Excursions in Cubical Type Theory

Siva Somayyajula

December 4, 2017

Contents

1 Motivation

In recent years, the *univalent foundations* program has aimed to develop a foundations of constructive mathematics with an enriched notion of equality. Technicality aside, recall that your average intensional type theory may only express intensional equalities between mathematical objects. Unlike extensional type theories, they may not express, for example, the extensional equality of functions. However, an intensional type theory with the *univalence axiom* may identify equivalent types as equal for an appropriate definition of equivalence. Consequently, function extensionality becomes a theorem in this system, and many results that require a coarser notion of equality become provable. As Jon Sterling said in an episode of the Type Theory Podcast, "in a sense, [this] is more extensional than extensional type theory," because the range of expressible equalities based on the behavior of objects is widened. In fact, univalence is expressly *inconsistent* with extensional type theory, so it seems that an intensional and univalent type theory poses many benefits over existing systems.

Homotopy type theory (HoTT), the first such type theory developed by this program, extends Martin-Löf type theory with the univalence axiom and other constructs to develop synthetic proofs of results in homotopy theory. Indeed, given our above exposition, HoTT seems to be the type theory to end all type theories. However, its critical weakness is that univalence lacks a computation rule. This is not an issue for those who intend to leverage HoTT for classical proofs, but what of programmers and constructivists? There are many situations where one may want to use univalence to seamlessly switch between different views of the same abstract type when writing a software application, or actually run one's proofs. While these programs and proofs would enjoy various type safety guarantees in HoTT, they would also just get *stuck*. Enter *cubical type theory* (CuTT), which is advertised as providing a "constructive interpretation of the univalence axiom;" that is, univalence gets a computation rule. We will give a brief introduction to CuTT and demonstrate its power and computational benefits in a variety of case studies motivated by concerns in mathematics and software engineering, all in Agda. Note that proofs pulled form external sources are cited either in the source code or in this paper; and, unless specified, definitions are taken from [5].

2 Cubical Type Theory

CuTT starts with intuitionistic type theory and then introduces a set of primitive types that ultimately capture the topological notion of an n-dimensional cube. We will begin with the type of *intervals*.

2.1 Intervals and Paths

Consider the unit interval $I = [0, 1]$ and some basic facts about it. In particular, I is bounded by least and greatest elements 0 and 1, respectively. We also have trivial equalities $1 - 0 = 1$ and $1 - 1 = 0$. Perhaps more interestingly, for all $r, s \in I$:

$$1 - \max(r, s) = \min(1 - r, 1 - s)$$
$$1 - \min(r, s) = \max(1 - r, 1 - s)$$

Together, these properties make $(I, \max, \min, 0, 1)$ a bounded distributive lattice and $r \mapsto 1 - r$ a De Morgan involution. In other words, $(I, \max, \min, 0, 1, r \mapsto 1 - r)$ is a De Morgan algebra. Thus, it is appropriate to refer to max as $\vee$, min as $\wedge$, and $r \mapsto 1 - r$ as $\sim$.

Now, fix a countable set of names N. We obtain a *homotopical* (that is, considering only the boundary elements) notion of I via the *set* `I` defined by the following grammar and quotiented by the aforementioned properties. Note that the inclusion of only `i0` and `i1` does not allow us to prove $(\sim r) \wedge r =$ `i0` or $(1 - r) \vee r =$ `i1`—`I` is faithful to the original definition of I.

$$r, s := \texttt{i0} \mid \texttt{i1} \mid i \in N \mid \sim r \mid r \wedge s \mid r \vee s$$

Note that CuTT defines `I` as a set, as opposed to a type, for technical reasons [4]. Now, consider that topological paths over a space A are defined as continuous functions from the I to A. We may recover a similar definition in CuTT via the *type* `Path`. However, we cannot define `Path` $:=$ `I` $\to$ `A` directly since `I` is not a type. Once again though, we are only concerned with the "boundary behavior" of a topological path f—the elements $f(0)$ and $f(1)$. As a result, we define a type `Path` $t\ u$ to be the type of paths between endpoints t, u : `A`. As a result, one may introduce a *path abstraction* $\lambda\ i \to e$: `Path` $e[\texttt{i0}/i]\ e[\texttt{i1}/i]$ where $i \in N$. Path abstractions and applications have the expected judgmental equalities with respect to lambda abstractions and applications, except our attention is restricted to `I` [1].

Perhaps that after dealing with this level of abstraction (pun intended), we deserve a few examples where we reason about the basic properties of paths.

Theorem 2.1 (Constant/identity path)**.** *If* x : A*, we have*

`idp` : `Path` $x\ x$

Proof. We want a path abstraction that is invariant on an element in `I`, so:

`idp` $\_ = x$

□

Theorem 2.2 (Path inversion)**.** $\_^{-1}$: `Path` $x\ y \to$ `Path` $y\ x$

Proof. Given a path p from x to y, $p\ (\sim$ `i0`$) = p$ `i1` $= y$ and $p\ (\sim$ `i1`$) = p$ `i0` $= x$, so:

$p^{-1} = \lambda\ i \to p\ (\sim i)$

□

Theorem 2.3 (Action on paths)**.** *If* f : `A` → `B` *and* x, y : `A`*, we have*

`ap` : `Path` $x\ y \to$ `Path` $(f\ x)\ (f\ y)$

Proof. Given a path p from x to y, $f\ (p$ `i0`$) = f\ x$ and $f\ (p$ `i1`$) = f\ y$, so:

`ap` $p\ i = f\ (p\ i)$

□

Paths look suspiciously like the identity type—in particular, these examples correspond to reflexivity, symmetry, and function substitutivity, respectively. Indeed, we will see how we may leverage paths to represent equalities; but first, we will see why CuTT is cubical. If P_n is a type family parameterized by n intervals, it induces the an n-dimensional cube; see the top of page 6 in [1]. The manipulation of paths will allow us to do mathematics and programming.

2.2 Modeling Equality

We claim that we can model equality á la Leibniz using paths, so let's introduce a convenient alias.

`_≡_` : $\forall\ \{\ell\}\ \{A$: `Set` $\ell\} \to A \to A \to$ `Set` ℓ

`_≡_` = `Path`

First, we will see how paths help us model extensional equality.

Definition 1 (Homotopy)**.** The space of *homotopies* between two functions is defined as follows.

`_∼_` : $(f\ g : (x : A) \to P\ x) \to$ `Set` $\_$

$f \sim g = \forall\ x \to f\ x \equiv g\ x$

Though topologically inspired, it is useful to think of the inhabitants of this type as proofs that two such functions are extensionally equal. As a result, we can prove the obvious: equal functions are *homotopic*.

Theorem 2.4. `app≡` : $f \equiv g \to f \sim g$

Proof. Given $p : f \equiv g$, we can produce a path $f\ x \equiv g\ x$ by the term $\lambda\ i \to p\ i\ x$. One can validate this result by doing the same case analysis on i as in the previous proofs.

`app≡` $p\ x\ i = p\ i\ x$

□

And, as desired, we can prove function extensionality.

Theorem 2.5 (Function extensionality)**.** `λ≡` : $f \sim g \to f \equiv g$

Proof. This proof is subtle—we need to construct a path abstraction that returns f on `i0` and g on `i1`. Given the homotopy h, $h\ x$ will give $f\ x \equiv g\ x$, so applying i will give $f\ x$ at `i0` and $g\ x$ at `i1`. By η-conversion, we have given f at `i0` and g at `i1`, as desired.

`λ≡` $h\ i\ x = h\ x\ i$

□

Furthermore, it is easy to see that $\texttt{app}\equiv$ and $\lambda\equiv$ are definitional inverses. It follows that the space of homotopies is equivalent to the space of paths on functions. Now, back to our regularly scheduled programming. Recall what is necessary for an equality to be Leibnizian [6]; it must satisfy:

1. Reflexivity
2. Function substitutivity
3. Indiscernibility of identicals

We have already shown (1) and (2), so it remains to show (3). First, we must define *type coercion*: given that $A \equiv B$, we can convert $x : A$ to a term of type B. It is not immediately clear that such an operation exists given the current machinery we have for paths. However, CuTT comes with a path composition primitive $\texttt{comp}$ that "completes" a partially specified cube.

Theorem 2.6 (Type coercion)**.** $\texttt{coe} : A \equiv B \to A \to B$

Proof. We realize a 0-dimensional or $\texttt{empty}$ cube (that is, a single point) by partially specifying it at $p\ \texttt{i0} = A$ with the given $x : A$. Thus, we get an element of type $p\ \texttt{i1} = B$.

$\texttt{coe}\ p\ x = \texttt{comp}\ (\lambda\ i \to p\ i)\ \_\ (\lambda\ \_ \to \texttt{empty})\ x$

□

Furthermore, we can prove that coercion on the constant path has no computational effect, as desired. This is also a *propositional computation rule* because it dictates, up to propositional equality represented by paths, how $\texttt{coe}$ behaves on certain inputs.

Theorem 2.7 (Degenerate coercion)**.** $\texttt{coe-}\beta : (x : A) \to \texttt{coe}\ \texttt{idp}\ x \equiv x$

Proof. Consider the square in figure 2.2 we would like to complete where $x \xrightarrow{p\ i} y$ denotes $p : \texttt{Path}\ x\ y$ applied to an element i of $\texttt{I}$.

$$\begin{tikzcd} \texttt{coe}\ \texttt{idp}\ x \arrow[r, dashed, "?"] & x \\ \mathrm{x} \arrow[u, dashed, "?"] \arrow[r, "\texttt{idp}\ i"'] & x \arrow[u, "\texttt{idp}\ j"'] \end{tikzcd}$$

Composition will complete the dashed sides as long as we specify the bottom face and the right side (where $i = \texttt{i1}$).

$\texttt{coe-}\beta\ x\ i = \texttt{comp}\ (\lambda\ \_ \to A)\ \_\ (\lambda\ \{\ j\ (i = \texttt{i1}) \to x\ \})\ x$

□

The indiscernibility of identicals follows immediately by coercion and action of paths.

Theorem 2.8 (Indiscernibility of identicals)**.** $\texttt{transport} : (P : A \to \texttt{Set}\ \ell_2)\ \{x\ y : A\} \to x \equiv y \to P\ x \to P\ y$
$\texttt{transport}^{-1} : (P : A \to \texttt{Set}\ \ell_2)\ \{x\ y : A\} \to x \equiv y \to P\ y \to P\ x$

Proof. Consider P to be a type-level function; then, the proof is trivial.

```
transport P = coe ∘ ap P
transport⁻¹ P = transport P ∘ _⁻¹
```

□

Now that we've proven to ourselves that paths are indeed a faithful representation of equality, we will develop a powerful inductive principle to reason about them—Paulin-Mohring's `J`, or *path induction*. But first, some definitions.

Definition 2 (Contractible space)**.** We say a space is *contractible* if and only if it is equivalent to a single point.

```
isContr : ∀ {ℓ} → Set ℓ → Set ℓ
isContr A = ∃ λ x → ∀ (y : A) → x ≡ y
```

We say that we can *contract* A to x.

Definition 3 (Singleton space [1])**.** The *singleton space* is the space of paths fixed at a base point.

```
singl : ∀ {ℓ} {A : Set ℓ} → A → Set ℓ
singl a = ∃ λ x → Path a x
```

We can immediately prove that the singleton space is contractible.

Theorem 2.9. `singlIsContr : isContr (singl x)`

Proof. We claim that we can contract this type to $(x,$ `idp`$)$ i.e. that for any (y, p), $(x, idp) \equiv (y, p)$. Thus, we construct a path abstraction valued (x, p) at `i0` and (y, p) at `i1`. Clearly, p takes care of the first component, but now we have to show $x \equiv p\ i$ for the second component. We claim that $\lambda\ j \rightarrow p\ (i \wedge j)$ works by case analysis on j—at `i0`, we have $p\ (i \wedge$ `i0`$) = p$ `i0` $= x$ and at `i1`, we have $p\ (i \wedge$ `i1`$) = p\ i$, as desired.

```
singlIsContr = (x , idp) , λ { (_ , p) i → p i , λ j → p (i ∧ j) }
```

□

This is a (known) remarkable result, with the CuTT proofs from [1]—under the right conditions, any path can be contracted to the constant path. Topologically, this makes sense—the singleton space consists of all paths fixed at a basepoint a, but free at another. So, we can "unhook" the path at the free endpoint and pull it back to a. As a result, this contraction is only possible if at least one endpoint is freely quantified. We can rephrase this contractibility result in terms of an induction principle. Paulin-Mohring's J, or *path induction* in homotopy type theory, is the principle that to prove any proposition over a path with at least one free endpoint, it suffices to prove the case for the constant path.

Theorem 2.10 (Paulin-Mohring's J)**.** `J : (P : ∀ y → Path x y → Set ℓ₂) (r : P x idp) {y : A} (p : Path x y) → P y p`

Proof. We transport the case for the constant path along the contractibility result to yield the proof for any path.

```
J P r {y} p = transport (uncurry P) (proj₂ singlIsContr (y , p)) r
```

□

We can go even further and prove a propositional computation rule for `J`; as expected, when applied to `idp`, it returns the provided case for `idp`. Unfortunately, this rule only holds up to paths, but the original CuTT paper defines Martin-Löf's *identity type* in terms of paths in a way where this rule holds definitionally [1].

Theorem 2.11. `J-β : J P r idp ≡ r`

Since `J` is defined in terms of `transport`, which is itself defined in terms of `coe`, this result is a corollary of its propositional computation rule.

Proof. `J-β = coe-β r`

□

Path induction is quite powerful because we can reduce many proofs involving paths to the simpler constant path case. In fact, we may easily define path composition, or transitivity of equality, using this principle.

Theorem 2.12 (Composition of paths)**.** `_•_ : Path x y → Path y z → Path x z`

Proof. Originally, we are asked to give the following unknown path.

$$\begin{tikzcd} x \arrow[r, "p"'] \arrow[rr, bend left, dashed, "?"] & y \arrow[r, "q"'] & z \end{tikzcd}$$

Using `J`, we can contract p, yielding the following.

$$\begin{tikzcd} x,y \arrow[r, "q"'] \arrow[r, bend left, dashed, "?"] & z \end{tikzcd}$$

Now, we can just return q!

```
_•_ = J (λ y _ → Path y z → Path x z) id
```

□

Theorem 2.13. *Paths form an equivalence relation.*

2.3 Equivalences

To prove (part of) univalence, the principle that equivalent types are equal, we need a type-theoretic notion of equivalence. One such definition is that $A \simeq B$ if and only if we have a function $f : A \to B$ such that for all $y : B$, there is a unique $x : A$ such that $y \equiv f\ x$, obeying a notion of bijectivity. However, univalence is the even stronger claim that the space of equivalences between two types is equivalent to the respective space of paths. As a result, given a proof e that f is an equivalence and if we have `equivToPath` and `pathToEquiv`, then `pathToEquiv` (`equivToPath` (f, e)) $\equiv (f, e)$. It follows that these procedures do not change f, and most importantly, e. This is impossible to guarantee as-is—even though we would still have proof that f is an equivalence, the original proof e might get lost in transit. Thus, we must require a proof-irrelevant notion of equivalence such that any modifications to e are indistinguishable in this proof system. We introduce the following definition of equivalence to get around this issue [1].

Definition 4 (Fiber space)**.** We co-opt this definition directly from set theory—the fiber of y under f is the inverse image of $\{y\}$ under f.

```
fiber : (f : A → B) (y : B) → Set _
fiber f y = Σ[ x ∈ A ] y ≡ f x
```

Now, we say that f is an equivalence not only if it obeys the condition we gave above, but also if the proof that $y \equiv f\ x$ is indistinguishable from any other such proof. That is, the fiber of every y under f is contractible.

Definition 5 (Equivalence)**.** `isEquiv : (A : Set ℓ₁) (B : Set ℓ₂) → (A → B) → Set _`
`isEquiv _ _ f = ∀ y → isContr (fiber f y)`

For convenience, we introduce the following alias that pairs f with proof that it is an equivalence.

```
infix 4 _≃_
_≃_ : ∀ {ℓ₁} {ℓ₂} → Set ℓ₁ → Set ℓ₂ → Set _
A ≃ B = Σ (A → B) (isEquiv A B)
```

Trivially, the identity function is an equivalence.

Definition 6 (Identity equivalence)**.** `ide : ∀ {ℓ} {A : Set ℓ} → A ≃ A`
`ide = id , λ y → singlIsContr {x = y}`

While this definition of equivalences certainly gets around the proof relevance problem, it is inconvenient to work with. In fact, it is more intuitive to think of equivalences as isomorphisms, like in the following definition.

Definition 7. A *quasi-inverse* of f is a function g with proofs that g is both a left and right inverse of f, up to homotopy.

```
record qinv {ℓ₁} {ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂} (f : A → B) : Set (ℓ₁ ⊔ ℓ₂) where
  constructor mkqinv
  field
    g : B → A
    ε : (g ∘ f) ∼ id
    η : (f ∘ g) ∼ id
```

This also corresponds to *homotopy equivalence* in homotopy theory.

We once again give a convenient alias for quasi-inverses.

```
_≈_ : ∀ {ℓ₁} {ℓ₂} (A : Set ℓ₁) (B : Set ℓ₂) → Set _
A ≈ B = Σ (A → B) qinv
```

Since quasi-inverses are not proof irrelevant, we cannot use them directly, but we may convert them to equivalences by the following theorem. The not-so-intuitive part of this proof is that we can develop proof irrelevant data from a quasi-inverse—you can see the proof of the *grad lemma* here by Andrea Vezzosi (Saizan).

Theorem 2.14. `qinvToEquiv : ∀ {ℓ₁} {ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂} → A ≈ B → A ≃ B`

Proof.

```
qinvToEquiv {A = A} {B} (f , mkqinv g ε η) = f , λ y → (g y , η y ⁻¹) ,
  λ { (z , p) i → gy≡z p i , square p i } where
```

□

2.4 Univalence, Almost

Now that we have a sufficient theory of equivalences, we would like to formalize part of *univalence* (the full proof can be found here).

Theorem 2.15. *For all types A and B, $(A \equiv B) \simeq (A \simeq B)$. That is, the space of equivalences is equivalent to the space of paths.*

Proof. We will give a function `pathToEquiv` and its quasi-inverse `ua` without the homotopy data. The former converts paths to equivalences by contracting them to the constant path and returning the identity equivalence.

```
pathToEquiv : A ≡ B → A ≃ B
pathToEquiv = J (λ B _ → A ≃ B) ide
```

The other direction is harder. CuTT adds a primitive operation `Glue` that allows us to directly give `ua` in a manner similar to `comp`. In particular, `Glue` allows us to partially specify the following square ($i \in$ `I`) where the top and bottom sides are paths but the left and right sides are equivalences, so that it can complete the missing bottom side—the desired path. The diagram and proof are taken from here.

$$\begin{tikzcd}
B \arrow[r, "\texttt{idp}\ i"] & B \\
A \arrow[u, "\mathrm{e}"] \arrow[r, dashed, "?"'] & B \arrow[u, "\texttt{ide}"']
\end{tikzcd}$$

```
ua : A ≃ B → A ≡ B
ua e i = Glue B _ (λ { (i = i0) → A; (i = i1) → B})
  λ { (i = i0) → e; (i = i1) → ide }
```

□

The power of univalence is that coercing along a univalent path (that is, a path constructed with `ua`) applies the underlying equivalence. This rule holds propositionally, as seen below, and gives us our first example of a nondegenerate use of coercion.

```
ua−β : (e : A ≃ B) → coe (ua e) ≡ proj₁ e
ua−β e = λ≡ λ x → coe−β _ • coe−β _ • coe−β _
```

Let's try a quick application given this knowledge. The `not` function is an equivalence on `Bool`, so coercing along this path with `true` should evaluate to `false`, which it does!

```
notq : Bool ≈ Bool
notq = not , mkqinv not
  (λ { true → idp; false → idp })
  (λ { true → idp; false → idp })

_ : coe (ua (qinvToEquiv notq)) true ≡ false
_ = idp
```

Did we do all this work for this one example? Of course not! Now, we get to good part—mathematics and programming with CuTT constructs.

3 Mathematics

Now that we have a minimal working library for cubical types, we can finally start doing something useful. Since CuTT, like its "predecessor" HoTT, models topological phenomena, it would be fitting to give a synthetic characterization of the *fundamental group* of a space i.e. the group of paths with the constant path and composition, and investigate the fundamental group of the circle, a *higher inductive type*. Thus, we must prove the following unit, inverse, and associativity laws. The base of our proofs is the left unit law.

3.1 Fundamental Group

Theorem 3.1. •–unitl : $(p$: Path $x\ y) \to$ idp • $p \equiv p$

Proof. Recall that composition is defined in terms of J. Since we are applying the constant path, we can directly apply the propositional computation rule for J.

•–unitl = app≡ (J–β $\{P = \lambda\ y\ \_ \to$ Path $x\ y \to$ Path $x\ y\})$

□

The rest of the unit and inverse laws are direct corollaries of the left unit law.

•–unitr : $(p$: Path $x\ y) \to p$ • idp $\equiv p$
•–unitr = J $(\lambda\ \_\ p \to p$ • idp $\equiv p)$ (•–unitl idp)

•–invl : $(p$: Path $x\ y) \to p^{-1}$ • $p \equiv$ idp
•–invl = J $(\lambda\ \_\ p \to p^{-1}$ • $p \equiv$ idp) (•–unitl idp)

•–invr : $(p$: Path $x\ y) \to p$ • $p^{-1} \equiv$ idp
•–invr = J $(\lambda\ \_\ p \to p$ • $p^{-1} \equiv$ idp) (•–unitl idp)

Thus, it remains to show associativity.

Theorem 3.2. •–assoc : $(p$: Path $x\ y)$ $(q$: Path $y\ z)$ $(r$: Path $z\ w) \to (p$ • $q)$ • $r \equiv p$ • q • r

Proof. By contracting p to the constant path using J, we apply the left unit law on both sides.

•–assoc = J $(\lambda\ \_\ p \to \forall\ q\ r \to (p$ • $q)$ • $r \equiv p$ • q • $r)$
  $\lambda\ q\ r \to$ ap $(\lambda\ x \to x$ • $r)$ (•–unitl q) • •–unitl $(q$ • $r)^{-1}$

□

With all these results, we would like to make a definitive statement about some type being a group. In HoTT, we define the *loop space* of A to be the space of paths fixed at both ends on a given basepoint.

Definition 8 (Loop space)**.** Ω[_, _] : ∀ {ℓ} (A : Set ℓ) → A → Set ℓ
Ω[_ , a] = Path $a\ a$

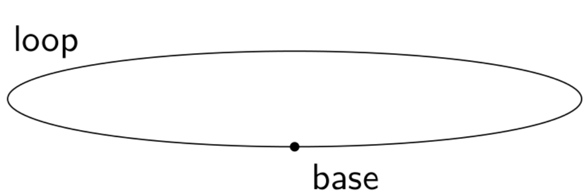
Thus, we get the following result.

Theorem 3.3 (Fundamental group)**.** *For all A and $a : A$,* Ω$[A, a]$ *is a group.*

For our intents and purposes, we may treat the loop space exactly as the fundamental group; see page 207 of [5] for more details.

3.2 Higher Inductive Types: The Circle

Now that we have the vocabulary to discuss the fundamental group of various spaces, it behooves us to characterize that of a nontrivial space, like *the circle*. Homotopically, a circle is simply a single basepoint base with a *nonconstant* path loop from base back to itself, as illustrated in 3.2 (figure 6.1 taken from [5]).



We can characterize this concept in CuTT via a *higher inductive type* `S¹`: not only is it inductively generated by `base`, but also by the path `loop`, which exists at a "higher" level than `base`. In general, any datatype that defines its own paths is higher inductive. The circle has a *topological recursion principle* `recS¹` that allows one to define a function f : `S¹` $\to A$ given the following data.

- A `base` case $b : A$
- A `loop` case $l : b \equiv b$

Then, we are guaranteed the definitional equalities f `base` $= b$ and `ap` f `loop` $= l$.

We would now like to show the following result.

Theorem 3.4. *The fundamental group of the circle is the integers i.e.* `Ω[S¹, base]` $\equiv \mathbb{Z}$.

Proof. A proof by Dan Licata is given at [3]. As we did with univalence, we will not discuss the homotopies, but the computational essence of the proof. The intuition is that we can ascribe a normal form to any path in $\Omega[S^1,$ `base`]: it is either `idp`, $\texttt{loop}^n = \underbrace{\texttt{loop} \cdot \ldots \cdot \texttt{loop}}_{n}$, or $\texttt{loop}^{-n} = \underbrace{\texttt{loop}^{-1} \cdot \ldots \cdot \texttt{loop}^{-1}}_{n}$. These look remarkably like the integers—as a result, a map $\Omega[S^1,$ `base`$] \to \mathbb{Z}$ sends `idp` to 0, $\texttt{loop}^n$ to n, and $\texttt{loop}^{-n}$ to $-n$. Then, the quasi-inverse simply takes an integer n and iterates `loop` n times in the direction of its sign, or `idp` if $n = 0$. □

The easier direction is actually backwards—giving the map from $\mathbb{Z}$ to `Ω[S¹, base]`.

Definition 9 (Winding path)**.** `loop^ : ℤ → Ω[ S¹ , base ]`

```
loop^ (+ 0)       = idp
loop^ (+ suc n) = loop • loop^ (+ n)
loop^ -[1+ 0 ]  = loop ⁻¹
loop^ -[1+ suc n ] = loop ⁻¹ • loop^ -[1+ n ]
```

The hard part is going forwards—how do we inspect the normal form of a path? We cannot do it directly, but we can do it with the topological recursion principle. First, we quickly prove that the successor function on the integers is an equivalence with the predecessor function as its quasi-inverse.

```
suc≡ : ℤ ≡ ℤ
suc≡ = ua $ qinvToEquiv
  (Data.Integer.suc ,
    mkqinv Data.Integer.pred
    (λ { (+ _) → idp; -[1+ 0 ] → idp; -[1+ suc _ ] → idp })
    (λ { (+ 0) → idp; (+ suc _) → idp; -[1+ _ ] → idp }))
```

We now define the *universal cover* of the circle, which sends `base` to $\mathbb{Z}$ and satisfies `ap Cover loop` $=$ `suc ≡`.

Definition 10 (Universal cover of the circle)**.** `Cover : S¹ → Set`

```
Cover = recS¹ ℤ suc≡
```

We finally give the desired map, which computes the *winding number* of a given path.

Definition 11 (Winding number)**.** Recall that `transport` $P =$ `coe` $\circ$ `ap` P. Since `ap Cover loop` $\equiv$ `suc ≡`, it follows that by the propositional computation rule `ua-β`:

$$\texttt{coe}\ (\texttt{ap Cover loop})\ 0 \equiv \texttt{coe}\ (\texttt{suc} \equiv) \equiv \texttt{suc}\ 0 = 1$$

Furthermore, `coe` (…) `idp` $0 \equiv 0$ by the propositional computation rule `coe-β`. We get the general case for $\texttt{loop}^n$ and $\texttt{loop}^{-n}$ by the functoriality of `transport` and `ua`:

1. $\texttt{transport Cover loop}^n\ 0 \equiv (\texttt{transport Cover loop})^n\ 0 \equiv \texttt{suc}^n\ 0 \equiv n$

2. $\texttt{transport Cover loop}^{-n}\ 0 \equiv \left((\texttt{transport Cover loop})^{-1}\right)^n\ 0 \equiv \left((\texttt{suc}\ \equiv)^{-1}\right)^n\ 0 \equiv \texttt{pred}^n\ 0 \equiv -n$

Thus, we give the following definition.

```
w : Ω[ S¹ , base ] → ℤ
w p = transport Cover p (+ 0)
```

Let us take a moment to appreciate the simplicity of these results—indeed, HoTT and CuTT are useful tools for topologists and homotopy theorists due to the ease of formalizing complex results.

4 Programming

Dan Licata gave a talk titled *Programming in Homotopy Type Theory* [2] in which he expounded upon the applications of univalence to software engineering. In particular, univalence does not allow a type theory to distinguish between equivalent types, so we can safely interchange use between them when convenient. This greatly increases code reuse—given code for one type, we can automatically generate code for all other equivalent types without resorting to metaprogramming or other ad hoc facilities. Furthermore, we can specify the behavior of one type and yield similar proofs of behavior for other equivalent types with the same mechanism. Consider the following examples.

4.1 Code Reuse

In Haskell and other functional programming languages, map/reduce algorithms are common when manipulating and consolidating data. Formally speaking, given a stream of data belonging to a monoid, we can "reduce" it via its associative operation.

Definition 12 (Monoid)**.** A monoid is a type equipped with an identity element and associative operation.

```
record Monoid {ℓ} (A : Set ℓ) : Set (lsuc ℓ) where
  infixr 7 _·_
  field
    e : A
    _·_ : A → A → A
    ·—unitl : ∀ x → (e · x) ≡ x
    ·—unitr : ∀ x → (x · e) ≡ x
    ·—assoc : ∀ x y z → (x · y · z) ≡ ((x · y) · z)
```

`foldl` performs this reduction operation on a finite stream from left-to-right.

Definition 13 (Fold/reduce)**.** `foldl : ∀ {ℓ} {A : Set ℓ} {{M : Monoid A}} → List A → A`

```
foldl [] = e
foldl (h :: t) = h · foldl t
```

Clearly, lists—the free monoid over any type—is a monoid. In fact, we give its instance of `Monoid` with the empty list as the identity element and "append" as the associative operation.

```
instance
  ListMonoid : ∀ {ℓ} {A : Set ℓ} → Monoid (List A)
  ListMonoid {A = A} = record
    { e   = []
    ; _·_ = _++_
```

```
; ·—unitl = λ _ → idp
; ·—unitr = unitr
; ·—assoc = assoc } where

unitr : (l : List A) → (l ++ []) ≡ l
unitr [] = idp
unitr (h :: t) = ap (_::_ h) (unitr t)

assoc : (x y z : List A) → (x ++ y ++ z) ≡ ((x ++ y) ++ z)
assoc [] _ _ = idp
assoc (h :: t) y z = ap (_::_ h) (assoc t y z)
```

This proof was quite tedious, but wait! It turns out that *vectors*—lists indexed by length—behave *exactly* like lists, so if the programmer would like to get the same functionality out of vectors, they would have to do essentially the same proof all over again...This is of course, a total waste of time. What if there was a way to generate the same `Monoid` instance for vectors given the one we have for lists? With univalence and `transport`, there is! First, consider the following datatype, which uses existential quantification to enclose over the type index for length.

Definition 14 (Vectors)**.** `VecList : ∀ {ℓ} → Set ℓ → Set ℓ`
`VecList A = Σ ℕ (Vec A)`

Now, we can state the intuitively obvious univalent path: lists are equivalent to vectors. We do so by converting lists to vectors and vice versa element-by-element, which preserves length.

```
ListIsVecList : ∀ {ℓ} {A : Set ℓ} → List A ≡ VecList A
ListIsVecList {A = A} = ua (qinvToEquiv (toVecList , mkqinv toList ε η)) where
  toVecList : List A → VecList A
  toVecList l = length l , fromList l

  toList : VecList A → List A
  toList (_ , v) = Data.Vec.toList v

  ε : (toList ∘ toVecList) ∼ id
  ε [] = idp
  ε (h :: t) = ap (_::_ h) (ε t)

  η : (toVecList ∘ toList) ∼ id
  η (_ , []) = idp
  η (suc n , h :: t) = ap (λ{(n , t) → ℕ.suc n , h :: t}) (η (n , t))
```

Now here is the cool part: we get an instance of `Monoid` for `VecList` for free, as desired. The indiscernibility of identicals is not only a statement about propositions, but about all *type-level functions*. Since `Monoid` is such a function, we can transport `ListMonoid` along the above path to yield the following instance.

```
instance
  VecListMonoid : ∀ {ℓ} {A : Set ℓ} → Monoid (VecList A)
  VecListMonoid = transport Monoid ListIsVecList ListMonoid
```

The utility and simplicity of this design pattern is astonishing—having a typesafe method of code generation will surely make sophisticated type theories more attractive to software engineers.

4.2 Abstract Types

Let us consider a similar application of univalence as in the last section. In particular, we often have different type-level views of the same concept. For example, in programming languages, typing contexts can simultaneously be thought of as functions from a domain of variables to a codomain of types, or as a set of ordered pairs between variables and types. Informally, we do not make a distinction between them, but contemporary type theories do, making formal reasoning in a proof assistant cumbersome. While this does not bother a lot of practitioners, this is a fundamental disconnect between the way we reason about mathematical objects versus the language we use to represent them. Philosophically, a univalent type theory is much better suited to mathematics and software engineering, because once again, it cannot distinguish between equivalent concepts, as we do not. Thus, consider extending the above example: another way to view vectors is as *arrays*—an array of length n with elements in A is a function from the set $\{0, \ldots, n-1\}$ to A, as follows.

Definition 15 (Arrays).
```
Array : ∀ {ℓ} → Set ℓ → ℕ → Set ℓ
Array A n = Fin n → A
```

Lists, vectors, and arrays are all *functors*, inspired by the analogous category-theoretic concept. Functors are types equipped with a map operation á la map/reduce that is coherent with the identity function and function composition.

Definition 16 (Functor).
```
record Functor {ℓ₁} {ℓ₂} (F : Set ℓ₁ → Set ℓ₂) : Set (lsuc ℓ₁ ⊔ ℓ₂) where
  infixl 4 _<$>_

  field
    _<$>_ : ∀ {A B} → (A → B) → F A → F B
    <$>-id : ∀ {A} (a : F A) → (id <$> a) ≡ a
    <$>-∘ : ∀ {A B C} (f : B → C) (g : A → B) (x : F A) →
      ((f ∘ g) <$> x) ≡ (f <$> (g <$> x))
```

Now, here is our scenario: it is *much* easier to reason about arrays than vectors because as functions, we can explicitly reason about the shape of each element. On the other hand, vectors are more easily memory optimized since they do not close on the environment like a function does at runtime. It follows that wherever we can, we reason about arrays, and then transport equivalent results to vectors. For example, consider the trivial `Functor` instance for arrays.

```
instance
  ArrayFunctor : ∀ {ℓ n} → Functor {ℓ} (flip Array n)
  ArrayFunctor = record
    { _<$>_ = λ f a i → f (a i)
    ; <$>-id = λ _ → idp
    ; <$>-∘ = λ _ _ _ → idp }
```

Let us consider how arrays and vectors are equivalent. We can convert an array to a vector by tabulating it from 0 to $n-1$. and vice versa by constructing a function that looks up a value by index from the given vector.

```
ArrayIsVec : ∀ {ℓ} (A : Set ℓ) (n : ℕ) → Array A n ≡ Vec A n
ArrayIsVec A n = ua (qinvToEquiv (tabulate , mkqinv lookup ε η)) where
  ε : (lookup ∘ tabulate) ∼ id
  ε f = λ≡ h where
    h : ∀ {n} {f : Array A n} → lookup (tabulate f) ∼ f
    h zero = idp
    h (suc x) = h x
```

```
η : ∀ {n} → (tabulate ∘ lookup {n = n}) ~ id
η [] = idp
η (h :: t) = ap (_::_ h) (η t)
```

Finally, we can automatically generate an instance of `Functor` for vectors and export a proof that the generated definition for the map operation obeys the identity function.

```
instance
  VecFunctor : ∀ {ℓ n} → Functor {ℓ} (flip Vec n)
  VecFunctor {n = n} = transport Functor (λ≡ (flip ArrayIsVec n)) (ArrayFunctor {n = n})

_ : ∀ {ℓ1} {A : Set ℓ1} {n} (v : Vec A n) → Functor._<$>_ VecFunctor id v ≡ v
_ = Functor.<$>-id VecFunctor
```

While this is nice, it would be more powerful to reason about *existing* definitions and not generated ones. Licata discusses an example where we have two views of sequences—`ListSeq`s with an operationally sequential map operation, and `PSeq`s with an operationally parallel one. Here is a link where he formalizes, in homotopy type theory, the ability to extract a proof that `PSeq`'s map function is coherent from an equality between `ListSeq` and `PSeq` as well as a proof of coherence for `ListSeq`'s map function.

5 Conclusion and Future Work

The power of cubical type theory lies in an alternative view of equality that has clear benefits to mathematics and programming—in general, we have demonstrated to ourselves that type theory is both a powerful foundations of mathematics and programming language. Not only does it have the potential to change the way we do math by realigning formalism with our intuitions about computation and equality, it could be the start of a class of programming tools that ends the many pain points of structuring software. It remains to develop type theory and proof assistants with univalent principles to a point where they become usable as daily drivers for mathematicians and software engineers alike.

References

[1] Cohen, C., Coquand, T., Huber, S., and Mörtberg, A. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR abs/1611.02108* (2016).

[2] Licata, D. Programming in homotopy type theory. IFIP Working Group 2.8 Meeting, 2012.

[3] Licata, D. A simpler proof that $\pi_1(s^1)$ is $\mathbb{Z}$, Jun 2012.

[4] Mörtberg, A. A hands-on introduction to cubicaltt, Sep 2017.

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

[6] Wikipedia. First-order logic — wikipedia, the free encyclopedia, 2017. [Online; accessed 4-December-2017].