Polynomial Universes and Dependent Types

C.B. Aberlé David I. Spivak

Abstract

Awodey, later with Newstead, showed how polynomial pseudomonads $(u, 1, \Sigma)$ with extra structure (termed "natural models" by Awodey) hold within them the syntax and rules for dependent type theory. Their work presented these ideas clearly but ultimately led them outside of the category of polynomial functors in order to explain all of the structure possessed by such models of type theory.

This paper builds off that work—explicating the syntax and rules for dependent type theory by axiomatizing them *entirely* in the language of polynomial functors. In order to handle the higher-categorical coherences required for such an explanation, we work with polynomial functors internally in the language of Homotopy Type Theory, which allows for higher-dimensional structures such as pseudomonads, etc. to be expressed purely in terms of the structure of a suitably-chosen ∞-category of polynomial functors. The move from set theory to Homotopy Type Theory thus has a twofold effect of enabling a simpler exposition of natural models, which is at the same time amenable to formalization in a proof assistant, such as Agda.

Moreover, the choice to remain firmly within the setting of polynomial functors reveals many additional structures of natural models that were otherwise left implicit or not considered by Awodey & Newstead. Chief among these, we highlight the fact that every polynomial pseudomonad $(u, 1, \Sigma)$ as above that is also equipped with structure to interpret dependent product types gives rise to a self-distributive law $u \triangleleft u \to u \triangleleft u$, which witnesses the usual distributive law of dependent products over dependent sums.

1 Introduction

Dependent type theory [Mar75] was founded by Per Martin-Löf in 1975 (among others) to formalize constructive mathematics. The basic idea is that the *order of events* is fundamental to the mathematical story arc: when playing out any specific example story in that arc, the beginning of the story affects not only the later events, but even the very terms with which the later events will be described. For example, in the story arc of conditional probability, one may say "now if the set P that we are asked to condition on happens to have measure zero, we must stop; but assuming that's not the case then the result will be a new probability measure.'' Here the story teller is saying that no terms will describe what happens if P has measure zero, whereas otherwise the terms of standard probability will apply.

Dependent types form a logical system with syntax, rules of computation, and methods of deduction. In [Awo14; AN18], Awodey and later Newstead show that there is a strong connection between dependent type theory and polynomial functors, via their concept of *natural models*, which cleanly solve the problem of *strictififying* certain identities that typically hold only up to isomorphism in arbitrary categories, but must hold *strictly* in order for these to soundly model dependent type theory. The solution to this problem offered by Awodey and Newstead makes use of the type-theoretic concept of a *universe*. Such universes then turn out to naturally be regarded as polynomial functors on a suitably-chosen category of presheaves, satisfying a certain *representability* condition.

Although the elementary structure of natural models is thus straightforwardly described by considering them as objects in the category of polynomial functors, Awodey and Newstead were ultimately led outside of this category in order to fully explicate those parts of natural models that require identities to hold only *up to isomorphism*, rather than strictly. There is thus an evident tension between *strict* and *weak* identities that has not yet been fully resolved in the story of natural models. In the present work, we build on Awodey and Newstead's work to fully resolve this impasse by showing how type universes can be fully axiomatized in terms of polynomial functors, by working with polynomial functors internally in the language of *Homotopy Type Theory* (HoTT). We thus come full circle from Awodey's original motivation to develop natural models *of* Homotopy Type Theory, to describing natural models *in* Homotopy Type Theory.

The ability for us to tell the story of natural models as set entirely in the category of polynomial functors has a great simplifying effect upon the resultant theory, and reveals many additional structures, both of polynomial universes, and of the category of polynomial functors as a whole. As an illustration of this, we show how every polynomial universe u, regarded as a polynomial pseudomonad with additional structure, gives rise to self-distributive law $u \triangleleft u \longrightarrow u \triangleleft u$, which in particular witnesses the usual distributive law of dependent products over dependent sums.

Moreover, the move from set theory to HoTT as a setting in which to tell this story enables new tools to be applied for its telling. In particular, the account of polynomial universes we develop is well-suited to formalization in a proof assistant, and we present such a formalization in Agda. This paper is thus itself a literate Agda document in which all results have been fully formalized and checked for validity.

```
{-# OPTIONS --without-K --rewriting #-}
module poly-universes where

open import Agda.Primitive
open import Agda.Builtin.Sigma
open import Agda.Builtin.Unit
```

The structure of this paper is as follows:

2 Background on Type Theory, Natural Models & HoTT

We begin with a recap of natural models, dependent type theory, and HoTT, taking this also as an opportunity to introduce the basics of our Agda formalization.

2.1 Dependent Types and their Categorical Semantics

The question "what is a type" is as deep as it is philosophically fraught. For present purposes, however, we need not concern ourselves so much directly with what (dependent) type *are*, as with what they can *do*, and how best to mathematically model this behavior. Suffice it to say, then, that a type specifies rules for both constructing and using the *inhabitants* of that type in arbitrary contexts of usage. Following standard conventions, we use the notation `a : A` to mean that `a` is an inhabitant of type `A`.

In Agda, one example of such a type is the *unit type* `⊤`, which is defined to have a single inhabitant `tt : ⊤`, such that for any other inhabitant `x : ⊤` we must have `x = tt`.

Another type (or rather, family of types) of particular importance is the *universe* of types `Type`, whose inhabitants themsleves represent types.[1] So e.g. to say that `⊤`, as defined above, is a type, we may simply write `⊤ : Type`.

```
Type : (ℓ : Level) → Set (lsuc ℓ)
Type ℓ = Set ℓ
```

Given a type `A`, one may in turn consider families of types `B x` indexed by, or *dependent* upon aribtrary inhabitants `x : A`. In agda, we represent such a type family `B` as a function `A → Type`.

Given a type `A : Type` and a family of types `B : A → Type` as above, two key examples of types we may construct are:

- The *dependent function type* `(x : A) → B x`, whose inhabitants are functions `λ x → f x` such that, for all `a : A`, we have `f a : B a`.
- The *dependent pair type* `Σ A B`, whose inhabitants are of the form `(a , b)` for `a : A` and `b : B a`, such that there are functions `fst : Σ A B → A` and `snd : (p : Σ A B) → B (fst p)`.

Note that in the case where `B` does not depend upon `x : A` (i.e. the variable `x` does not appear in the expression for `B`), these correspond to the more familiar function type `A → B` and pair type `A × B`, respectively. E.g. we can define the Cartesian product of two types `A` and `B` as follows:

```
_×_ : ∀ {ℓ κ} (A : Type ℓ) (B : Type κ) → Type (ℓ ⊔ κ)
A × B = Σ A (λ _ → B)
```

[1] For consistency with the usage of the term "set" in HoTT (whereby sets are types satisfying a certain *truncation* condition, to be explained shortly,) we relabel Agda's universes of types as `Type`, rather than the default `Set`.

In more traditional type-theoretic notation, one might see the rules for these types written as follows:

$$\Gamma \vdash \top : \mathsf{Type} \qquad \Gamma \vdash \mathsf{tt} : \top \qquad \frac{\Gamma \vdash x : \top}{\Gamma \vdash x = tt}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B[x] : \mathsf{Type}}{\Gamma \vdash \Pi x : A.B[x] : \mathsf{Type}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B[x] : \mathsf{Type}}{\Gamma \vdash \Sigma x : A.B[x] : \mathsf{Type}}$$

$$\frac{\Gamma, x : A \vdash f[x] : B[x]}{\Gamma \vdash \lambda x.f[x] : \Pi x : A.B[x]} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : \Sigma x : A.B[x]}$$

$$\frac{\Gamma \vdash f : \Pi x : A.B[x] \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[a]} \qquad \frac{\Gamma \vdash p : \Sigma x : A.B[x]}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : \Sigma x : A.B[x]}{\Gamma \vdash \pi_2(p) : B[\pi_1(p)]}$$

$$(\lambda x.f[x])a = f[a] \qquad \pi_1(a, b) = a \quad \pi_2(a, b) = b$$

$$f = \lambda x.fx \qquad p = (\pi_1(p), \pi_2(p))$$

The constructors λ and $(-,-)$ are called the *introduction* forms of $\Pi x : A.B[x]$ and $\Sigma x : A.B[x]$, while fa and $\pi_1(p)$, $\pi_2(p)$ are called the *elimination* forms of these types, respectively. One may wonder why all typing judgments in the above rules have been decorated with annotations of the form $\Gamma \vdash$, for some Γ. In these cases, Γ is the *context* of the corresponding judgment, used to keep track of the types of variables that may appear in that judgment.

Although contexts may seem rather trivial from a syntactic perspective, they are key to understanding the categorical semantics of dependent type theory. In particular, when modelling a dependent type theory as a category, it is the *contexts* which form the objects of this category, with morphisms between contexts being *substitutions* of terms in the domain context for the variables of the codomain context. A type A dependent upon variables in a context Γ is then interpreted as a morphism (i.e. substitution) $\Gamma, x : A \to \Gamma$, whose domain represents the context Γ extended with a variable of type A. We then interpret a term a of type A in context Γ as a *section* of the display map representing A, i.e.

$$\begin{tikzcd} \Gamma \arrow[r, "a"] \arrow[dr, equal] & \Gamma, x : A \arrow[d, "A"] \\ & \Gamma \end{tikzcd}$$

Hence for each context Γ, there is a category $\mathbf{Ty}[\Gamma]$, which is the full subcategory of the slice category $\mathcal{C}/\Gamma$ consisting of all display maps, wherein objects correspond to types in context Γ, and morphisms correspond to terms.

In typical categorical semantics, given a substitution $f : \Gamma \to \Delta$, and a type $A : \Delta, x : A \to \Delta$, we then interpret the action of f on A as a pullback:

$$\begin{tikzcd} \Gamma, x : A[f] \arrow[r] \arrow[d, "A[f]"'] \arrow[dr, phantom, "\lrcorner", very near start] & \Delta, x : A \arrow[d, "A"] \\ \Gamma \arrow[r, "f"'] & \Delta \end{tikzcd}$$

In particular, then, any display map $A : \Gamma, x : A \to \Gamma$ induces a functor $\mathbf{Ty}[\Gamma] \to \mathbf{Ty}[\Gamma, x : A]$ by substitution along A. The left and right adjoints to this functor (if they exist) then correspond to dependent pair and dependent function types, respectively.

So far, we have told a pleasingly straightforward story of how to interpret the syntax of dependent type theory categorically. Unfortunately, this story is a fantasy, and the interpretation of type-theoretic syntax into categorical semantics sketched above is unsound, as it stands. The problem in essentials is that, in the syntax of type theory, substitution is strictly associative – i.e. given substitutions $f : \Gamma \to \Delta$ and $g : \Delta \to \Theta$ and a type $\mathtt{A}$, we have $A[g][f] = A[g[f]]$; however, in the above categorical semantics, such iterated substitution is interpreted via successively taking pullbacks, which is in general only associative up to isomorphism. It seems, then, that something more is needed to account for this kind of *strictness* in the semantics of dependent type theory. [2] It is precisely this problem which natural models exist to solve.

2.2 Natural Models

The key insight of Awodey in formulating the notion of a natural model is that the problem of strictness in the semantics of type theory has, in a sense, already been solved by the notion of *type universes*, such as $\mathtt{Type}$ as introduced above. Given a universe of types $\mathcal{U}$, rather than representing dependent types as display maps, and substitution as pullback, we can simply represent a family of types $B[x]$ dependent upon a type A as a function $A \to \mathcal{U}$, with substitution then given by precomposition, which is automatically strictly associative.

To interpret the syntax of dependent type theory in a category $\mathcal{C}$ of contexts and substitutions, it therefore suffices to *embed* $\mathcal{C}$ into a category whose type-theoretic internal language posesses such a universe whose types correspond to those of $\mathcal{C}$. For this purpose, we work in the category of *prehseaves* $\mathbf{Set}^{\mathcal{C}^{op}}$, with the embedding $\mathcal{C} \hookrightarrow \mathbf{Set}^{\mathcal{C}^{op}}$ being nothing other than the Yoneda embedding.

The universe $\mathcal{U}$ is then given by an object of $\mathbf{Set}^{\mathcal{C}^{op}}$, i.e. an assignment, to each context Γ, of a set $\mathsf{Ty}[\Gamma]$ of types in context Γ, with functions $\mathsf{Ty}[\Delta] \to \mathsf{Ty}[\Gamma]$ for each substitution $f : \Gamma \to \Delta$ that compose associatively, together with a $\mathcal{U}$-indexed family of objects $u \in \mathbf{Set}^{\mathcal{C}^{op}}/\mathcal{U}$, i.e. a natural transformation $u : \mathcal{U}_\bullet \Rightarrow \mathcal{U}$, where for each context Γ and type $A \in \mathsf{Ty}[\Gamma]$, the fibre of u_Γ over A is the set $\mathsf{Tm}[\Gamma, A]$ of inhabitants of A in context Γ.

The condition that all types in $\mathcal{U}$ "belong to $\mathcal{C}$", in an appropriate sense, can then be expressed by requiring u to be *representable*, i.e. for any representable $\gamma \in \mathbf{Set}^{\mathcal{C}^{op}}$ with a natural transformation $\alpha : \gamma \Rightarrow \mathcal{U}$, the pullback

$$\begin{tikzcd}
\gamma \times_{\alpha,u} \mathcal{U}_\bullet \arrow[r, Rightarrow] \arrow[d, Rightarrow, "u[\alpha]"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathcal{U}_\bullet \arrow[d, Rightarrow, "u"] \\
\gamma \arrow[r, Rightarrow, "\alpha"'] & \mathcal{U}
\end{tikzcd}$$

of u along α is representable.

[2] Something something Beck-Chevalley...

The question, then, is how to express that $\mathcal{C}$ has dependent pair types, dependent function types, etc., in terms of the structure of u. A further insight of Awodey, toward answering this question, is that u gives rise to a functor (indeed, a *polynomial functor*) $\overline{u} : \mathbf{Set}^{\mathcal{C}^{op}} \to \mathbf{Set}^{\mathcal{C}^{op}}$, defined as follows

$$\overline{u}(P)(\Gamma) = \sum_{A:\mathsf{Ty}[\Gamma]} P(\Gamma)^{\mathsf{Tm}[\Gamma,A]}$$

and much of the type-theoretic structure of u can be accounted for in terms of this functor. For instance (for reasons to be explained shortly), dependent pair types are given by a natural transformation $\sigma : \overline{u} \circ \overline{u} \Rightarrow \overline{u}$, that is *Cartesian* in that, for every $\alpha : P \Rightarrow Q$, the following naturality square is a pullback

$$\begin{tikzcd}
\overline{u}(\overline{u}(P)) \arrow[r, Rightarrow, "\overline{u}(\overline{u}(\alpha))"] \arrow[d, Rightarrow, "\sigma_P"'] \arrow[dr, phantom, "\lrcorner"] & \overline{u}(\overline{u}(Q)) \arrow[d, Rightarrow, "\sigma_Q"] \\
\overline{u}(P) \arrow[r, Rightarrow, "\overline{u}(\alpha)"'] & \overline{u}(Q)
\end{tikzcd}$$

A question that arises, then, is what structure such a natural transformation interpreting dependent pair types must possess. It is natural to think that σ, along with a suitably-chosen natural transformation $Id \Rightarrow \overline{u}$, ought to give $\overline{u}$ the structure of a monad. However, this turns out to be too strong a requirement, as it amounts to asking that $\Sigma x : A.(\Sigma y : B[x].C[x,y]) = \Sigma(x,y) : (\Sigma x : A.B[x]).C[x,y]$, when in general this identity only holds up to isomorphism. Hence we seem to have crossed over from Scylla of our semantics for dependent type theory not being strict enough to interpret those identities we expect to hold strictly, to the Charybdis of them now being too strict to interpret the identities we expect to hold only up to isomorphism. It was for this reason that Awodey & Newstead were forced to ultimately go beyond Polynomial functors in their accounts of natural models.

However, another possibility exists to solve this dilemma – to use the language of HoTT itself to reason about such equality-up-to-isomorphism in natural models. For this purpose, rather than taking natural models to be certain (representable) morphisms in $\mathbf{Set}^{\mathcal{C}^{op}}$, we can instead expand the mathematical universe in which these models live to $\infty\mathbf{Grpd}^{\mathcal{C}^{op}}$, which, as an ∞-topos, has HoTT as its internal language. Taking advantage of this fact, we can use HoTT itself as a language for studying the semantics of type theory, by postulating an abstract type $\mathcal{U}$ together with a type family $u : \mathcal{U} \to \mathsf{Type}$, corresponding to a representable natural transformation $u : \mathcal{U}_\bullet \Rightarrow \mathcal{U}$ as above.

What remains, then, is to show how the various type-theoretic properties of such natural models can be expressed in terms of polynomial functors in the language of HoTT, and the complex identities to which these give rise. For this purpose, we begin with a recap of the basics of HoTT, before launching into a development of the theory of polynomial functors within HoTT, with an eye toward the latter's use in the study of natural models.

2.3 Homotopy Type Theory

2.3.1 The Identity Type

Given elements `a,b : A` for some type `A`, the identity type `a ≡ b` is inductively generated from the single constructor `refl : {x : A} → x ≡ x`, witnessing reflexivity of equality.

```
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

The core insight of Homotopy Type Theory is that the presence of (intensional) identity types in a system of dependent type theory endows each type with the structure of an ∞-groupoid, and endows each function between types with the structure of a functor between ∞-groupoids, etc. This allows a wealth of higher-categorical properties and structures to be defined and studied *internally* in the language of dependent type theory.

Since an invocation of reflexivity typically occurs at the end of an equality proof, we introduce the notation □ as a shorthand for `refl` as follows:

```
_□ : ∀ {ℓ} {A : Type ℓ} (a : A) → a ≡ a
a □ = refl
```

The inductive generation of `a ≡ b` from `refl` then gives rise to the operation of *transport* that allows an inhabitant of the type `B a` to be converted to an inhabitant of `B b` for any type family `B : (x : A) → Type`.

```
transp : ∀ {ℓ κ} {A : Type ℓ} (B : A → Type κ) {a a' : A}
         → (e : a ≡ a') → B a → B a'
transp B refl b = b
```

Transitivity of equality then follows in the usual way. Note, however, that we take advantage of Agda's support for mixfix notation to present transitivity in such a way as to streamline both the reading and writing of equality proofs:

```
_≡⟨_⟩_ : ∀ {ℓ} {A : Type ℓ} (a : A) {b c : A}
          → a ≡ b → b ≡ c → a ≡ c
a ≡⟨ e ⟩ refl = e
```

Symmetry of equality follows similarly:

```
sym : ∀ {ℓ} {A : Type ℓ} {a a' : A} → a ≡ a' → a' ≡ a
sym refl = refl
```

As mentioned above, each function `f : A → B` in HoTT is canonically endowed with the structure of a functor between ∞-groupoids, where the action of such a function `f` on paths (i.e. elements of the identity type) is as follows:

```
ap : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {a a' : A}
     → (f : A → B) → a ≡ a' → (f a) ≡ (f a')
ap f refl = refl
```

By the same token, given a proof `f ≡ g` for two functions `f,g : (x : A) → B x`, it follows that for any `a : A` we have `f a ≡ g a`.

```
coAp : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f g : A → B}
       → f ≡ g → (x : A) → f x ≡ g x
coAp refl x = refl
```

Moreover, to show that two pairs `(a , b)` and `(a' , b')` are equal, it suffices to show that there is an identification `e : a ≡ a'` together with `e' : transp B e b ≡ b'`.

```
pairEq : ∀ {ℓ κ} {A : Type ℓ} {B : A → Type κ}
         → {a a' : A} {b : B a} {b' : B a'}
         → (e : a ≡ a') (e' : transp B e b ≡ b')
         → (a , b) ≡ (a' , b')
pairEq refl refl = refl
```

2.3.2 Truncation, Bracket Types & Factorization

We say that a type `A` is:

1. *contractible* (aka (-2)-truncated) if `A` is uniquely inhabited
2. a (mere) *proposition* (aka (-1)-truncated) if any two elements of `A` are identical
3. a *set* (aka 0-truncated) if for any `a,b : A`, the type `a ≡ b` is a proposition.

```
isContr : ∀ {ℓ} → Type ℓ → Type ℓ
isContr A = Σ A (λ a → (b : A) → a ≡ b)

isProp : ∀ {ℓ} → Type ℓ → Type ℓ
isProp A = {a b : A} → a ≡ b

isSet : ∀ {ℓ} → Type ℓ → Type ℓ
isSet A = (a b : A) → isProp (a ≡ b)
```

We additionally postulate the existence of a *propositional truncation*, or *bracket type* operation, that takes a type `A` to the least proposition (wrt entailment) entailed by inhabitation of `A`.

```
postulate
    ‖_‖ : ∀ {ℓ} (A : Type ℓ) → Type lzero
    in‖-‖ : ∀ {ℓ} {A : Type ℓ} → A → ‖ A ‖
    ‖-‖IsProp : ∀ {ℓ} {A : Type ℓ} → isProp (‖ A ‖)
    ‖-‖Rec : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ}
             → isProp B → (A → B) → ‖ A ‖ → B
```

When the type ‖ A ‖ is inhabited, we say that A is *merely* inhabited.

From this operation on types, we straightforwardly obtain the higher analogue of the usual epi-mono factorization system on functions between sets, as follows:

Given a function f : A → B and an element b : B, the *fibre* of f at b is the type of elements of a equipped with a proof of f a ≡ b:

```
Fibre : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → B → Type (ℓ ⊔ κ)
Fibre {A = A} f b = Σ A (λ a → f a ≡ b)
```

By the same token, given such an f, its *image* is the type of elements of B whose fibres are merely inhabited.

```
Im : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → Type κ
Im {B = B} f = Σ B (λ b → ‖ Fibre f b ‖)
```

We say that f is *(-1)-truncated* (i.e. a monomorphism), if for each b : B, the fibre of f at b is a proposition.

```
isMono : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → Type (ℓ ⊔ κ)
isMono {B = B} f = (b : B) → isProp (Fibre f b)
```

Likewise, we say that f is *(-1)-connected* (i.e. an epimorphism), if all of its fibres are merely inhabited.

```
isEpi : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → Type κ
isEpi {B = B} f = (b : B) → ‖ Fibre f b ‖
```

Every function f can then be factored into a (-1)-connected map onto its image followed by a (-1)-truncated map onto its codomain:

```
factor1 : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (f : A → B) → A → Im f
factor1 f a = (f a) , in∣-∣ (a , refl)

factor2 : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (f : A → B) → Im f → B
factor2 f (b , _) = b

factor≡ : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ}
          → (f : A → B) (a : A) → factor2 f (factor1 f a) ≡ f a
factor≡ f a = refl

factor1IsEpi : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ}
               → (f : A → B) → isEpi (factor1 f)
factor1IsEpi f (b , x) =
    ∣-∣Rec ∣-∣IsProp
          (λ {(a , refl) → in∣-∣ (a , pairEq refl ∣-∣IsProp)})
          x
```

```
factor2IsMono : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ}
                → (f : A → B) → isMono (factor2 f)
factor2IsMono f b {a = ((b' , x) , refl)} {b = ((b'' , x') , refl)} =
    pairEq (pairEq refl ∥-∥IsProp) (lemma ∥-∥IsProp)
    where lemma : (e : x ≡ x') → transp (λ a → factor2 f a ≡ b)
                                        (pairEq refl e) refl ≡ refl
          lemma refl = refl
```

2.3.3 Equivalences

A pivotal notion, both for HoTT in general, and for the content of this paper, is that of a function `f : A → B` being an *equivalence* of types. The reader familiar with HoTT will know that there are several definitions – all equivalent – of this concept appearing in the HoTT literature. For present purposes, we make use of the *bi-invertible maps* notion of equivalence. Hence we say that a function `f : A → B` is an equivalence if it has both a left inverse and a right inverse:

```
isEquiv : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → Type (ℓ ⊔ κ)
isEquiv {A = A} {B = B} f =
      (Σ (B → A) (λ g → (a : A) → g (f a) ≡ a))
    × (Σ (B → A) (λ h → (b : B) → f (h b) ≡ b))
```

A closely-related notion is that of a function `f` being an *isomorphism*, i.e. having a single two-sided inverse:

```
Iso : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} → (A → B) → Type (ℓ ⊔ κ)
Iso {A = A} {B = B} f =
    (Σ (B → A) (λ g → ((a : A) → g (f a) ≡ a)
                    × ((b : B) → f (g b) ≡ b)))
```

One might be inclined to wonder, then, why we bother to define equivalence via the seemingly more complicated notion of having both a left and a right inverse when the familiar notion of isomorphism can just as well be defined, as above. The full reasons for this are beyond the scope of this paper, though see [**hottbook**] for further discussion. Suffice it to say that, for subtle reasons due to the higher-categorical structure of types in HoTT, the plain notion of isomorphism given above is not a *good* notion of equivalence, whereas that of bi-invertible maps is. In particular, the type `Iso f` is not necessarily a proposition for arbitrary `f`, whereas `isEquiv f` is.

We may nonetheless move more-or-less freely back and forth between the notions of equivalence and isomorphism given above, thanks to the following functions, which allow us to convert isomorphisms to equivalences and vice versa:

```
Iso→isEquiv : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B}
              → Iso f → isEquiv f
Iso→isEquiv (g , l , r) = ((g , l) , (g , r))
```

```
isEquiv→Iso : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B}
              → isEquiv f → Iso f
isEquiv→Iso {f = f} ((g , l) , (h , r)) =
    h , (λ x → (h (f x))        ≡⟨ sym (l (h (f x))) ⟩
               (g (f (h (f x))) ≡⟨ ap g (r (f x)) ⟩
               ((g (f x))       ≡⟨ l x ⟩
               (x □)))) , r
```

Straightforwardly, the identity function at each type is an equivalence, and equivalences are closed under composition:

```
idIsEquiv : ∀ {ℓ} {A : Type ℓ} → isEquiv {A = A} (λ x → x)
idIsEquiv = ((λ x → x) , (λ x → refl)) , ((λ x → x) , (λ x → refl))

compIsEquiv : ∀ {ℓ ℓ' ℓ''} {A : Type ℓ} {B : Type ℓ'} {C : Type ℓ''}
              → (g : B → C) (f : A → B) → isEquiv g → isEquiv f
              → isEquiv (λ a → g (f a))
compIsEquiv g f ((g' , lg) , (g'' , rg)) ((f' , lf) , (f'' , rf)) =
      ( (λ c → f' (g' c))
      , λ a → (f' (g' (g (f a))))   ≡⟨ ap f' (lg (f a)) ⟩
              (f' (f a)             ≡⟨ lf a ⟩
              (a                    □)))
    , ((λ c → f'' (g'' c))
      , λ c → (g (f (f'' (g'' c)))) ≡⟨ ap g  (rf (g'' c)) ⟩
              (g (g'' c)            ≡⟨ rg c ⟩
              (c                    □)))
```

And by the above translation between equivalences and isomorphisms, each equivalence has a corresponding inverse map in the opposite direction, which is itself an equivalence:

```
inv : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B} → isEquiv f → B → A
inv (_ , (h , _)) = h

isoInv : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B}
         → (isof : Iso f) → Iso (fst isof)
isoInv {f = f} (g , l , r) = (f , r , l)

invIsEquiv : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B}
             → (ef : isEquiv f) → isEquiv (inv ef)
invIsEquiv ef = Iso→isEquiv (isoInv (isEquiv→Iso ef))
```

We close this section by noting that, for each type family `B : A → Type`, the map `B a → B a'` induced by transport along `e : a ≡ a'` for any `a, a' : A` is an equivalence with inverse given by transport along `sym e`, as follows:

```
syml : ∀ {ℓ κ} {A : Type ℓ} {B : A → Type κ} {a b : A}
       → (e : a ≡ b) (x : B a) → transp B (sym e) (transp B e x) ≡ x
syml refl x = refl

symr : ∀ {ℓ κ} {A : Type ℓ} {B : A → Type κ} {a b : A}
       → (e : a ≡ b) (y : B b) → transp B e (transp B (sym e) y) ≡ y
symr refl x = refl
```

And...

```
transpAp : ∀ {ℓ ℓ' κ} {A : Type ℓ} {A' : Type ℓ'} {a b : A}
           → (B : A' → Type κ) (f : A → A') (e : a ≡ b) (x : B (f a))
           → transp (λ x → B (f x)) e x ≡ transp B (ap f e) x
transpAp B f refl x = refl

≡siml : ∀ {ℓ} {A : Type ℓ} {a b : A}
        → (e : a ≡ b) → refl ≡ (b ≡⟨ sym e ⟩ e)
≡siml refl = refl

≡idr : ∀ {ℓ} {A : Type ℓ} {a b : A}
       → (e : a ≡ b) → e ≡ (a ≡⟨ refl ⟩ e)
≡idr refl = refl

conj : ∀ {ℓ} {A : Type ℓ} {a b c d : A}
       → (e1 : a ≡ b) (e2 : a ≡ c) (e3 : b ≡ d) (e4 : c ≡ d)
       → (a ≡⟨ e1 ⟩ e3) ≡ (a ≡⟨ e2 ⟩ e4)
       → e3 ≡ (b ≡⟨ sym e1 ⟩(a ≡⟨ e2 ⟩ e4))
conj e1 e2 refl refl refl = ≡siml e1

nat : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f g : A → B} {a b : A}
      → (α : (x : A) → f x ≡ g x) (e : a ≡ b)
      → ((f a) ≡⟨ α a ⟩ (ap g e)) ≡ ((f a) ≡⟨ ap f e ⟩ (α b))
nat {a = a} α refl = ≡idr (α a)

cancel : ∀ {ℓ} {A : Type ℓ} {a b c : A}
         → (e1 e2 : a ≡ b) (e3 : b ≡ c)
         → (a ≡⟨ e1 ⟩ e3) ≡ (a ≡⟨ e2 ⟩ e3)
         → e1 ≡ e2
cancel e1 e2 refl refl = refl

apId : ∀ {ℓ} {A : Type ℓ} {a b : A}
       → (e : a ≡ b) → ap (λ x → x) e ≡ e
apId refl = refl

apComp : ∀ {ℓ ℓ' ℓ''} {A : Type ℓ} {B : Type ℓ'} {C : Type ℓ''} {a b : A}
```

```
        → (f : A → B) (g : B → C) (e : a ≡ b)
        → ap (λ x → g (f x)) e ≡ ap g (ap f e)
apComp f g refl = refl

apHtpy : ∀ {ℓ} {A : Type ℓ} {a : A}
        → (i : A → A) (α : (x : A) → i x ≡ x)
        → ap i (α a) ≡ α (i a)
apHtpy {a = a} i α =
    cancel (ap i (α a)) (α (i a)) (α a)
           ((i (i a) ≡⟨ ap i (α a) ⟩ α a)
           ≡⟨ sym (nat α (α a)) ⟩
           ((i (i a) ≡⟨ α (i a) ⟩ ap (λ z → z) (α a))
           ≡⟨ ap (λ e → i (i a) ≡⟨ α (i a) ⟩ e) (apId (α a)) ⟩
           ((i (i a) ≡⟨ α (i a) ⟩ α a) □)))

HAdj : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ}
       → (A → B) → Set (ℓ ⊔ κ)
HAdj {A = A} {B = B} f =
    Σ (B → A) (λ g →
      Σ ((x : A) → g (f x) ≡ x) (λ η →
        Σ ((y : B) → f (g y) ≡ y) (λ ϵ →
          (x : A) → ap f (η x) ≡ ϵ (f x))))

Iso→HAdj : ∀ {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B}
           → Iso f → HAdj f
Iso→HAdj {f = f} (g , η , ϵ) =
    g , (η
    , ( (λ y →
           f (g y)          ≡⟨ sym (ϵ (f (g y))) ⟩
          (f (g (f (g y))) ≡⟨ ap f (η (g y)) ⟩
          (f (g y)          ≡⟨ ϵ y ⟩
          (y                □))))
      , λ x → conj (ϵ (f (g (f x)))) (ap f (η (g (f x))))
                   (ap f (η x)) (ϵ (f x))
                   (((f (g (f (g (f x)))) ≡⟨ ϵ (f (g (f x))) ⟩ ap f (η x)))
                    ≡⟨ nat (λ z → ϵ (f z)) (η x) ⟩
                    (((f (g (f (g (f x)))) ≡⟨ ap (λ z → f (g (f z))) (η x) ⟩ ϵ (f x)))
                    ≡⟨ ap (λ e → (f (g (f (g (f x)))) ≡⟨ e ⟩ ϵ (f x)))
                          ((ap (λ z → f (g (f z))) (η x))
                           ≡⟨ apComp (λ z → g (f z)) f (η x) ⟩
                           ((ap f (ap (λ z → g (f z)) (η x)))
                           ≡⟨ ap (ap f) (apHtpy (λ z → g (f z)) η) ⟩
                           (ap f (η (g (f x))) □))) ⟩
                    (((f (g (f (g (f x)))) ≡⟨ ap f (η (g (f x))) ⟩ ϵ (f x))) □)))))
```

```
pairEquiv1 : ∀ {ℓ ℓ' κ} {A : Type ℓ} {A' : Type ℓ'} {B : A' → Type κ}
             → (f : A → A') → isEquiv f
             → isEquiv {A = Σ A (λ x → B (f x))} {B = Σ A' B}
                       (λ (x , y) → (f x , y))
pairEquiv1 {A = A} {A' = A'} {B = B} f ef =
  Iso→isEquiv
    ( (λ (x , y) → (g x , transp B (sym (ϵ x)) y))
    , ( (λ (x , y) → pairEq (η x) (lemma x y))
      , λ (x , y) → pairEq (ϵ x) (symr (ϵ x) y) ) )
  where
    g : A' → A
    g = fst (Iso→HAdj (isEquiv→Iso ef))
    η : (x : A) → g (f x) ≡ x
    η = fst (snd (Iso→HAdj (isEquiv→Iso ef)))
    ϵ : (y : A') → f (g y) ≡ y
    ϵ = fst (snd (snd (Iso→HAdj (isEquiv→Iso ef))))
    ρ : (x : A) → ap f (η x) ≡ ϵ (f x)
    ρ = snd (snd (snd (Iso→HAdj (isEquiv→Iso ef))))
    lemma : (x : A) (y : B (f x))
            → transp (λ z → B (f z)) (η x)
                     (transp B (sym (ϵ (f x))) y)
              ≡ y
    lemma x y = (transp (λ z → B (f z)) (η x)
                        (transp B (sym (ϵ (f x))) y))
                ≡⟨ transpAp B f (η x)
                             (transp B (sym (ϵ (f x))) y) ⟩
                ( transp B (ap f (η x))
                           (transp B (sym (ϵ (f x))) y)
                ≡⟨ ap (λ e → transp B e
                                 (transp B (sym (ϵ (f x))) y))
                      (ρ x) ⟩
                ( (transp B (ϵ (f x))
                          (transp B (sym (ϵ (f x))) y))
                ≡⟨ (symr (ϵ (f x)) y) ⟩
                (y □)))

pairEquiv2 : ∀ {ℓ κ κ'} {A : Type ℓ} {B : A → Type κ} {B' : A → Type κ'}
             → (g : (x : A) → B x → B' x) → ((x : A) → isEquiv (g x))
             → isEquiv {A = Σ A B} {B = Σ A B'}
                       (λ (x , y) → (x , g x y))
pairEquiv2 g eg =
    let isog = (λ x → isEquiv→Iso (eg x))
    in Iso→isEquiv ( (λ (x , y) → (x , fst (isog x) y))
```

```
                    , ( (λ (x , y) →
                           pairEq refl (fst (snd (isog x)) y))
                      , λ (x , y) →
                          pairEq refl (snd (snd (isog x)) y)))

pairEquiv : ∀ {ℓ ℓ' κ κ'} {A : Type ℓ} {A' : Type ℓ'}
            → {B : A → Type κ} {B' : A' → Type κ'}
            → (f : A → A') (g : (x : A) → B x → B' (f x))
            → isEquiv f → ((x : A) → isEquiv (g x))
            → isEquiv {A = Σ A B} {B = Σ A' B'}
                      (λ (x , y) → (f x , g x y))
pairEquiv f g ef eg =
    compIsEquiv (λ (x , y) → (f x , y))
                (λ (x , y) → (x , g x y))
                (pairEquiv1 f ef)
                (pairEquiv2 g eg)
```

3 Polynomials in HoTT

3.1 Basics

Remark: for the sake of concision, since essentially all of the categorical structures treated in this paper will be infinite-dimensional, we shall generally omit the prefix "∞-" from our descriptions these structures. Hence hereafter "category" generally means ∞-category, "functor" means ∞-functor, etc.

Let **Type** be the category of types and functions between them. Given a type `A`, let y^A denote the corresponding representable functor **Type** → **Type**.

A *polynomial functor* is a coproduct of representable functors **Type** → **Type**, i.e. an endofunctor on **Type** of the form

$$\sum_{a:A} y^{B(a)}$$

for some type `A` and family of types `B : A → Type`. The data of a polynomial functor is thus uniquely determined by the choice of `A` and `B`. Hence we may represent such functors in Agda simply as pairs (A , B) of this form:

```
Poly : (ℓ κ : Level) → Type ((lsuc ℓ) ⊔ (lsuc κ))
Poly ℓ κ = Σ (Set ℓ) (λ A → A → Set κ)
```

A basic example of such a polynomial functor is the identity functor y consisting of a single term of unit arity – hence represented by the pair `(⊤ , λ _ → ⊤)`.

```
y : Poly lzero lzero
y = (⊤ , λ _ → ⊤)
```

The observant reader may note the striking similarity of the above-given formula for a polynomial functor and the endofunctor on $\mathbf{Set}^{\mathcal{C}^{op}}$ defined in the previous section on natural models. Indeed, this is no accident, given a type U and a function `u : U → Type` corresponding to a natural model as described previously, we obtain the corresponding polynomial u `: Poly` as the pair `(U , u)`. Hence we can study the structure of U and `u` in terms of u, and this, as we shall see, allows for some significant simplifications in the theory of natural models.

Given polynomial functors $p = \sum_{a:A} y^{B(a)}$ and $q = \sum_{c:C} y^{D(c)}$, a natural transformation $p \Rightarrow q$ is equivalently given by the data of a *forward* map `f : A → B` and a *backward* map `g : (a : A) → D (f a) → B a`, as can be seen from the following argument via Yoneda:

$$\begin{aligned} & \int_{y \in \mathbf{Type}} \left(\sum_{a:A} y^{B(a)}\right) \to \sum_{c:C} y^{D(c)} \\ \simeq\ & \prod_{a:A} \int_{y \in \mathbf{Type}} y^{B(a)} \to \sum_{c:C} y^{D(c)} \\ \simeq\ & \prod_{a:A} \sum_{c:C} B(a)^{D(c)} \\ \simeq\ & \sum_{f:A \to C} \prod_{a:A} B(a)^{D(f(c))} \end{aligned}$$

We use the notation $p \leftrightarrows q$ to denote the type of natural transformations from p to q (aka *lenses* from p to q), which may be written in Agda as follows:

```
_⇆_ : ∀ {ℓ ℓ' κ κ'} → Poly ℓ κ → Poly ℓ' κ' → Type (ℓ ⊔ ℓ' ⊔ κ ⊔ κ')
(A , B) ⇆ (C , D) = Σ (A → C) (λ f → (a : A) → D (f a) → B a)
```

By application of function extensionality, we derive the following type for proofs of equality between lenses:

```
EqLens : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
         → (p ⇆ q) → (p ⇆ q) → Set (ℓ ⊔ ℓ' ⊔ κ ⊔ κ')
EqLens (A , B) (C , D) (f , g) (f' , g') =
  (a : A) → Σ (f a ≡ f' a)
              (λ e → (b : D (f a)) → g a b ≡ g' a (transp D e b))
```

For each polynomial p, the correspnding *identity* lens is given by the following data:

```
id : ∀ {ℓ κ} (p : Poly ℓ κ) → p ⇆ p
id p = ( (λ a → a) , λ a b → b )
```

And given lenses $p \leftrightarrows q$ and $q \leftrightarrows r$, their composition may be computed as follows:

```
comp : ∀ {ℓ ℓ' ℓ'' κ κ' κ''}
       → (p : Poly ℓ κ) (q : Poly ℓ' κ') (r : Poly ℓ'' κ'')
       → p ⇆ q → q ⇆ r → p ⇆ r
comp p q r (f , g) (h , k) =
    ( (λ a → h (f a)) , λ a z → g a (k (f a) z) )
```

Hence we have a category **Poly** of polynomial functors and lenses between them. Our goal, then, is to show how the type-theoretic structure of a natural model naturally arises from the structure of this category. In fact, **Poly** is replete with categorical structures of all kinds, of which we now mention but a few:

3.2 The Vertical-Cartesian Factorization System on Poly

We say that a lens `(f , f♯) : (A , B) ⇆ (C , D)` is *vertical* if `f : A → C` is an equivalence, and Cartesian if for every `a : A`, the map `f♯ a : D[f a] → B a` is an equivalence.

```
isVertical : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
           → p ⇆ q → Set (ℓ ⊔ ℓ')
isVertical p q (f , f♯) = isEquiv f

isCartesian : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
            → p ⇆ q → Set (ℓ ⊔ κ ⊔ κ')
isCartesian (A , B) q (f , f♯) = (a : A) → isEquiv (f♯ a)
```

Every lens `(A , B) ⇆ (C , D)` can then be factored as a vertical lens followed by a Cartesian lens:

```
vertfactor : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
           → (f : p ⇆ q) → p ⇆ (fst p , λ x → snd q (fst f x))
vertfactor p q (f , f♯) = (λ x → x) , (λ a x → f♯ a x)

vertfactorIsVert : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ)
                 → (q : Poly ℓ' κ') (f : p ⇆ q)
                 → isVertical p (fst p , λ x → snd q (fst f x))
                              (vertfactor p q f)
vertfactorIsVert p q f = idIsEquiv

cartfactor : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
           → (f : p ⇆ q) → (fst p , λ x → snd q (fst f x)) ⇆ q
cartfactor p q (f , f♯) = f , λ a x → x

cartfactorIsCart : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ)
                 → (q : Poly ℓ' κ') (f : p ⇆ q)
                 → isCartesian (fst p , λ x → snd q (fst f x)) q
                               (cartfactor p q f)
cartfactorIsCart p q f x = idIsEquiv

vertcartfactor≡ : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ)
                → (q : Poly ℓ' κ') (f : p ⇆ q)
                → EqLens p q f
                         (comp p (fst p , λ x → snd q (fst f x)) q
                               (vertfactor p q f)
                               (cartfactor p q f))
vertcartfactor≡ p q f a = refl , (λ b → refl)
```

Of these two classes of morphisms in **Poly**, it is *Cartesian* lenses that shall be of principal interest to us. If we view a polynomial `p = (A , B)` as an `A`-indexed family of types, given by `B`, then given a lens `(f , f♯) : p ⇆ u`, we can think of the map `f♯ a : u (f a) → B a`, as an *elimination form* for the type `u (f a)`, i.e. a way of *using* elements of the type `u (f a)`. If we then ask that `(f , f♯)` isCartesian, this implies that the type `u (f a)` is completely characterized (up to equivalence) by this elimination form, and in this sense, *u contains* the type `B a`, for all `a : A`.[3]

We can therefore use Cartesian lenses to detect which types are contained in a natural model *u*.

A further fact about Cartesian lenses is that they are closed under identity and composition, as a direct consequence of the closure of equivalences under identity and composition:

```
idCart : ∀ {ℓ κ} (p : Poly ℓ κ)
         → isCartesian p p (id p)
idCart p a = idIsEquiv

compCartesian : ∀ {ℓ ℓ' ℓ'' κ κ' κ''}
                → (p : Poly ℓ κ) (q : Poly ℓ' κ') (r : Poly ℓ'' κ'')
                → (f : p ⇆ q) (g : q ⇆ r)
                → isCartesian p q f → isCartesian q r g
                → isCartesian p r (comp p q r f g)
compCartesian p q r f g cf cg a =
    compIsEquiv (snd f a) (snd g (fst f a)) (cf a) (cg (fst f a))
```

Hence there is a category $\mathbf{Poly}^{\mathbf{Cart}}$ defined as the wide subcategory of **Poly** whose morphisms are precisely Cartesian lenses. As we shall see, much of the categorical structure of natural models qua polynomial functors can be derived from the subtle interplay between $\mathbf{Poly}^{\mathbf{Cart}}$ and **Poly**.

3.2.1 Epi-Mono Factorization on $\mathbf{Poly}^{\mathbf{Cart}}$

In fact, $\mathbf{Poly}^{\mathbf{Cart}}$ itself inherits a factorization system from the epi-mono factorization on types considered previously.

Define a Cartesian lens `(f , f♯) : p ⇆ q` to be a *Cartesian embedding* if `f` is a monomorphism, and a *Cartesian surjection* if `f` is an epimorphism.

```
isCartesianEmbedding : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                       → (f : p ⇆ q) → isCartesian p q f → Set (ℓ ⊔ ℓ')
isCartesianEmbedding p q (f , f♯) cf = isMono f
```

[3]Those familiar with type theory may recognize this practice of defining types in terms of their elimination forms as characteristic of so-called *negative* types (in opposition to *positive* types, which are characterized by their introduction forms). Indeed, there are good reasons for this, having to do with the fact that negative types are equivalently those whose universal property is given by a *representable* functor, rather than a *co-representable* functor, which reflects the fact that natural models are defined in terms of *presheaves* on a category of contexts, rather than *co-presheaves*.

```
isCartesianSurjection : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                      → (f : p ⇆ q) → isCartesian p q f → Set ℓ'
isCartesianSurjection p q (f , f♯) cf = isEpi f
```

Then every Cartesian lens can be factored into a Cartesian surjection followed by a Cartesian embedding.

```
factorcart1 : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
            → (f : p ⇆ q) → isCartesian p q f
            → p ⇆ (Im (fst f) , λ (x , _) → snd q x)
factorcart1 p q (f , f♯) cf =
    (factor1 f) , f♯

factorcart1IsCart : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                  → (f : p ⇆ q) (cf : isCartesian p q f)
                  → isCartesian p
                                (Im (fst f) , λ (x , _) → snd q x)
                                (factorcart1 p q f cf)
factorcart1IsCart p q (f , f♯) cf = cf

factorcart1IsEpi : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                 → (f : p ⇆ q) (cf : isCartesian p q f)
                 → isCartesianSurjection p
                     (Im (fst f) , λ (x , _) → snd q x)
                     (factorcart1 p q f cf)
                     (factorcart1IsCart p q f cf)
factorcart1IsEpi p q (f , f♯) cf = factor1IsEpi f

factorcart2 : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
            → (f : p ⇆ q) → isCartesian p q f
            → (Im (fst f) , λ (x , _) → snd q x) ⇆ q
factorcart2 p q (f , f♯) cf = (factor2 f) , λ (x , _) y → y

factorcart2IsCart : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                  → (f : p ⇆ q) (cf : isCartesian p q f)
                  → isCartesian (Im (fst f) , λ (x , _) → snd q x) q
                                (factorcart2 p q f cf)
factorcart2IsCart p q (f , f♯) cf x = idIsEquiv

factorcart2IsMono : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
                  → (f : p ⇆ q) (cf : isCartesian p q f)
                  → isCartesianEmbedding
                      (Im (fst f) , λ (x , _) → snd q x) q
                      (factorcart2 p q f cf)
```

```
                        (factorcart2IsCart p q f cf)
factorcart2IsMono p q (f , f♯) cf = factor2IsMono f

factorcart≡ : ∀ {ℓ ℓ' κ κ'} (p : Poly ℓ κ) (q : Poly ℓ' κ')
              → (f : p ⇆ q) (cf : isCartesian p q f)
              → EqLens p q f
                       (comp p (Im (fst f) , λ (x , _) → snd q x) q
                             (factorcart1 p q f cf)
                             (factorcart2 p q f cf))
factorcart≡ p q f cf x = refl , λ y → refl
```

3.3 Composition of Polynomial Functors

As endofunctors on **Type**, polynomial functors may straightforwardly be composed. To show that the resulting composite is itself (equivalent to) a polynomial functor, we can reason via the following chain of equivalences: given polynomials `(A , B)` and `(C , D)`, their composite, evaluated at a type y is

$$\begin{array}{rl} & \sum_{a:A} \prod_{b:B(a)} \sum_{c:C} y^{D(c)} \\ \simeq & \sum_{a:A} \sum_{f:B(a)\to C} \prod_{b:B(a)} y^{D(f(b))} \\ \simeq & \sum_{(a,f):\sum_{a:A} C^{B(a)}} y^{\sum_{b:B(a)} D(f(b))} \end{array}$$

This then defines a monoidal product ◃ on **Poly** with monoidal unit given by the identity functor y:

```
_◃_ : ∀ {ℓ ℓ' κ κ'} → Poly ℓ κ → Poly ℓ' κ' → Poly (ℓ ⊔ κ ⊔ ℓ') (κ ⊔ κ')
(A , B) ◃ (C , D) = (Σ A (λ a → B a → C) , λ (a , f) → Σ (B a) (λ b → D (f b)))

◃Lens : ∀ {ℓ ℓ' ℓ'' ℓ''' κ κ' κ'' κ'''}
        → (p : Poly ℓ κ) (p' : Poly ℓ' κ')
        → (q : Poly ℓ'' κ'') (q' : Poly ℓ''' κ''')
        → p ⇆ p' → q ⇆ q' → (p ◃ q) ⇆ (p' ◃ q')
◃Lens p p' q q' (f , g) (h , k) =
    ((λ (a , c) → (f a , λ b' → h (c (g a b')))))
    , λ (a , c) (b' , d') → ((g a b') , k (c (g a b')) d'))
```

where `◃Lens` is the action of ◃ on lenses.

By construction, the existence of a Cartesian lens `(σ , σ♯) : u ◃ u ⇆ u` effectively shows that u is closed under Σ-types, since:

- σ maps a pair (A , B) consisting of `A : U` and `B : (u A) → U` to a term `σ(A,B)` representing the Σ type. This corresponds to the type formation rule

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B[x]\ \mathsf{Type}}{\Gamma \vdash \Sigma x : A.B[x]\ \mathsf{Type}}$$

- For all `(A , B)` as above, $\sigma\sharp$ `(A , B)` takes a term of type σ `(A , B)` and yields a term `fst (`$\sigma\sharp$ `(A , B)) : A` along with a term `snd (`$\sigma\sharp$ `(A , B)) : B (fst (`$\sigma\sharp$ `(A , B)))`, corresponding to the elimination rules

$$\frac{\Gamma \vdash p : \Sigma x : A.B[x]}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : \Sigma x : A.B[x]}{\Gamma \vdash \pi_2(p) : B[\pi_1(p)]}$$

- The fact that $\sigma\sharp$ `(A , B)` has is an equivalence implies it has an inverse $\sigma\sharp^{-1}$ `(A , B) : Σ (u A) (λ x → u (B x)) → u (`σ `(A , B))`, which takes a pair of terms to a term of the corresponding pair type, and thus corresponds to the introduction rule

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : \Sigma x : A.B[x]}$$

- The fact that $\sigma\sharp^{-1}(A, B)$ is both a left and a right inverse to $\sigma\sharp$ then implies the usual β and η laws for dependent pair types

$$\pi_1(a, b) = a \quad \pi_2(a, b) = b \quad p = (\pi_1(p), \pi_2(p))$$

Similarly, the existence of a Cartesian lens $(\eta, \eta\sharp) : y \leftrightarrows u$ implies that u contains (a type equivalent to) the unit type, in that:

- There is an element η `tt : U` which represents the unit type. This corresponds to the type formation rule

$$\frac{}{\Gamma \vdash \top : \mathsf{Type}}$$

- The "elimination rule" $\eta\sharp$ `tt : u(`η `tt) → ⊤`, applied to an element `x : u(`η `tt)` is trivial, in that it simply discards `x`. This corresponds to the fact that, in the ordinary type-theoretic presentation, $\top$ does not have an elimination rule.
- However, since this trivial elimination rule has an inverse $\eta\sharp^{-1}$ `tt : ⊤ → u (`η `tt)`, it follows that there is a (unique) element $\eta\sharp^{-1}$ `tt tt : u (`η `tt)`, which corresponds to the introduction rule for $\top$:

$$\frac{}{\Gamma \vdash \mathsf{tt} : \top}$$

- Moreover, the uniqueness of this element corresponds to the η-law for $\top$:

$$\frac{\Gamma \vdash x : \top}{\Gamma \vdash x = \mathsf{tt}}$$

But then, what sorts of laws can we expect Cartesian lenses as above to obey, and is the existence of such a lens all that is needed to ensure that the natural model u has dependent pair types in the original sense of Awodey & Newstead's definition in terms of Cartesian (pseudo)monads, or is some further data required? And what about Π types, or other type formers? To answer these questions, we will need to study the structure of $\triangleleft$, along with some closely related functors, in a bit more detail. In particular, we shall see that the structure of $\triangleleft$ as a monoidal product on **Poly** reflects many of the basic identities one expects to hold of Σ types.

For instance, the associativity of $\triangleleft$ corresponds to the associativity of Σ-types,

```
◁assoc : ∀ {ℓ ℓ' ℓ'' κ κ' κ''}
         → (p : Poly ℓ κ) (q : Poly ℓ' κ') (r : Poly ℓ'' κ'')
         → ((p ◁ q) ◁ r) ⇆ (p ◁ (q ◁ r))
◁assoc p q r =
    ((λ ((a , f) , g) → (a , (λ b → (f b , λ d → g (b , d)))))
    , λ ((a , f) , g) (b , (d , x)) → ((b , d) , x))

◁assocCart : ∀ {ℓ ℓ' ℓ'' κ κ' κ''}
             → (p : Poly ℓ κ) (q : Poly ℓ' κ') (r : Poly ℓ'' κ'')
             → isCartesian ((p ◁ q) ◁ r) (p ◁ (q ◁ r)) (◁assoc p q r)
◁assocCart p q r (a , f) =
    Iso→isEquiv ( (λ ((b , d) , x) → b , d , x)
                , ( (λ (b , d , x) → refl)
                  , λ ((b , d) , x) → refl))
```

while the left and right unitors of ◁ correspond to the fact that ⊤ is both a left and a right unit for Σ-types.

```
◁unitl : ∀ {ℓ κ} (p : Poly ℓ κ) → (y ◁ p) ⇆ p
◁unitl p = (λ (tt , a) → a tt) , λ (tt , a) x → tt , x

◁unitlCart : ∀ {ℓ κ} (p : Poly ℓ κ)
             → isCartesian (y ◁ p) p (◁unitl p)
◁unitlCart p (tt , a) =
    Iso→isEquiv ( (λ (tt , b) → b)
                , (λ b' → refl)
                , (λ b' → refl) )

◁unitr : ∀ {ℓ κ} (p : Poly ℓ κ) → (p ◁ y) ⇆ p
◁unitr p = (λ (a , f) → a) , λ (a , f) b → b , tt

◁unitrCart : ∀ {ℓ κ} (p : Poly ℓ κ)
             → isCartesian (p ◁ y) p (◁unitr p)
◁unitrCart p (a , f) =
    Iso→isEquiv ( (λ (b , tt) → b)
                , (λ b → refl)
                , (λ (b , tt) → refl) )
```

In fact, ◁ restricts to a monoidal product on $\mathbf{Poly}^{\mathbf{Cart}}$, since the functorial action of ◁ on lenses preserves Cartesian lenses:

```
◁LensCart : ∀ {ℓ ℓ' ℓ'' ℓ''' κ κ' κ'' κ'''}
            → (p : Poly ℓ κ) (q : Poly ℓ' κ')
            → (r : Poly ℓ'' κ'') (s : Poly ℓ''' κ''')
            → (f : p ⇆ q) (g : r ⇆ s)
```

```
            → isCartesian p q f → isCartesian r s g
            → isCartesian (p ◃ r) (q ◃ s)
                          (◃Lens p q r s f g)
◃LensCart p q r s (f , f♯) (g , g♯) cf cg (a , h) =
    pairEquiv (f♯ a) (λ x → g♯ (h (f♯ a x)))
              (cf a) (λ x → cg (h (f♯ a x)))
```

We should expect, then, for these equivalences to be somehow reflected in the structure of a Cartesian lenses $\eta : y \leftrightarrows u$ and $\mu : u \triangleleft u \leftrightarrows u$. This would be the case, e.g., if the following diagrams in $\mathbf{Poly}^{\mathbf{Cart}}$ were to commute

$$\begin{tikzcd} y \triangleleft \mathfrak{u} \arrow[r, "\eta \triangleleft \mathfrak{u}"] \arrow[dr, "\triangleleft\mathsf{unitl}"'] & \mathfrak{u} \triangleleft \mathfrak{u} \arrow[d, "\mu"] & \mathfrak{u} \triangleleft y \arrow[l, "\mathfrak{u} \triangleleft \eta"'] \arrow[dl, "\triangleleft\mathsf{unitr}"] \\ & \mathfrak{u} & \end{tikzcd} \qquad \begin{tikzcd} (\mathfrak{u} \triangleleft \mathfrak{u}) \triangleleft \mathfrak{u} \arrow[r, "\triangleleft\mathsf{assoc}"] \arrow[d, "\mu \triangleleft \mathfrak{u}"'] & \mathfrak{u} \triangleleft (\mathfrak{u} \triangleleft \mathfrak{u}) \arrow[r, "\mathfrak{u} \triangleleft \mu"] & \mathfrak{u} \triangleleft \mathfrak{u} \arrow[d, "\mu"] \\ \mathfrak{u} \triangleleft \mathfrak{u} \arrow[rr, "\mu"'] & & \mathfrak{u} \end{tikzcd}$$

One may recognize these as the usual diagrams for a monoid in a monoidal category, hence (since $\triangleleft$ corresponds to composition of polynomial endofunctors) for a *monad* as typically defined. However, because of the higher-categorical structure of types in HoTT, we should not only ask for these diagrams to commute, but for the cells exhibiting that these diagrams commute to themselves be subject to higher coherences, and so on, giving u not the structure of a (Cartesian) monad, but rather of a (Cartesian) ∞-*monad*.

Yet demonstrating that u is an ∞-monad involves specifying a potentially infinite amount of coherence data. Have we therefore traded both the Scylla of equality-up-to-isomorphism and the Charybdis of strictness for an even worse fate of higher coherence hell? The answer to this question, surprisingly, is negative, as there is a way to implicitly derive all of this data from a single axiom, which corresponds to the characteristic axiom of HoTT itself: univalence. To show this, we now introduce the central concept of this paper – that of a *polynomial universe*.

4 Polynomial Universes

4.1 Univalence

For any polynomial u `= (A , B)` and elements `a,b : A`, we may define a function that takes a proof of `a ≡ b` to an equivalence `B a ≃ B b`.

```
idToEquiv : ∀ {ℓ κ} (p : Poly ℓ κ) (a b : fst p)
            → a ≡ b → Σ (snd p a → snd p b) isEquiv
idToEquiv p a b e =
      transp (snd p) e
    , Iso→isEquiv (transp (snd p) (sym e) , (syml e , symr e))
```

We say that a polynomial u is *univalent* if for all `a,b : A`, this function is an equivalence.

```
isUnivalent : ∀ {ℓ κ} → Poly ℓ κ → Type (ℓ ⊔ κ)
isUnivalent (A , B) =
    (a b : A) → isEquiv (idToEquiv (A , B) a b)
```

We call this property of polynomials univalence by analogy with the usual univalence axiom of HoTT. Indeed, the univalence axiom simply states that the polynomial functor `(Type , λ X → X)` is itself univalent.

```
postulate
    ua : ∀ {ℓ} → isUnivalent (Type ℓ , λ X → X)
```

A key property of polynomial universes – qua polynomial functors – is that every polynomial universe u is a *subterminal object* in $\mathbf{Poly}^{\mathbf{Cart}}$, i.e. for any other polynomial `p`, the type of Cartesian lenses `p ⇆ u` is a proposition, i.e. any two Cartesian lenses with codomain u are equal.

```
isSubterminal : ∀ {ℓ κ} (u : Poly ℓ κ) → Setω
isSubterminal u = ∀ {ℓ' κ'} (p : Poly ℓ' κ')
                  → (f g : p ⇆ u)
                  → isCartesian p u f
                  → isCartesian p u g
                  → EqLens p u f g
```

To show this, we first prove the following *transport lemma*, which says that transporting along an identity `a ≡ b` induced by an equivalence `f : B a ≃ B b` in a univalent polynomial `p = (A , B)` is equivalent to applying `f`.

```
transpLemma : ∀ {ℓ κ} {u : Poly ℓ κ}
              → (univ : isUnivalent u)
              → {a b : fst u} (f : snd u a → snd u b)
              → (ef : isEquiv f) (x : snd u a)
              → transp (snd u) (inv (univ a b) (f , ef)) x ≡ f x
transpLemma {u = u} univ {a = a} {b = b} f ef x =
    coAp (ap fst (snd (snd (univ a b)) ((f , ef)))) x
```

The result then follows:

```
univ→Subterminal : ∀ {ℓ κ} (u : Poly ℓ κ)
                   → isUnivalent u
                   → isSubterminal u
univ→Subterminal u univ p f g cf cg a =
    ( inv univfg (fg⁻¹ , efg⁻¹)
    , (λ b → sym ((snd g a (transp (snd u)  (inv univfg (fg⁻¹ , efg⁻¹)) b))
                  ≡⟨ (ap (snd g a) (transpLemma univ fg⁻¹ efg⁻¹ b)) ⟩
                  ((snd g a (fg⁻¹ b))
                  ≡⟨ snd (snd (cg a)) (snd f a b) ⟩
```

```
                ((snd f a b) □)))))
    where univfg : isEquiv (idToEquiv u (fst f a) (fst g a))
          univfg = univ (fst f a) (fst g a)
          fg⁻¹ : snd u (fst f a) → snd u (fst g a)
          fg⁻¹ x = inv (cg a) (snd f a x)
          efg⁻¹ : isEquiv fg⁻¹
          efg⁻¹ = compIsEquiv (inv (cg a)) (snd f a)
                              (invIsEquiv (cg a)) (cf a)
```

We shall refer to polynomial functors with this property of being subterminal objects in $\mathbf{Poly}^{\mathbf{Cart}}$ as *polynomial universes.* As we shall see, such polynomial universes have many desirable properties as models of type theory.

If we think of a polynomial p as representing a family of types, then what this tells us is that if u is a polynomial universe, there is essentially at most one way for u to contain the types represented by p, where Containment is here understood as existence of a Cartesian lens $\mathtt{p} \leftrightarrows u$. In this case, we say that u *classifies* the types represented by p.

As a direct consequence of this fact, it follows that every diagram consisting of parallel Cartesian lenses into a polynomial universe automatically commutes, and moreover, every higher diagram that can be formed between the cells exhibiting such commutation must also commute, etc.

Hence the fact that u must satisfy the laws of a monad if there are Cartesian lenses $\eta : y \leftrightarrows u$ and $\mu : u \triangleleft u \leftrightarrows u$ follows immediately from the above theorem and the closure of Cartesian lenses under composition:

```
univ◁unitl : ∀ {ℓ κ} (u : Poly ℓ κ) → isUnivalent u
             → (η : y ⇆ u) (μ : (u ◁ u) ⇆ u)
             → isCartesian y u η → isCartesian (u ◁ u) u μ
             → EqLens (y ◁ u) u
                      (◁unitl u)
                      (comp (y ◁ u) (u ◁ u) u
                            (◁Lens y u u u η (id u)) μ)
univ◁unitl u univ η μ cη cμ =
    univ→Subterminal
        u univ (y ◁ u) (◁unitl u)
        (comp (y ◁ u) (u ◁ u) u
              (◁Lens y u u u η (id u)) μ)
        (◁unitlCart u)
        (compCartesian (y ◁ u) (u ◁ u) u
                       (◁Lens y u u u η (id u)) μ
                       (◁LensCart y u u u η (id u)
                                  cη (idCart u)) cμ)

univ◁unitr : ∀ {ℓ κ} (u : Poly ℓ κ) → isUnivalent u
             → (η : y ⇆ u) (μ : (u ◁ u) ⇆ u)
             → isCartesian y u η → isCartesian (u ◁ u) u μ
```

```
              → EqLens (u ◃ y) u
                       (◃unitr u)
                       (comp (u ◃ y) (u ◃ u) u
                             (◃Lens u u y u (id u) η) μ)
univ◃unitr u univ η μ cη cμ =
    univ→Subterminal
        u univ (u ◃ y) (◃unitr u)
        (comp (u ◃ y) (u ◃ u) u
              (◃Lens u u y u (id u) η) μ)
        (◃unitrCart u)
        (compCartesian (u ◃ y) (u ◃ u) u
                       (◃Lens u u y u (id u) η) μ
                       (◃LensCart u u y u (id u) η
                                  (idCart u) cη) cμ)


univ◃assoc : ∀ {ℓ κ} (u : Poly ℓ κ) → isUnivalent u
             → (η : y ⇆ u) (μ : (u ◃ u) ⇆ u)
             → isCartesian y u η → isCartesian (u ◃ u) u μ
             → EqLens ((u ◃ u) ◃ u) u
                      (comp ((u ◃ u) ◃ u) (u ◃ u) u
                            (◃Lens (u ◃ u) u u u μ (id u)) μ)
                      (comp ((u ◃ u) ◃ u) (u ◃ (u ◃ u)) u
                            (◃assoc u u u)
                            (comp (u ◃ (u ◃ u)) (u ◃ u) u
                                  (◃Lens u u (u ◃ u) u
                                         (id u) μ) μ))
univ◃assoc u univ η μ cη cμ =
    univ→Subterminal
        u univ ((u ◃ u) ◃ u)
        (comp ((u ◃ u) ◃ u) (u ◃ u) u
              (◃Lens (u ◃ u) u u u μ (id u)) μ)
        (comp ((u ◃ u) ◃ u) (u ◃ (u ◃ u)) u
              (◃assoc u u u)
              (comp (u ◃ (u ◃ u)) (u ◃ u) u
                    (◃Lens u u (u ◃ u) u (id u) μ) μ))
        (compCartesian ((u ◃ u) ◃ u) (u ◃ u) u
                       (◃Lens (u ◃ u) u u u μ (id u)) μ
                       (◃LensCart (u ◃ u) u u u μ (id u)
                                  cμ (idCart u)) cμ)
        (compCartesian ((u ◃ u) ◃ u) (u ◃ (u ◃ u)) u
                       (◃assoc u u u)
                       (comp (u ◃ (u ◃ u)) (u ◃ u) u
                             (◃Lens u u (u ◃ u) u
```

```
                    (id u) μ) μ)
        (◃assocCart u u u)
        (compCartesian
          (u ◃ (u ◃ u)) (u ◃ u) u
          (◃Lens u u (u ◃ u) u (id u) μ) μ
          (◃LensCart u u (u ◃ u) u (id u) μ
                    (idCart u) cμ) cμ))
```

And more generally, all the higher coherences of an ∞-monad should follow from the contractibility of the types of Cartesian lenses p $\leftrightarrows$ u that can be formed using μ and η.

4.1.1 Rezk Completion of Polynomial Functors

We have so far seen that polynomial universes are quite special objects in the theory of polynomial functors in HoTT, but what good would such special objects do us if they turned out to be exceedingly rare or difficult to construct? In fact, we can show that for *any* polynomial functor, there exists a corresponding polynomial universe, using a familiar construct from the theory of categories in HoTT – the *Rezk Completion.* We will show that this construction allows us to quotient any polynomial functor to a corresponding polynomial universe.

By our assumption of the univalence axiom, every polynomial functor p is classified by *some* univalent polynomial:

```
classifier : ∀ {ℓ κ} (p : Poly ℓ κ) → p ⇆ (Type κ , λ X → X)
classifier (A , B) = (B , λ a b → b)

classifierCart : ∀ {ℓ κ} (p : Poly ℓ κ)
                 → isCartesian p (Type κ , λ X → X)
                                (classifier p)
classifierCart p a = idIsEquiv
```

We then obtain the Rezk completion of p as the image factorization in $\mathbf{Poly}^{\mathbf{Cart}}$ of this classifying lens:

```
Rezk : ∀ {ℓ κ} (p : Poly ℓ κ) → Poly (lsuc κ) κ
Rezk (A , B) = (Im B) , (λ (X , _) → X)

→Rezk : ∀ {ℓ κ} (p : Poly ℓ κ) → p ⇆ (Rezk p)
→Rezk {κ = κ} p =
    factorcart1 p (Type κ , λ X → X)
                  (classifier p)
                  (classifierCart p)

Rezk→ : ∀ {ℓ κ} (p : Poly ℓ κ) → (Rezk p) ⇆ (Type κ , λ X → X)
Rezk→ {κ = κ} p =
```

```
    factorcart2 p (Type κ , λ X → X)
                  (classifier p)
                  (classifierCart p)
```

Because the map `Rezk→` defined above is a Cartesian embedding, and the polynomial `(Type κ , λ X → X)` is univalent, it follows that `Rezk p` is a polynomial universe:

```
RezkSubterminal : ∀ {ℓ κ} (p : Poly ℓ κ) → isSubterminal (Rezk p)
RezkSubterminal {κ = κ} p q (f , f♯) (g , g♯) cf cg x =
    ( pairEq (inv (ua (fst (f x)) (fst (g x)))
                  ( (λ y → inv (cg x) (f♯ x y))
                  , compIsEquiv (inv (cg x))
                                (f♯ x)
                                (invIsEquiv (cg x))
                                (cf x))) ∥-∥IsProp
    , λ y → f♯ x y
            ≡⟨ sym (g♯ x (transp (λ X → X)
                                 (inv (ua (fst (f x)) (fst (g x)))
                                      ((λ z → inv (cg x) (f♯ x z)) , (compIsEquiv (inv (c
                                                   (invIsEquiv (cg x))
                                                   (cf x)))) y)
                     ≡⟨ (ap (g♯ x)
                            (transpLemma ua
                               (λ z → inv (cg x) (f♯ x z))
                               (compIsEquiv (inv (cg x)) (f♯ x)
                                            (invIsEquiv (cg x)) (cf x))
                               y)) ⟩
                     snd (snd (cg x)) (f♯ x y)) ⟩
            ap (g♯ x) (sym (lemma1 ∥-∥IsProp y)) )
    where lemma1 : {a b : fst (Rezk p)}
                   → {e : fst a ≡ fst b}
                   → (e' : transp (λ c → ∥ (Fibre (snd p) c) ∥)
                                  e (snd a)
                           ≡ (snd b))
                   → (z : fst a)
                   → transp (snd (Rezk p)) (pairEq e e') z
                     ≡ transp (λ X → X) e z
          lemma1 {e = refl} refl z = refl
```

5 Π-Types, Jump Monads & Distributive Laws

We have so far considered how polynomial universes may be equipped with structure to interpret the unit type and dependent pair types. We have not yet, however, said much in the way of *dependent function types.* In order to rectify this omission, it will

first be prudent to consider some additional structure on the category of polynomial functors – specifically a new functor $\uparrow\uparrow : \mathbf{Poly}^{\mathbf{Cart}} \times \mathbf{Poly} \to \mathbf{Poly}$ that plays a similar role for Π types as the composition $\triangleleft : \mathbf{Poly} \times \mathbf{Poly} \to \mathbf{Poly}$ played for Σ types, and which in turn bears a close connection to a class of structured morphisms in **Poly**, which we refer to as *jump morphisms*.

5.1 The ↑↑ Functor & Jump Morphisms

The ↑↑ functor can be loosely defined as the solution to the following problem: given a polynomial universe u, find $u \uparrow\uparrow u$ such that u classifies $u \uparrow\uparrow u$ if and only if u has the structure to interpret Π types (in the same way that u classifies $u \triangleleft u$ if and only if u has the structure to interpret Σ types). Generalizing this to arbitrary pairs of polynomials $p = (A, B)$, $q = (C, D)$ then yields the following formula for $p \uparrow\uparrow q$:

$$p \uparrow\uparrow q = \sum_{(a,f):\sum_{a:A} C^{B(a)}} y^{\prod_{b:B(a)} D(f(b))}$$

```
_↑↑_ : ∀ {ℓ ℓ' κ κ'} → Poly ℓ κ → Poly ℓ' κ' → Poly (ℓ ⊔ κ ⊔ ℓ') (κ ⊔ κ')
(A , B) ↑↑ (C , D) =
    ( Σ A (λ a → B a → C)
    , (λ (a , f) → (b : B a) → D (f b)))
```

Note that this construction is straightforwardly functorial with respect to arbitrary lenses in its 2nd argument, but is only functorial with respect to Cartesian lenses in its first argument:

```
↑↑Lens : ∀ {ℓ ℓ' ℓ'' ℓ''' κ κ' κ'' κ'''}
        → (p : Poly ℓ κ) (q : Poly ℓ' κ')
        → (r : Poly ℓ'' κ'') (s : Poly ℓ''' κ''')
        → (f : p ⇆ r) → isCartesian p r f
        → (g : q ⇆ s) → (p ↑↑ q) ⇆ (r ↑↑ s)
↑↑Lens p q r s (f , f♯) cf (g , g♯) =
    ( (λ (a , h) → (f a) , (λ b → g (h (f♯ a b))))
    , λ (a , h) k b
      → g♯ (h b) (transp (snd s)
                    (ap g (ap h (snd (snd (cf a)) b)))
                    (k (inv (cf a) b))))
```

By construction, the existence of a Cartesian lens `(π , π♯) : u ◁ u ⇆ u` effectively shows that u is closed under Π-types, since:

- `π` maps a pair `(A , B)` consisting of `A : U` and `B : u(A) → U` to a term `π(A,B)` representing the corresponding Π type. This corresponds to the type formation rule

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B[x]\ \mathsf{Type}}{\Gamma \vdash \Pi x : A.B[x]\ \mathsf{Type}}$$

- The "elimination rule" $\pi\sharp$ `(A , B)`, for any pair `(A , B)` as above, maps an element `f : π(A,B)` to a function $\pi\sharp$ `(A , B) f : (a : u(A)) → u (B x)` which takes an element `x` of `A` and yields an element of `B x`. This corresponds to the rule for function application:

$$\frac{\Gamma \vdash f : \Pi x : A.B[x] \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a]}$$

- Since $\pi\sharp$ `(A , B)` is an equivalence, it follows that there is an inverse $\pi\sharp^{-1}$ `(A , B) : ((x : u(A)) → u(B(x)) → u(π(A,B))`, which corresponds to λ-abstraction:

$$\frac{\Gamma, x : A \vdash f[x] : B[x]}{\Gamma \vdash \lambda x.f[x] : \Pi x : A.B[x]}$$

- The fact that $\pi\sharp^{-1}$ `(A , B)` is both a left and a right inverse to $\pi\sharp$ then corresponds to the β and η laws for Π types.

$$(\lambda x.f[x])\ a = f[a] \qquad f = \lambda x.f\ x$$

5.2 From Jump Morphisms to Distributive Laws

6 Other Type Formers in Polynomial Universes

6.1 Identity Types

6.2 Positive Types

7 Conclusion