Higher Inductive Type Eliminators Without Paths

Nils Anders Danielsson
University of Gothenburg, Sweden

Abstract

Cubical Agda has support for higher inductive types. Paths are integral to the working of this feature. However, there are other notions of equality. For instance, Cubical Agda comes with an identity type family for which the J rule computes in the usual way when applied to the canonical proof of reflexivity, whereas typical implementations of the J rule for paths do not.

This text shows how one can use some of the higher inductive types definable in Cubical Agda with arbitrary notions of equality satisfying certain axioms. The method works for several examples taken from the HoTT book, including the interval, the circle, suspensions, pushouts, the propositional truncation, a general truncation operator, and set quotients.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Cubical Agda, higher inductive types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.10

Supplementary Material Accompanying Agda code is available to download [3].

Acknowledgements I would like to thank Anders Mörtberg and Andrea Vezzosi for helping me get to grips with Cubical Agda. I would also like to thank some anonymous reviewers for useful feedback.

1 Introduction

Higher inductive types provide a way to define things like propositional truncation, (set) quotients and other things in type theory [7]. Recently support for higher inductive types has been added to Agda [1, 8]. As an example propositional truncation can be defined in the following way (where $\mathit{Type}\ a$ is the universe at level a):

```
data ∥_∥ (A : Type a) : Type a where
  |_|     : A → ∥ A ∥
  trivial : (x y : ∥ A ∥) → x ≡ y
```

This type family has a regular constructor $|\_|$ which states that $\| A \|$ is inhabited if A is. It also has a *higher* constructor trivial which states that every element of $\| A \|$ is equal to every other, i.e. that $\| A \|$ is a (mere) proposition. In the type of trivial the equality $x \equiv y$ stands for the type of *paths* from x to y. Paths in A are a kind of functions from the *interval* I to A. When trivial $x\ y$ is applied to an element i of the interval we get a value in $\| A \|$: this value is definitionally equal to x if i is $\underline{0}$, and y if i is $\underline{1}$, where $\underline{0}$ and $\underline{1}$ are the endpoints of the interval. Thus all constructors of $\| A \|$ target the same type.

One can define functions from $\| A \|$ using pattern matching. For instance, here is a map function:

```
map : (A → B) → ∥ A ∥ → ∥ B ∥
map f ∣ x ∣            = ∣ f x ∣
map f (trivial x y i) = trivial (map f x) (map f y) i
```

25th International Conference on Types for Proofs and Programs (TYPES 2019).
Editors: Marc Bezem and Assia Mahboubi; Article No. 10; pp. 10:1–10:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Let us consider the second case. Note that trivial is applied to three arguments. The right-hand side of the function must be an expression of type $\| B \|$ that satisfies two side-conditions: if $\underline{0}$ is substituted for i, then the value must be definitionally equal to $map\ f\ x$ (because trivial $x\ y\ \underline{0}$ is definitionally equal to x), and if $\underline{1}$ is substituted for i, then the value must be definitionally equal to $map\ f\ y$. The given right-hand side satisfies these conditions.

This text is concerned with the following question: What if you want to use higher inductive types, but you do not want to use paths? Cubical Agda, the variant of Agda with support for higher inductive types, comes with two notions of equality: paths and an identity type family [2, 6]. One can prove the J rule for paths, but so far no one has managed to do this in such a way that the J rule computes in the usual way when applied to reflexivity. The identity type family comes with a J rule that does compute in the usual way when applied to reflexivity. People with code that relies on this computational behaviour of J might not want to switch to using paths. This text shows one way in which one can avoid doing this, and still make use of (at least some) higher inductive types:

- The approach works for any notion of equality that satisfies certain axioms (see Section 2). There is work in progress on adding proper support for inductive families to Cubical Agda. Such support would mean that the approach would work also for equality defined in the following way:

  ```
  data _≡_ {A : Type a} (x : A) : A → Type a where
    refl : x ≡ x
  ```

- The basic idea of the approach is to define variants of the higher constructors that use the other notion of equality instead of paths, and to define eliminators that refer to these variants of the constructors. It might seem obvious that this can be done, because any notion of equality that satisfies the axioms is equivalent to path equality. However, in Cubical Agda it is natural to express eliminators for many higher inductive types using a heterogeneous notion of path equality (see Section 4). Fortunately heterogeneous paths can be expressed using homogeneous paths (see Section 4; this result was proved together with Anders Mörtberg and Andrea Vezzosi).
- The eliminators are defined in such a way that they compute in the "right" way for constructors that do not involve paths. For higher constructors propositional "computation" rules are proved.
- The approach works for at least the following higher inductive types: the circle (see Section 5), set quotients (Section 6), the propositional truncation operator given above (Section 7), suspensions (Section 7), and some types that are not discussed in detail in the paper, but are treated in accompanying Agda code: the interval, pushouts, and a general truncation operator. The obtained eliminators are close to the induction principles given in the HoTT book [7]. No attempt is made to characterise exactly when the method works, but there is some discussion of when the method might be usable in Section 9.
- It might not come as a surprise that something like this can be done. A key contribution of the paper is, in my opinion, some functions that make it *easy*—at least for the higher inductive types mentioned above—to define the eliminators and to prove the computation rules (see Sections 4 and 5).

  Dependent eliminators are defined by using the eliminators for paths, plus one of two lemmas for each higher constructor (one lemma for truncation constructors, and one for the rest). All corresponding "computation" rules (one for each higher constructor, except for the truncation constructors) are proved using the same lemma, applied to a proof of reflexivity. These three lemmas suffice for all the examples mentioned above. There are also similar lemmas for non-dependent eliminators. Section 7 demonstrates that the lemmas developed in previous sections work also for other higher inductive types.

The text is accompanied by machine-checked Agda proofs [3] (but there is no guarantee that Agda is free of bugs). Note that there are small differences between the accompanying code and the code presented below.

2 An Axiomatisation of Equality With J

Let us assume that _≡_ has the following type:

$$\_\equiv\_ : \{A : \mathit{Type}\ a\} \to A \to A \to \mathit{Type}\ a$$

(Arguments in braces are implicit arguments, that do not need to be given explicitly if Agda can infer them. To avoid clutter some implicit argument declarations, like the one for the universe level a, are omitted from type signatures.) This type family is assumed to satisfy the following axioms:

$$\begin{aligned}
&\mathit{refl} &&: (x : A) \to x \equiv x\\
&J &&: (P : \{x\ y : A\} \to x \equiv y \to \mathit{Type}\ p) \to (\forall\ x \to P\ (\mathit{refl}\ x)) \to (eq : x \equiv y) \to P\ eq\\
&J\text{-}\mathit{refl} &&: (P : \{x\ y : A\} \to x \equiv y \to \mathit{Type}\ p)\ (r : \forall\ x \to P\ (\mathit{refl}\ x)) \to J\ P\ r\ (\mathit{refl}\ x) \equiv r\ x
\end{aligned}$$

There should be a canonical proof of reflexivity, *refl*, there should be a J rule, and the usual computation rule for J should hold up to the given notion of equality.

Any two notions of equality satisfying these axioms are pointwise equivalent, in the sense of the HoTT book [7], using one of the notions to define what it means to be equivalent:

$$\equiv\simeq\equiv : (x \equiv_1 y) \simeq (x \equiv_2 y)$$

(Hofmann and Streicher have proved a very similar result [4, Section 5.2].) Furthermore this proof maps *refl* to *refl*, in both directions. The proofs of these properties are easy and omitted.

Cubical Agda's path and identity type families are instances of these axioms, as is the equality type family defined as an inductive family with a single constructor refl as in Section 1. For paths this is shown in Section 3.

From now on _≡_ will be used to refer to an arbitrary notion of equality satisfying the axioms above, whereas the path type family will be called *Path*. (It might be the case that _≡_ also refers to the path type family.)

3 Homogeneous Paths

This section contains an introduction to paths, or more specifically *homogeneous* paths. Section 4 discusses *heterogeneous* paths.

The path type constructor has the following type:

$$\mathit{Path} : \{A : \mathit{Type}\ a\} \to A \to A \to \mathit{Type}\ a$$

The type $\mathit{Path}\ \{A = A\}\ x\ y$ (where the notation $\{A = A\}$ is used to explicitly give A as the implicit argument A) is a kind of function space from the interval I to A. It is subject to the restriction that when values in this type are applied to the endpoints of the interval, $\underline{0}$ and $\underline{1}$, we get x and y, respectively.

We can prove that the path type family is reflexive in the following way:

$$\begin{aligned}
&\mathit{refl}^P : (x : A) \to \mathit{Path}\ x\ x\\
&\mathit{refl}^P\ x = \lambda\ _ \to x
\end{aligned}$$

Note that $\mathit{refl}^P\ x\ i$ is equal to x for all values of i.

Cubical Agda comes with some interval operations. There is a maximum operation, *max*, with $\underline{0}$ as a definitional unit and $\underline{1}$ as a definitional zero. Similarly *min* is a minimum operation, with $\underline{1}$ as a definitional unit and $\underline{0}$ as a definitional zero. Furthermore there is a negation operation, $-\_$, that maps $\underline{0}$ to $\underline{1}$ and $\underline{1}$ to $\underline{0}$.

Cubical Agda also comes with a primitive transport operation:

$$\mathit{transport} : \{p : I \to \mathit{Level}\}\ (P : (i : I) \to \mathit{Type}\ (p\ i)) \to I \to P\ \underline{0} \to P\ \underline{1}$$

(*Level* is the type of universe levels.) If the interval argument is $\underline{0}$, then the computational behaviour of *transport* depends on the type family P. However, if the interval argument is $\underline{1}$, then *transport* returns its final argument. In this case there is a side-condition on the use of *transport* that is not captured in its type: the type family P must be definitionally constant. (The interval argument might be an expression that does not reduce to $\underline{0}$ or $\underline{1}$, and the type family might mention interval variables used in the interval argument. In this case the application is accepted if Agda can verify that the type family is constant whenever the constraint $i = \underline{1}$ holds, where i is the interval argument [8].)

The primitive transport operation can be used to prove the J rule for paths (the notation $\{x = x\}$ is used to bind the implicit argument x to the name x):

$$\begin{aligned}
&J^P : (P : \{x\ y : A\} \to \mathit{Path}\ x\ y \to \mathit{Type}\ p) \to (\forall\ x \to P\ (\mathit{refl}^P\ x)) \to \\
&\qquad (\mathit{eq} : \mathit{Path}\ x\ y) \to P\ \mathit{eq} \\
&J^P\ \{x = x\}\ P\ p\ \mathit{eq} = \mathit{transport}\ (\lambda\ i \to P\ (\lambda\ j \to \mathit{eq}\ (\mathit{min}\ i\ j)))\ \underline{0}\ (p\ x)
\end{aligned}$$

Note that when i is $\underline{0}$, then the expression $P\ (\lambda\ j \to \mathit{eq}\ (\mathit{min}\ i\ j))$ is definitionally equal to $P\ (\lambda\ \_ \to x)$, which is the type of $p\ x$. When i is $\underline{1}$, then the expression is definitionally equal to $P\ \mathit{eq}$.

The computation rule for J does not hold by definition for J^P. However, it can be proved using the following lemma (following Anders Mörtberg [1]):

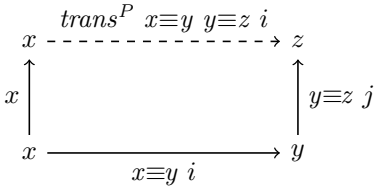
$$\begin{aligned}
&\mathit{transport\text{-}refl} : \mathit{Path}\ (\mathit{transport}\ (\lambda\ i \to \mathit{refl}^P\ A\ i)\ \underline{0})\ (\lambda\ x \to x) \\
&\mathit{transport\text{-}refl}\ \{A = A\} = \lambda\ i \to \mathit{transport}\ (\lambda\ _ \to A)\ i
\end{aligned}$$

Note that the first argument given to the final occurrence of *transport* is constant when i is $\underline{1}$, as required.

Cubical Agda also has support for *composition* of paths [2, 8]. There are two variants, homogeneous and heterogeneous. Here the homogeneous variant is used to prove that path equality is transitive [2] (this can also be proved using the J rule):

$$\mathit{trans}^P : \mathit{Path}\ x\ y \to \mathit{Path}\ y\ z \to \mathit{Path}\ x\ z$$

The basic idea of the proof is to construct three sides of a square, and to use the composition operation to compute the square's fourth side. Instead of showing the Agda code I have included a diagram:



Every arrow in the diagram is a path between the expressions at the arrow's endpoints, and the expressions between the endpoints stand for arbitrary "points" on the paths. The left-hand side of the diagram corresponds to i being $\underline{0}$, and the right-hand side to i being $\underline{1}$. Similarly, the bottom corresponds to j being $\underline{0}$, and the top to j being $\underline{1}$. The solid arrows are constructed using the two arguments given to $trans^P$ ($x{\equiv}y$ and $y{\equiv}z$), as well as a constant path (the left-hand side). The composition operation is then used to construct the dashed arrow.

The homogeneous composition operation requires that (roughly speaking) every "point" on the left and right sides of the square have the same type. The heterogeneous operation, which is used in Section 4, is more general in that it (roughly speaking) allows the types of the points on the left and right sides to vary with j.

4 Heterogeneous Paths

The path type family discussed above is a homogeneous special case of a heterogenous notion of path:

$$Path^H : (P : I \to Type\ p) \to P\ \underline{0} \to P\ \underline{1} \to Type\ p$$

$Path$ is defined using $Path^H$:

$$\begin{aligned}&Path : \{A : Type\ a\} \to A \to A \to Type\ a\\&Path\ \{A = A\} = Path^H\ (\lambda\ \_ \to A)\end{aligned}$$

A typical eliminator for a regular inductive type takes one argument per constructor (plus some other arguments). What should the type of such an argument be for a higher constructor? It turns out that one can, at least in some cases, use heterogeneous paths to give suitable types to such arguments.

Consider the following incomplete definition of an eliminator for the propositional truncation operator:

$$\begin{aligned}&elim^P : (P : \|\ A\ \| \to Type\ p) \to ((x : A) \to P\ |\ x\ |) \to\ ? \to (x : \|\ A\ \|) \to P\ x\\&elim^P\ P\ f\ t\ |\ x\ |\ \ \ \ \ \ \ \ \ \ = f\ x\\&elim^P\ P\ f\ t\ (\mathsf{trivial}\ x\ y\ i) = ?\end{aligned}$$

There are two side-conditions on the right-hand side of the last clause: when i is $\underline{0}$, then it must be definitionally equal to $elim^P\ P\ f\ t\ x$ (because $\mathsf{trivial}\ x\ y\ \underline{0}$ is definitionally equal to x), and when i is $\underline{1}$, then it must be definitionally equal to $elim^P\ P\ f\ t\ y$. These requirements can be captured using $Path^H$:

$$\begin{aligned}&elim^P\ P\ f\ t\ (\mathsf{trivial}\ x\ y\ i) = rhs\ i\\&\quad \textbf{where}\\&\quad rhs : Path^H\ (\lambda\ i \to P\ (\mathsf{trivial}\ x\ y\ i))\ (elim^P\ P\ f\ t\ x)\ (elim^P\ P\ f\ t\ y)\\&\quad rhs = ?\end{aligned}$$

There are no side-conditions on the right-hand side of rhs. Here is a complete definition of the eliminator:

$$\begin{aligned}&elim^P : (P : \|\ A\ \| \to Type\ p) \to\\&\qquad ((x : A) \to P\ |\ x\ |) \to\\&\qquad (\{x\ y : \|\ A\ \|\}\ (p : P\ x)\ (q : P\ y) \to Path^H\ (\lambda\ i \to P\ (\mathsf{trivial}\ x\ y\ i))\ p\ q) \to\\&\qquad (x : \|\ A\ \|) \to P\ x\end{aligned}$$

$$elim^P\ P\ f\ t\ |\ x\ |\ = f\ x$$
$$elim^P\ P\ f\ t\ (\mathsf{trivial}\ x\ y\ i) = t\ (elim^P\ P\ f\ t\ x)\ (elim^P\ P\ f\ t\ y)\ i$$

The goal here is to define eliminators that use an arbitrary notion of equality that satisfies the axioms from Section 2, not necessarily paths, either heterogeneous or homogeneous. As mentioned above homogeneous path equality is pointwise equivalent to any other notion of equality satisfying the axioms. How do heterogeneous paths fit into this picture?

$Path^H$ can, up to equivalence, be expressed using $Path$:

$$Path^H{\simeq}Path :$$
$$(P : I \to Type\ p)\ \{p : P\ \underline{0}\}\ \{q : P\ \underline{1}\} \to Path^H\ P\ p\ q \simeq Path\ (transport\ P\ \underline{0}\ p)\ q$$

It turns out that it is very easy to prove this equivalence.[1] One can use *transport* to construct the corresponding path:

$$Path^H{\equiv}Path :$$
$$(P : I \to Type\ p)\ (p : P\ \underline{0})\ (q : P\ \underline{1}) \to Path\ (Path^H\ P\ p\ q)\ (Path\ (transport\ P\ \underline{0}\ p)\ q)$$
$$Path^H{\equiv}Path\ P\ p\ q\ i =$$
$$Path^H\ (\lambda\ j \to P\ (max\ i\ j))\ (transport\ (\lambda\ j \to P\ (min\ i\ j))\ (-\ i)\ p)\ q$$

When i is $\underline{0}$, then the type family argument given to *transport* is constant, as required, and the right-hand side is definitionally equal to $Path^H\ P\ p\ q$. Furthermore, when i is $\underline{1}$, then the right-hand side is definitionally equal to $Path\ (transport\ P\ \underline{0}\ p)\ q$. Once the equality has been established in this way one can turn it into an equivalence by using $subst^P$:

$$subst^P : (P : A \to Type\ p) \to Path\ x\ y \to P\ x \to P\ y$$
$$subst^P\ P\ x{\equiv}y\ p = transport\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ \underline{0}\ p$$

A function like $subst^P$ can also be defined for the arbitrary notion of equality by using the J rule. In order to support different definitions, like $subst^P$ for paths, let us assume that our arbitrary notion of equality comes with a function *subst*, along with a propositional computation rule for *subst*:

$$subst\ \ \ : (P : A \to Type\ p) \to x \equiv y \to P\ x \to P\ y$$
$$subst\text{-}refl : subst\ P\ (refl\ x)\ p \equiv p$$

As noted in Section 2 the arbitrary notion of equality is pointwise equivalent to path equality. Let $from\text{-}path : Path\ x\ y \to x \equiv y$ denote one direction of the equivalence, and *to-path* the other. We can now relate *subst* to $subst^P$:

$$subst{\equiv}subst^P : (x{\equiv}y : Path\ x\ y) \to subst\ P\ (from\text{-}path\ x{\equiv}y)\ p \equiv subst^P\ P\ x{\equiv}y\ p$$

The proof uses the J rule for paths and the following calculation (recall that *from-path* maps canonical reflexivity proofs to canonical reflexivity proofs):

$$\begin{aligned}
&subst\ P\ (from\text{-}path\ (refl^P\ x))\ p\ &&\equiv\\
&subst\ P\ (refl\ x)\ p\ &&\equiv\\
&p\ &&\equiv\\
&subst^P\ P\ (refl^P\ x)\ p
\end{aligned}$$

The last step follows from *transport-refl*.

[1] In retrospect. Anders Mörtberg had implemented a corresponding logical equivalence [5]. I asked Anders and Andrea Vezzosi if and how the corresponding equivalence could be proved. Andrea gave me some useful hints. I managed to finish the proof, only to find out that Andrea had proved it a couple of days before me. However, both proofs were rather complicated. The simple proof presented here was found by Anders quite some time later.

4.1 Consequences of the Equivalence

Let us now discuss some consequences of the equivalence $Path^H{\simeq}Path$. By combining it with $subst{\equiv}subst^P$ we get the following equivalence:

$$\begin{aligned}&subst{\equiv}{\simeq}Path^H : \{x{\equiv}y : Path\ x\ y\} \to \\ &\qquad (subst\ P\ (from\text{-}path\ x{\equiv}y)\ p \equiv q) \simeq Path^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ p\ q\end{aligned}$$

We can calculate in the following way:

$$\begin{aligned}&subst\ P\ (from\text{-}path\ x{\equiv}y)\ p \equiv q &&\simeq \\ &subst^P\ P\ x{\equiv}y\ p \equiv q &&\simeq \\ &Path\ (subst^P\ P\ x{\equiv}y\ p)\ q &&\simeq \\ &Path^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ p\ q\end{aligned}$$

Thus heterogeneous paths are closely related to the dependent paths used in the types of eliminators for several higher inductive types in the HoTT book [7]. Let us denote the forward direction of the equivalence by $subst{\equiv}{\to}Path^H$ and the other direction by $Path^H{\to}subst{\equiv}$. Let us also give a name to the forward direction of the first two steps of the calculation above:

$$\begin{aligned}&subst{\equiv}{\to}subst^P{\equiv} : \{x{\equiv}y : Path\ x\ y\} \to \\ &\qquad subst\ P\ (from\text{-}path\ x{\equiv}y)\ p \equiv q \to Path\ (subst^P\ P\ x{\equiv}y\ p)\ q\end{aligned}$$

The HoTT book also makes use of a function called *apd*, defined using J [7]. Let us assume that the arbitrary notion of equality comes with such a function, along with a propositional computation rule:

$$\begin{aligned}&cong^D : (f : (x : A) \to P\ x)\ (x{\equiv}y : x \equiv y) \to subst\ P\ x{\equiv}y\ (f\ x) \equiv f\ y \\ &cong^D\text{-}refl : (f : (x : A) \to P\ x) \to cong^D\ f\ (refl\ x) \equiv subst\text{-}refl\end{aligned}$$

We can prove a similar property for paths [5]:

$$\begin{aligned}&cong^H : (f : (x : A) \to P\ x)\ (x{\equiv}y : Path\ x\ y) \to Path^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ (f\ x)\ (f\ y) \\ &cong^H\ f\ x{\equiv}y\ i = f\ (x{\equiv}y\ i)\end{aligned}$$

The functions can be related in the following way:

$$\begin{aligned}&cong^D{\equiv}cong^H : \\ &\quad \{x{\equiv}y : Path\ x\ y\}\ (f : (x : A) \to P\ x) \to \\ &\quad cong^D\ f\ (from\text{-}path\ x{\equiv}y) \equiv Path^H{\to}subst{\equiv}\ (cong^H\ f\ x{\equiv}y)\end{aligned}$$

We can prove $cong^D{\equiv}cong^H$ by defining $cong^{DP}$ (a variant of $cong^D$ for paths) and relating this variant to $cong^D$ as well as $cong^H$.

Given the definition of $subst^P$ above one can define $cong^{DP}$ using *transport* in the following way:

$$\begin{aligned}&cong^{DP} : (f : (x : A) \to P\ x)\ (x{\equiv}y : Path\ x\ y) \to Path\ (subst^P\ P\ x{\equiv}y\ (f\ x))\ (f\ y) \\ &cong^{DP}\ \{P = P\}\ f\ x{\equiv}y = \lambda\ i \to transport\ (\lambda\ j \to P\ (x{\equiv}y\ (max\ i\ j)))\ i\ (f\ (x{\equiv}y\ i))\end{aligned}$$

The proof of the following property relating $cong^D$ and $cong^{DP}$ is omitted (see the accompanying code for details):

$$\begin{aligned}&cong^D{\equiv}cong^{DP} : \\ &\quad \{x{\equiv}y : Path\ x\ y\} \to \\ &\quad subst{\equiv}{\to}subst^P{\equiv}\ (cong^D\ f\ (from\text{-}path\ x{\equiv}y)) \equiv cong^{DP}\ f\ x{\equiv}y\end{aligned}$$

Let us instead focus on the proof of the following property that relates $cong^{DP}$ to $cong^{H}$ ($\_\simeq\_.to$ gives the forward direction of an equivalence, and $\_\simeq\_.from$ the other one):

$$
\begin{aligned}
&cong^{DP}{\equiv}cong^{H} : \\
&\quad \{x{\equiv}y : Path\ x\ y\}\ (f : (x : A) \to P\ x) \to \\
&\quad Path\ (cong^{DP}\ f\ x{\equiv}y)\ (\_\simeq\_.to\ (Path^{H}{\simeq}Path\ (\lambda\ i \to P\ (x{\equiv}y\ i)))\ (cong^{H}\ f\ x{\equiv}y))
\end{aligned}
$$

We can start by using the J rule for the path $x{\equiv}y$, and then calculate in the following way:

$$
\begin{aligned}
&cong^{DP}\ f\ (refl^{P}\ x) && \equiv \\
&(\lambda\ i \to transport\ (\lambda\ \_ \to P\ x)\ i\ (f\ x)) && \equiv \\
&transport\ (\lambda\ i \to Path\ (transport\ (\lambda\ \_ \to P\ x)\ (-\ i)\ (f\ x))\ (f\ x))\ \underline{0}\ (refl^{P}\ (f\ x)) && \equiv \\
&transport\ (\lambda\ i \to Path\ (transport\ (\lambda\ \_ \to P\ x)\ (-\ i)\ (f\ x))\ (f\ x))\ \underline{0} \\
&\quad (transport\ (\lambda\ \_ \to Path\ (f\ x)\ (f\ x))\ \underline{0}\ (refl^{P}\ (f\ x))) && \equiv \\
&\_\simeq\_.to\ (Path^{H}{\simeq}Path\ (\lambda\ i \to P\ (refl^{P}\ x\ i)))\ (cong^{H}\ f\ (refl^{P}\ x))
\end{aligned}
$$

The first and last steps hold by definition. The third step follows from *transport-refl*. Finally the second step uses heterogeneous composition to construct the dashed arrow of the following square:

$$
\begin{array}{ccc}
transport\ (\lambda\ i \to Path\ (transport\ (\lambda\ \_ \to P\ x)\ (-\ i)\ (f\ x))\ (f\ x))\ \underline{0}\ (refl^{P}\ (f\ x)) & & \\
\uparrow & & \\
\lambda\ i \to transport\ (\lambda\ \_ \to P\ x)\ i\ (f\ x) & & \uparrow\ r \\
\uparrow\ l & & \\
refl^{P}\ (f\ x) & \xrightarrow[b]{} & transport\ (\lambda\ \_ \to Path\ (f\ x)\ (f\ x))\ \underline{0}\ (refl^{P}\ (f\ x))
\end{array}
$$

The bottom line in the diagram (b) is defined in the following way:

$$transport\ (\lambda\ \_ \to Path\ (f\ x)\ (f\ x))\ (-\ i)\ (refl^{P}\ (f\ x))$$

The left-hand side of the diagram (l) corresponds to i being $\underline{0}$, and the right-hand side (r) to i being $\underline{1}$. Similarly, the bottom (b) corresponds to j being $\underline{0}$, and the dashed arrow to j being $\underline{1}$. The diagram's left-hand side (l) is defined in the following way:

$$\lambda\ k \to transport\ (\lambda\ \_ \to P\ x)\ (max\ k\ (-\ j))\ (f\ x)$$

The right-hand side (r) is defined in the following way:

$$transport\ (\lambda\ k \to Path\ (transport\ (\lambda\ \_ \to P\ x)\ (-\ min\ k\ j)\ (f\ x))\ (f\ x))\ \underline{0}\ (refl^{P}\ (f\ x))$$

As mentioned above the heterogeneous composition operation allows the types of l and r to vary with j. In this case these expressions have the following type:

$$Path\ (transport\ (\lambda\ \_ \to P\ x)\ (-\ j)\ (f\ x))\ (f\ x)$$

With $cong^{D}{\equiv}cong^{DP}$ and $cong^{DP}{\equiv}cong^{H}$ in place it is easy to prove $cong^{D}{\equiv}cong^{H}$. As a corollary we get the following property that will be used below:

$$
\begin{aligned}
&dependent\text{-}computation\text{-}rule\text{-}lemma : \\
&\quad \{x{\equiv}y : Path\ x\ y\}\ \{fx{\equiv}fy : subst\ P\ (from\text{-}path\ x{\equiv}y)\ (f\ x) \equiv f\ y\} \to \\
&\quad cong^{H}\ f\ x{\equiv}y \equiv subst{\equiv}{\to}Path^{H}\ fx{\equiv}fy \to cong^{D}\ f\ (from\text{-}path\ x{\equiv}y) \equiv fx{\equiv}fy
\end{aligned}
$$

5 The Circle Without Paths

Let us now see how we can make the definition of a higher inductive type—the circle [7]—usable with the arbitrary notion of equality satisfying the axioms from Section 2 (and with *subst*, *subst-refl*, $cong^D$ and $cong^D$-*refl* instantiated in some way, as discussed in Section 4).

Here is the definition of the circle, using paths:

```
data 𝕊¹ : Type where
  base  : 𝕊¹
  loopᴾ : Path base base
```

It is easy to define a variant of loop^P that uses the arbitrary notion of equality instead of a path:

```
loop : base ≡ base
loop = from-path loopᴾ
```

What about the eliminator? An eliminator that uses paths can be defined in the following way:

```
elimᴾ : (P : 𝕊¹ → Type p) (b : P base) → Pathᴴ (λ i → P (loopᴾ i)) b b → (x : 𝕊¹) → P x
elimᴾ P b ℓ base      = b
elimᴾ P b ℓ (loopᴾ i) = ℓ i
```

Now it is easy to use $subst{\equiv}{\to}Path^H$ to construct an eliminator that uses the arbitrary notion of equality instead. The type signature matches the one given in the HoTT book [7]:

```
elim : (P : 𝕊¹ → Type p) (b : P base) (ℓ : subst P loop b ≡ b) (x : 𝕊¹) → P x
elim P b ℓ = elimᴾ P b (subst≡→Pathᴴ ℓ)
```

The HoTT book gives two computation rules for the eliminator. The one for the point constructor base is stated to be definitional, and that is the case here. The one for the higher constructor is given as an equality, and we can do the same thing:

```
elim-loop : congᴰ (elim P b ℓ) loop ≡ ℓ
elim-loop = dependent-computation-rule-lemma (refl _)
```

The proof simply applies *dependent-computation-rule-lemma* to reflexivity. Things have been set up in such a way that $cong^H\ (elim\ P\ b\ \ell)\ \mathsf{loop}^\mathsf{P}$ is definitionally equal to $subst{\equiv}{\to}Path^H\ \ell$; every step of the following calculation holds by definition:

```
congᴴ (elim P b ℓ) loopᴾ                          ≡
(λ i → elim P b ℓ (loopᴾ i))                      ≡
(λ i → elimᴾ P b (subst≡→Pathᴴ ℓ) (loopᴾ i))       ≡
(λ i → subst≡→Pathᴴ ℓ i)                          ≡
subst≡→Pathᴴ ℓ
```

We can also define a non-dependent eliminator. This definition does not require most of the machinery introduced above. Here is a non-dependent eliminator for paths:

```
recᴾ : (b : A) → Path b b → 𝕊¹ → A
recᴾ = elimᴾ _
```

This variant can be used to define an eliminator for the arbitrary notion of equality:

$$
\begin{aligned}
&\mathit{rec} : (b : A) \to b \equiv b \to \mathbb{S}^1 \to A\\
&\mathit{rec}\ b\ \ell = \mathit{rec}^P\ b\ (\mathit{to\text{-}path}\ \ell)
\end{aligned}
$$

We simply convert the equality to a path. The computation rule for the higher constructor is stated using *cong*, a function that, along with a propositional computation rule, is assumed to come with our arbitrary notion of equality (these functions could be defined using J):

$$
\begin{aligned}
&\mathit{cong}\ \ \ \ \ \ : (f : A \to B) \to x \equiv y \to f\ x \equiv f\ y\\
&\mathit{cong\text{-}refl} : \mathit{cong}\ f\ (\mathit{refl}\ x) \equiv \mathit{refl}\ (f\ x)
\end{aligned}
$$

The computation rule can be stated and proved in the following way:

$$
\begin{aligned}
&\mathit{rec\text{-}loop} : \mathit{cong}\ (\mathit{rec}\ b\ \ell)\ \mathit{loop} \equiv \ell\\
&\mathit{rec\text{-}loop} = \mathit{non\text{-}dependent\text{-}computation\text{-}rule\text{-}lemma}\ (\mathit{refl}\ \_)
\end{aligned}
$$

Here *non-dependent-computation-rule-lemma* is a lemma that is easy to prove:

$$
\begin{aligned}
&\mathit{non\text{-}dependent\text{-}computation\text{-}rule\text{-}lemma} :\\
&\quad \{x{\equiv}y : \mathit{Path}\ x\ y\}\ \{fx{\equiv}fy : f\ x \equiv f\ y\} \to\\
&\quad \mathit{cong}^H\ f\ x{\equiv}y \equiv \mathit{to\text{-}path}\ fx{\equiv}fy \to \mathit{cong}\ f\ (\mathit{from\text{-}path}\ x{\equiv}y) \equiv fx{\equiv}fy
\end{aligned}
$$

An alternative is to define the non-dependent eliminator in terms of the dependent one:

$$
\begin{aligned}
&\mathit{rec}' : (b : A) \to b \equiv b \to \mathbb{S}^1 \to A\\
&\mathit{rec}'\ b\ \ell = \mathit{elim}\ \_\ b\ (\mathit{trans}\ \mathit{subst\text{-}const}\ \ell)
\end{aligned}
$$

Here *trans* and *subst-const* have the following types:

$$
\begin{aligned}
&\mathit{trans}\ \ \ \ \ \ \ \ : x \equiv y \to y \equiv z \to x \equiv z\\
&\mathit{subst\text{-}const} : \mathit{subst}\ (\lambda\ \_ \to A)\ x{\equiv}y\ z \equiv z
\end{aligned}
$$

The third argument to *elim* thus captures the following calculation, where the first step uses *subst-const* and the second uses ℓ:

$$
\begin{aligned}
&\mathit{subst}\ (\lambda\ \_ \to A)\ \mathit{loop}\ b\ \ \equiv\\
&b\ \equiv\\
&b
\end{aligned}
$$

The computation rule can be proved using the computation rule for the dependent eliminator:

$$
\begin{aligned}
&\mathit{rec}'\text{-}\mathit{loop} : \mathit{cong}\ (\mathit{rec}'\ b\ \ell)\ \mathit{loop} \equiv \ell\\
&\mathit{rec}'\text{-}\mathit{loop} = \mathit{cong}^D{\equiv}{\to}\mathit{cong}{\equiv}\ \mathit{elim\text{-}loop}
\end{aligned}
$$

The proof uses the following lemma:

$$
\begin{aligned}
&\mathit{cong}^D{\equiv}{\to}\mathit{cong}{\equiv} :\\
&\quad \{x{\equiv}y : x \equiv y\}\ \{fx{\equiv}fy : f\ x \equiv f\ y\} \to\\
&\quad \mathit{cong}^D\ f\ x{\equiv}y \equiv \mathit{trans}\ \mathit{subst\text{-}const}\ fx{\equiv}fy \to \mathit{cong}\ f\ x{\equiv}y \equiv fx{\equiv}fy
\end{aligned}
$$

6 Set Quotients Without Paths

The higher inductive type given for the circle does not include any truncation constructor, i.e. a constructor that states directly that the type has a certain *h-level*. As an example of such a higher inductive type this section treats set quotients [7], which come with a truncation constructor that ensures that the resulting types are sets (in the sense of the HoTT book [7]).

A type of h-level n is an $(n - 2)$-type:

```
Contractible : Type a → Type a
Contractible A = Σ A λ x → (y : A) → x ≡ y

H-level : ℕ → Type a → Type a
H-level zero          A = Contractible A
H-level (suc zero)    A = (x y : A) → x ≡ y
H-level (suc (suc n)) A = {x y : A} → H-level (suc n) (x ≡ y)
```

Propositions (or mere propositions) are types of h-level 1, and sets are types of h-level 2:

```
Is-proposition : Type a → Type a
Is-proposition = H-level 1

Is-set : Type a → Type a
Is-set = H-level 2
```

Let $\mathit{H\text{-}level}^P$, $\mathit{Is\text{-}proposition}^P$ and $\mathit{Is\text{-}set}^P$ refer to the corresponding concepts defined using paths instead of the arbitrary notion of equality.

Now we can define set quotients (the definition is similar to the one in the HoTT book [7], but the relations are not required to be propositional, following Mörtberg [5]):

```
data _/_ (A : Type a) (R : A → A → Type r) : Type (a ⊔ r) where
  [_]                    : A → A / R
  []-respects-relationᴾ  : R x y → Path [ x ] [ y ]
  /-is-setᴾ              : Is-setᴾ (A / R)
```

(Here _⊔_ is a maximum operator for universe levels.) The constructor [_]—box—takes values from the underlying type to the quotient, and the type of []-respects-relation$^{\mathsf{P}}$ implies that box maps related values to equal values. If we expand $\mathit{Is\text{-}set}^P$, then we see that the /-is-set$^{\mathsf{P}}$ constructor takes two paths between quotient values, and returns a path between paths:

```
{x y : A / R} (eq₁ eq₂ : Path x y) → Path eq₁ eq₂
```

A direct definition of an eliminator could take the following form:

```
elimᴾ′ : (P : A / R → Type p)
         (f : ∀ x → P [ x ])
         (g : ∀ {x y} (r : R x y) → Pathᴴ (λ i → P ([]-respects-relationᴾ r i)) (f x) (f y)) →
         (∀ {x y} {eq₁ eq₂ : Path x y} {p : P x} {q : P y}
           (eq₃ : Pathᴴ (λ i → P (eq₁ i)) p q) (eq₄ : Pathᴴ (λ i → P (eq₂ i)) p q) →
           Pathᴴ (λ i → Pathᴴ (λ j → P (/-is-setᴾ eq₁ eq₂ i j)) p q) eq₃ eq₄) →
         (x : A / R) → P x
```

$$
\begin{aligned}
&\mathit{elim}^{P\prime}\ P\ f\ g\ h\ [\ x\ ] &&= f\ x\\
&\mathit{elim}^{P\prime}\ P\ f\ g\ h\ ([]\text{-}\mathsf{respects\text{-}relation}^{\mathsf{P}}\ r\ i) &&= g\ r\ i\\
&\mathit{elim}^{P\prime}\ P\ f\ g\ h\ (/\text{-}\mathsf{is\text{-}set}^{\mathsf{P}}\ p\ q\ i\ j) &&=\\
&\quad h\ (\lambda\ i \to \mathit{elim}^{P\prime}\ P\ f\ g\ h\ (p\ i))\ (\lambda\ i \to \mathit{elim}^{P\prime}\ P\ f\ g\ h\ (q\ i))\ i\ j
\end{aligned}
$$

However, the type of the penultimate argument might look somewhat daunting. It can be replaced by the requirement that the motive P is a family of sets. (For performance reasons the accompanying code defines the eliminator in a slightly different way: the second, third and fourth arguments are bundled up using a record type with η-equality turned off. Other eliminators from this section are also defined in this way in the accompanying code.)

Before defining an alternative eliminator, let us prove some lemmas. Consider the following variant of the final clause of $\mathit{H\text{-}level}^P$:

$$
\begin{aligned}
&\mathit{H\text{-}level}^P\text{-}\mathit{suc}{\simeq}\mathit{H\text{-}level}^P\text{-}\mathit{Path}^H\ :\\
&\quad \{P : I \to \mathit{Type}\ p\} \to\\
&\quad \mathit{H\text{-}level}^P\ (\mathsf{suc}\ n)\ (P\ i) \simeq ((x : P\ \underline{0})\ (y : P\ \underline{1}) \to \mathit{H\text{-}level}^P\ n\ (\mathit{Path}^H\ P\ x\ y))
\end{aligned}
$$

This variant can be proved by calculating in the following way:

$$
\begin{aligned}
&\mathit{H\text{-}level}^P\ (\mathsf{suc}\ n)\ (P\ i) && \simeq\\
&\mathit{H\text{-}level}^P\ (\mathsf{suc}\ n)\ (P\ \underline{1}) && \simeq\\
&((x\ y : P\ \underline{1}) \to \mathit{H\text{-}level}^P\ n\ (x \equiv y)) && \simeq\\
&((x : P\ \underline{0})\ (y : P\ \underline{1}) \to \mathit{H\text{-}level}^P\ n\ (\mathit{Path}\ (\mathit{transport}\ P\ \underline{0}\ x)\ y)) && \simeq\\
&((x : P\ \underline{0})\ (y : P\ \underline{1}) \to \mathit{H\text{-}level}^P\ n\ (\mathit{Path}^H\ P\ x\ y))
\end{aligned}
$$

The first step follows from the following equality:

$$
\begin{aligned}
&\mathit{index\text{-}irrelevant} : (P : I \to \mathit{Type}\ p) \to \forall\ i\ j \to \mathit{Path}\ (P\ i)\ (P\ j)\\
&\mathit{index\text{-}irrelevant}\ P\ i\ j\ k = P\ (\mathit{max}\ (\mathit{min}\ i\ (-\ k))\ (\mathit{min}\ j\ k))
\end{aligned}
$$

The second step is related to Lemma 3.11.10 from the HoTT book [7], and the fourth step uses $\mathit{Path}^H{\simeq}\mathit{Path}$. The third step uses *index-irrelevant* again, as well as the following preservation lemma:

$$
\begin{aligned}
&\Pi\text{-}\mathit{cong} : (A{\simeq}B : A \simeq B) \to (\forall\ x \to P\ x \simeq Q\ (\_{\simeq}\_.\mathit{to}\ A{\simeq}B\ x)) \to\\
&\qquad ((x : A) \to P\ x) \simeq ((x : B) \to Q\ x)
\end{aligned}
$$

This step also uses *transport-refl* and the following equality:

$$
\begin{aligned}
&\mathit{transport\text{-}transport} : (P : I \to \mathit{Type}\ p)\ \{p : P\ \underline{0}\} \to\\
&\qquad\qquad \mathit{Path}\ (\mathit{transport}\ (\lambda\ i \to P\ (-\ i))\ \underline{0}\ (\mathit{transport}\ P\ \underline{0}\ p))\ p
\end{aligned}
$$

We can use $\mathit{H\text{-}level}^P\text{-}\mathit{suc}{\simeq}\mathit{H\text{-}level}^P\text{-}\mathit{Path}^H$ to prove that a heterogeneous notion of proof irrelevance holds for families of propositions:

$$
\begin{aligned}
&\mathit{heterogeneous\text{-}irrelevance}\ :\\
&\quad (\forall\ x \to \mathit{Is\text{-}proposition}^P\ (P\ x)) \to\\
&\quad \{x{\equiv}y : \mathit{Path}\ x\ y\}\ \{p : P\ x\}\ \{q : P\ y\} \to \mathit{Path}^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ p\ q
\end{aligned}
$$

We can reason in the following way:

$$
\begin{aligned}
&(\forall\ x \to \mathit{Is\text{-}proposition}^P\ (P\ x)) && \to\\
&\mathit{Is\text{-}proposition}^P\ (P\ x) && \to
\end{aligned}
$$

$$
\begin{array}{ll}
\mathit{Is\text{-}proposition}^P\ (P\ (x{\equiv}y\ \underline{0})) & \to \\
\mathit{Contractible}^P\ (\mathit{Path}^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ p\ q) & \to \\
\mathit{Path}^H\ (\lambda\ i \to P\ (x{\equiv}y\ i))\ p\ q &
\end{array}
$$

The third step follows from $\mathit{H\text{-}level}^P\text{-}\mathit{suc}{\simeq}\mathit{H\text{-}level}^P\text{-}\mathit{Path}^H$, and the last step uses the fact that contractible types are inhabited. We can also prove a similar result for families of sets:[2]

$$
\begin{array}{l}
\mathit{heterogeneous\text{-}UIP} : \\
\quad (\forall\ x \to \mathit{Is\text{-}set}^P\ (P\ x)) \to \\
\quad \{\mathit{eq}_1\ \mathit{eq}_2 : \mathit{Path}\ x\ y\}\ \{\mathit{eq}_3 : \mathit{Path}\ \mathit{eq}_1\ \mathit{eq}_2\}\ \{p_1 : P\ x\}\ \{p_2 : P\ y\} \\
\quad (\mathit{eq}_4 : \mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_1\ j))\ p_1\ p_2) \\
\quad (\mathit{eq}_5 : \mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_2\ j))\ p_1\ p_2) \to \\
\quad \mathit{Path}^H\ (\lambda\ i \to \mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_3\ i\ j))\ p_1\ p_2)\ \mathit{eq}_4\ \mathit{eq}_5
\end{array}
$$

The proof is very similar to the previous one:

$$
\begin{array}{ll}
(\forall\ x \to \mathit{Is\text{-}set}^P\ (P\ x)) & \to \\
\mathit{Is\text{-}set}^P\ (P\ x) & \to \\
\mathit{Is\text{-}set}^P\ (P\ (\mathit{eq}_3\ \underline{0}\ \underline{0})) & \to \\
\mathit{Is\text{-}proposition}^P\ (\mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_3\ \underline{0}\ j))\ p_1\ p_2) & \to \\
\mathit{Contractible}^P\ (\mathit{Path}^H\ (\lambda\ i \to \mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_3\ i\ j))\ p_1\ p_2)\ \mathit{eq}_4\ \mathit{eq}_5) & \to \\
\mathit{Path}^H\ (\lambda\ i \to \mathit{Path}^H\ (\lambda\ j \to P\ (\mathit{eq}_3\ i\ j))\ p_1\ p_2)\ \mathit{eq}_4\ \mathit{eq}_5 &
\end{array}
$$

Here $\mathit{H\text{-}level}^P\text{-}\mathit{suc}{\simeq}\mathit{H\text{-}level}^P\text{-}\mathit{Path}^H$ is used twice, once in the third step and once in the fourth.

Now let us go back to the set quotients. Using *heterogeneous-UIP* it is easy to implement the following dependent eliminator and a corresponding non-dependent eliminator:

$$
\begin{array}{l}
\mathit{elim}^P : (P : A\ /\ R \to \mathit{Type}\ p) \\
\qquad (f : \forall\ x \to P\ [\ x\ ]) \to \\
\qquad (\forall\ \{x\ y\}\ (r : R\ x\ y) \to \mathit{Path}^H\ (\lambda\ i \to P\ ([]\text{-respects-relation}^{\mathsf{P}}\ r\ i))\ (f\ x)\ (f\ y)) \to \\
\qquad (\forall\ x \to \mathit{Is\text{-}set}^P\ (P\ x)) \to \\
\qquad (x : A\ /\ R) \to P\ x \\
\mathit{elim}^P\ P\ f\ g\ s = \mathit{elim}^{P\prime}\ P\ f\ g\ (\mathit{heterogeneous\text{-}UIP}\ s)
\end{array}
$$

$$
\begin{array}{l}
\mathit{rec}^P : (f : A \to B) \to (\forall\ \{x\ y\} \to R\ x\ y \to \mathit{Path}\ (f\ x)\ (f\ y)) \to \mathit{Is\text{-}set}^P\ B \to A\ /\ R \to B \\
\mathit{rec}^P\ f\ g\ s = \mathit{elim}^P\ \_\ f\ g\ (\lambda\ \_ \to s)
\end{array}
$$

With the non-dependent eliminator one can define a function from $A\ /\ R$ to a set B by giving a function from A to B that respects the relation.

Let us now define variants without paths of the higher constructors and the last two eliminators. The first higher constructor can be treated in the same way as before:

$$
\begin{array}{l}
[]\text{-}\mathit{respects\text{-}relation} : R\ x\ y \to \_{\equiv}\_\ \{A = A\ /\ R\}\ [\ x\ ]\ [\ y\ ] \\
[]\text{-}\mathit{respects\text{-}relation} = \mathit{from\text{-}path} \circ []\text{-respects-relation}^{\mathsf{P}}
\end{array}
$$

For the other one we can start by noting that the two definitions of h-levels given above are pointwise equivalent:

$$
\mathit{H\text{-}level}{\simeq}\mathit{H\text{-}level}^P : \forall\ n \to \mathit{H\text{-}level}\ n\ A \simeq \mathit{H\text{-}level}^P\ n\ A
$$

[2] Zesen Qian has proved more or less the same result, but in a different way [5, Git commit 9a4f3cf3c733db82344bfc98b82f405101df816a].

It is then easy to define the variant of the second constructor:

```
/-is-set : Is-set (A / R)
/-is-set = _≃_.from (H-level≃H-level^P 2) /-is-set^P
```

The dependent eliminator can be defined in the following way, using $subst{\equiv}{\to}Path^H$ and $H\text{-}level{\simeq}H\text{-}level^P$:

```
elim : (P : A / R → Type p)
       (f : ∀ x → P [ x ]) →
       (∀ {x y} (r : R x y) → subst P ([]-respects-relation r) (f x) ≡ f y) →
       (∀ x → Is-set (P x)) →
       (x : A / R) → P x
elim P f g s = elim^P P f (subst≡→Path^H ∘ g) (_≃_.to (H-level≃H-level^P 2) ∘ s)
```

Finally it is easy to define a variant of rec^P:

```
rec : (f : A → B) → (∀ {x y} → R x y → f x ≡ f y) → Is-set B → A / R → B
rec f g s = rec^P f (to-path ∘ g) (_≃_.to (H-level≃H-level^P 2) s)
```

I have not included computation rules for the higher constructors, because sets have propositional equality types (i.e. any two equality proofs of the same type are equal).

7 More Examples

Let us now consider more examples. No new functionality is introduced in this section: the functions introduced above suffice to handle a large number of examples.

The suspension type constructor and corresponding eliminators can be defined in the following way [7]:

```
data Susp (A : Type a) : Type a where
  north      : Susp A
  south      : Susp A
  meridian^P : A → Path north south

elim^P : (P : Susp A → Type p) (n : P north) (s : P south) →
         (∀ x → Path^H (λ i → P (meridian^P x i)) n s) →
         (x : Susp A) → P x
elim^P _ n s n≡s north          = n
elim^P _ n s n≡s south          = s
elim^P _ n s n≡s (meridian^P x i) = n≡s x i

rec^P : (n s : B) → (A → Path n s) → Susp A → B
rec^P = elim^P _
```

Variants of the higher constructor and the eliminators, and two propositional computation rules, can then be defined in the following way:

```
meridian : A → _≡_ {A = Susp A} north south
meridian = from-path ∘ meridian^P
```

```
elim : (P : Susp A → Type p) (n : P north) (s : P south) →
       (∀ x → subst P (meridian x) n ≡ s) →
       (x : Susp A) → P x
elim P n s n≡s = elim^P P n s (subst≡→Path^H ∘ n≡s)

elim-meridian : (P : Susp A → Type p) (n : P north) (s : P south)
                (n≡s : ∀ x → subst P (meridian x) n ≡ s) →
                cong^D (elim P n s n≡s) (meridian x) ≡ n≡s x
elim-meridian _ _ _ _ = dependent-computation-rule-lemma (refl _)

rec : (n s : B) → (A → n ≡ s) → Susp A → B
rec n s n≡s = rec^P n s (to-path ∘ n≡s)

rec-meridian : (n s : B) (n≡s : A → n ≡ s) →
               cong (rec n s n≡s) (meridian x) ≡ n≡s x
rec-meridian _ _ _ = non-dependent-computation-rule-lemma (refl _)
```

Note that most of the text consists of type signatures, and that the lemmas introduced above can be used unchanged.

As a second example of a higher inductive type with a truncation constructor, let us now return to the propositional truncation operator from Section 1 (with trivial renamed to $\mathsf{trivial}^{\mathsf{P}}$):

```
data ∥_∥ (A : Type a) : Type a where
  |_|       : A → ∥ A ∥
  trivial^P : (x y : ∥ A ∥) → Path x y
```

The following eliminator was given in Section 4 (but it was called $elim^P$):

```
elim^P′ : (P : ∥ A ∥ → Type p) →
          ((x : A) → P ∣ x ∣) →
          ({x y : ∥ A ∥} (p : P x) (q : P y) → Path^H (λ i → P (trivial^P x y i)) p q) →
          (x : ∥ A ∥) → P x
elim^P′ P f t ∣ x ∣             = f x
elim^P′ P f t (trivial^P x y i) = t (elim^P′ P f t x) (elim^P′ P f t y) i
```

Just like for the set quotients in Section 6 we can define an alternative eliminator and a corresponding non-dependent eliminator:

```
elim^P : (P : ∥ A ∥ → Type p) →
         ((x : A) → P ∣ x ∣) →
         (∀ x → Is-proposition^P (P x)) →
         (x : ∥ A ∥) → P x
elim^P P f p = elim^P′ P f (λ _ _ → heterogeneous-irrelevance p)

rec^P : (A → B) → Is-proposition^P B → ∥ A ∥ → B
rec^P f p = elim^P _ f (λ _ → p)
```

Variants of the higher constructor and the eliminators can then be defined in the following way:

```
trivial : Is-proposition ∥ A ∥
trivial = _≃_.from (H-level≃H-level^P 1) trivial^P
```

```
elim : (P : ∥ A ∥ → Type p) →
       ((x : A) → P ∣ x ∣) →
       (∀ x → Is-proposition (P x)) →
       (x : ∥ A ∥) → P x
elim P f p = elim^P P f (_≃_.to (H-level≃H-level^P 1) ∘ p)

rec : (A → B) → Is-proposition B → ∥ A ∥ → B
rec f p = rec^P f (_≃_.to (H-level≃H-level^P 1) p)
```

Note again that the lemmas introduced above can be used unchanged. (Computation rules for the higher constructors are omitted, because propositions have propositional equality types.)

8 An Alternative Approach

This section discusses an alternative approach, suggested by an anonymous reviewer. The circle is used as an example.

We can write down the formation, introduction, elimination and computation rules of the circle using Σ-types in the following way:

```
Circle : (p : Level) → Type (lsuc p)
Circle p =
  Σ Type          λ 𝕊¹ →
  Σ 𝕊¹            λ base →
  Σ (base ≡ base) λ loop →
    (P : 𝕊¹ → Type p) (b : P base) (ℓ : subst P loop b ≡ b) →
    Σ ((x : 𝕊¹) → P x) λ elim →
    Σ (elim base ≡ b)  λ elim-base →
      subst (λ b → subst P loop b ≡ b) elim-base (cong^D elim loop) ≡ ℓ
```

Note that the definition is parametrised by the level of the universe into which the eliminator should eliminate (*lsuc* is a successor operation for levels). Note also that the computation rule for *loop* is more complicated than in Section 5; the reason is that the computation rule for *base* does not hold by definition.

We can also write down a corresponding definition that uses paths instead of the arbitrary notion of equality and prove that the two definitions are pointwise equivalent:

```
Circle^P          : (p : Level) → Type (lsuc p)
Circle^P≃Circle : Circle^P p ≃ Circle p
```

Finally we can prove that $Circle^P\ p$ is inhabited and use the equivalence to derive an implementation of *Circle p*:

```
circle^P : Circle^P p
circle   : Circle p
```

It is even possible to do this in such a way that the derived eliminator gets the "right" computational behaviour for the point constructor (see the accompanying code for details). This implies that the less complicated computation rule given for the higher constructor in Section 5 can be proved.

However, when I followed this method I ended up with quite a bit more code (excluding library code) than when I used the approach demonstrated above. Here is an implementation of $circle^P$, with the proof of the second computation rule omitted:

$$\mathbb{S}^1 \text{ , } \mathsf{base} \text{ , } \mathsf{loop}^{\mathsf{P}} \text{ , } \lambda\ P\ b\ \ell \to elim^P\ P\ b\ (subst{\equiv}{\to}Path^H\ \{P = P\}\ \ell) \text{ , } refl^P\ b \text{ , } \ldots$$

The code uses a variant of $subst{\equiv}{\to}Path^H$ for paths, and the omitted proof of the second computation rule uses a variant of *dependent-computation-rule-lemma* for paths, plus an extra lemma (due to the more complicated formulation of the computation rule). If we do not count the size of library code then this code ($Circle^P$ and $circle^P$) is already about as large as the definition of the eliminator and computation rule given in Section 5. In addition we have to prove $Circle^P{\simeq}Circle$, and if we are not careful we can end up with an eliminator that does not have the right computational behaviour for the point constructor. Finally we can write a little more code to establish the less complicated formulation of the computation rule for the higher constructor.

9 Discussion

The approach used for suspensions in Section 7 works for several other higher inductive types from the HoTT book [7], including the interval, pushouts, and a general truncation operator (see the accompanying code for details). Furthermore I have shown how one can handle propositional truncation and set truncation constructors. The higher constructors of these types—with the exception of the truncation constructors—satisfy the following two properties:

- All constructors return paths between points.
- No constructor takes a path involving the type family that is being defined as input (ignoring the possibility of later instantiating parameters to such paths).

This might seem to be a serious limitation. However, the HoTT book mentions a method for avoiding paths as inputs [7, Section 6.9], involving the use of auxiliary higher inductive types. It also discusses a method for avoiding higher constructors that return paths between paths, using a "hub" and "spokes" [7, Section 6.7]. The hub-and-spokes construction is used to define the general truncation operator. A potential drawback of the hub-and-spokes construction is that the resulting eliminators may have different computational behaviour [7, Remark 6.7.2], but that is already the case for the methods discussed here.

If the computational behaviour for higher constructors is important, then I do not advocate using the techniques discussed above. Working directly with paths can lead to better computational behaviour: the J rule might not compute in the usual way, but instead there are new definitional equalities. To illustrate this point: The HoTT book mentions another higher inductive type, the torus, given using the hub-and-spokes construction. The definition refers to the circle. I could use a variation of the method described in this text to give an interface to the torus. However, when doing this I found it easier to work with the path-based interface to the circle than the one using an arbitrary notion of equality. (For details of my definition, see the accompanying code.)

References

1 The Agda Team. *Agda User Manual, Release 2.6.1*, 2020. URL: `https://readthedocs.org/projects/agda/downloads/pdf/v2.6.1/`.

2 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs, TYPES 2015*, number 69 in LIPIcs, page 5:1–5:34, 2018. `doi:10.4230/LIPIcs.TYPES.2015.5`.

3 Nils Anders Danielsson. Code related to the paper "Higher Inductive Type Eliminators Without Paths", 2020. doi:10.5281/zenodo.3941063.

4 Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five Years of Constructive Type Theory: Proceedings of a Congress Held in Venice, October 1995*, volume 36 of *Oxford Logic Guides*, page 83–111. Oxford University Press, 1998.

5 Anders Mörtberg, Andrea Vezzosi, et al. An experimental library for Cubical Agda. Agda code, 2020. URL: https://github.com/agda/cubical/.

6 Andrew Swan. An algebraic weak factorisation system on 01-substitution sets: A constructive proof. *Journal of Logic & Analysis*, 8(1):1–35, 2016. doi:10.4115/jla.2016.8.1.

7 The Univalent Foundations Program. Homotopy type theory, 2013. First edition. URL: https://homotopytypetheory.org/book/.

8 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87:1–87:29, 2019. doi:10.1145/3341691.