A Cubical Approach to Synthetic Homotopy Theory

Daniel R. Licata
Wesleyan University
dlicata@wesleyan.edu

Guillaume Brunerie
Université de Nice Sophia Antipolis
guillaume.brunerie@gmail.com

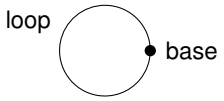
***Abstract*—Homotopy theory can be developed synthetically in homotopy type theory, using types to describe spaces, the identity type to describe paths in a space, and iterated identity types to describe higher-dimensional paths. While some aspects of homotopy theory have been developed synthetically and formalized in proof assistants, some seemingly easy examples have proved difficult because the required manipulations of paths becomes complicated. In this paper, we describe a cubical approach to developing homotopy theory within type theory. The identity type is complemented with higher-dimensional cube types, such as a type of squares, dependent on four points and four lines, and a type of three-dimensional cubes, dependent on the boundary of a cube. Path-over-a-path types and higher generalizations are used to describe cubes in a fibration over a cube in the base. These higher-dimensional cube and path-over types can be defined from the usual identity type, but isolating them as independent conceptual abstractions has allowed for the formalization of some previously difficult examples.**

I. Introduction

Homotopy theory is the study of spaces by way of their points, paths (between points), homotopies (paths or continuous deformations between paths), homotopies between homotopies (paths between paths between paths), and so on. This area of mathematics can be developed *synthetically* by using the homotopy-theoretic structure of types in Martin-Löf type theory [3, 13, 14, 20, 30, 31, 32]. Using principles inspired by these semantics, such as higher inductive types [21, 22, 26] and Voevodsky's univalence axiom [16, 31], some aspects of homotopy theory have been developed and formalized using the Agda [24] and Coq [10] proof assistants. These include calculations of some homotopy groups of spheres [17, 19, 29]; constructions of the Hopf fibration [29], of covering spaces [12], and of Eilenberg-MacLane spaces [18]; and proofs of the Freudenthal suspension theorem [29], the Blakers-Massey theorem, the van Kampen theorem [29], and the Mayer-Vietoris theorem [8]. Ideas from synthetic homotopy theory have also been applied to represent the patch theories that arise in version control using higher inductive types [2].

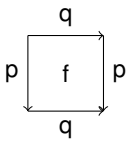
Many of the results mentioned above were posed as challenge problems during the 2012–2013 year on univalent foundations at the Institute for Advanced Study. One additional challenge problem from that year, which was anticipated to be *less* difficult than those listed above, was to show that the higher inductive definition of the torus is equivalent to a product of two circles.

In homotopy type theory, the elements of a type correspond to points of a space, and the equality proofs in a type correspond to paths (we write $\mathsf{Path\ a\ b}$ for the equality type). A higher inductive type for the circle (see [19, 29]) is generated by a point and a loop. A circle



corresponds to a higher inductive type with one point constructor $\mathsf{base} : \mathsf{S}^1$ and one path constructor $\mathsf{loop} : \mathsf{Path\ base\ base}$.

Similarly, we can describe a torus by identifying the opposite sides of a square (glue two sides together to form a cylinder, and then glue the two ends of the cylinder together):



Writing $\mathsf{p} \cdot \mathsf{q}$ for composition of paths in diagramatic order, the torus can be represented by a higher inductive type with the following constructors (see [29, Section 6.6]):

$\mathsf{a} : \mathsf{T}$
$\mathsf{p} : \mathsf{Path\ a\ a}$
$\mathsf{q} : \mathsf{Path\ a\ a}$
$\mathsf{f} : \mathsf{Path}\ (\mathsf{p} \cdot \mathsf{q})\ (\mathsf{q} \cdot \mathsf{p})$

The f ("face") constructor generates a path between paths. It represents the inside of the above square as a disc between the "left then bottom" and "top then right" composites. Algebraically, the torus is generated by two commuting loops.

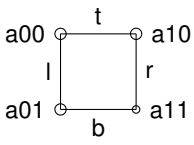
To prove that the torus is equivalent to the product of two circles means to give functions $\mathsf{t2c} : \mathsf{T} \to \mathsf{S}^1 \times \mathsf{S}^1$ and $\mathsf{c2t} : \mathsf{S}^1 \times \mathsf{S}^1 \to \mathsf{T}$ and show that they are mutually inverse (up to paths). At first glance, it seems like it should be simple to define the functions back and forth and prove that they are mutually inverse using the recursion and induction principles for the circle and the torus. And indeed, it is not difficult to define the two functions. However, at the end of the IAS year, this problem had not been solved, though Sojakova and Lumsdaine had each given proof sketches, and Sojakova's was

This material is based on research sponsored by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

later written up in a detailed 25-page proof [28]. The reason for the complexity is that the path manipulation required to prove the path-between-path goals gets quite involved.

In this paper, we develop a cubical approach to synthetic homotopy theory. Using this approach and the libraries we develop, the proof that the torus is the product of two circles can be formalized in Agda in around 100 lines of code.[1] The approach has also proved useful for the formalization of a "three-by-three" lemma about pushouts that is used in the construction of the Hopf fibration,[2] and in resolving a question about a patch theory represented as a higher inductive type [2]. The approach was also used by Cavallo to simplify the proof of the Mayer-Vietoris theorem [8].

Inspired by heterogeneous equality [23], the cubical sets models of type theory [5], and cubical methods in higher category theory [6], the main idea of the approach is to work with *cube types* that generalize the path type Path a b. For example, in this paper, we will consider a type of squares Square l t b r, dependent on four paths that fit into a square:



We will also consider a type of cubes dependent on six squares giving its sides. Another key ingredient is to work systematically with path-over-a path and higher cube-over-a-cube types to represent cubes in a dependent type.

While our approach fits nicely with work in progress on new cubical type theories [1, 7, 11, 25], the present paper can be conducted entirely by making appropriate definitions in "book homotopy type theory" [29]—Martin-Löf type theory with the univalence axiom and higher inductive types. Higher cubes can be defined in terms of higher paths, and cube-over-a-cube types can be reduced to homogeneous paths. Thus, our constructions can be interpreted in the known models of homotopy type theory with univalence and higher inductive types (see [16, 22, 27]). While cubes can be reduced away in this way, for engineering reasons, we have found it convenient in Agda to use new inductive families to represent cube types.

We begin by discussing a notion of heterogeneous equality (Section II), and a related path-over-a-path type (Section III). Then, we discuss squares (Section IV) and cubes (Section V). Next, we discuss the torus example (Section VI), and the three-by-three pushout lemma (Section VII).

II. Heterogeneous equality

The path type Path {A} a0 a1 (an inductive family with one constructor id : Path a0 a0) is sometimes called *homogeneous equality*, because it relates two elements a0 and a1 whose types are definitionally/judgementally equal. McBride [23] introduced a *heterogeneous equality*, which is an equality type a:A = b:B that relates two elements a:A and b:B which may have two judgementally distinct types, though the reflexivity constructor applies only when both the two types and the two terms are judgementally equal. Heterogeneous equality is used to elide the reasoning why equations type check from the equations themselves, which simplifies some constructions. However, McBride's heterogeneous equality is logically equivalent to a homogeneous equality type satisfying uniqueness of identity proofs [23], which is undesirable in homotopy type theory, because not all types should be sets.

This paper provides an investigation of how to manage the reasons why equations type check in a setting where these reasons are proof-relevant. While we cannot ignore the reason why an equation type checks entirely, we can still keep the evidence "off to the side", rather than embedding it in the equation itself. For example, we can define a type HEq A B α a b where α : Path {Type} A B and a:A and b:B, as an inductive family with one reflexivity constructor hid : HEq A A id a a, which relates a to itself along the reflexivity path id. This heterogeneous equality relates two elements of two different types *along a specific equality* α *between the types*.

However, this notion of heterogeneous equality need not be taken as a primitive, because it can be reduced to the homogeneous equality type in several equivalent ways. Writing coe : Path {Type} A B → A → B for the function (defined by path induction, as transport (λ X → X)) that coerces along a homogeneous equality, the following types are equivalent (under the definition of equivalence in [29]):

1) The inductive family HEq A B α a b defined above.
2) Path {B} (coe α a) b – send a over to B using the type equality α, and compare the result with b.
3) Path {A} a (coe (! α) b) – send b over to A using the equality α (inverted), and compare the result with a.
4) Define heterogeneous equality by path induction into the universe: when the type equality α is id, a heterogeneous equality is a homogeneous equality: HEq' A A id a b = Path {A} a b

The equivalences between these types are all immediate by path induction or induction on HEq: keeping the evidence that the equation type checks "off to the side" is equivalent to embedding it in the equation on either side, and to the more symmetric fourth option. We will argue that, even though it could be defined away, it is useful to think in terms of such "off to the side" abstractions.

III. Path Over a Path

A. Type Definition

The heterogeneous equalities HEq A B α a a' we consider will often have the property that some of the outer structure of A and B is the same, and the important part of α happens inside this outer structure. A typical example is

```
HEq (Vec Nat (n + m)) (Vec Nat (m + n))
    (ap (Vec Nat) (+-comm n m)) v1 v2
```

[1]The proof is available at `github.com/dlicata335/hott-agda`, tag `torus-is-s1s1`, in the file `homotopy/TS1S1.agda`; use Agda version 2.4.2.2.

[2]Agda code is in the repository `github.com/HoTT/HoTT-Agda`, but this was checked using old versions of Agda and the cubical libraries, on a machine with 250GB of memory.

where Vec Nat k represents vectors of length k, and v1 : Vec Nat (n + m) and v2 : Vec Nat (m + n). In this example, the two types both have the form Vec Nat -, and the reason why the two types are equal is essentially commutativity of addition—but we need to use use ap (congruence of equality) to apply Vec Nat to both sides of the commutativity proof.

Heterogeneous equalities of this form can be simplified using a *factored* heterogeneous equality type, which separates a context (like Vec Nat -) from an equality on the insides of the context. This is called a *path over a path* or *path-over* type (it is discussed briefly in [29]), and it can be defined as an inductive family as follows:

```
data PathOver {A : Type} (C : A → Type) {a1 : A} :
              {a2 : A} (α : Path a1 a2)
              (c1 : C a1) (c2 : C a2) → Type where
  id : {c1 : C a1} → PathOver C id c1 c1
```

Given a1, a2 : A connected by a path α, along with a dependent type C : A → Type, this type relates an element of C a1 to an element of C a2. The endpoints a1 and a2 are implicit arguments because they can typically be inferred. The constructor id (note the use of constructor overloading) represents *reflexivity over reflexivity*, and says that any reflexive equation where α is also reflexivity holds. Using path-over, the above example is

```
PathOver (Vec Nat) (+-comm n m) v1 v2
```

Here C is Vec Nat, which is applied to n+m to get the type of v1, to m+n to get the type of v2, and to +-comm n m to get the proof that the two types are equal.

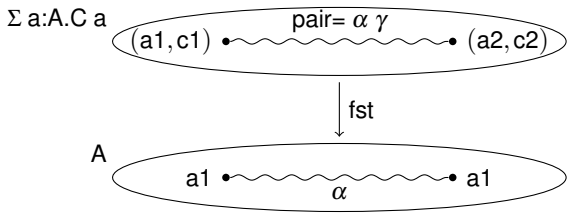
Because types are elements of a universe, HEq A B α a1 a2 is the special case of PathOver (λ (X : Type) → X) α a1 a2. Conversely, PathOver can be expressed in terms of heterogeneous equality using ap as above. Indeed, the following types are equivalent:

1) The inductive family PathOver C {a1} {a2} α c1 c2
2) HEq (C a1) (C a2) (ap C α) c1 c2
3) Path {C a2} (transport C α c1) c2
4) Path {C a1} c1 (transport C (! α) c1)
5) PathOver defined by path induction into the universe as

```
PathOver C id c1 c2 = Path c1 c2
```

The equivalences are all simple to construct using path induction or HEq-induction or path-over induction. The final three options are analogous to the final three ways to render heterogeneous equality described above, though using transport C α instead of the equivalent coe (ap C α).

While we have motivated PathOver as a factored heterogeneous equality, there is also a geometric intuition. Dependent types correspond to fibrations, so a type C : A → Type can be pictured as its total space Σ a:A. C a projecting down to A by first projection. A path-over γ : PathOver C α c1 c2 represents a path σ in Σ a:A. C a between (a1, c1) and (a2, c2), such that ap fst σ is exactly α. That is, it is a path in the total space that projects down to, or *lies over*, α (path pairing pair= α γ will be made precise below):



We have experimented with two implementations of path-over in two different Agda libraries. In one library, it is defined as in the fifth option above (by path induction into the universe). In another library, it is defined as inductive family, which is convenient because we can eliminate on a path-over using Agda's support for pattern matching. Moreover, the inductive family implementation does not really require extending the semantics of type theory with this new type constructor: If we defined PathOver C α c1 c2 as Path {C a2} (transport C α c1) c2, then the inductive family elimination rule is definable and satisfies the required β-reduction rule definitionally. Therefore, assuming that everything in Agda could be translated to eliminators (see [9]), the eliminator for path-over could then be implemented in terms of homogeneous paths, before interpreting in a model.

B. Library

Next, we give a tour of some of the facts about path-overs that are commonly used. Though we use Agda notation, we sometimes elide universal quantifiers, implicitly quantifying variables with their most general types.

First, applying a dependent function to a homogeneous path gives a path over it:

```
apdo : {A : Type} {C : A → Type} (f : (a : A) → C a)
       {a1 a2 : A} (α : Path a1 a2)
       → PathOver C α (f a1) (f a2)
apdo f id = id
```

The name apdo is for "dependent ap producing a path-over".

Next, we define the pairing of paths discussed above: A path in a Σ-type can be constructed by pairing together a path between the left-hand sides and a path over it between the right-hand sides:

```
pair= : {A : Type} {B : A → Type} {a1 a2 : A} (α : Path a1 a2)
        {b1 : A a1} {b2 : A a2} (β : PathOver B α b1 b2)
        → Path (a1, b1) (a2, b2)
pair= .id id = id
```

In fact, this is an equivalence, with inverse given by ap fst and apdo snd—these three behave like introduction and elimination rules for paths in a Σ-type.

We have the type equivalence (written ≃) between PathOver and a homogeneous equation using transport:

```
hom-to-over/left-eqv : Path (transport C α c1) c2
                       ≃ PathOver C α c1 c2
```

In the special case where α is id, this gives that paths over reflexivity are the same as paths:

```
hom-to-over-eqv : {A : Type} {C : A → Type}
  {a1 : A} {c1 c2 : C a1}
  → (Path {C a1} c1 c2) ≃ (PathOver C id c1 c2)
```

Next, we have lemmas characterizing path-overs based on the dependent type C, which are analogous to the rules for transport in each dependent type. A path-over in a constant fibration is the same as a homogeneous path:

```
PathOver-constant-eqv : {A : Type} {C : Type}
  {a1 a2 : A} {c1 c2 : C} {α : Path a1 a2}
  → (PathOver (λ _ → C) α M1 M2) ≃ (Path c1 c2)
```

A path-over in a (function) composition can be re-associated, moving part of the fibration into the path (when A is $(\lambda\, X \to X)$, this is the equivalence between HEq and PathOver mentioned above).

```
over-o-ap-eqv : {A B : Type} (C : B → Type)
  {f : A → B} {a1 a2 : A} {α : Path a1 a2}
  {c1 : C a1} {c2 : C a2} →
  (PathOver (C o f) α c1 c2) ≃ (PathOver C (ap f α) c1 c2)
```

This is the path-over equivalent of re-associating between transport (C o f) α and transport C (ap f α).

Finally, we have rules for each type constructor. For example for Π-types, we have

```
PathOverΠ-eqv : {A : Type} {B : A → Type}
  {C : Σ B → Type} {a1 a2 : A} {α : Path a1 a2}
  {f : (x : B a1) → C (a1,x)} {g : (x : B a2) → C (a2,x)}
  → (PathOver (λ a → (x : B a) → C (a,x)) α f g)
  ≃ ((x : B a1) (y : B a2) (β : PathOver B α x y) →
       PathOver C (pair= α β) (f x) (g y))
```

This is a path-over version of function extensionality; it says that two functions are equal (over α) if they take two equal arguments (over α) to two equal results (over both α and β, because f x : C (a1,x) and g y : C (a2,y)). This can be proved from the usual function extensionality for homogeneous paths.

C. Example: Circle Elimination

For the circle type S^1, with constructors base : S^1 and loop : Path $\{S^1\}$ base base, the elimination rule is

```
S¹-elimo : (C : S¹ → Type) (b : C base)
           (l : PathOver C loop c c) (x : S¹) → C x
S¹-elimo C b l base ≡ b
βloop/elimo : Path (apdo (S¹-elimo C b l) loop) l
```

We write S^1-elimo ("S^1 elimination with path-over") for circle elimination, which we will also call circle induction. To eliminate from the circle into a dependent type C, we give a point b in C base as the image of the base point, and a path from c to itself *over the loop* as the image of loop. We have a definitional computation rule for points and a propositional computation rule for applying the eliminator (using apdo) to the loop. By the equivalence between PathOver C loop c c and Path (transport C loop c) c, these rules are equivalent to the usual ones in terms of homogeneous path.

In the case for loop, we will typically "reduce" the PathOver C loop c c goal using the type-directed moves described above. For example, when calculating $\pi_1(S^1)$ [19], circle induction is used to define a function

```
decode : (x : S¹) → Cover x → Path base x
```

where Cover is the universal cover fibration. In this case, we apply circle elimination with C x := Cover x → Path base x. In the case for base, we supply a function loop^ : Int → Path base base (by definition Cover base is Int). In the case for loop, PathOverΠ-eqv is used to reduce the goal to

```
(x y : Cover base) (β : PathOver Cover loop x y) →
  PathOver (λ p → Path base (fst p)) (pair= loop β)
           (loop^ x) (loop^ y)
```

Because we are defining a non-dependent function, the function's range type does not mention snd p, so using over-o-ap-eqv to reassociate, and then reducing ap fst (pair= loop β) to loop, we need to show

```
(x y : Cover base) (β : PathOver Cover loop x y) →
  PathOver (λ a → Path base a) loop (loop^ x) (loop^ y)
```

Cover is defined so that PathOver Cover loop x y is equivalent to Path (x + 1) y, so we need to show

```
PathOver (λ a → Path base a) loop (loop^ x) (loop^ (x + 1))
```

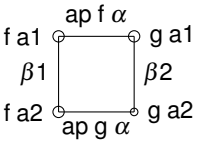
For this, we need a rule for reducing PathOver in a Path type, which we discuss next.

IV. Squares

To understand the rule for path-over in a path type, it is helpful to generalize from the above example to

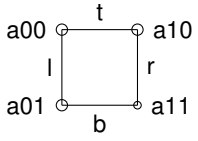
```
PathOver (λ x → Path (f x) (g x)) α β1 β2
```

where f g : A → B and α : Path $\{A\}$ a1 a2 and $\beta 1$: Path (f a1) (g a1) and $\beta 2$: Path (f a2) (g a2). The key idea is that this data naturally fits into a square as follows:



A. Definition

Given points and paths that form a square



we would like to define a type Square l t b r, where an element of this type represents the inside of such a square. One possible definition is as path-over in a path type:

```
PathOver (λ (x:A, y:A) → Path x y) (pair-line t b) l r
```

where pair-line is a non-dependent version of pair= for A × B, which takes two homogeneous paths. Another is as a disc (path-between-paths) between composites Path (l · b) (t · r). We can also define a new inductive family dependent on four points, which we make implicit arguments, and four lines, representing squares:

```
data Square {A : Type} {a00 : A} : {a01 a10 a11 : A}
  : Path a00 a01 → Path a00 a10 →
    Path a01 a11 → Path a10 a11 → Type where
  id : Square id id id id
```

All of these types are equivalent:

1) The inductive family Square l t b r
2) Path (l · b) (t · r)
3) PathOver (λ (x:A, y:A) → Path x y) (pair= t b) l r
4) A definition by path-induction:

$$\text{Square l id id r} = \text{Path l r}$$

The second definition again satisfies the inductive family elimination rule with a judgemental β rule, so in Agda we use the inductive family for convenience but think of it as a derived notion semantically.

B. Library

Next, we develop some operations on squares. We have the equivalence with discs, and the equivalence between path-overs in a path type and certain squares:

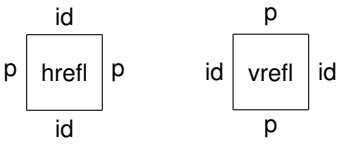
```
square-disc-eqv : Square l t b r ≃ Path (l · b) (t · r)

PathOver-=-eqv : {A B : Type} {f g : A → B}
  {a1 a2 : A} {α : Path a1 a2}
  {β1 : Path (f a1) (g a1)} {β2 : Path (f a2) (g a2)}
  → (PathOver (λ x → Path (f x) (g x)) α β1 β2)
  ≃ (Square β1 (ap f α) (ap g α) β2)
```

This equivalence includes maps into and out of a path-over in a path type, which we will write as in-PathOver= and out-PathOver=; we use the in- and out- notation analogously for other equivalences.

For a given path, there are horizontal and vertical reflexivity squares, with reflexivity paths in the other dimension:

```
hrefl-square : {A : Type} {a00 a01 : A}
  {p : Path a00 a01} → Square p id id p
vrefl-square : {A : Type} {a00 a01 : A}
  {p : Path a00 a01} → Square id p p id
```



We can apply a function to a square, yielding a square between the action of the function on each side:

```
ap-square : {A B : Type} (f : A → B)
  {a00 a01 a10 a11 : A} {l : Path a00 a01}
  {t : Path a00 a10} {b : Path a01 a11} {r : Path a10 a11}
  → Square l t b r → Square (ap f l) (ap f t) (ap f b) (ap f r)
```

We have rules for introducing and eliminating squares in each type. For example, for A × B, we can pair a square in A with a square in B to get a square in A × B, whose boundary sides are the pairs of the sides of the given squares:

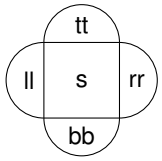
```
pair-line : Path {A} a0 a1 → Path {B} b0 b1
          → Path (a0, a1) (b0, b1)

pair-square :
  Square {A} la ta ba ra → Square {B} lb tb bb rb
  → Square (pair-line la lb) (pair-line ta tb)
           (pair-line ba bb) (pair-line ra rb)
```

Because Square is a dependent type, by path induction we can "retype" the sides of a square by paths-between-paths:

```
whisker-square : {A : Type} {a00 a01 a10 a11 : A}
  {l l' : Path a00 a01} {t t' : Path a00 a10}
  {b b' : Path a01 a11} {r r' : Path a10 a11}
  (ll : Path l l') (tt : Path t t') (bb : Path b b') (rr : Path r r')
  (s : Square l t b r) → Square l' t' b' r'
```

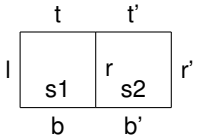
This creates a new square that is the composite of the original square s with these paths:



We can compose squares vertically and horizontally:

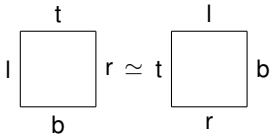
```
_·-square-v_ : Square l t b r → Square l' b b' r'
  → Square (l · l') t b' (r · r')
_·-square-h_ : Square l t b r → Square r t' b' r'
  → Square l (t · t') (b · b') r'
```

For example, s1 ·-square-h s2 represents the composite



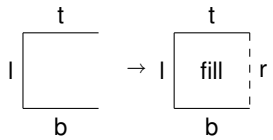
Symmetry interchanges the horizontal and vertical sides:

```
square-symmetry-eqv : Square l t b r ≃ Square t l r b
```



Another operation we will need is *Kan filling* [15]. For squares, this says that given any three sides, we can find a fourth that fits in a square. For example:

```
fill-square-right : {A : Type} {a00 a01 a10 a11 : A}
  (l : Path a00 a01) (t : Path a00 a10) (b : Path a01 a11)
  → Σ [r : Path a10 a11] Square l t b r
```



Though both the groupoid structure (identity, composition, inverses, the groupoid laws) and the Kan filling result from path induction, it is illustrative to construct them directly from

each other. To derive the Kan filler, we can define the right side r to be $!\,t \cdot l \cdot b$, and then, as a disc between composites, the required square is a $\mathsf{Path}\ (l \cdot b)\ (t \cdot (!\,t \cdot l \cdot b))$ using the groupoid laws. From the Kan filling we can define $p \cdot q$ as $\mathsf{fst}\ (\mathsf{fill}\ p\ \mathsf{id}\ q)$, and then use $\mathsf{snd}\ (\mathsf{fill}\ p\ \mathsf{id}\ q)$ to show the groupoid laws.

C. Example: Circle induction, continued

Returning to the example from Section III-C, we need a

```
PathOver (λ a → Path base a) loop (loop^ x) (loop^ (x + 1))
```

By PathOver-=-eqv, this is the same as a

```
Square (loop^ x) (ap (λ _ → base) loop)
       (ap (λ a → a) loop) (loop^ (x + 1))
```

After reducing the ap's using whisker-square, we need a

```
Square (loop^ x) id loop (loop^ (x + 1))
```

But loop^ (x+1) is defined to be loop^x · loop, so we need a

```
Square (loop^ x) id loop (loop^ x · loop)
```

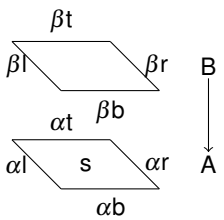
which is the characterization of composition as a Kan filler.

D. Square over a square

Just as we had a type for a path in a dependent type over a path in the base, it will be useful to have a type of squares in a dependent type over a square in the base:

```
data SquareOver {A : Type} (B : A → Type)
  {a00 : A} {b00 : B a00} : {a01 a10 a11 : A}
  {αl : Path a00 a01} {αt : Path a00 a10}
  {αb : Path a01 a11} {αr : Path a10 a11}
  (s : Square αl αt αb αr)
  {b01 : B a01} {b10 : B a10} {b11 : B a11}
  (βl : PathOver B αl b00 b01)
  (βt : PathOver B αt b00 b10)
  (βb : PathOver B αb b01 b11)
  (βr : PathOver B αr b10 b11)
  → Type where
    id : SquareOver B id id id id id
```

A SquareOver B f βl βt βb βr relates four path-overs, each of which lays over one side of the square s (the boundary of s and the points in B are implicit arguments). Visually, an element of this type is the inside of the top square in the following:



To avoid introducing a new inductive family, we could define square-over by square induction, saying that a square over id is a homogeneous square. Alternatively, it can be defined as a higher disc directly, using several transports.

E. Example: Torus

The torus is generated by a point constructor, two path constructors, and a square whose opposite sides are identified:

```
a : T
p : Path a a
q : Path a a
f : Square p q q p
```

A simply-typed function from the torus is defined by giving the image of each constructor:

```
T-rec : {C : Type} (a' : C) (p' q' : Path a a) (f' : Square p' q' q' p')
      → T → C
```

The dependent elimination rule is analogous, but the image of each constructor lays over that constructor:

```
T-elim : (C : T → Type) (a' : C a)
  (p' : PathOver C p a' a') (q' : PathOver C q a' a')
  (f' : SquareOver C f p' q' q' p')
  (x : T) → C x
```

For contrast, writing out the type of T-elim using homogeneous paths directly gives

```
T-elim : (C : T → Type) (a' : C a)
  (p' : Path (transport C p a') a')
  (q' : Path (transport C q a') a')
  (f' : Path (transport (λ x → Path (transport C x a') a') f
                ((transport-· C p q) · (ap (transport C q) p') · q'))
             ((transport-· C q p) · (ap (transport C p) q') · p'))
  → (x : T) → C x
```

and, in the absence of the square-over abstraction, this was difficult to use in proofs.

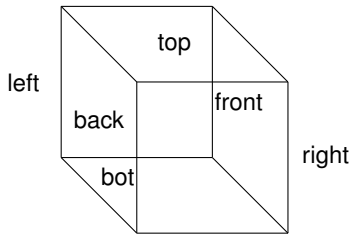
When we prove that the torus is equivalent to the product of two circles, we will define functions c2t and t2c, and then use T-elim to prove $(x : T) \to \mathsf{Path}\ (\mathsf{c2t}\ (\mathsf{t2c}\ x))\ x$. This means that the induction formula C will itself be a path type, so for the f' goal, we will need to give a SquareOver in a path type. Just as a path-over in a path type is a square, a SquareOver in a path type is a 3-dimensional cube.

V. Cubes

Just as we had a type of square insides, dependent on the boundary of a square, we have a type of cubes insides, dependent on eight points, twelve lines, and six faces:

```
data Cube {A : Type} {a000 : A} :
  {a010 a100 a110 a001 a011 a101 a111 : A}
  {p0-0 : Path a000 a010} {p-00 : Path a000 a100}
  {p-10 : Path a010 a110} {p1-0 : Path a100 a110}
  (left : Square p0-0 p-00 p-10 p1-0)
  {p0-1 : Path a001 a011} {p-01 : Path a001 a101}
  {p-11 : Path a011 a111} {p1-1 : Path a101 a111}
  (right : Square p0-1 p-01 p-11 p1-1)
  {p00- : Path a000 a001} {p01- : Path a010 a011}
  {p10- : Path a100 a101} {p11- : Path a110 a111}
  (back : Square p0-0 p00- p01- p0-1)
  (top : Square p-00 p00- p10- p-01)
  (bot: Square p-10 p01- p11- p-11)
  (front : Square p1-0 p10- p11- p1-1)
  → Type where
  id : Cube id id id id id id
```

An element of this type represents the inside of a cube



All the points and lines are implicit arguments. The order of faces is different than for squares: we write Cube left right back top bot front to prioritize the left and right sides, because we will mostly use cubes as an equality between the left and right squares, along the "tube" given by the back and top and bot and front. As usual, we could avoid introducing a new inductive family by instead defining a cube using square induction, to say that when the back, top, bottom, and front are the identity squares, a cube is a path between the left and the right.

Many of the lemmas about cubes are analogous to (and dependent on) those for squares. For example, we can compose two cubes horizontally:

```
_·-cube-h_ : Cube lf rt bk tp bt fr
  → Cube rt rt' bk' tp' bt' fr'
  → Cube lf rt'
         (bk ·-square-h bk') (tp ·-square-h tp')
         (bt ·-square-h bt") (fr ·-square-h fr')
```

We can apply a function to a cube, to get a cube between the action of the function on each face square:

```
ap-cube : (f : A → B) → Cube left right back top bot front
  → Cube (ap-square f left) (ap-square f right)
         (ap-square f back) (ap-square f top)
         (ap-square f bot) (ap-square f front)
```

There are symmetries that switch the dimensions, for example moving the left side to the top, and rearranging and applying symmetries to the other faces as necessary:

```
cube-symmetry-left-to-top :
  Cube left right back top bot front
  → Cube (square-symmetry back) (square-symmetry front)
         (square-symmetry top) left
         right (square-symmetry bot)
```

Next, there are Kan filling operations, which say "any open box has a lid, and an inside." For example, we can create the left side of a cube from the other five sides:

```
fill-cube-left : (right : Square p0-1 p-01 p-11 p1-1)
  (back : Square p0-0 p00- p01- p0-1)
  (top : Square p-00 p00- p10- p-01)
  (bot : Square p-10 p01- p11- p-11)
  (front : Square p1-0 p10- p11- p1-1)
  → Σ [left : Square p0-0 p-00 p-10 p1-0]
      Cube left right back top bot front
```

Cubes arise any time a square type occurs inside a path type, or vice versa. For example, a square-over in a path type is equivalent to a cube (we will only need logical equivalence):

```
SquareOver-= :
  Cube (out-PathOver-= q0-) (out-PathOver-= q1-)
       (out-PathOver-= q-0) (ap-square f fa)
       (ap-square g fa) (out-PathOver-= q-1)
  ↔ SquareOver (λ x → Path (f x) (g x)) fa q0- q-0 q-1 q1-
```

Similarly, a path-over in a square type can be given by a cube:

```
in-PathOver-Square :
  Cube f1 f2
    (out-PathOver-= (apdo p0- δ)) (out-PathOver-= (apdo p-0 δ))
    (out-PathOver-= (apdo p-1 δ)) (out-PathOver-= (apdo p1- δ)))
  → PathOver (λ x → Square (p0- x) (p-0 x) (p-1 x) (p1- x))
             δ f1 f2
```

A typical use of cubes is to "reduce" squares up to reduction on their boundaries. For example, just as there is a path between ap (g o f) p and ap g (ap f p), we would like ap-square (g o f) s to be equal to ap-square g (ap-square f s). However, the sides of these two squares differ by the reductions on paths. Thus, we can phrase this reduction as a cube, which equates these two squares along the reductions between their sides:

```
ap-o : Square (ap (g o f) p) id id (ap g (ap f p))
ap-square-o : (s : Square l t b r) →
  Cube (ap-square (g o f) s) (ap-square g (ap-square f s))
       (ap-o g f l) (ap-o g f t) (ap-o g f b) (ap-o g f r)
```

Similarly, the propositional reduction rules for torus recursion T-rec can be phrased as squares and cubes.

```
βp/rec : Square (ap (T-rec a' p' q' f') p) id id p'
βq/rec : Square (ap (T-rec a' p' q' f') q) id id q'
βf/rec : Cube (ap-square (T-rec a' p' q' f') f) f'
              βp/rec βq/rec βq/rec βp/rec
```

For p and q we have the usual paths, rephrased as horizontally degenerate cubes. For f, we have a cube between the application of T-rec to the face constructor and the provided image f', along the previous reduction rules.

VI. Torus ≃ Product of Two Circles

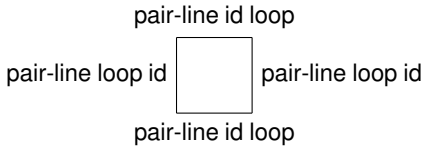
The main idea of the correspondence between T and $S^1 \times S^1$ is that a loop in the first component of the pair corresponds to p, and a loop in the second component to q.

A. Torus to circles

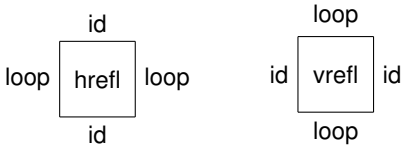
The map from the torus to the circles uses torus recursion:

```
t2c : T → S¹ × S¹
t2c = T-rec (base, base)
            (pair-line loop id)
            (pair-line id loop)
            (pair-square hrefl-square vrefl-square)
```

This function sends the point a on the torus to the point (base, base). As the images of p and q, we need two elements of Path {$S^1 \times S^1$} (base, base) (base, base). For p, we pair together loop on the left with reflexivity on the right; for q, we pair reflexivity on the left with loop on the right. Next, as the image of f, we need a square



which is given by pairing together the squares



B. Circles to torus

We define c2t : $S^1 \times S^1 \to T$ to be the uncurrying of a map c2t' : $S^1 \to S^1 \to T$. For intuition, to be inverse to t2c, we would like that c2t' behaves as follows when applied (using appropriate ap's, which we omit here) to constructors:

```
c2t' base base = a
c2t' base loop = q
c2t' loop base = p
c2t' loop loop = f
```

We code matching on two arguments as nested circle eliminations, which roughly have the form "S^1-elim (S^1-elim a q) (S^1-elim p f)." That is, when the first argument is base, we get a function that sends base to a and loop to q; when the first argument is loop, we get a function that sends base to p and loop to f. We now make this precise:

```
c2t-square-and-cube :
  Σ [s : Square p (ap (S¹-rec a q) loop)
                  (ap (S¹-rec a q) loop) p]
  Cube s f hrefl-square βsquare βsquare hrefl-square
c2t-square-and-cube = (fill-cube-left _ _ _ _ _)
c2t' : S¹ → S¹ → T
c2t' = S¹-rec (S¹-rec a q)
              (λ≃ (S¹-elimo _ p
                (in-PathOver-= (fst c2t-square-and-cube))))
```

The match on the first argument is a simply-typed circle recursion, so we need a point and a loop in $S^1 \to T$. The point is again defined by circle recursion, sending base to a and loop to q. The loop must be a Path $\{S^1 \to T\}$ (S^1-rec a q) (S^1-rec a q), which, by (homogeneous) function extensionality, is equivalent to (x : S^1) → Path (S^1-rec a q x) (S^1-rec a q x). Using circle elimination, we need a Path $\{T\}$ a a (because S^1-rec a q base ≡ a), which we take to be p, and then a PathOver (λ x → Path (S^1-rec a q x) (S^1-rec a q x)) loop p p. Applying PathOver-=-eqv to reduce a path-over in a path type to a square, we need a square s with the type given as the first component of c2t-square-and-cube. But (ap (S^1-rec a q) loop) reduces (propositionally) to q, and the square we want is the f constructor for the torus composed with this propositional reduction. Writing

```
βsquare : Square (ap (S¹-rec a q) loop) id id q
```

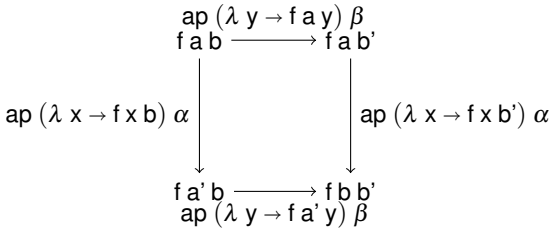
for the reduction, it turns out to be convenient to obtain the necessary square using Kan filling, because then we also get a cube relating s to f along the reduction (and along reflexivity for the p positions).

Reduction rules: Next, we prove propositional reduction rules for c2t' on the constructors, elaborating on the informal versions given above. On points, c2t' base base is indeed judgementally equal to a. The more precise version of the next two equations is

```
ap (λ x → c2t' x base) loop = p
ap (λ y → c2t' base y) loop = q
```

Proving these equations will involve reducing circle eliminations on the loop constructor, so they will only hold propositionally.

For the final equation, we first need to clarify how to apply c2t' to the loop in both positions. For any curried function f : A → B → C and paths α : Path $\{A\}$ a a' and β : Path $\{B\}$ b b', there is a square



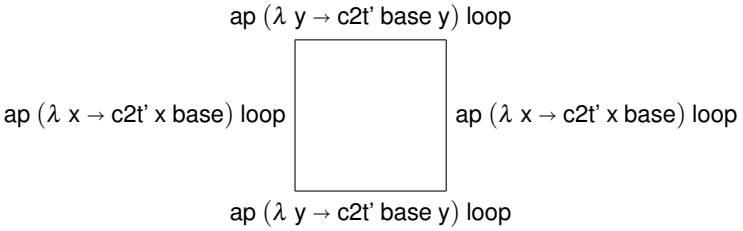
defined by the iterated application of f to α and β:

```
apdo-ap f α β =
  out-PathOver-= (apdo (λ y → ap (λ x → f x y) α) β)
```

To see that this type checks, for any y, the term ap (λ x → f x y) α has type Path (f a y) (f a' y), so applying this to β gives a

```
PathOver (λ y → Path (f a y) (f a' y)) β
         (ap (λ x → f x b) α) (ap (λ x → f x b') α)
```

and turning this path-over into a square gives the result. The specific case of apdo-ap c2t' loop loop is a square



and note that the path reduction rules above equate the sides of this square the sides of f.

Thus, the desired propositional reduction rules for c2t' are

```
c2t'-β : Σ [βl2 : Square (ap (λ y → c2t' base y) loop) id id q]
         Σ [βl1 : Square (ap (λ x → c2t' x base) loop) id id p]
           Cube (apdo-ap c2t' loop loop) f βl1 βl2 βl2 βl1
```

This says that we want two squares for "c2t' base loop" and "c2t' loop base", and then a cube along these squares for the reduction on "c2t' loop loop". It will be important below that this cube's top equals its bottom and front equals its back.

We could proceed by defining βl2 (as βsquare from above, for example) and βl1 and then trying to find an appropriate cube for the third component. However, there is a simpler way: The only property we need of the βl1 and βl2 squares is that they exist and fit into the cube above. Moreover, it turns out that we can define the cube goal in such a way that it determines suitable βl1 and βl2! In Agda, unification fills in βl2 and βl1 from the definition of the cube.

To define a cube whose left side is (apdo-ap c2t' loop loop) and whose right side is f, we compose six cubes horizontally, whose middle sides are as follows:

```
    apdo-ap c2t' loop loop
≡   out-PathOver=
      (apdo (λ y → (ap (λ x → c2t' x y) loop)) loop)
□= out-PathOver=
      (apdo (λ y → (ap (λ f → f y) (ap c2t' loop))) loop)
□= out-PathOver= (apdo (λ y → (ap (λ f → f y)
      (λ≃ (S¹-elimo _ p
        (in-PathOver-= (fst c2t-square-and-cube)))))) loop)
□= out-PathOver= (apdo
      (S¹-elimo _ p
        (in-PathOver-= (fst c2t-square-and-cube))) loop)
□= out-PathOver= (in-PathOver-= (fst c2t-square-and-cube))
□= fst c2t-square-and-cube
□= f
```

We think of this as an equation chain between these eight squares, but the proof of each step is really a cube, rather than a homogeneous path (we write □= to emphasize this). In order, the justifications for the steps are (0) by definition, (1) un-fusing ap (λ x → c2t' x y) loop to ap (λ f → f y) (ap c2t' loop), (2) reducing c2t' (which is a circle recursion) on loop, (3) reducing ap (λ f → f y) on a function extensionality, (4) reducing S¹-elimo on the loop, (5) collapsing the two sides of the PathOver-= equivalence, and (6) using snd c2t-square-and-cube. Thus, we do what looks like an equational proof that the square (apdo-ap c2t' loop loop) "equals" the square f, but each step may also contribute to the back-top-bottom-front tube that connects the boundaries of these two squares. For example, step (6) using snd c2t-square-and-cube contributes βsquare on the top and bottom. Because each of the cubes used in steps (1) through (6) has the property that its front is equal to its back and its top is equal to its bottom, βl1 and βl2 can be defined to be the composites of these sides, and the overall cube has the required boundary.

C. Torus to circles to torus

Next, we need to show

```
t2c2t : (x : T) → Path (c2t (t2c x)) x
```

We proceed by torus induction. In the case for a, the result holds definitionally. After a bit of massaging (using PathOver-= to mediate between a path-over in a path type and a square; collapsing round-trips of the PathOver-= equivalence; using in-SquareOver-= to create a square-over from a cube; using cube-symmetry-left-to-top to put the important faces on the left-right sides), the remaining goals are

```
p-case : Square (ap (λ z → c2t (t2c z)) p) id
                id (ap (λ z → z) p)
q-case : Square (ap (λ z → c2t (t2c z)) q) id
                id (ap (λ z → z) q)
f-case : Cube (ap-square (λ z → c2t (t2c z)) f)
              (ap-square (λ z → z) f)
              p-case q-case q-case p-case
```

This simply says that we need to check the composite on each of the constructors, where the case for f is a cube along the cases for p and q. Once again, we can solve the f case and let that determine the p and q cases. The f case is a horizontal composition of cubes whose middle faces are as follows:

```
    ap-square (λ z → c2t (t2c z)) f
□= ap-square c2t (ap-square t2c f)
□= ap-square c2t (pair-square hrefl-square vrefl-square)
□= apdo-ap c2t' loop loop
□= f
□= ap-square (λ z → z) f
```

That is, we (1) un-fuse the ap-square using ap-square-o, (2) reduce t2c (defined by torus recursion) on the f constructor, (3) change an ap-square into an apdo-ap, (4) use the c2t'-β cube for apdo-ap c2t' loop loop, and (5) expand ap-square with the identity function. The proof is again given as a series of horizontal composites of cubes, and the boundary of this cube solves the p and q cases:

```
p-case = _
q-case = _
f-case = ap-square-o c2t t2c f ·-cube-h
         ap-cube c2t βfcube ·-cube-h
         apdo-ap-cube-hv c2t' loop loop ·-cube-h
         snd (snd c2t'-β) ·-cube-h
         ap-square-id! f
```

In step (3), we use a cube

```
apdo-ap-cube-hv : Cube
  (ap-square (uncurry f) (pair-square (hrefl p) (vrefl q)))
  (apdo-ap f p q)
  (ap-id-snd-square f p) (ap-id-fst-square f q)
  (ap-id-fst-square f q) (ap-id-snd-square f p)
```

This lemma is an analogue of currying for applying a function to a pair of paths: apdo-ap f p q (which is like "f p q") is the same as square-applying uncurry f to the pair of p (as a horizontally trivial square) and q (as a vertically trivial square). The remaining sides equate ap (uncurry f) (pair-line id q) with ap (f a) q and similarly for the second component.

In step (5), we use a Cube s (ap-square (λ x → x) s) ... whose remaining sides are the paths between ap (λ x → x) p and p.

D. Circles to torus to circles

Finally, we check the other composite:

```
c2t2c : (x y : S¹) → Path (t2c (c2t' x y)) (x, y)
```

The outer structure of the proof consists of nested circle inductions, together with uses of function extensionality,

PathOver-= and in-PathOver-Square, some massaging (reducing an S^1-elimo on loop and a round-trip of PathOver=), and (for convenience) a use of cube-symmetry-left-to-top. After applying these lemmas, the remaining goals are

```
loop1-case : Square (ap (λ x → t2c (c2t' x base)) loop) id
                    id (ap (λ x → x, base) loop)
loop2-case : Square (ap (λ y → t2c (c2t' base y)) loop) id
                    id (ap (λ y → base, y) loop)
looploop-case :
  Cube (apdo-ap (λ x y → t2c (c2t' x y)) loop loop)
       (apdo-ap (λ x y → x, y) loop loop)
       loop1-case loop2-case loop2-case loop1-case
```

That is, we need to check that the theorem holds for when the composite is applied to loop base and base loop and loop loop. Once again, we can solve the loop loop case and let that determine the others. The reduction in question is a horizontal composite of cubes with the following middle faces

```
    apdo-ap (λ x y → t2c (c2t' x y)) loop loop
□= ap-square t2c (apdo-ap c2t' loop loop)
□= ap-square t2c f
□= pair-square hrefl-square vrefl-square
□= ap-square (λ x → x)
      (pair-square hrefl-square vrefl-square)
□= apdo-ap _, _ loop loop
```

The justifications are (1) un-fuse the apdo-ap of a composition of a functions (a lemma analogous to ap-square-o), (2) use c2t'-β from above, (3) reduce the t2c torus elimination on f, (4) expand ap-square $(\lambda\, x \to x)$ and (5) use apdo-ap-cube-hv to mediate between an ap-square and a apdo-ap. The proof is the composite of these five cubes, and loop1-case and loop2-case are inferred by unification:

```
loop1-case = _
loop2-case = _
looploop-case =
  apdo-ap-o t2c c2t' loop loop ·-cube-h
  ap-cube t2c (snd (snd c2t'-β)) ·-cube-h
  βfcube ·-cube-h
  ap-square-id! _ ·-cube-h
  apdo-ap-cube-hv _, _ loop loop
```

VII. The 3×3 lemma

In this section, we present another example of a problem whose formalization in Agda has been made possible using cubical ideas. This problem, called the 3×3 lemma, has been used in particular in the construction of the Hopf fibration and in the computation of $\pi_4\, S^3$.

We first need to define the notion of pushout: given three types A, B and C and two maps $f : C \to A$ and $g : C \to B$

$$A \xleftarrow{f} C \xrightarrow{g} B$$

their pushout is the higher inductive type Pushout f g with the following three constructors:

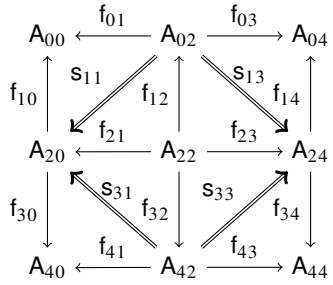
```
inl : A → Pushout f g
inr : B → Pushout f g
push : (c : C) → Path (inl (f c)) (inr (g c))
```

A pushout is like a sum type A + B, except certain instances of inl and inr are "glued" together (see [29]). The corresponding elimination rule is:

```
Pushout-elimo : {A B C : Type} {f : C → A} {g : C → B}
  (P : Pushout f g → Type)
  (l : (a : A) → P (inl a)) (r : (b : B) → P (inr b))
  (p : (c : C) → PathOver P (push c) (l (f c)) (r (g c)))
  (x : Pushout f g) → P x
```

As usual, we have definitional reduction rules on points and propositional reduction rules on paths.

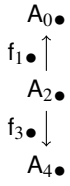
The problem is now the following. Given nine types A_{ij}, twelve maps f_{ij} and four equalities s_{ij} as follows:



where the double arrows mean that we have for instance

$$s_{11} : (x : A_{22}) \to \mathsf{Path}\ (f_{01}\ (f_{12}\ x))\ (f_{10}\ (f_{21}\ x))$$

we want to compute its "two-dimensional pushout". There are at least two ways to do that. We can either first compute the pushout of each of the three lines, which fit together in a diagram as follows:



and then compute the pushout of the resulting diagram, which we will denote by $A_{■●}$. But we can also first compute the pushouts of the three columns:

$$A_{\bullet 0} \xleftarrow{f_{\bullet 1}} A_{\bullet 2} \xrightarrow{f_{\bullet 3}} A_{\bullet 4}$$

and then the pushout $A_{●■}$ of the resulting diagram. The 3×3 lemma states that the two types $A_{■●}$ and $A_{●■}$ are equivalent.

First, we explain how to construct the map $f_{1●}$ (the maps $f_{3●}$, $f_{●1}$ and $f_{●3}$ are defined in a similar way). To make things clearer, until the end of this section we will annotate the constructors inl, inr and push by the $_{ij}$ corresponding to their return type.

The map $f_{1●} : A_{2●} \to A_{0●}$ is defined using the recursion rule of the pushout. It sends the point $\mathsf{inl}_{2●}\ x$ to $\mathsf{inl}_{0●}\ (f_{10}\ x)$. It sends the point $\mathsf{inr}_{2●}\ x$ to $\mathsf{inr}_{0●}\ (f_{14}\ x)$. Finally, the path $\mathsf{push}_{2●}\ x$ (which goes from $\mathsf{inl}_{2●}\ (f_{21}\ x)$ to $\mathsf{inr}_{2●}\ (f_{23}\ x)$) has to be sent to a path from $\mathsf{inl}_{0●}\ (f_{10}\ (f_{21}\ x))$ to $\mathsf{inr}_{0●}\ (f_{14}\ (f_{23}\ x))$. The path $\mathsf{push}_{0●}\ (f_{12}\ x)$ doesn't work directly, because it goes from

$\mathsf{inl}_{0\bullet}\,(\mathsf{f}_{01}\,(\mathsf{f}_{12}\,\mathsf{x}))$ to $\mathsf{inr}_{0\bullet}\,(\mathsf{f}_{03}\,(\mathsf{f}_{12}\,\mathsf{x}))$, but we can compose to the left and to the right with the equalities s_{11} and s_{13} to make the endpoints match. Given the direction of the arrows, this can be done conveniently using fill-square-right or an analogous fill-square-left operation, which produces the left side of a square from the top and right and bottom. We notate these operations as Kan-right and Kan-left in this section.

Hence, the complete Agda definition of $\mathsf{f}_{1\bullet}$ is the following:

```
f1• : A2• → A0•
f1• = Pushout-rec (λ x → inl0• (f10 x))
                  (λ x → inr0• (f14 x))
                  (λ y → Kan-right (ap inl0• (s11 y))
                                   (ap inr0• (s13 y))
                                   (push0• (f12 y)))
```

To prove that $\mathsf{A}_{\blacksquare\bullet}\simeq\mathsf{A}_{\bullet\blacksquare}$, we construct two maps back and forth and prove that the two compositions are the identity. Even though the statement of the problem involves only higher inductive types (pushouts) with only point and path constructors, we are nesting them by considering a pushout of pushouts, so we will have to deal with squares. The construction of the maps and of the equalities go by double induction on the pushouts, hence there will be essentially nine steps each time: four for the points coming from A_{00}, A_{40}, A_{04} and A_{44}, four for the lines coming from A_{02}, A_{42}, A_{20}, A_{40}, and one for the squares coming from A_{22}.

It is worth noting that something nontrivial happens in the case of the squares coming from A_{22}. Indeed, the one-dimensional constructor of $\mathsf{A}_{\blacksquare\bullet}$ is

$$\mathsf{push}_{\blacksquare\bullet} : (\mathsf{y} : \mathsf{A}_{2\bullet}) \to \mathsf{Path}\,(\mathsf{inl}_{\blacksquare\bullet}\,(\mathsf{f}_{1\bullet}\,\mathsf{y}))\,(\mathsf{inr}_{\blacksquare\bullet}\,(\mathsf{f}_{3\bullet}\,\mathsf{y}))$$

and the one-dimensional constructor of $\mathsf{A}_{2\bullet}$ is

$$\mathsf{push}_{2\bullet} : (\mathsf{x} : \mathsf{A}_{22}) \to \mathsf{Path}\,(\mathsf{inl}_{2\bullet}\,(\mathsf{f}_{21}\,\mathsf{x}))\,(\mathsf{inr}_{2\bullet}\,(\mathsf{f}_{23}\,\mathsf{x}))$$

hence the square in $\mathsf{A}_{\blacksquare\bullet}$ corresponding to a point $\mathsf{x} : \mathsf{A}_{22}$ is the term $\mathsf{apdo}\,\mathsf{push}_{\blacksquare\bullet}\,(\mathsf{push}_{2\bullet}\,\mathsf{x})$ which is of type

```
PathOver (λ y → Path (inl■• (f1• y)) (inr■• (f3• y))) (push2• x)
         (push■• (inl2• (f21 x))) (push■• (inr2• (f23 x)))
```

Using PathOver-=-eqv we get a square of type

```
Square (push■• (inl2• (f21 x)))
       (ap (λ y → inl■• (f1• y)) (push2• x))
       (ap (λ y → inr■• (f3• y)) (push2• x))
       (push■• (inr2• (f23 x)))
```

Using whisker-square, we can reduce the top and bottom. For the top: unfusing the ap gives $\mathsf{ap}\ \mathsf{inl}_{\blacksquare\bullet}\ (\mathsf{ap}\ \mathsf{f}_{1\bullet}\ (\mathsf{push}_{2\bullet}\ \mathsf{x}))$, and reducing the pushout recursion on push goes from $\mathsf{ap}\ \mathsf{f}_{1\bullet}\ (\mathsf{push}_{2\bullet}\ \mathsf{x})$ to $\mathsf{Kan\text{-}right}\ (\mathsf{ap}\ \mathsf{inl}_{0\bullet}\ (\mathsf{s}_{11}\ \mathsf{y}))\ (\mathsf{ap}\ \mathsf{inr}_{0\bullet}\ (\mathsf{s}_{13}\ \mathsf{y}))\ (\mathsf{push}_{0\bullet}\ (\mathsf{f}_{12}\ \mathsf{x}))$. Finally, using the fact that Kan operations commute with ap, this is the same as

```
Kan-right (ap inl■• (ap inl0• (s11 x)))
          (ap inl■• (ap inr0• (s13 x)))
          (ap inl■• (push0• (f12 x)))
```

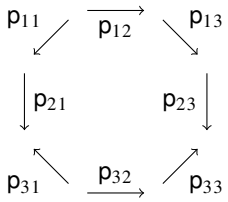
Hence, the square in $\mathsf{A}_{\blacksquare\bullet}$ we have is of type

```
Square (push■• (inl2• (f21 x)))
       (Kan-right (ap inl■• (ap inl0• (s11 x)))
                  (ap inl■• (ap inr0• (s13 x)))
                  (ap inl■• (push0• (f12 x))))
       (Kan-right (ap inr■• (ap inl4• (s31 x)))
                  (ap inr■• (ap inr4• (s33 x)))
                  (ap inr■• (push4• (f32 x))))
       (push■• (inr2• (f23 x)))
```

which we will shorten to

$$\mathsf{Square}\ \mathsf{p}_{21}\ (\mathsf{Kan\text{-}right}\ \mathsf{p}_{11}\ \mathsf{p}_{13}\ \mathsf{p}_{12})\ (\mathsf{Kan\text{-}right}\ \mathsf{p}_{31}\ \mathsf{p}_{33}\ \mathsf{p}_{32})\ \mathsf{p}_{23}$$

Note that the p_{ij} fit in the following diagram:



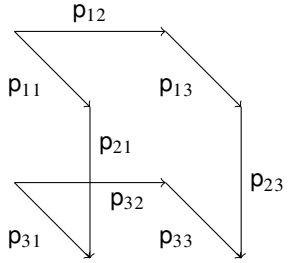
However, when constructing the map $\mathsf{A}_{\bullet\blacksquare} \to \mathsf{A}_{\blacksquare\bullet}$ using a double induction on the pushouts, we can check that what we need is a square in $\mathsf{A}_{\blacksquare\bullet}$ of type

$$\mathsf{Square}\ \mathsf{p}_{12}\ (\mathsf{Kan\text{-}left}\ \mathsf{p}_{11}\ \mathsf{p}_{31}\ \mathsf{p}_{21})\ (\mathsf{Kan\text{-}left}\ \mathsf{p}_{13}\ \mathsf{p}_{33}\ \mathsf{p}_{23})\ \mathsf{p}_{32}$$

Hence to define the map $\mathsf{A}_{\bullet\blacksquare} \to \mathsf{A}_{\blacksquare\bullet}$, we need a map between those two square types, and in order to do the complete proof that $\mathsf{A}_{\bullet\blacksquare} \simeq \mathsf{A}_{\blacksquare\bullet}$ we will need an equivalence between them.

One way to do it is as follows: we first do a path induction on p_{11}, p_{13}, p_{31} and p_{33} and use the fact that for every p there is a path between Kan-right id id p and p and a path between Kan-left id id p and p, and then we can apply square-symmetry-eqv.

Another way to see it is as a three-dimensional cube:



If we fill the left, right, top, and bottom, the front and back faces are exactly the two square types that we want to prove equivalent. Thus, it suffices to observe that, by the Kan filling operations, any four faces of a cube that form a tube determine an equivalence between the other two opposing faces.

VIII. Related and Future Work

The various definitions of path-over, and their equivalence, were discussed during the IAS special year (see [29, Remark 6.3.2]). The idea to use inductive families to define shapes other than lines was explored in a simplicial setting by Coquand (e.g. a triangle in a type). At the time, it was considered an open question whether it was helpful to consider shapes other than the homogeneous globes that arise by iterating the

identity type. In this paper, we have argued that there are benefits to working with cube and cube-over types. Higher cubes and cube-overs arise naturally when higher inductive types have two-dimensional (or higher) constructors (like the torus), when higher inductive types are nested (a pushout of pushouts, like in the three-by-three lemma), and when eliminations on higher inductive types are nested (like when mapping out of two circles). Developing lemmas in terms of these abstractions has enabled the new formalizations described in this paper.

At present, we have developed cube types in "offshoots" of Agda homotopy type theory libraries; an interesting direction for future work would be to revise the libraries to use cubes throughout, and to revisit existing results to see if they can be simplified. Another direction for future work is to define higher cube and cube-over types in a dimension-polymorphic way, rather than implementing each dimension in isolation, as we have done here.

Our work is carried out in the setting of dependent type theory with axioms for univalence and higher inductives, but there have also been extensions of dependent type theory based on cubical ideas. These include a type theory with a computational interpretation of parametricity [4], and new cubical type theories [1, 7, 11, 25] inspired by the cubical sets model of homotopy type theory [5]. We have begun to translate the examples discussed here to cubical type theory, to compare these new type theories against working axiomatically. With Cohen, Coquand, Huber, and Mörtberg, using their implementation,[3] we have coded a special case of the 3x3 lemma that is used in $\pi_4(S^3)$, and found that the amount of memory needed to type check is greatly reduced, because new definitional equalities keep the proof terms smaller. To construct models of these new cubical type theories, it would be convenient to translate back to axiomatic homotopy type theory; the low-dimensional libraries and examples presented here provide some empirical evidence that this might be possible.

Acknowledgments We thank Joseph Lee, who attempted to prove the torus-circles equivalence with the first author in summer 2012. We thank the participants of the IAS year on univalent foundations and the Paris trimester on semantics of proofs and certified mathematics for many helpful discussions. We thank the anonymous reviewers for helpful feedback.

References

[1] T. Altenkirch and A. Kaposi. A syntax for cubical type theory. Available from http://www.cs.nott.ac.uk/~txa/publ/ctt.pdf, 2014.

[2] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper. Homotopical patch theory. In *ACM SIGPLAN International Conference on Functional Programming*, 2014.

[3] S. Awodey and M. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 2009.

[4] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *IEEE Symposium on Logic in Computer Science*, 2012.

[5] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. Preprint, September 2013.

[6] R. Brown and P. J. Higgins. On the algebra of cubes. *Journal of Pure and Applied Algebra*, 21(3):233–260, 1981.

[7] G. Brunerie and D. R. Licata. A cubical type theory. Talk at Oxford Workshop on Homotopy Type Theory, November 2014.

[8] E. Cavallo. The mayer-vietoris sequence in HoTT. Talk at Oxford Workshop on Homotopy Type Theory, November 2014.

[9] J. Cockx, D. Devriese, and F. Piessens. Pattern matching without K. In *ACM SIGPLAN International Conference on Functional Programming*, pages 257–268. ACM, September 2014.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. INRIA, 2009. Available from http://coq.inria.fr/.

[11] T. Coquand. Variations on cubical sets. Available from http://www.cse.chalmers.se/~coquand/comp.pdf, 2014.

[12] K.-B. H. (Favonia). Covering spaces in homotopy type theory. Talk at TYPES, May 2014.

[13] N. Gambino and R. Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(3):94–109, 2008.

[14] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[15] D. M. Kan. Abstract homotopy i. In *Proceedings of National Academy of Sciences USA*, volume 41, pages 1092–1096, 1955.

[16] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. arXiv:1211.2851, 2012.

[17] D. R. Licata and G. Brunerie. $\pi_n(s^n)$ in homotopy type theory. In *Certified Programs and Proofs*, 2013.

[18] D. R. Licata and E. Finster. Eilenberg-maclane spaces in homotopy type theory. In *IEEE Symposium on Logic in Computer Science*, 2014.

[19] D. R. Licata and M. Shulman. Calculating the fundamental group of the cirlce in homotopy type theory. In *IEEE Symposium on Logic in Computer Science*, 2013.

[20] P. L. Lumsdaine. Weak ω-categories from intensional type theory. In *International Conference on Typed Lambda Calculi and Applications*, 2009.

[21] P. L. Lumsdaine. Higher inductive types: a tour of the menagerie. http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/, April 2011.

[22] P. L. Lumsdaine and M. Shulman. Higher inductive types. In preparation, 2013.

[23] C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000.

[24] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[25] A. Polonsky. Internalization of extensional equality. Available from http://arxiv.org/abs/1401.1148, 2014.

[26] M. Shulman. Homotopy type theory VI: higher inductive types. http://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html, April 2011.

[27] M. Shulman. Univalence for inverse diagrams, oplax limits, and gluing, and homotopy canonicity. arXiv:1203.3253, 2013.

[28] K. Sojakova. The torus as a product of two circles in homotopy type theory. Technical Report CMU-CS-15-105, Carnegie Mellon University, 2015. Available from http://reports-archive.adm.cs.cmu.edu/.

[29] The Univalent Foundations Program, Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations Of Mathematics*. Available from homotopytypetheory.org/book, 2013.

[30] B. van den Berg and R. Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[31] V. Voevodsky. A very short note on homotopy λ-calculus. http://www.math.ias.edu/vladimir/files/2006_09_Hlambda.pdf, September 2006.

[32] M. A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008.

[3] https://github.com/simhu/cubical/tree/connections