathsf{rel}(M,P,U,y)"'] & & \mathsf{pt}(N) \arrow[dd, "\mathsf{rel}(N,Q,W,y)"] \\
{} & & & & \\
& & \mathsf{pt}(P) \arrow[rr, dashed] & & \mathsf{pt}(Q)
\end{tikzcd}$$

By feeding the upper horseshoe into hcom at the inductive type, we obtain a term in x connecting the bottom two pt terms. For canonicity to hold, this term must reduce to a canonical value of the inductive type with the same boundary. Unless R happens to be symmetric and transitive, however, there is no reason for a such a term to exist among the constructor terms of $\mathsf{quo}(A, R)$.

Intuitively, an inductive type is generated not only by its constructor terms, but also by the Kan operations. With this in mind, we declare the values of an inductive type to consist not only of constructor terms but also *composition values*, that is, hcom terms.[3] A side effect of doing this is that we must explain how the *eliminator* for an inductive type reduces when applied to a composition value. The solution, modulo technicalities, is to send composition values in the inductive type to compositions in the target type. Intuitively, the inductive type has a freely generated composition structure, so to define into any other type with a composition structure, it suffices to explain where to send the constructors.

By contrast, *coercion* in ordinary higher inductive types can be implemented without introducing new values; in brief, a coercion applied to a constructor or composition value pushes inside its arguments. For indexed inductive types, however, coercion values *are* required—even when the type has no higher constructors.

The simplest non-trivial example of an indexed inductive type is Martin-Löf's identity type. Given a type A, the identity type family $\mathsf{Id}_A(-,-)$ is parameterized by a pair of elements of A and has a single constructor which inhabits the reflexive instances of the relation.

$$\begin{array}{l} A : \mathsf{type} \vdash \mathsf{data}\ \mathsf{Id}(a_0 : A, a_1 : A)\ \mathsf{where} \\ \mid \mathsf{refl}(a : A) \in \mathsf{Id}(a, a) \end{array}$$

To see why coercion values are necessary to implement this type in cubical type theory, suppose we have a path $P \in \mathsf{Path}_A(M, N)$. Using coercion, we can turn the reflexive identification at M into an identification between M and N.

$$\mathsf{coe}^{0 \rightsquigarrow 1}_{x.\mathsf{Id}_A(M, P@x)}(\mathsf{refl}(M)) \in \mathsf{Id}_A(M, N)$$

Unless M and N happen to be exactly equal, there is no constructor value to which this coercion can reduce. Again, our solution is to introduce a kind of coercion value. More specifically, we add values for coercion *between the indices* of the inductive type.[4]

$$\frac{A\ \mathsf{type}\ [\Psi] \qquad M_0, M_1 \in A\ [\Psi, x] \qquad P \in \mathsf{Id}_A(M_0\langle r/x\rangle, M_1\langle r/x\rangle)\ [\Psi]}{\mathsf{fcoe}^{r \rightsquigarrow s}_{x.(M_0, M_1)}(P) \in \mathsf{Id}_A(M_0\langle s/x\rangle, M_1\langle s/x\rangle)\ [\Psi]}$$

[3]A simple restriction on the shape of compositions, introduced in [5, Definition 12], can ensure that there are no composition values in an empty dimension context. Thus, for example, a zero-dimensional computation of type int will always evaluate to a pos or neg term, never a composition.

[4]We cannot simply introduce values for *all* coercions, for predicativity reasons: to do so, we would need to quantify over all type lines $x.B$ such that $B\langle s/x\rangle$ is an instance of the inductive type, but such lines can be "larger" than the inductive type itself.

The values of an indexed inductive type thus consist of constructor values, fcoe values, and composition values (again necessary even without higher constructors, because non-trivial composition problems can be formed using coercion values). It is then possible to implement general coercion using a combination of fcoe values, composition values, and coercion in the input type A. As with composition values in ordinary inductive types, the eliminator for an indexed inductive type takes coercion values to coercions in the target type.

2.2.3 Related work

With regard to the design of schemata, the earliest work is by Basold, Geuvers, and van der Weide [7] and Dybjer and Moeneclaey [33], who design formats for types with 1-dimensional and 2-dimensional constructors respectively. A more comprehensive format is developed by Kaposi and Kovács [41]; it includes indexed and inductive-inductive types as well as general n-dimensional constructors. (However, because higher constructors are described as elements of iterated Martin-Löf identity types—there is no native notion of "square"—reasoning with constructors above dimension one rapidly becomes infeasibly complicated.) None of these address questions of canonicity. Dybjer and Moeneclaey give a semantics for their theory in the groupoid model of Hofmann and Streicher [35]. Kaposi, Kovács, and Altenkirch separately develop a semantics for a schema in a setting with *uniqueness of identity proofs* (UIP), the principle that any pair of paths are themselves connected by a path [42]. These models exhibit 1- and 0-dimensional structure respectively; they are incompatible with an infinite hierarchy of univalent universes.

The first systematic work on infinite-dimensional models of higher inductive types is due to Lumsdaine and Shulman [48]. Their approach is purely semantic: rather than giving a syntactic schema, they develop a semantic notion of *cell monad* for simplicial model categories and show it can be used to interpret specific examples of syntactically specified higher inductive types. Again, this work is not concerned with constructivity, for which simplicial sets are known to be problematic [16]. Also, they are unable to model universes closed under parameterized higher inductive types such as the quotient or truncation.

To develop a univalent type theory with higher inductive types and a canonicity property, cubical type theory is the most promising route. The initial papers by Cohen et al. and Angiuli et al. include simple examples of higher inductive types, but not a schema. More recently, Coquand, Huber, and Mörtberg have worked out further examples in detail and sketched a schema, accompanied by a semantics in cubical sets [30].

2.3 Internal parametricity

With a system for higher inductive types in place, we can begin to study their properties. This is not such a simple task; study of the *spheres*, a particular simple sequence of higher inductive types, forms the notoriously complex backbone of homotopy theory. However, even some questions that are classically straightforward are inordinately difficult to answer in higher-dimensional type theory, typically thanks to a lack of recourse to an internal "strict" (i.e., proof-irrelevant) equality. As an example, take the aforementioned *smash product*. The smash product $- \wedge -$ is a binary operator on *pointed types* $A, B \in \mathcal{U}_* := (X{:}\mathcal{U}) \times X$ that can be defined in type theory as a higher inductive type [62, §6.8]. While it is not difficult to prove that this operator is commutative (up to equivalence of types), showing that it is associative is a challenge. Showing that the commutator $c_{X,Y} \in X \wedge Y \simeq Y \wedge X$ and associator $a_{X,Y,Z} \in (X \wedge Y) \wedge Z \simeq X \wedge (Y \wedge Z)$ interact appropriately is even more difficult [64, 18].

The field of programming languages suggests a productive line of attack: *parametricity*. Originally developed by Reynolds [56], parametricity studies the uniformity principles automatically satisfied by polymorphic functions expressible in type theory. In brief, Reynolds' *abstraction theorem* states that the set-theoretic denotation of any polymorphic function in the simply-typed λ-calculus acts on relations. For example, consider a term $F \in X \to X$ defined over a type variable X. Reynolds concludes that for any two sets A, B, relation $R \subseteq A \times B$, and elements $(a, b) \in R$, we have $(\llbracket F \rrbracket_A(a), \llbracket F \rrbracket_B(b)) \in R$, where $\llbracket F \rrbracket_C$ is the denotation of F when X is interpreted as C. The content of the proof of Reynolds' abstraction theorem is a construction of a relational interpretation of simple type theory.

The uniformity principle provided by parametricity is quite powerful. For any $F \in X \to X$ as above, we can use its action on relations to show that $\llbracket F \rrbracket = \llbracket \lambda a.a \rrbracket$: F must be the polymorphic identity function. As a more interesting example, for any $F \in X \times Y \to X \times Y$, we can again show that $\llbracket F \rrbracket = \llbracket \lambda p.p \rrbracket$, which implies in particular that any pair of functions $X \times Y \leftrightarrows Y \times X$ "automatically" constitutes an isomorphism. For ordinary product types, it is straightforward enough to define such an isomorphism directly; for the smash product, on the other hand, this kind of principle could drastically simplify existing proofs.

Only recently has Reynolds' work been extended to dependent type theory [13, 10, 45, 47, 6]. A notable change from the simply-typed setting is that, thanks to the expressivity of dependent types, the consequences of parametricity can be stated *within* the dependent type theory under study. As shown by Bernardy, Jansson, and Paterson, one can define an (external) operator that generates a proof of the parametricity property for a given type. The internalization process can go even further: Bernardy and Moulin show how the operator computing the relational interpretation of a type can be internalized as a type constructor [11, 14]. In further work, they simplify this extended theory by use of a technique familiar to us: dimension variables. In fact, the demands of internal parametricity are remarkably similar to those of univalence. Univalence means in particular that all constructions act on equivalences, in the same way that parametricity expresses that all constructions act on relations. Indeed, both principles serve to make a uniformity property implicit in type theory available to the programmer.

Robert Harper and I have designed an internally parametric extension to cubical type theory [21]. In Section 3.2, I propose to prove meta-theoretic results connecting this theory to ordinary cubical type theory and to apply it to prove results in cubical type theory such as those described above.

2.3.1 The structure of internally parametric type theory

I will present an outline of internal parametricity as an extension to cubical type theory in particular, following my work with Robert Harper. Aside from the cubical additions and choice of notation, the theory presented in this section is due to Bernardy et al. [14]. As in cubical type theory, the first move is to extend the basic judgments with a context of dimension variables $\Phi = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$; we call these *bridge variables* to distinguish from the existing *path variable* context Ψ, following the terminology of Nuyts et al. [55].

$$A = B \ \mathsf{type}\ [\Phi \mid \Psi] \qquad M = N \in A\ [\Phi \mid \Psi]$$

Where we think of a type A varying in a path dimension x as representing an equivalence between its endpoints $A\langle 0/x\rangle$ and $A\langle 1/x\rangle$, we think of A varying in a bridge dimension $\boldsymbol{x}$ as standing for a *relation* on $A\langle \mathbf{0}/\boldsymbol{x}\rangle$ and $A\langle \mathbf{1}/\boldsymbol{x}\rangle$. In contrast to the path context, it is apparently essential that the bridge context behave *substructurally.* This manifests in a restriction of substitution to *fresh* variables: we can only substitute $\boldsymbol{r}$ for $\boldsymbol{x}$ in M when $\boldsymbol{r}$ is not a variable already occurring in M.

$$\frac{\boldsymbol{r} \in \Phi \cup \{\mathbf{0}, \mathbf{1}\} \qquad M \in A\ [\Phi^{\setminus \boldsymbol{r}}, \boldsymbol{x} \mid \Psi]}{M\langle \boldsymbol{r}/\boldsymbol{x}\rangle \in A\ [\Phi \mid \Psi]}$$

In the above rule, $\Phi^{\setminus \boldsymbol{r}}$ is the context obtained by removing $\boldsymbol{r}$ from Φ if it is a variable (and doing nothing if $\boldsymbol{r}$ is $\mathbf{0}$ or $\mathbf{1}$). We will see in a moment why this restriction is key. Just as path types internalize dependence on path variables, we introduce *bridge types* to internalize dependence on variables.

$$\frac{A\ \mathsf{type}\ [\Phi, \boldsymbol{x} \mid \Psi] \qquad P \in A\ [\Phi, \boldsymbol{x} \mid \Psi]}{\lambda^{\mathbf{2}}\boldsymbol{x}.P \in \mathsf{Bridge}_{\boldsymbol{x}.A}(P\langle \mathbf{0}/\boldsymbol{x}\rangle, P\langle \mathbf{1}/\boldsymbol{x}\rangle)\ [\Phi \mid \Psi]}$$

These obey similar rules to those for path types, except that bridges can only be applied at fresh variables.

$$\frac{\boldsymbol{r} \in \Phi \cup \{\mathbf{0}, \mathbf{1}\} \qquad N \in \mathsf{Bridge}_{\boldsymbol{x}.A}(M_0, M_1)\ [\Phi^{\setminus \boldsymbol{r}} \mid \Psi]}{N @ \boldsymbol{r} \in A\langle \boldsymbol{r}/\boldsymbol{x}\rangle\ [\Phi \mid \Psi]}$$

The Bridge type former implements one direction of our desired correspondence between type lines and relations: given $A\ \mathsf{type}\ [\Phi, \boldsymbol{x} \mid \Psi]$, its associated relation is the family $a_0 : A\langle \mathbf{0}/\boldsymbol{x}\rangle, a_1 : A\langle \mathbf{1}/\boldsymbol{x}\rangle \vdash \mathsf{Bridge}_{\boldsymbol{x}.A}(a_0, a_1)\ \mathsf{type}\ [\Psi]$. Note that this is a *type-valued* relation. Just as cubical type theory is a theory of proof-relevant equality, so internally parametric type theory is a theory of proof-relevant relations.

The substructurality of bridge variables comes into play when we consider bridges at compound types. We expect these to match the definitions of the relational interpretation in ordinary parametricity. For example, a bridge over a product type $\boldsymbol{x}.A \times B$ ought to correspond to a pair of bridges over $\boldsymbol{x}.A$ and $\boldsymbol{x}.B$ respectively.

$$\mathsf{Bridge}_{\boldsymbol{x}.A\times B}(M_0, M_1) \simeq \mathsf{Bridge}_{\boldsymbol{x}.A}(\mathsf{fst}(M_0), \mathsf{fst}(M_1)) \times \mathsf{Bridge}_{\boldsymbol{x}.B}(\mathsf{snd}(M_0), \mathsf{snd}(M_1))$$

The same principle holds in cubical type theory if we replace bridges with paths, and we can prove it in the same way we would for paths. In the forward direction, we send P to $\langle \lambda^{\mathbf{2}}\boldsymbol{x}.\mathsf{fst}(P@x), \lambda^{\mathbf{2}}\boldsymbol{x}.\mathsf{snd}(P@x)\rangle$; in the backward direction, we send Q to $\lambda^{\mathbf{2}}\boldsymbol{x}.\langle \mathsf{fst}(Q)@\boldsymbol{x}, \mathsf{snd}(Q)@\boldsymbol{x}\rangle$. To reiterate, this "extensionality" principle follows from the fact that the rules for product introduction and elimination apply in all dimension contexts.

Function types, on the other hand, are more delicate. In the cubical type theory I presented in Section 2.1, we have following principle when x does not occur in A.

$$\mathsf{Path}_{x.A\to B}(F_0, F_1) \simeq (a{:}A) \to \mathsf{Path}_{x.B}(F_0 a, F_1 a)$$

This is what we want for paths: equality of functions is pointwise equality. Like products, it is proven by shifting dimension binders past constructors: the function from right to left takes H to $\lambda^{\mathbb{I}}x.\lambda a.Ha@x$. In this case, however, the argument does not translate into a substructural type theory: in the expression $\lambda^{\mathbb{I}}x.\lambda a.Ha@x$, the variable a is introduced when x is already in scope, so x is not fresh for Ha, and therefore $Ha@x$ is ill-formed in a substructural theory. This is a feature, not a bug: for bridges, the principle we need is not the equivalence above but the following.

$$\begin{gathered}\mathsf{Bridge}_{\boldsymbol{x}.A\to B}(F_0, F_1)\\ \simeq \\ (a_0{:}A\langle \mathbf{0}/\boldsymbol{x}\rangle)(a_1{:}A\langle \mathbf{1}/\boldsymbol{x}\rangle) \to \mathsf{Bridge}_{\boldsymbol{x}.A}(a_0, a_1) \to \mathsf{Bridge}_{\boldsymbol{x}.B}(F_0 a_0, F_1 a_1)\end{gathered}$$

That is, a bridge over $\boldsymbol{x}.A \to B$ ought to correspond to a function taking bridges over $\boldsymbol{x}.A$ to bridges over $\boldsymbol{x}.B$. In fact, it is precisely because bridge dimensions are substructural that this principle holds. Intuitively, the idea is that we can take Q in the latter type to "$\lambda^{\mathbf{2}}\boldsymbol{x}.\lambda a.Q(a\langle \mathbf{0}/\boldsymbol{x}\rangle)(a\langle \mathbf{1}/\boldsymbol{x}\rangle)(\lambda^{\mathbf{2}}\boldsymbol{x}.a)@\boldsymbol{x}$" in the former, capturing the "occurrences" of $\boldsymbol{x}$ in a to extract a bridge over $\boldsymbol{x}.A$ and then applying Q to obtain a bridge over $\boldsymbol{x}.B$. The technical details are more involved—in particular, the capture must be delayed until a suitably concrete term is substituted for a—but the essential point is that variable capture only commutes with dimension substitution when dimension variables are substructural.

2.3.2 The relativity principle

In the same way that univalence is the centerpiece of cubical type theory, so the bedrock of parametric type theory is a characterization of bridges in the universe of

Formation

$$\frac{\boldsymbol{r} \in \Phi \cup \{\mathbf{0},\mathbf{1}\} \qquad A, B \ \mathsf{type}\ [\Phi^{\backslash \boldsymbol{r}} \mid \Psi] \qquad R \in A \times B \to \mathcal{U}\ [\Phi^{\backslash \boldsymbol{r}} \mid \Psi]}{\mathsf{Gel}_{\boldsymbol{r}}(A, B, R)\ \mathsf{type}\ [\Phi \mid \Psi]}$$

$$\mathsf{Gel}_{\mathbf{0}}(A, B, R) = A \qquad \mathsf{Gel}_{\mathbf{1}}(A, B, R) = B$$

Introduction

$$\frac{\boldsymbol{r} \in \Phi \cup \{\mathbf{0},\mathbf{1}\} \qquad M \in A\ [\Phi^{\backslash \boldsymbol{r}} \mid \Psi] \qquad N \in B\ [\Phi^{\backslash \boldsymbol{r}} \mid \Psi] \qquad P \in R\langle M, N\rangle\ [\Phi^{\backslash \boldsymbol{r}} \mid \Psi]}{\mathsf{gel}_{\boldsymbol{r}}(M, N, P) \in \mathsf{Gel}_{\boldsymbol{r}}(A, B, R)\ [\Phi \mid \Psi]}$$

$$\mathsf{gel}_{\mathbf{0}}(M, N, P) = M \in A \qquad \mathsf{gel}_{\mathbf{1}}(M, N, P) = N \in B$$

Elimination

$$\frac{Q \in \mathsf{Gel}_{\boldsymbol{x}}(A, B, R)\ [\Phi, \boldsymbol{x} \mid \Psi]}{\mathsf{ungel}(\boldsymbol{x}.Q) \in R\langle Q\langle \mathbf{0}/\boldsymbol{x}\rangle, Q\langle \mathbf{1}/\boldsymbol{x}\rangle\rangle\ [\Phi \mid \Psi]}$$

$$\mathsf{ungel}(\boldsymbol{x}.\mathsf{gel}_{\boldsymbol{x}}(M, N, P)) = P \in R\langle M, N\rangle$$

$$Q\langle \boldsymbol{r}/\boldsymbol{x}\rangle = \mathsf{gel}_{\boldsymbol{x}}(Q\langle \mathbf{0}/\boldsymbol{x}\rangle, Q\langle \mathbf{1}/\boldsymbol{x}\rangle, \boldsymbol{x}.\mathsf{ungel}(\boldsymbol{x}.Q)) \in \mathsf{Gel}_{\boldsymbol{r}}(A, B, R)$$

Figure 1: Rules for Gel types. For notational simplicity, I have assumed that the relation is "small" ($\mathcal{U}$-valued); the general definition takes $a : A, b : B \vdash R\ \mathsf{type}\ [\Phi \mid \Psi]$.

types. In keeping with our original intuition of bridges of types as standing for relations on their endpoints, the target is the following principle, which we call *relativity*.

$$\mathsf{Bridge}_{\mathcal{U}}(A, B) \simeq A \times B \to \mathcal{U}$$

More specifically, we want the map $C \mapsto \mathsf{Bridge}_{\boldsymbol{x}.C@\boldsymbol{x}}(-,-)$, which extracts a relation from a bridge of types $C \in \mathsf{Bridge}_{\mathcal{U}}(A, B)$, to be an equivalence. To give an inverse, we define a type former that takes a relation R on A and B and creates a bridge of types between A and B. The rules for this type former, which we call Gel, are shown in Figure 1. The introduction and elimination rules make it possible to convert evidence for $R\langle M, N\rangle$ into a bridge between M and N over $\boldsymbol{x}.\mathsf{Gel}_{\boldsymbol{x}}(A, B, R)$ and vice-versa. The situation parallels the G, Glue, and V types used by Bezem et al., Cohen et al., and Angiuli et al. respectively to validate univalence in cubical type theory—particularly the first, on account of the shared use of substructural dimensions.

The formation rule gives the candidate inverse map to $C \mapsto \mathsf{Bridge}_{\boldsymbol{x}.C@\boldsymbol{x}}(-,-)$; the introduction and elimination rules show that $\mathsf{Bridge}_{\boldsymbol{x}.\mathsf{Gel}_{\boldsymbol{x}}(A,B,R)}(M, N) \simeq R\langle M, N\rangle$ for any $M \in A$ and $N \in B$, which establishes one of the inverse conditions that the two maps must satisfy. A more elaborate argument establishes the opposite condition, which states that $\mathsf{Gel}_{\boldsymbol{r}}(A, B, \mathsf{Bridge}_{\boldsymbol{x}.C@\boldsymbol{x}}(-,-)) \simeq C@\boldsymbol{r}$.

Note that I have implicitly relied on univalence and function extensionality in the above. For the first inverse condition, for example, what we actually need is a path connecting $\mathsf{Bridge}_{\boldsymbol{x}.\mathsf{Gel}_{\boldsymbol{x}}(A,B,R)}(-,-)$ and R in the type $A \times B \to \mathcal{U}$. To construct such a path, we need an understanding of paths in function types and the universe. In the work of Bernardy, Coquand, and Moulin, which uses "ordinary" dependent type theory where we use cubical type theory, such principles are not available. Instead, the inverse conditions are satisfied by adding them as primitive definitional equalities.

$$\mathsf{Bridge}_{\boldsymbol{x}.\mathsf{Gel}_{\boldsymbol{x}}(A,B,R)}(M,N) = R\langle M,N\rangle$$
$$\mathsf{Gel}_{\boldsymbol{r}}(A,B,\mathsf{Bridge}_{\boldsymbol{x}.C@\boldsymbol{x}}(-,-)) = C@\boldsymbol{r}$$

However, care must be taken to construct a model that validates these equations. In place of the (Kan) presheaf model typical of cubical type theory, Bernardy et al. introduce an ad-hoc notion of *refined presheaf* in order to ensure that they hold. By contrast, adding relational structure to cubical type theory requires no such complication, a benefit of a base theory with a well-behaved internal equality.

2.3.3 Consequences of parametricity

Using Gel types, it is simple to establish basic theorems of parametricity, such as the fact that all functions $F \in (X{:}\mathcal{U}) \to X \to X$ are equal to the identity function. For that example, we essentially follow the classical proof. For any such F, type $X : \mathcal{U}$, and $a : X$, we define the following relation.

$$R := \lambda\langle a', \_\rangle.\mathsf{Path}_X(a', a) \in X \times \mathsf{unit} \to \mathcal{U}$$

In words, $a' : X$ is related to $u : \mathsf{unit}$ when a' is a. (In this case, we are essentially defining a unary predicate rather than a relation.) We then introduce a fresh bridge variable $\boldsymbol{x}$ and apply F at the Gel type corresponding to R in direction $\boldsymbol{x}$.

$$F(\mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R)) \in \mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R) \to \mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R)$$

We then apply the result to $\mathsf{gel}_{\boldsymbol{x}}(a,*,\lambda^{\mathbb{I}}\_.a)$, the term of Gel type corresponding to the fact that $a : X$ and $* : \mathsf{unit}$ are related by R.

$$F(\mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R))(\mathsf{gel}_{\boldsymbol{x}}(a,*,\lambda^{\mathbb{I}}\_.a)) \in \mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R)$$

Next, we use the eliminator for Gel to turn this element into a proof that its endpoints stand in the relation R. Note that if we substitute $\langle \mathbf{0}/\boldsymbol{x}\rangle$ in the above, the Gel and gel terms reduce to their first arguments, giving FXa; if we substitute $\langle \mathbf{1}/\boldsymbol{x}\rangle$, we get $F(\mathsf{unit})(*)$.

$$\mathsf{ungel}(\boldsymbol{x}.F(\mathsf{Gel}_{\boldsymbol{x}}(X,\mathsf{unit},R))(\mathsf{gel}_{\boldsymbol{x}}(a,*,\lambda^{\mathbb{I}}\_.a))) \in R\langle FXa, F(\mathsf{unit})(*)\rangle$$

By definition of R, this term has type $\mathsf{Path}_X(FXa, a)$. As $X : \mathcal{U}$ and $a : A$ were arbitrary, we obtain a term of type $\mathsf{Path}_{(X:\mathcal{U})\to X\to X}(F, \lambda X.\lambda a.a)$ by function extensionality. From this, it is straightforward to establish that $((X{:}\mathcal{U}) \to X \to X) \simeq \mathsf{unit}$.

Related consequences of parametricity follow by similar arguments; for example, we can identify elements of $(X{:}\mathcal{U}) \to X \to (X \to X) \to X$ with natural numbers. However, not all standard theorems appear to be true. This is thanks to the lack of an *identity extension lemma* [56], which states that the relational interpretation of a closed type is the equality relation on that type. In this setting, the corresponding statement would say that $\mathsf{Bridge}_{\_.A}(M, N) \simeq \mathsf{Path}_{\_.A}(M, N)$: when A is a constant line, bridges over A are the same as paths over A. Such a property is typically needed to prove parametricity theorems that involve some externally quantified type: for example, that $A \simeq (X{:}\mathcal{U}) \to (A \to X) \to X$ for all $A : \mathcal{U}$. Luckily, Robert Harper and I observe that one can internally isolate the types that satisfy it as a sub-universe $\mathcal{U}_{\mathsf{BDisc}}$ of *bridge-discrete types* in $\mathcal{U}$ which is closed under the standard type formers and even satisfies the relativity property [22, §10]. One can then show that $A \simeq (X{:}\mathcal{U}) \to (A \to X) \to X$ for all $A : \mathcal{U}_{\mathsf{BDisc}}$.[5]

2.3.4 Related work

As I have already described, internal parametricity begins with Bernardy and Moulin [11], and the system I have presented is largely similar to the cleaner system later described by Bernardy, Coquand, and Moulin [14]. In [21], we extend the system to accommodate cubical type theory and so inherit the benefits of that theory. In particular, function extensionality is especially relevant for working with Church encodings. By relying on univalence rather than exact equations to prove relativity, we avoid needing the technical device of refined presheaves in constructing a model.[6]

We also advance the understanding of theorem proving using internal parametricity. We introduce the sub-universe of bridge-discrete types and show that it is closed under various type-formers; for function types, this requires function extensionality, while the argument for inductive types (shown for the special case of booleans) is non-trivial and uses the relativity property. Using the fact that the booleans are bridge-discrete, we observe that parametric cubical type theory refutes the law of the excluded middle as formulated in [62, §3.4]. We also prove a simple representative case of a parametricity theorem for a higher inductive type, showing that the type $(X{:}\mathcal{U}) \to \mathsf{susp}(X) \to \mathsf{susp}(X)$ of polymorphic functions on the *suspension* [62, §6.5] has exactly four elements.

In addition to the work of Bernardy et al., there is a second line of work by Nuyts, Vezzosi, and Devriese which takes a different approach to internal parametricity [55, 54]. Nuyts et al. introduce a system of modalities which distinguish between ordinary ("continuous") and parametric dependency on a variable. In brief, a term is parametric in a variable when it takes bridges in that variable to paths. Types and elements are checked under different modalities; where the work of Bernardy et al. centers on the relational interpretation aspect of parametricity, this work thus

[5]Or that $A \simeq (X{:}\mathcal{U}_{\mathsf{BDisc}}) \to (A \to X) \to X$ for all $A : \mathcal{U}_{\mathsf{BDisc}}$, for that matter.

[6]Our model is operational, while theirs is denotational, but one may essentially mechanically translate between the two approaches.

focuses on the distinction between uses of variables in type- and element-level positions. While Nuyts et al. do use dimension variables to represent bridges, their variables are structural. The consequences of this choice are mitigated by asserting the identity extension lemma for small types, but one result is that *iterated parametricity*—the use of parametricity to prove theorems about terms constructed using parametricity—is impossible. Moreover, said assertion is axiomatic, so it is not clear if a computational interpretation exists. In later work, Nuyts and Devriese recover iterated parametricity, but only by extending from merely paths and bridges to an infinite tower of increasingly coarse relations [54].

To combine internal parametricity and cubical type theory, it is essential that we have a *proof-relevant* concept of relation. First of all, we cannot hope to get results like those in Section 2.3.3 up to exact equality. For example, *not* every term of type $(X{:}\mathcal{U}) \to X \to X$ is exactly equal to the identity function: $\lambda X.\lambda a.\mathsf{coe}^{0\rightsquigarrow 1}_{\_.X}(a)$ and $\lambda X.\lambda a.a$ are equal up to a path but not up to exact equality.[7] $\mathsf{Path}_{x.A}(-,-)$ is a $\mathcal{U}$-valued relation, so if we want to construct paths using parametricity, we must work with $\mathcal{U}$-valued relations. Benton et al. [9], Ghani et al. [34], and Sojakova and Johann [57] study proof-relevant parametricity from a semantic perspective, complementing the syntactic parametricity of Bernardy et al. and Nuyts et al. In particular, Johann and Sojakova [39] define a notion of infinite-dimensional parametric models of System F using cubical categories.

3 Proposed Work

In the area of higher inductive types, I propose to flesh out the simple schema I developed with Robert Harper in [22] in two ways:

- incorporating higher inductive-inductive types,
- designing a more expressive language for constructors and boundaries.

For cubical internal parametricity, I intend to put the theory defined in [21] to good use. This program can be divided into two pieces:

- establishing a relationship between theorems proven in parametric and ordinary cubical type theory,
- using parametric type theory to prove a useful result from cubical type theory in a substantially simpler way.

Finally, I propose to complete the implementation of higher inductive types in the experimental cubical proof assistant `redtt` [61] and to incorporate parametric cubical type theory as an extension.

[7]The fact that coercion in a constant line of types is not exactly the identity function is known as the failure of *regularity*. Early attempts at cubical type theory (e.g., [27]) required coercion to be regular, but this is problematic for univalent universes; see [28, 59] for details.

3.1 Extending higher inductive types

3.1.1 Higher inductive-inductive types

Introduced by Nordvall Forsberg and Setzer [52], inductive-inductive definitions allow the simultaneous definition of an inductive type and an inductive family indexed by that type. For example, given a type $A \in \mathcal{U}$ with an order $O \in A \to A \to \mathcal{U}$, one may define a type $\mathsf{sorted} \in \mathcal{U}$ of sorted lists of elements of A together with a predicate $\mathsf{leq} \in A \to \mathsf{sorted} \to \mathcal{U}$ that tests when a number is less than or equal to all elements of a list [51, Example 3.2].

```
data sorted where
| nil ∈ sorted
| cons($a : A, s :$ sorted$, p :$ leq$(a, s)$) ∈ sorted

and leq($a : A, s :$ sorted) where
| leqnil($a : A$) ∈ leq($a$, nil)
| leqcons($a : A, b : A, s :$ sorted$, l : O(a, b), p :$ leq$(b, s)$) ∈ leq($a$, cons($b, s, p$))
```

Other classic examples include internal definitions of type theory as an inductive type $\mathsf{ctx} \in \mathcal{U}$ of contexts (generated by an empty context and context extension) together with a family $\mathsf{ty} \in \mathsf{ctx} \to \mathcal{U}$ of types in each context (generated by the type formers) [51, Example 3.1]. Hugunin has shown by example that inductive-inductive types can be encoded using indexed inductive types in cubical type theory [38]. Note that constructing ordinary (i.e., non-higher) inductive-inductive types in cubical type theory is already non-trivial; they raise the same issues as those described in Section 2.2.2 for indexed inductive types.

In the "HoTT Book" [62], a *higher* inductive-inductive definition is used to formulate a type of Cauchy reals, avoiding issues with choice raised by the standard definition in a constructive setting. As the name suggests, this is simply an inductive-inductive definition where constructors may introduce identities in addition to elements. More recently, Kaposi and Kovács have developed a schema for higher inductive types that includes inductive-inductive definitions [41]. However, this work does not address the problem of constructing instances of that schema. Kaposi, Kovács, and Altenkirch have now shown how to construct higher inductive-inductive types in a setting with UIP [42], but the non-truncated case remains open.

I intend to extend our schema to encompass inductive-inductive types and give a computational interpretation. I conjecture that such an extension requires few new insights. As far as design of the schema goes, the work of Kaposi and Kovács provides a blueprint, which need only be adapted to the cubical setting. As to the computational interpretation, Hugunin's encoding suggests the issues with constructing inductive-inductive types in cubical type theory are already raised with indexed inductive types, which our schema presently includes.[8] Moreover, it seems likely

[8]The likely existence of an encoding of higher inductive-inductive types as indexed higher in-

that the "higher" aspect of higher inductive-inductive types will be orthogonal to the inductive-inductive aspect.

Time permitting, I also plan to investigate *inductive-recursive types* [32]. Where an inductive-inductive type simultaneously defines a type inductively and a family over that type *inductively*, an inductive-recursive type defines a type inductively and a family over that type *recursively*. I am aware of no prior work on combining higher inductive types with induction-recursion.

3.1.2 Constructor language

The schema we design in [22] is designed to cover most commonly-recognized higher inductive types while being as simple as possible. The result is that there is plenty of room for polish. Recall the grammars of argument types and elements from Section 2.2.1.

$$
\begin{aligned}
\mathrm{A} &::= \mathsf{X}(\overline{I}) \mid (a{:}A) \to \mathrm{A} \\
\mathrm{M} &::= \mathsf{intro}_\ell(\overline{M}, \overline{\mathrm{M}}, \overline{r}) \mid \mathsf{hcom}_{\overline{I}}^{r \rightsquigarrow s}(\mathrm{M}; \overrightarrow{\xi_i \hookrightarrow x.\mathrm{M}}) \mid \lambda a.\mathrm{M} \mid \mathrm{M}M
\end{aligned}
$$

The most glaring absence here, at least for those familiar with homotopy type theory, is that of path types in the type being constructed. These could be used to define *higher truncations*, the higher-dimensional generalizations of the (-1)-truncation type trunc introduced in. For example, the *0-truncation* could be defined as follows.

$$
\begin{aligned}
&A : \mathsf{type} \vdash \mathsf{data}\ \mathsf{trunc0}\ \mathsf{where} \\
&\mid \mathsf{pt}(a : A) \\
&\mid \mathsf{squash}(c : \mathsf{trunc0}, d : \mathsf{trunc0}, p : \mathsf{Path}_{\mathsf{trunc0}}(c, d), q : \mathsf{Path}_{\mathsf{trunc0}}(c, d), x : \mathbb{I}, y : \mathbb{I}) \\
&\quad [x = 0 \hookrightarrow p@y \mid x = 1 \hookrightarrow q@y \mid y = 0 \hookrightarrow c \mid y = 1 \hookrightarrow d]
\end{aligned}
$$

Where the (-1)-truncation identifies every pair of *points* in a type, the 0-truncation identifies every pair of *paths*. Intuitively, it trivializes the higher-dimensional structure of A, leaving a "set" behind. While it is possible to define the 0-truncation indirectly using the schema we present (by way of the so-called "hub-and-spokes" construction [62, §6.7]), such an encoding is inefficient and more difficult to program with than the one shown above.

A second issue is the inability to define argument terms by case analysis on (ordinary) terms of positive type. It is less clear that such an extension is practically useful. However, it is of theoretical interest: it is used in Lumsdaine and Shulman's example of a higher inductive type not definable from pushouts [48, §9].

As with inductive-inductive types, these two extensions are already present in the schema of Kaposi and Kovács and given a semantics in the presence of UIP by Kaposi, Kovács, and Altenkirch, so the goal is to present a cubical version and give a (computational) semantics in the presence of univalence.

ductive types does not obviate the value of a direct definition, as existing encodings are not at all computationally efficient.

3.2 Internal parametricity

3.2.1 Connecting parametric and ordinary type theory

Suppose we have a type $A\ \mathsf{type}\ [\cdot \mid \Psi]$ which is well-formed in both ordinary and parametric cubical type theory. It is highly unlikely that any proof of A in parametric cubical type theory can be transformed into a proof in the ordinary fragment. For example, I mentioned in Section 2.3.4 that the parametric theory refutes the law of the excluded middle. Conversely, it is probable that the cubical set models of type theory validate the excluded middle (assuming a classical metatheory), though to my knowledge this is an open problem.[9] Despite this, it should be possible to give a transfer theorem for a restricted class of types A.

There are multiple levels on which we can pose the question: formal logics, computational interpretations, or denotational models. Fix a formal type theory—a collection of inference rules—consisting of the constructs of cubical and parametric type theory as described in this proposal. Such a theory can be interpreted into the computational interpretation described in [21]. One can mechanically translate this construction into a denotational model in Kan presheaves (suitably formulated) on the product of the cartesian cube category (for the path direction) and the BCH cube category (for the bridge direction); I will construct this model explicitly in my thesis. The theory and interpretations have ordinary cubical counterparts: formal cartesian cubical type theory, its computational interpretation [5], and its model in cubical sets [3]. Note that the formal cubical type theory is a fragment of the parametric theory, so it also has an interpretation in the parametric models.

The following theorem, which gives one way of relating the parametric and cubical models (denotational or computational), follows by straightforward inspection of the interpretation functions.[10]

Proposition. *Let $\Gamma \vdash A\ \mathsf{type}\ [\cdot \mid \Psi]$ be derivable in the fragment of formal cubical type theory without function types. If this type's parametric interpretation is inhabited, then its cubical interpretation is also inhabited.*

The idea behind the proof is that in either the computational or denotational interpretation, we can "truncate" a semantic parametric type to obtain a semantic cubical type, remembering only its elements in an empty bridge variable context. This truncation operation commutes with all of the cubical type formers except the function type. The trouble with the semantics of functions is that they quantify over future dimension substitutions (à la Kripke semantics), so that the semantics of $A \to B$ in an empty bridge context depends on the semantics of A and B in non-empty contexts.

[9] Kapulkin and Lumsdaine have verified that the law of the excluded middle holds in the simplicial model of homotopy type theory [44]. One may see from their proof that the question is not entirely trivial.

[10] For the computational version of the theorem, we assume that the cubical and parametric interpretations are built on the same untyped programming language.

We can also ask whether the theorem holds on the level of formal theories: if $\Gamma \vdash A\ \mathsf{type}\ [\cdot \mid \Psi]$ is as above and is inhabited in the parametric theory, is it also inhabited in the cubical theory? The semantics results suggests that this may be the case, and I plan to test that hypothesis. Theorems of this kind are sufficient to transfer the results I will discuss in the following section to cubical type theory; they do not concern higher order functions, so can be written as open sequents without function types.

However, this still leaves us far from recovering the full power of external parametricity. For example, one may use external parametricity to show that any *closed* term of type $(X{:}\mathcal{U}) \to X \to (X \to X) \to X$ in formal Martin-Löf type theory is equal to $\lambda X.\lambda z.\lambda s.s^n z$ for some closed $n \in \mathsf{nat}$ (see, for example, [13]). This is likewise provable in parametric type theory, but the theorem above does not suffice to transfer it to cubical type theory thanks to the presence of a higher order function. I propose to push the boundaries of transferability, with the goal of matching the power of external parametricity.

3.2.2 Applications

The central novelty in bringing parametricity to cubical type theory is that we can prove uniformity properties of higher inductive types. I intend to realize this potential using the smash product example discussed in the introduction.

The smash product is a binary operator on pointed types, types with a distinguished element. I write $\mathcal{U}_* := (X{:}\mathcal{U}) \times X$ for the universe of pointed types, $|A| := \mathsf{fst}(A) \in \mathcal{U}$ for the underlying type of a pointed type, and $\mathsf{pt}(A) := \mathsf{snd}(A) \in |A|$ for its distinguished point. The smash product is a higher inductive type defined as follows [64, Definition 4.3.6].

$$
\begin{aligned}
&X : \mathcal{U}_*, Y : \mathcal{U}_* \vdash \mathsf{data\ smash\ where} \\
&\mid \mathsf{smprod}(a : |X|, b : |Y|) \in \mathsf{smash} \\
&\mid \mathsf{basel} \in \mathsf{smash} \\
&\mid \mathsf{baser} \in \mathsf{smash} \\
&\mid \mathsf{gluel}(b : |Y|, x : \mathbb{I}) \in \mathsf{smash}\ [x = 0 \hookrightarrow \mathsf{gluel}(\mathsf{pt}(X), b), x = 1 \hookrightarrow \mathsf{basel}] \\
&\mid \mathsf{gluer}(a : |X|, x : \mathbb{I}) \in \mathsf{smash}\ [x = 0 \hookrightarrow \mathsf{gluer}(a, \mathsf{pt}(Y)), x = 1 \hookrightarrow \mathsf{baser}]
\end{aligned}
$$

I will abbreviate $\mathsf{smash}(A, B)$ as $A \wedge B$. Intuitively, the smash product is a quotient of the product $|A| \times |B|$ obtained by identifying elements of the form $\langle \mathsf{pt}(A), b\rangle$ and $\langle a, \mathsf{pt}(B)\rangle$ with the base-point $\langle \mathsf{pt}(A), \mathsf{pt}(B)\rangle$. It appears frequently in algebraic topology, as it provides a (monoidal) product structure on pointed spaces which is left adjoint to the internal hom functor (taking A, B to the space $A \to_* B$ of base-point-preserving functions between them). It also provides a means of relating the higher-dimensional spheres to each other: $\mathbb{S}^m \wedge \mathbb{S}^n = \mathbb{S}^{m+n}$.

However, as discussed previously, the basic algebraic properties of the smash product—unit laws, commutativity, associativity, and coherences between these—are difficult to establish, a fact which has prevented the complete formalization of multiple projects in synthetic homotopy theory [18, 64].[11] This is the result of coherence problems arising from nested induction on elements of the smash product. To define a map out of $(X \wedge Y) \wedge Z$, for example, one must induct on both the inner and outer product. Among the cases is, for example, that of $\mathsf{gluer}(\mathsf{gluer}(a, y), x)$ where $a : |X|$ and $x, y : \mathbb{I}$, a two-dimensional term which must be mapped to a two-dimensional term in the target type. As soon as the lower-dimensional cases are the least bit non-trivial, the higher-dimensional cases that relate them become quite painful.[12] Proving an *equality between* maps out of $(X \wedge Y) \wedge Z$ adds another layer of dimensionality. So defining the associator $\alpha : (X \wedge Y) \wedge Z \to X \wedge (Y \wedge Z)$ requires proving with 2-dimensional terms, and proving it is an equivalence requires constructing 3-dimensional terms. Then there is Mac Lane's pentagon, which establishes that the two different ways of re-associating a quaternary smash product are the same.

$$\begin{tikzcd}
 & & (X \wedge Y) \wedge (Z \wedge W) \arrow[drr, "\alpha"] & & \\
((X \wedge Y) \wedge Z) \wedge W \arrow[urr, "\alpha"] \arrow[dr, "\alpha \wedge W"'] & & & & X \wedge (Y \wedge (Z \wedge W)) \\
 & (X \wedge (Y \wedge Z)) \wedge W \arrow[rr, "\alpha"] & & X \wedge ((Y \wedge Z) \wedge W) \arrow[ur, "X \wedge \alpha"'] &
\end{tikzcd}$$

Proving this requires 4-dimensional terms! In fact, while they have been not been required in formalization efforts thus far, there are infinitely many properties of increasing dimensionality satisfied by the smash product (relating, for example, different ways of combining uses of the pentagon).[13]

I contend that parametricity provides a path to discharging all these obligations uniformly. Beyond the definitions of the commutator and associator and their candidate inverses, they can all be rephrased in terms of maps of the following form.

$$(X_1, \ldots, X_n{:}\mathcal{U}_*) \to \textstyle\bigwedge_{i \leq n} X_i \to_* \bigwedge_{i \leq n} X_i$$

For example, if we have an associator $\alpha : (X \wedge Y) \wedge Z \to X \wedge (Y \wedge Z)$ and a candidate inverse $\beta : X \wedge (Y \wedge Z) \to (X \wedge Y) \wedge X$, showing α is an equivalence amounts to showing that $\beta \circ \alpha$ and $\alpha \circ \beta$ are identity maps. Once we know that α is an equivalence,

[11]These proofs are notably simpler in classical algebraic topology. In the classical definitions, the gluel and gluer paths are replaced by *strict* equations, which are easier to work with than paths. However, this definition does not a priori ensure that $- \wedge -$ respects equivalences: it is not necessarily *homotopy invariant*. In homotopy or cubical type theory, only homotopy invariant constructions are possible (thanks to univalence), so an alternative definition is required. *Two-level type theories* [67, 1, 5] aim to make strict equality available in higher-dimensional type theory. However, it is not possible in existing theories to establish internally that a type is homotopy invariant.

[12]For a concrete demonstration of this phenomenon, one may see `pointed.smash` in the redtt library [61].

[13]It is not currently clear whether the full spectrum of conditions can even be formulated in type theory, but see Stay [58, §4] for the next level of coherences.

the pentagon can be read as expressing that the round-trip around its perimeter is the identity. I therefore propose to prove the following conjecture.

Conjecture. *In parametric cubical type theory, for any $n : \mathsf{nat}$, any term of type*

$$(X_1, \ldots, X_n{:}\mathcal{U}_*) \to \bigwedge_{i \le n} X_i \to_* \bigwedge_{i \le n} X_i$$

(where both products are associated in the same way) is either the polymorphic constant pointed function (sending everything to the basepoint) or the polymorphic identity function.

Crucially, I conjecture that this can be proven *uniformly* in n. This means that it takes the same amount of effort to prove that the commutator and associator are equivalences, that the pentagon commutes, or that to verify any higher coherence condition one can cook up.

For example, we can say that $\beta \circ \alpha : (X \wedge Y) \wedge Z \to (X \wedge Y) \wedge Z$ above is either a constant function or the identity (uniformly in X, Y, Z). To determine which is true is a matter of computation. The type of booleans is a unit for the smash product (this is not so hard to verify), so if we instantiate each type variable with bool, we get the following map.

$$\mathsf{bool} \xrightarrow{\sim} (\mathsf{bool} \wedge \mathsf{bool}) \wedge \mathsf{bool} \xrightarrow{\beta \circ \alpha} (\mathsf{bool} \wedge \mathsf{bool}) \wedge \mathsf{bool} \xrightarrow{\sim} \mathsf{bool}$$

By applying this map to true and false and evaluating it (which is completely mechanical, as we are working in a system with a canonicity property), we can determine whether it is the identity or constant, which then determines the behavior of $\beta \circ \alpha$ for every X, Y, Z!

In addition to this specific example, I propose to seek out other applications of internal parametricity to the practice of cubical type theory; particularly in the area of synthetic homotopy theory, where higher inductive types are most liberally employed. Candidate applications include proving properties of the *join* [62, §6.8], a related binary operator.

3.3 Implementation

I propose to implement my extensions to cubical type theory as part of `redtt`, an experimental proof assistant for cubical type theory currently under development [61].

Where higher inductive types are concerned, `redtt` supports a fragment of the schema presented in [22]: it lacks indexed higher inductive types and recursive arguments of function type (*generalized inductive types* in Dybjer's terminology [31]). I will extend `redtt` to accommodate the full schema from [22] as well as the additions I proposed in Section 3.1.

I also propose to develop an extension to `redtt` for iterated parametricity. This is a more ambitious task, because it requires dealing with substructural variables. The judgmental apparatus used to manage apartness in my work with Robert Harper is based on that of Cheney's nominal dependent type theory [23]; that work has not been implemented to my knowledge, but does describe a method of algorithmic type-checking. However, it is not clear whether the additional demands of dimension variables (principally, the presence of constants 0 and 1) complicates the story. The `cubical` type-checker [24] is also related, but it is a type-checker for (some variation of) homotopy type theory combined with an interpretation into a cubical semantic domain, rather than a type-checker for a cubical type theory per se.

3.4 Timeline

Jul–Aug 2019 Develop an expanded schema for higher inductive types, including in particular inductive-inductive types.

Sept–Nov 2019 Establish connections between internally parametric and ordinary type theory, first on the semantic and then on the syntactic level.

Dec 2019–Feb 2020 Extend the `redtt` implementation with the expanded higher inductive type schema and with internal parametricity, giving priority to the latter. Use the implementation to explore applications of internal parametricity.

March–May 2020 Write thesis.

Bibliography

[1] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29, 2016.

[2] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. Homotopical patch theory. *J. Funct. Program.*, 26:E18, 2016.

[3] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Cartesian cubical type theory. December 2017. URL https://github.com/dlicata335/cart-cube.

[4] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 680–693, 2017.

[5] 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, United Kingdom*, 2018.

[6] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516, 2014.

[7] Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *J. UCS*, 23(1):63–88, 2017.

[8] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 164–172, 2017.

[9] Nick Benton, Martin Hofmann, and Vivek Nigam. Abstract effects and proof-relevant logical relations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 619–632, 2014.

[10] Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 108–122, 2011.

[11] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 135–144, 2012.

[12] Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 61–72, 2013.

[13] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 345–356, 2010.

[14] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015.

[15] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, pages 107–128, 2013.

[16] Marc Bezem, Thierry Coquand, and Erik Parmann. Non-constructivity in kan simplicial sets. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, pages 92–106, 2015.

[17] Marc Bezem, Thierry Coquand, and Simon Huber. The univalence axiom in cubical sets. arXiv:1710.10941, October 2017.

[18] Guillaume Brunerie. Computer-generated proofs for the monoidal structure of the smash product, 2018. URL https://youtu.be/JEUvWyd1mTk. Video of a talk in the Homotopy Type Theory Electronic Seminar Talks series.

[19] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda. URL https://github.com/HoTT/HoTT-Agda.

[20] Evan Cavallo and Robert Harper. Computational higher type theory IV: Inductive types. arXiv:1801.01568, January 2018.

[21] Evan Cavallo and Robert Harper. Parametric cubical type theory. arXiv:1901.00489, January 2019.

[22] Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *PACMPL*, 3(POPL):1:1–1:27, 2019.

[23] James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8(1), 2012.

[24] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. `cubical`. URL https://github.com/simhu/cubical.

[25] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, pages 5:1–5:34, 2015.

[26] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[27] Thierry Coquand. Variation on cubical sets. http://www.cse.chalmers.se/ coquand/comp.pdf, 2014.

[28] Thierry Coquand. Re: [HoTT] a cubical type theory. Mailing list post, May 2015. URL groups.google.com/d/msg/homotopytypetheory/oXQe5u_Mmtk/3HEDk5g5uq4J.

[29] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, pages 50–66, 1988.

[30] Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In *33nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 9-12, 2018*, 2018.

[31] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[32] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[33] Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. In *Mathematical Foundations of Programming Semantics, 33rd International Conference, Ljubljana, Slovenia*, 2017.

[34] Neil Ghani, Patricia Johann, Fredrik Nordvall Forsberg, Federico Orsanigo, and Tim Revell. Bifibrational functorial semantics of parametric polymorphism. *Electr. Notes Theor. Comput. Sci.*, 319:165–181, 2015.

[35] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[36] Kuen-Bang Hou (Favonia) and Michael Shulman. The seifert-van kampen theorem in homotopy type theory. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, pages 22:1–22:16, 2016.

[37] Simon Huber. *Cubical Interpretations of Type Theory*. PhD thesis, University of Gothenburg, 2016.

[38] Jasper Hugunin. Constructing inductive-inductive types in cubical type theory. To appear in *FoSSaCS 2019*, 2019.

[39] Patricia Johann and Kristina Sojakova. Cubical categories for higher-dimensional parametricity. arXiv:1701.06244, January 2017.

[40] Daniel M. Kan. Abstract homotopy. I. *Proceedings of the National Academy of Sciences of the United States of America*, 41(12):1092–1096, 1955. ISSN 00278424.

[41] Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *3nd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, 2018.

[42] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *PACMPL*, 3(POPL):2:1–2:24, 2019.

[43] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). arXiv:1211.2851, November 2012.

[44] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. Law of excluded middle in the simplicial model. https://www.uwo.ca/math/faculty/kapulkin/notes/LEM_in_sSet.pdf, December 2018.

[45] Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pages 381–395, 2012.

[46] Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 595–604, 2016.

[47] Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, pages 432–451, 2013.

[48] Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. arXiv:1705.07088, May 2017.

[49] 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. North-Holland, 1975.

[50] Per Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen, J. Łoś, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, Methodology and Philosophy of Science*, volume VI, pages 153–175, 1982.

[51] Fredrik Nordvall Forsberg. *Inductive-Inductive Definitions.* PhD thesis, Swansea University, 2013.

[52] Fredrik Nordvall Forsberg and Anton Setzer. Inductive-inductive definitions. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pages 454–468, 2010.

[53] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers, Göteborg University, 2007.

[54] Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788, 2018.

[55] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017.

[56] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[57] Kristina Sojakova and Patricia Johann. A general framework for relational parametricity. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 869–878, 2018.

[58] Michael Stay. Compact closed bicategories. *Theory and Applications of Categories*, 31(26):755–798, August 2016.

[59] Andrew Swan. Separating path and identity types in presheaf models of univalent type theory. arXiv:1808.00920, August 2018.

[60] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, 2012.

[61] The RedPRL Development Team. `redtt`, 2018. URL https://github.com/RedPRL/redtt.

[62] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[63] Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 122–129, 2016.

[64] Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory*. PhD thesis, Carnegie Mellon University, 2018.

[65] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy type theory in lean. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 479–495, 2017.

[66] Vladimir Voevodsky. Univalent Foundations Project (a modified version of an NSF grant application). http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf, October 2010.

[67] Vladimir Voevodsky. A simple type system with two identity types. Talk at Andre Joyal's 70th birthday conference. (Slides available at https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS_slides.pdf), 2 2013. URL https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf.