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 categorical semantics 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 categorical semantics of 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

The central idea of dependent type theory (c.f. [Mar75]) is that *order of events* is fundamental to the mathematical story arc: when playing out any specific example story within 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 robust categorical semantics. 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 a 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) [**Voevodsky:2013a**]. 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
```

The structure of this paper is as follows:

- In Section 2, we give an introductory presentation of dependent type theory and natural models, followed by a recap the basics of HoTT that will be used throughout the rest of the paper.
- In Section 3, we outline the basic theory of polynomial functors in HoTT, culminating in a demonstration of how to model dependent pair types using polynomial

functors. In order to show that these polynomial functors are in fact monads, however, we will need some additional technology, to which we devote the following section.

- In Section 4, we introduce the key concept of a *polynomial universe* as a polynomial functor satisfying a certain *univalence* condition, that allows us to straightforwardly derive the monad laws for polynomial universes equipped with the structure to interpret dependent pair types.
- In Section 5, building on the ideas of the previous sections, we show how to model dependent function types with polynomial functors, and demonstrate that any polynomial universe equipped with this structure – along with the aforementioned structure for interpreting dependent pair types – gives rise to a self-distributive law of the corresponding monad.
- In Section 6, we conclude the paper by sketching how this theory may be further developed to handle identity types, inductive types, and other key concepts from dependent type theory.

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

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`.

[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`. We also note in passing that, due to size issues, the universe `Type` is not in fact one type, but rather a whole family of types, stratified by a hierarchy of *levels*. However, this structure of levels is not of much concern to us in this paper, so we shall do our best to ignore it.

```
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)
```

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

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

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

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

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

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

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

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.

$$\begin{array}{ccc} \Gamma & \xrightarrow{a} & \Gamma, x : A \\ & \searrow & \downarrow{\scriptstyle A} \\ & & \Gamma \end{array}$$

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:

$$\begin{array}{ccc} \Gamma, x : A[f] & \longrightarrow & \Delta, x : A \\ {\scriptstyle A[f]}\downarrow & \lrcorner & \downarrow{\scriptstyle A} \\ \Gamma & \xrightarrow[f]{} & \Delta \end{array}$$

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. It is precisely this problem which natural models exist to solve.

2.2 Natural Models

The key insight of Awodey [Awo14] 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 `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

$$\begin{array}{ccc} \gamma \times_{\alpha,u} \mathcal{U}_\bullet & \Longrightarrow & \mathcal{U}_\bullet \\ {\scriptstyle u[\alpha]}\Downarrow & \lrcorner & \Downarrow{\scriptstyle u} \\ \gamma & \underset{\alpha}{\Longrightarrow} & \mathcal{U} \end{array}$$

of u along α is representable.

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

$$\begin{array}{ccc} \overline{u}(\overline{u}(P)) & \overset{\overline{u}(\overline{u}(\alpha))}{\Longrightarrow} & \overline{u}(\overline{u}(Q)) \\ {\scriptstyle \sigma_P}\Downarrow & \lrcorner & \Downarrow{\scriptstyle \sigma_Q} \\ \overline{u}(P) & \underset{\overline{u}(\alpha)}{\Longrightarrow} & \overline{u}(Q) \end{array}$$

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 $u : \mathcal{U} \longrightarrow \mathsf{Type}$, 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 **[Voevodsky:2013a]** 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.[2]:

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

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

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

{-# REWRITE comprewrite #-}
```

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

We additionally have the following "dependent" form of ap as above, allowing us to apply a dependent function to both sides of an identity, in a suitable manner:

```
apd : ∀ {ℓ0 ℓ1 κ} {A : Type ℓ0} {B : Type ℓ1} {f : A → B}
      → (C : B → Type κ) {a a' : A}
      → (g : (x : A) → C (f x)) → (e : a ≡ a')
      → transp C (ap f e) (g a) ≡ g a'
apd B f refl = refl
```

[2]We also 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:

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'`.

```
module PairEq {ℓ κ} {A : Type ℓ} {B : A → Type κ}
              {a a' : A} {b : B a} {b' : B a'} where

    pairEq : (e : a ≡ a') (e' : transp B e b ≡ b') → (a , b) ≡ (a' , b')
    pairEq refl refl = refl
```

We then have the following laws governing equality proofs for pairs.

```
    pairEqβ1 : (e : a ≡ a') (e' : transp B e b ≡ b')
             → ap fst (pairEq e e') ≡ e
    pairEqβ1 refl refl = refl

    pairEqη : (e : (a , b) ≡ (a' , b'))
            → pairEq (ap fst e) (apd B snd e) ≡ e
    pairEqη refl = refl

open PairEq public
```

2.3.2 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))
```

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 : ∀ {ℓ0 ℓ1 ℓ2} {A : Type ℓ0} {B : Type ℓ1} {C : Type ℓ2}
            → {g : B → C} {f : A → B} → isEquiv g → isEquiv f
            → isEquiv (λ a → g (f a))
compIsEquiv {g = g} {f = 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                    □)))
```

A closely-related notion to equivalence 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 **[Voevodsky:2013a]** 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:

```
module Iso↔Equiv {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B} where

    Iso→isEquiv : Iso f → isEquiv f
    Iso→isEquiv (g , l , r) = ((g , l) , (g , r))

    isEquiv→Iso : isEquiv f → Iso f
    isEquiv→Iso ((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

open Iso↔Equiv public
```

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:

```
module InvEquiv {ℓ κ} {A : Type ℓ} {B : Type κ} {f : A → B} where

    inv : isEquiv f → B → A
    inv (_ , (h , _)) = h

    isoInv : (isof : Iso f) → Iso (fst isof)
    isoInv (g , l , r) = (f , r , l)

    invIsEquiv : (ef : isEquiv f) → isEquiv (inv ef)
    invIsEquiv ef = Iso→isEquiv (isoInv (isEquiv→Iso ef))

open InvEquiv public
```

We note 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:

```
module TranspEquiv {ℓ κ} {A : Type ℓ} {B : A → Type κ}
                   {a b : A} (e : a ≡ b) where

    syml : (x : B a) → transp B (sym e) (transp B e x) ≡ x
    syml x rewrite e = refl

    symr : (y : B b) → transp B e (transp B (sym e) y) ≡ y
    symr y rewrite e = refl

    transpIsEquiv : isEquiv {A = B a} {B = B b} (λ x → transp B e x)
    transpIsEquiv =
        Iso→isEquiv ((λ x → transp B (sym e) x) , (syml , symr))

open TranspEquiv public
```

2.3.3 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 ∥)
    ∥-∥≡Contr : ∀ {ℓ} {A : Type ℓ} {a b : ∥ A ∥} {e : a ≡ b}
              → ∥-∥IsProp ≡ e
    ∥-∥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:

```
module Epi-Mono {ℓ κ} {A : Type ℓ} {B : Type κ} (f : A → B) where
```

We say that a function `f : A → B` is *injective* (i.e. a monomorphism), if for all `a , b : A` the map `ap f : a ≡ b → f a ≡ f b` is an equivalence

```
    isMono : Type (ℓ ⊔ κ)
    isMono = {a b : A} → isEquiv (ap {a = a} {a' = b} f)
```

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

We then say that `f` is *surjective* (i.e. an epimorphism), if all of its fibres are merely inhabited.

```
    isEpi : Type κ
    isEpi = (b : B) → ∥ Fibre b ∥
```

```
open Epi-Mono public
```

```
module EMFactor {ℓ κ} {A : Type ℓ} {B : Type κ} (f : A → B) where
```

Given a function `f`, its *image* is the type of elements of `B` whose fibres are merely inhabited.

```
  Im : Type κ
  Im = Σ B (λ 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 : A → Im
  factor1 a = (f a) , in‖-‖ (a , refl)

  factor2 : Im → B
  factor2 (b , _) = b

  factor≡ : (a : A) → factor2 (factor1 a) ≡ f a
  factor≡ a = refl

  factor1IsEpi : isEpi factor1
  factor1IsEpi (b , x) =
      ‖-‖Rec ‖-‖IsProp
             (λ {(a , refl) → in‖-‖ (a , pairEq refl ‖-‖IsProp)})
             x

  factor2IsMono : isMono factor2
  factor2IsMono {a = (a , α)} {b = (b , β)} =
      Iso→isEquiv ( (λ e → pairEq e ‖-‖IsProp)
                  , ( (λ e → (pairEq (ap factor2 e) ‖-‖IsProp)
                             ≡⟨ (ap (pairEq (ap factor2 e)) ‖-‖≡Contr) ⟩
                             ( _
                             ≡⟨ pairEqη e ⟩
                             (e □)))
                    , λ e → pairEqβ1 e ‖-‖IsProp))
```

```
open EMFactor public
```

Some additional facts about the identity type, that will be used in formalizing the results of this paper, are given in Appendix A.

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** $\to$ **Type**.

A *polynomial functor* is a coproduct of representable functors **Type** $\to$ **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 (Type ℓ) (λ A → A → Type 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:

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

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:

```
_⇆_ : ∀ {ℓ0 ℓ1 κ0 κ1} → Poly ℓ0 κ0 → Poly ℓ1 κ1 → Type (ℓ0 ⊔ ℓ1 ⊔ κ0 ⊔ κ1)
(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 : ∀ {ℓ0 ℓ1 κ0 κ1}
         → {p : Poly ℓ0 κ0} (q : Poly ℓ1 κ1)
```

```
        → (f g : p ⇆ q) → Type (ℓ0 ⊔ ℓ1 ⊔ κ0 ⊔ κ1)
EqLens {p = (A , B)} (C , D) (f , f♯) (g , g♯) =
  Σ ((a : A) → f a ≡ g a)
    (λ e → (a : A) (d : D (f a))
          → f♯ a d ≡ g♯ a (transp D (e a) d))
```

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 : ∀ {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2}
       → {p : Poly ℓ0 κ0} {q : Poly ℓ1 κ1} (r : Poly ℓ2 κ2)
       → p ⇆ q → q ⇆ r → p ⇆ r
comp r (f , f♯) (g , g♯) =
     ( (λ a → g (f a)) , λ a z → f♯ a (g♯ (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.[3]

```
module Vert-Cart {ℓ0 ℓ1 κ0 κ1} {p : Poly ℓ0 κ0}
                 (q : Poly ℓ1 κ1) (f : p ⇆ q) where

    isVertical : Set (ℓ0 ⊔ ℓ1)
    isVertical = isEquiv (fst f)

    isCartesian : Set (ℓ0 ⊔ κ0 ⊔ κ1)
    isCartesian = (a : fst p) → isEquiv (snd f a)

open Vert-Cart public
```

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

[3] For a proof that this notion of Cartesian morphism between polynomials is equivalent to the one given previously in Section 2.2, see Chapter 5.5 of [**spivak2022poly**]

```
module VertCartFactor {ℓ0 ℓ1 κ0 κ1} {p : Poly ℓ0 κ0}
                      (q : Poly ℓ1 κ1) (f : p ⇆ q) where

    vcIm : Poly ℓ0 κ1
    vcIm = (fst p , λ x → snd q (fst f x))

    vertfactor : p ⇆ vcIm
    vertfactor = ( (λ x → x) , (λ a x → snd f a x) )

    vertfactorIsVert : isVertical vcIm vertfactor
    vertfactorIsVert = idIsEquiv

    cartfactor : vcIm ⇆ q
    cartfactor = ( fst f , (λ a x → x) )

    cartfactorIsCart : isCartesian q cartfactor
    cartfactorIsCart x = idIsEquiv

    vertcartfactor≡ : EqLens q f (comp q vertfactor cartfactor)
    vertcartfactor≡ = ( (λ a → refl) , (λ a b → refl) )

open VertCartFactor public
```

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`.[4]

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 (id p)
idCart p a = idIsEquiv
```

[4]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*.

```
compCartesian : ∀ {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2}
                → {p : Poly ℓ0 κ0} {q : Poly ℓ1 κ1} (r : Poly ℓ2 κ2)
                → {f : p ⇆ q} {g : q ⇆ r}
                → isCartesian q f → isCartesian r g
                → isCartesian r (comp r f g)
compCartesian r {f = (f , f♯)} {g = (g , g♯)} cf cg a =
    compIsEquiv (cf a) (cg (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 lens `(f , f♯) : p ⇆ q` to be a *vertical embedding* if `f` is a monomorphism, and a *vertical surjection* if `f` is an epimorphism.

```
module VertEpi-Mono {ℓ0 ℓ1 κ0 κ1} {p : Poly ℓ0 κ0}
                    (q : Poly ℓ1 κ1) (f : p ⇆ q) where

    isVerticalEmbedding : Set (ℓ0 ⊔ ℓ1)
    isVerticalEmbedding = isMono (fst f)

    isVerticalSurjection : Set ℓ1
    isVerticalSurjection = isEpi (fst f)

open VertEpi-Mono public
```

Then every Cartesian lens can be factored into a vertical surjection followed by a vertical embedding, both of which are Cartesian.

```
module CartEMFactorization {ℓ0 ℓ1 κ0 κ1} {p : Poly ℓ0 κ0}
           (q : Poly ℓ1 κ1) (f : p ⇆ q) (cf : isCartesian q f) where

    cartIm : Poly ℓ1 κ1
    cartIm = (Im (fst f) , λ (x , _) → snd q x)

    factorcart1 : p ⇆ cartIm
    factorcart1 = ( factor1 (fst f) , snd f )

    factorcart1IsCart : isCartesian cartIm factorcart1
    factorcart1IsCart = cf
```

```
    factorcart1IsEpi : isVerticalSurjection cartIm factorcart1
    factorcart1IsEpi = factor1IsEpi (fst f)

    factorcart2 : cartIm ⇆ q
    factorcart2 = ( factor2 (fst f) , (λ _ y → y) )

    factorcart2IsCart : isCartesian q factorcart2
    factorcart2IsCart _ = idIsEquiv

    factorcart2IsMono : isVerticalEmbedding q factorcart2
    factorcart2IsMono = factor2IsMono (fst f)

    factorcart≡ : EqLens q f (comp q factorcart1 factorcart2)
    factorcart≡ = ( (λ x →  refl) , (λ x y → refl) )

open CartEMFactorization public
```

We note in passing that the vertical embeddings are indeed the monomorphisms in $\mathbf{Poly}^{\mathbf{Cart}}$, i.e. if `f : q ⇆ r` is a both Cartesian and a vertical embedding, then for any Cartesian `g h : p ⇆ q` such that `f ∘ g ≡ f ∘ h`, we have `g = h`.

```
VertEmbedding→PolyCartMono :
    ∀ {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} {p : Poly ℓ0 κ0}
    → {q : Poly ℓ1 κ1} (r : Poly ℓ2 κ2) {f : q ⇆ r}
    → isCartesian r f → isVerticalEmbedding r f
    → {g h : p ⇆ q} → isCartesian q g → isCartesian q h
    → EqLens r (comp r g f) (comp r h f)
    → EqLens q g h
VertEmbedding→PolyCartMono {p = p} {q = q} r {f = (f , f♯)} cf vef
                           {g = (g , g♯)} {h = (h , h♯)} cg ch (e , e♯) =
    ( (λ a → inv vef (e a))
    , (λ a d → (g♯ a d)
                  ≡⟨ ap (g♯ a) (sym (snd (snd (cf (g a))) d)) ⟩
              ( _ ≡⟨ (e♯ a (inv (cf (g a)) d)) ⟩
              ( _ ≡⟨ (ap (h♯ a)
                         ( _ ≡⟨ (ap (f♯ (h a))
                                    (transpPre vef
                                      (λ x y → inv (cf x) y)
                                      (e a))) ⟩
                         ( _ ≡⟨ snd (snd (cf (h a))) _ ⟩
                         ( _ □)))) ⟩
              ((h♯ a (transp (snd q) (inv vef (e a)) d)) □)))) )
```

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

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

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

```
_◃_ : ∀ {ℓ0 ℓ1 κ0 κ1} → Poly ℓ0 κ0 → Poly ℓ1 κ1
      → Poly (ℓ0 ⊔ κ0 ⊔ ℓ1) (κ0 ⊔ κ1)
(A , B) ◃ (C , D) =
    ( Σ A (λ a → B a → C)
    , λ (a , f) → Σ (B a) (λ b → D (f b)) )

_◃◃[_]_ : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
        → {p : Poly ℓ0 κ0} {q : Poly ℓ2 κ2} → p ⇆ q
        → {r : Poly ℓ1 κ1} (s : Poly ℓ3 κ3) → r ⇆ s
        → (p ◃ r) ⇆ (q ◃ s)
(f , f♯) ◃◃[ s ] (g , g♯) =
    ((λ (a , γ) → (f a , λ b' → g (γ (f♯ a b'))))
    , λ (a , γ) (b' , d') → ((f♯ a b') , g♯ (γ (f♯ a b')) d'))
```

where `◃Lens` is the action of $\triangleleft$ 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, `σ♯ (A , B)` takes a term of type `σ (A , B)` and yields a term `fst (σ♯ (A , B)) : A` along with a term `snd (σ♯ (A , B)) : B (fst (σ♯ (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 `σ♯ (A , B)` has is an equivalence implies it has an inverse `σ♯⁻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 [Awo14; AN18], 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.

```
module ◃Assoc {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} (p : Poly ℓ0 κ0)
              (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2) where

  ◃assoc : ((p ◃ q) ◃ r) ⇆ (p ◃ (q ◃ r))
  ◃assoc = ( (λ ((a , γ) , δ)
                → (a , (λ b → (γ b , λ d → δ (b , d))))))
```

```
            , (λ _ (b , (d , x)) → ((b , d) , x)) )

  ◃assoc⁻¹ : (p ◃ (q ◃ r)) ⇆ ((p ◃ q) ◃ r)
  ◃assoc⁻¹ = ( (λ (a , γ) → ( (a , (λ x → fst (γ x)))
                            , (λ (x , y) → snd (γ x) y) ))
             , λ _ ((x , y) , z) → (x , (y , z)) )

open ◃Assoc public
```

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

```
module ◃LRUnit {ℓ κ} (p : Poly ℓ κ) where

  ◃unitl : (y ◃ p) ⇆ p
  ◃unitl = ( (λ (_ , a) → a tt) , λ (_ , a) x → (tt , x) )

  ◃unitl⁻¹ : p ⇆ (y ◃ p)
  ◃unitl⁻¹ = ( (λ a → (tt , λ _ → a)) , (λ a (_ , b) → b ) )

  ◃unitr : (p ◃ y) ⇆ p
  ◃unitr = ( (λ (a , γ) → a) , (λ (a , γ) b → (b , tt)) )

  ◃unitr⁻¹ : p ⇆ (p ◃ y)
  ◃unitr⁻¹ = ( (λ a → a , (λ _ → tt)) , (λ a (b , _) → b) )

open ◃LRUnit public
```

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

```
◃◃Cart : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
         → {p : Poly ℓ0 κ0} (q : Poly ℓ2 κ2) {f : p ⇆ q}
         → {r : Poly ℓ1 κ1} (s : Poly ℓ3 κ3) {g : r ⇆ s}
         → isCartesian q f → isCartesian s g
         → isCartesian (q ◃ s) (f ◃◃[ s ] g)
◃◃Cart q {f = (f , f♯)} s {g = (g , g♯)} cf cg (a , γ) =
    pairEquiv (f♯ a) (λ x → g♯ (γ (f♯ a x)))
              (cf a) (λ x → cg (γ (f♯ a x)))
```

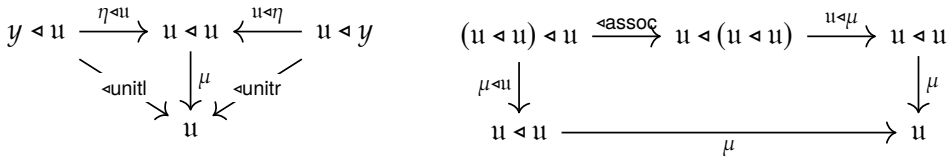
and all of the above-defined structure morphisms for ◃ are Cartesian.

```
module ◃AssocCart {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} (p : Poly ℓ0 κ0)
                  (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2) where

  ◃assocCart : isCartesian (p ◃ (q ◃ r)) (◃assoc p q r)
```

```
    ◃assocCart _ =
        Iso→isEquiv
            ( snd (◃assoc⁻¹ p q r) _
            , ( (λ _ → refl) , (λ _ → refl) ) )

    ◃assoc⁻¹Cart : isCartesian ((p ◃ q) ◃ r) (◃assoc⁻¹ p q r)
    ◃assoc⁻¹Cart _ =
        Iso→isEquiv
            ( snd (◃assoc p q r) _
            , ( (λ _ → refl) , (λ _ → refl) ) )

open ◃AssocCart public

module ◃LRUnitCart {ℓ κ} (p : Poly ℓ κ) where

    ◃unitlCart : isCartesian p (◃unitl p)
    ◃unitlCart _ =
        Iso→isEquiv
            ( snd (◃unitl⁻¹ p) _
            , ((λ _ → refl) , (λ _ → refl)) )

    ◃unitl⁻¹Cart : isCartesian (y ◃ p) (◃unitl⁻¹ p)
    ◃unitl⁻¹Cart _ =
        Iso→isEquiv
            ( snd (◃unitl p) _
            , ((λ _ → refl) , (λ _ → refl)) )

    ◃unitrCart : isCartesian p (◃unitr p)
    ◃unitrCart _ =
        Iso→isEquiv
            ( snd (◃unitr⁻¹ p) _
            , ((λ _ → refl) , (λ _ → refl)) )

    ◃unitr⁻¹Cart : isCartesian (p ◃ y) (◃unitr⁻¹ p)
    ◃unitr⁻¹Cart _ =
        Iso→isEquiv
            ( snd (◃unitr p) _
            , ((λ _ → refl) , (λ _ → refl)) )

open ◃LRUnitCart public
```

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



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 = `(U , El)`, we say that u is *univalent* if u is a *subterminal object* in $\mathbf{Poly}^{\mathbf{Cart}}$, i.e. for any other polynomial `p`, the type of Cartesian lenses `p` $\leftrightarrows$ u is a proposition, i.e. any two Cartesian lenses with codomain u are equal.

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

We call this property of polynomials univalence in analogy with the usual univalence axiom of HoTT. Indeed, the univalence axiom can be equivalently stated as the fact that the polynomial functor `(Type , λ X → X)` is itself univalent.

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

We shall refer to univalent polynomial functors as *polynomial universes*. f 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:

```
module UnivMonad {ℓ κ} (u : Poly ℓ κ) (univ : isUnivalent u)
                 (η : y ⇆ u) (μ : (u ◃ u) ⇆ u)
                 (cη : isCartesian u η) (cμ : isCartesian u μ) where

    univ◃unitl : EqLens u (◃unitl u) (comp u (η ◃◃[ u ] (id u)) μ)
    univ◃unitl = univ (◃unitlCart u)
                      (compCartesian u (◃◃Cart u u cη (idCart u)) cμ)

    univ◃unitr : EqLens u (◃unitr u) (comp u ((id u) ◃◃[ u ] η) μ)
    univ◃unitr = univ (◃unitrCart u)
                      (compCartesian u (◃◃Cart u u (idCart u) cη) cμ)


    univ◃assoc : EqLens u (comp u (μ ◃◃[ u ] (id u)) μ)
                          (comp u (◃assoc u u u)
                                  (comp u ((id u) ◃◃[ u ] μ) μ))
    univ◃assoc = univ (compCartesian u (◃◃Cart u u cμ (idCart u)) cμ)
                      (compCartesian u (◃assocCart u u u)
                                       (compCartesian u
                                         (◃◃Cart u u (idCart u) cμ) cμ))

open UnivMonad public
```

And more generally, all the higher coherences of an ∞-monad would follow – if we bothered to write them out – from the contractibility of the types of Cartesian lenses $\mathtt{p} \leftrightarrows u$ that can be formed using μ and η.

4.1.1 Examples of Polynomial Universes

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, polynomial universes are surprisingly plentiful in univalent type theory. We have already seen how the univalence axiom implies that `(Type , λ X → X)` is a polynomial universe. From this single example, a plethora of others can be seen to follow, many of which encompass familiar constructs from programming and mathematics.

In a sense, the polynomial `(Type , λ X → X)` is *universal* among polynomials in $\mathbf{Poly}^{\mathbf{Cart}}$ in that, for any polynomial `p`, there is a (necessarily unique, by univalence) Cartesian morphism `p ⇆ (Type , λ X → X)`. Or rather, there would be, were it not for the size issues preventing `(Type , λ X → X)` from being a single object. Instead, it can more accurately be said that the family of polynomials `(Type ℓ , λ X → X)` for all `ℓ : Level` is universal among polynomials in $\mathbf{Poly}^{\mathbf{Cart}}$ – this can be shown straightforwardly as follows:

```
module PolyCartUniv {ℓ κ} (p : Poly ℓ κ) where

    classifier : p ⇆ (Type κ , λ X → X)
    classifier = (snd p , λ _ b → b)

    classifierCart : isCartesian (Type κ , λ X → X) classifier
    classifierCart _ = idIsEquiv
```

In other words, every polynomial functor `p` is classified by some polynomial universe. Moreover, if the classifying morphism `p ⇆ (Type κ , λ X → X)` is a Vertical embedding (i.e. a monomorphism in $\mathbf{Poly}^{\mathbf{Cart}}$), then `p` itself is also a polynomial universe – for any pair of Cartesian morphisms `f g : q ⇆ p`, since `(Type κ , λ X → X)` is univalent, we have `classifier ∘ f ≡ classifier ∘ g`, but then since `classifier` is assumed to be a monomorphism, this implies that `f ≡ g`.

```
    polyCartUniv : isVerticalEmbedding (Type κ , λ X → X) classifier
                   → isUnivalent p
    polyCartUniv veclassifier cf cg =
        VertEmbedding→PolyCartMono
            (Type κ , λ X → X) classifierCart veclassifier cf cg
            (ua (compCartesian _ cf classifierCart)
                (compCartesian _ cg classifierCart))

open PolyCartUniv public
```

It follows that, for any type family `P : Type → Type`, we can create a polynomial *sub-universe* of `(Type , λ X → X)` by restricting to those types `X` for which there *merely* exists an inhabitant of `P X`.

```
module SubUniv {ℓ κ} (P : Type ℓ → Type κ) where

    subUniv : Poly (lsuc ℓ) ℓ
    subUniv = (Σ (Type ℓ) (λ X → ‖ P X ‖) , λ (X , _) → X)
```

```
    subUnivClassifierVertEmb :
        isVerticalEmbedding (Type ℓ , λ X → X) (classifier subUniv)
    subUnivClassifierVertEmb =
        Iso→isEquiv
            ( (λ e → pairEq e ∥-∥IsProp)
            , ( ( λ e → (pairEq (ap (fst (classifier subUniv)) e)
                                ∥-∥IsProp)
                      ≡⟨ ap (λ e' →
                                pairEq (ap (fst (classifier subUniv))
                                           e) e')
                            ∥-∥≡Contr ⟩ ( _
                      ≡⟨ (pairEqη e) ⟩
                      (e □)))
                , (λ e → pairEqβ1 e ∥-∥IsProp) ) )

    subUnivIsUniv : isUnivalent subUniv
    subUnivIsUniv = polyCartUniv subUniv subUnivClassifierVertEmb

open SubUniv public
```

As a first example of a polynomial universe other than `(Type , λ X → X)`, then, we may consider the polynomial universe of *propositions* P:

```
module PropUniv where

    P : Poly (lsuc lzero) lzero
    P = subUniv isProp
```

If we write out explicitly the polynomial endofunctor defined by P we see that it has the following form:

$$y \mapsto \sum_{\phi:\mathbf{Prop}} y^{\phi}$$

This endofunctor (in fact it is a monad) is well-known in type theory by another name – the *partiality* monad. Specifically, this is the monad `M` whose kleisli morphisms `A → M B` correspond to *partial functions* from `A` to `B`, that associate to each element `a : A`, a proposition `def f a` indicating whether or not the value of `f` is defined at `a`, and a function `val : def f a → B` that takes a proof that `f` is defined at `a` to its value at `a`.

If we return to the original example of the polynomial universe `(Type , λ X → X)` we see that the associated polynomial endofunctor (which, by the above argument, is also a monad) has a similar form.

$$y \mapsto \sum_{X:\mathbf{Type}} y^{X}$$

In this case, we can think of this as a "proof relevant" partiality monad `M`, such that a function `f : A → M B` associates to each element `a : A` a *type* `Def f a` of proofs that `f` is defined at `a`, and a function `val : Def f a → B`.[5]

More generally, we can say that, for any polynomial universe closed under dependent pair types, the associated monad will be a kind of (potentially proof-relevant) partiality monad, where the structure of the polynomial universe serves to dictate which types can count as *evidence* for whether or not a value is defined.

Rezk Completion

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*. [**ahrens2015univalent**] We will show that this construction allows us to quotient any polynomial functor to a corresponding univalent polynomial, i.e. a polynomial universe.

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

```
module RezkCompletion {ℓ κ} (p : Poly ℓ κ) where

  Rezk : Poly (lsuc κ) κ
  Rezk = cartIm (Type κ , λ X → X)
         (classifier p) (classifierCart p)

  →Rezk : p ⇆ Rezk
  →Rezk = factorcart1 (Type κ , λ X → X)
              (classifier p) (classifierCart p)

  Rezk→ : Rezk ⇆ (Type κ , λ X → X)
  Rezk→ = factorcart2 (Type κ , λ X → X)
              (classifier p) (classifierCart p)
```

The polynomial `Rezk` defined above can be seen to have the same form as a subuniverse of `(Type , λ X → X)`; hence it is a polynomial universe, as desired.

```
  RezkUniv : isUnivalent Rezk
  RezkUniv = subUnivIsUniv (λ X → Σ (fst p) (λ a → (snd p a) ≡ X))

open RezkCompletion public
```

As an example of how the Rezk completion allows us to "upgrade" a polynomial functor (a polynomial monad, even) into a polynomial universe, consider the following definition of the finite ordinals as a family of types indexed by the type `Nat` of natural numbers:

[5] the conception of the monad determined by `(Type , λ X → X)` as a "proof relevant" partiality monad was communicated to the first author during private conversations with Jonathan Sterling.

```
module FinUniv where
  open import Agda.Builtin.Nat
```

We first define the standard ordering on natural numbers:

```
  data _<_ : Nat → Nat → Type lzero where
    zero< : {n : Nat} → zero < suc n
    succ< : {n m : Nat} → n < m → (suc n) < (suc m)
```

We then define the `n`th finite ordinal as the subtype of `Nat` consisting of all numbers `m` strictly less than `n`:

```
  Fin : Nat → Type lzero
  Fin n = Σ Nat (λ m → m < n)
```

From these data, we can straightforwardly define a polynomial as follows

```
  ω : Poly lzero lzero
  ω = (Nat , Fin)
```

If we once again write out the polynomial endofunctor determined by these data

$$y \mapsto \sum_{n \in \mathbb{N}} y^{\{m \in \mathbb{N} | m < n\}}$$

we see that this functor has a familiar form – it is the *list monad* that maps a type y to the disjoint union of the types of n-tuples of elements of y, for all $n \in \mathbb{N}$.

As defined, ω is not a polynomial universe; the type `Nat` is a set, and so for any `n : Nat`, the type `n ≡ n` is contractible, i.e. it has a single inhabitant, while the type of equivalences `Fin n ≃ Fin n` consists of all permutations of `n` elements, so these two types cannot be equivalent. However, we can now use the Rezk completion to obtain a polynomial universe from ω.

```
  Fin : Poly (lsuc lzero) lzero
  Fin = Rezk ω
```

If we write out an explicit description of `Fin`, we see that it is the subuniverse of types `X` that are merely equivalent to some `Fin n`. In constructive mathematics, these types (they are necessarily sets) are known as *Bishop finite sets*. Hence the polynomial universe obtained by Rezk completion of the list monad is precisely the subuniverse of types spanned by (Bishop) finite sets.

5 Π-Types & 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 $⇈[\_] : \mathsf{Tw}(\mathbf{Poly}) \times \mathbf{Poly} \to \mathbf{Poly}$ 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 *distributive laws* in **Poly**.

5.1 The ⇈ and ⇈[_][_] Functors

The ⇈ functor can be loosely defined as the solution to the following problem: given a polynomial universe u, find $u \mathbin{⇈} u$ such that u classifies $u \mathbin{⇈} u$ 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 $p ⇈ q$:

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

```
_⇈_ : ∀ {ℓ0 ℓ1 κ0 κ1} → Poly ℓ0 κ0 → Poly ℓ1 κ1
      → Poly (ℓ0 ⊔ κ0 ⊔ ℓ1) (κ0 ⊔ κ1)
(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. Functoriality of the 1st argument is trickier, however. For reasons that will become apparent momentarily, we define the functorial action $p ⇈ q \leftrightarrows p' ⇈ q$ of ⇈ on a lens $f : p \leftrightarrows p'$ equipped with a left inverse $f' : p' \leftrightarrows p$, i.e. such that $f' \circ f = \mathrm{id}_p$. [6]

```
⇈Lens : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
        → {p : Poly ℓ0 κ0} (r : Poly ℓ2 κ2)
        → {q : Poly ℓ1 κ1} (s : Poly ℓ3 κ3)
        → (f : p ⇆ r) (f' : r ⇆ p)
        → EqLens p (id p) (comp p f f')
        → (g : q ⇆ s) → (p ⇈ q) ⇆ (r ⇈ s)
⇈Lens {p = p} r s (f , f♯) (f' , f'♯) (e , e♯) (g , g♯) =
    ( (λ (a , γ) → (f a , (λ x → g (γ (f♯ a x)))))
    , (λ (a , γ) F x →
        g♯ (γ x)
           (transp (λ y → snd s (g (γ y)))
```

[6] To see why this is the right choice of morphism for which '⇈' is functorial in its first argument, we note that pairs consisting of a morphism and a left inverse for it are equivalently the morphisms between identity morphisms in the *twisted arrow category* of **Poly**, i.e. diagrams of the following form:

$$\begin{array}{ccc} p & \to & q \\ = & & = \\ p & \leftarrow & q \end{array}$$

```
(sym (e♯ a x))
(F (f'♯ (f a) (transp (snd p) (e a) x))))) )
```

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" `π♯ (A , B)`, for any pair `(A , B)` as above, maps an element `f : π(A,B)` to a function `π♯ (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 `π♯ (A , B)` is an equivalence, it follows that there is an inverse `π♯⁻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 `π♯⁻¹ (A , B)` is both a left and a right inverse to `π♯` then corresponds to the β and η laws for Π types.
$$(\lambda x.f[x])\ a = f[a] \qquad f = \lambda x.f\ x$$

Although it is clear enough that the $\Uparrow$ functor serves its intended purpose of characterizing Π types in polynomial universes, its construction seems somewhat more ad hoc than that of $\triangleleft$, which similarly characterized Σ types in polynomial universes while arising quite naturally from composition of polynomial functors. We would like to better understand what additional properties $\Uparrow$ must satisfy, and how these in turn are reflected as properties of polynomial universes with Π types. In fact, we will ultimately show that this construction is intimately linked with a quite simple structure on polynomial universes u, namely a *distributive law* of u (viewed as a monad) over itself. Before that, however, we note some other key properties of $\Uparrow$.

Specifically, let $\mathbf{Poly}_R$ be the category whose objects are polynomials and whose morphisms are lenses equipped with left inverses. Straightforwardly, $\triangleleft$ restricts to a monoidal product on $\mathbf{Poly}_R$, since it is functorial in both arguments and must preserve left/right inverses. Hence $\Uparrow$ can be viewed as a functor $\mathbf{Poly}_R \times \mathbf{Poly} \to \mathbf{Poly}$. Then $\Uparrow$ moreover naturally carries the structure of an *action* on $\mathbf{Poly}$ of the monoidal category $\mathbf{Poly}_R$ equipped with $\triangleleft$, in that there are natural transformations
$$y \Uparrow p \to p \quad \text{and} \quad (p \triangleleft q) \Uparrow r \to p \Uparrow (q \Uparrow r)$$
which are moreover *Cartesian*:

```
module Unit⇈ {ℓ κ} (p : Poly ℓ κ) where

    y⇈ : (y ⇈ p) ⇆ p
    y⇈ = ( (λ (_ , a) → a tt) , λ (_ , a) b tt → b)

    y⇈Cart : isCartesian p y⇈
    y⇈Cart (_ , x) =
        Iso→isEquiv ( (λ F → F tt)
                    , ( (λ a → refl)
                      , λ b → refl))

open Unit⇈ public

module ◃⇈ {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} (p : Poly ℓ0 κ0)
          (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2) where

    ⇈Curry : ((p ◃ q) ⇈ r) ⇆ (p ⇈ (q ⇈ r))
    ⇈Curry = ( (λ ((a , h) , k)
                  → (a , (λ b → ( (h b)
                                , (λ d → k (b , d))))))
             , (λ ((a , h) , k) f (b , d) → f b d) )

    ⇈CurryCart : isCartesian (p ⇈ (q ⇈ r)) ⇈Curry
    ⇈CurryCart ((a , h) , k) =
        Iso→isEquiv ( (λ f b d → f (b , d))
                    , ( (λ f → refl)
                      , (λ f → refl) ) )

open ◃⇈ public
```

The fact that ⇈Curry is Cartesian corresponds to the usual currying isomorphism that relating dependent functions types to dependent pair types:

$$\Pi(x, y) : \Sigma x : A.B[x].C[x, y] \simeq \Pi x : A.\Pi y : B[x].C[x, y]$$

Similarly, $\Uparrow$ is colax with respect to $\triangleleft$ in its second argument, in that there are Cartesian natural transformations

$$p \Uparrow y \to y \quad \text{and} \quad p \Uparrow (q \triangleleft r) \to (p \Uparrow q) \triangleleft (p \Uparrow r)$$

```
module ⇈Unit {ℓ κ} (p : Poly ℓ κ) where

    ⇈y : (p ⇈ y) ⇆ y
    ⇈y = ( (λ (a , γ) → tt) , λ (a , γ) tt b → tt )

    ⇈yCart : isCartesian y ⇈y
```

```
    ⇈yCart (x , γ) =
        Iso→isEquiv ( (λ x → tt)
                    , ( (λ a → refl)
                      , λ b → refl))

open ⇈Unit public

module ⇈◃ {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} (p : Poly ℓ0 κ0)
          (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2) where

    ⇈Distr : (p ⇈ (q ◃ r)) ⇆ ((p ⇈ q) ◃ (p ⇈ r))
    ⇈Distr = ( (λ (a , h)
                  → ( (a , (λ b → fst (h b)))
                    , λ f → (a , (λ b → snd (h b) (f b))) ))
             , (λ (a , h) (f , g) b → (f b , g b)) )

    ⇈DistrCart : isCartesian ((p ⇈ q) ◃ (p ⇈ r)) ⇈Distr
    ⇈DistrCart (a , h) =
        Iso→isEquiv ( (λ f → ( (λ b → fst (f b))
                             , (λ b → snd (f b)) ))
                    , ( (λ (f , g) → refl)
                      , (λ f → refl) ) )

open ⇈◃ public
```

The fact that ⇈Distr is Cartesian corresponds to the distributive law of Π types over Σ types, i.e.

$$\Pi x : A.\Sigma y : B[x].C[x, y] \simeq \Sigma f : \Pi x : A.B[x].\Pi x : A.C[x, f(x)]$$

One may wonder, then, whether this distributive law is somehow related to a distributive law ofg the monad structure on a polynomial universe u given by Σ types (as discussed in the previous section) over itself, i.e. a morphism

$$\mathfrak{u} \triangleleft \mathfrak{u} \leftrightarrows \mathfrak{u} \triangleleft \mathfrak{u}$$

subject to certain laws. Indeed, given a Lens π : $(u \Uparrow u) \leftrightarrows u$ (intuitively – corresponding to the structure of Π types in u), one can define a morphism of this form as follows:

```
distrLaw? : ∀ {ℓ κ} (u : Poly ℓ κ) → (u ⇈ u) ⇆ u
          → (u ◃ u) ⇆ (u ◃ u)
distrLaw? u (π , π♯) =
    ( (λ (a , b) → π (a , b) , (λ x → a))
    , λ (a , b) (f , x) → (x , (π♯ ((a , b)) f x)) )
```

The question then becomes whether this morphism has the structure of a distributive law when u has the structure of a polynomial universe with Σ types, and π is Cartesian (i.e. u also has Π types). Answering this question in the affirmative shall be our task in the remainder of this section.

As a first step in this direction, we make a perhaps unexpected move of further generalizing the $\Uparrow$ functor to a functor $\mathsf{Tw}(\mathbf{Poly}) \times \mathbf{Poly} \to \mathbf{Poly}$, where $\mathsf{Tw}(\mathbf{Poly})$ is the *twisted arrow category* of **Poly**, i.e. the category whose objects are lenses and whose morphisms are *twisted* commuting squares of the form

$$\begin{array}{ccc} p & \to & p' \\ \downarrow & & \downarrow \\ q & \leftarrow & q' \end{array}$$

```
_⇑⇑[_][_]_ : ∀ {ℓ ℓ' ℓ'' κ κ' κ''}
           → (p : Poly ℓ κ) (q : Poly ℓ' κ')
           → (p ⇆ q) → (r : Poly ℓ'' κ'')
           → Poly (ℓ ⊔ κ ⊔ ℓ'') (κ' ⊔ κ'')
(A , B) ⇑⇑[ (C , D) ][ (f , f♯) ] (E , F) =
  ( (Σ A (λ a → B a → E))
  , (λ (a , ϵ) → (d : D (f a)) → F (ϵ (f♯ a d))))

module ⇑⇑[]Functor {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 ℓ5 κ0 κ1 κ2 κ3 κ4 κ5}
         {p : Poly ℓ0 κ0} {p' : Poly ℓ3 κ3}
         (q : Poly ℓ1 κ1) {q' : Poly ℓ4 κ4}
         {r : Poly ℓ2 κ2} (r' : Poly ℓ5 κ5)
         (f : p ⇆ q) (f' : p' ⇆ q')
         (g : p ⇆ p') (h : q' ⇆ q) (k : r ⇆ r')
         (e : EqLens q f (comp q g (comp q f' h))) where

  ⇑⇑[]Lens : (p ⇑⇑[ q ][ f ] r) ⇆ (p' ⇑⇑[ q' ][ f' ] r')
  ⇑⇑[]Lens =
    ( (λ (a , γ) → (fst g a , λ x → fst k (γ (snd g a x))))
    , λ (a , γ) F x →
        snd k (γ (snd f a x))
          (transp (λ y → snd r' (fst k (γ y)))
                  (sym (snd e a x))
                  (F (snd h (fst f' (fst g a))
                            (transp (snd q) (fst e a) x)))) )
```

Straightforwardly, we have that `p ⇑⇑ q = p ⇑⇑[ p ][ id p ] q`. In particular, we have `⇑⇑Lens p p' q q' f f' e g = ⇑⇑[]Lens p p' p p' q q' (id p) (id p') f f' g e`, which serves to motivate the definition of `⇑⇑Lens` in terms of morphisms equipped with left inverses.

The functor `_⇑⇑[_][_]_` defined above moreover preserves Cartesian morphisms in all of its arguments, and so restricts to a functor $\mathsf{Tw}(\mathbf{Poly}^{\mathbf{Cart}}) \times \mathbf{Poly}^{\mathbf{Cart}} \to \mathbf{Poly}^{\mathbf{Cart}}$.

```
    ⇈[]LensCart : isCartesian q h → isCartesian r' k
                  → isCartesian (p' ⇈[ q' ][ f' ] r') ⇈[]Lens
    ⇈[]LensCart ch ck (a , γ) =
        compIsEquiv
            (PostCompEquiv (λ x → snd k (γ (snd f a x)))
                           (λ x → ck (γ (snd f a x))))
            (compIsEquiv
                (PostCompEquiv
                    (λ x → transp (λ y → snd r' (fst k (γ y)))
                                  (sym (snd e a x)))
                    (λ x → transpIsEquiv (sym (snd e a x))))
                (compIsEquiv
                    (PreCompEquiv (transp (snd q) (fst e a))
                                  (transpIsEquiv (fst e a)))
                    (PreCompEquiv (λ x → snd h (fst f' (fst g a)) x)
                                  (ch (fst f' (fst g a))))))

open ⇈[]Functor public
```

Moreover, all the properties of _⇈_ noted above generalize to _⇈[_][_]_. For instance, we now have natural transformations

$$y⇈[y][\mathrm{id}_y]p \to p \quad \text{and} \quad (p \triangleleft r)⇈[q \triangleleft s][f \triangleleft g]t \to p⇈[q][f](r⇈[s][g]t)$$

as follows:

```
y⇈[] : ∀ {ℓ κ} (p : Poly ℓ κ) → (y ⇈[ y ][ (id y) ] p) ⇆ p
y⇈[] p = ((λ (_ , γ) → γ tt) , λ (_ , γ) F _ → F)

⇈[]Curry : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 κ0 κ1 κ2 κ3 κ4}
           → (p : Poly ℓ0 κ0) (q : Poly ℓ1 κ1)
           → (r : Poly ℓ2 κ2) (s : Poly ℓ3 κ3) (t : Poly ℓ4 κ4)
           → (f : p ⇆ q) (g : r ⇆ s)
           → ((p ◃ r) ⇈[ q ◃ s ][ f ◃◃[ s ] g ] t)
             ⇆ (p ⇈[ q ][ f ] (r ⇈[ s ][ g ] t))
⇈[]Curry p q r s t f g =
    ( (λ ((a , h) , k) → a , (λ b → (h b) , (λ d → k (b , d))))
    , λ ((a , h) , k) F (b , d) → F b d)
```

And similarly, we have natural transformations

$$p⇈[q][f]y \to y \quad \text{and} \quad p⇈[r][g \circ f](s \triangleleft t) \to (p⇈[q][f]s) \triangleleft (q⇈[r][g]t)$$

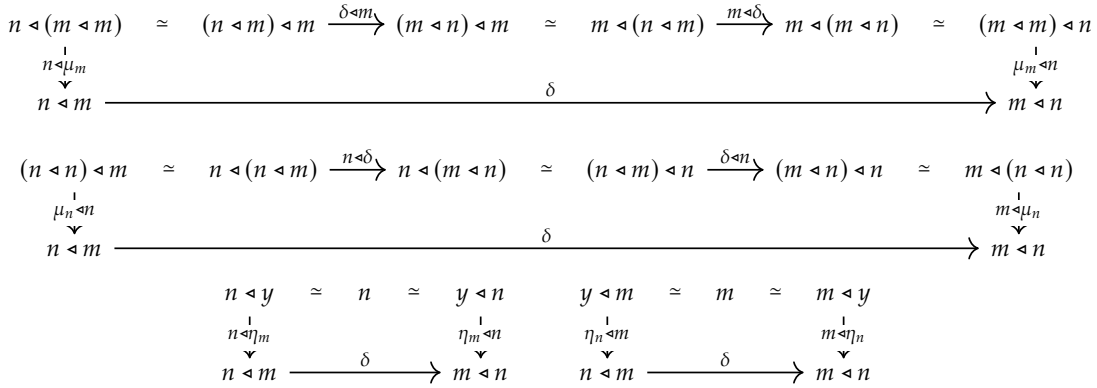
```
⇈[]y : ∀ {ℓ0 κ0 ℓ1 κ1} (p : Poly ℓ0 κ0) (q : Poly ℓ1 κ1)
       → (f : p ⇆ q) → (p ⇈[ q ][ f ] y) ⇆ y
⇈[]y p q f = ((λ _ → tt) , λ _ _ _ → tt)
```

```
⇈[]Distr : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 κ0 κ1 κ2 κ3 κ4}
           → (p : Poly ℓ0 κ0) (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2)
           → (s : Poly ℓ3 κ3) (t : Poly ℓ4 κ4)
           → (f : p ⇆ q) (g : q ⇆ r)
           → (p ⇈[ r ][ comp r f g ] (s ◃ t))
             ⇆ ((p ⇈[ q ][ f ] s) ◃ (q ⇈[ r ][ g ] t))
⇈[]Distr p q r s t (f , f♯) (g , g♯) =
    ( (λ (a , h) → ( (a , (λ x → fst (h x)))
                   , (λ k1 → ( f a , λ x → snd (h (f♯ a x))
                                               (k1 x) )) ))
    , (λ (a , h) (k1 , k2) d → ( (k1 (g♯ (f a) d)) , k2 d )) )
```

As we shall now see, these structures on `_⇈[_][_]_` are intimately connected to a class of morphisms in **Poly**, which we call *distributors*.

5.2 Distributors

Given polynomials `p,q,r,s`, a *distributor* of `p,q` over `r,s` is a morphism of the form `(p ◃ r) ⇆ (s ◃ q)` in **Poly**. The name "distributor" is here drawn from the fact that, given polynomial monads `m,n` with η_m `: y ⇆ m`, η_m `: y ⇆ n` and μ_m `: (m ◃ m) ⇆ m`, μ_m `: (n ◃ n) ⇆ n`, a *distributive law* of `m` over `n` consists of a distributor of `n,n` over `m,m` (i.e. a morphism δ `: (n ◃ m) ⇆ (m ◃ n)`) such that the following diagrams commute:



By inspection, it can be seen that all the composite morphisms required to commute by the above diagrams are themselves distributors of various forms. Understanding the closure properties of such distributors that give rise to these diagrams, then, will be a central aim of this section.

By function extensionality, we obtain the following type of equality proofs for distributors:

```
EqDistributor : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
                → (p : Poly ℓ0 κ0) (q : Poly ℓ1 κ1)
```

```
                → (r : Poly ℓ2 κ2) (s : Poly ℓ3 κ3)
                → (p ◃ r) ⇆ (s ◃ q) → (p ◃ r) ⇆ (s ◃ q)
                → Type (ℓ0 ⊔ ℓ1 ⊔ ℓ2 ⊔ ℓ3 ⊔ κ0 ⊔ κ1 ⊔ κ2 ⊔ κ3)
EqDistributor p q r s (f , f♯) (g , g♯) =
    (a : fst p) (γ : snd p a → fst r)
    → Σ (fst (f (a , γ)) ≡ fst (g (a , γ)))
        (λ e1 → (x : snd s (fst (f (a , γ))))
                → Σ ((snd (f (a , γ)) x)
                    ≡ (snd (g (a , γ))
                           (transp (snd s) e1 x)))
                    (λ e2 → (y : snd q (snd (f (a , γ)) x))
                            → (f♯ (a , γ) (x , y))
                              ≡ (g♯ (a , γ)
                                    ( (transp (snd s) e1 x)
                                    , (transp (snd q) e2 y)))))
```

Moreover, for any polynomial `u` with `π : (u ↑↑ u) ⇆ u`, the morphism `distrLaw? u π` defined above is a distributor of `u,u` over itself. In fact, we can straightforwardly generalize the construction of `distrLaw?` to a transformation

$$(p \uparrow\uparrow [q][f] r) \leftrightarrows s \implies (p \triangleleft r) \leftrightarrows (s \triangleleft q)$$

as follows:

```
↑↑→Distributor : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
                → {p : Poly ℓ0 κ0} (q : Poly ℓ1 κ1)
                → (r : Poly ℓ2 κ2) {s : Poly ℓ3 κ3}
                → {f : p ⇆ q}
                → (p ↑↑[ q ][ f ] r) ⇆ s
                → (p ◃ r) ⇆ (s ◃ q)
↑↑→Distributor q r {f = (f , f♯)} (g , g♯) =
    ( (λ (a , h) → g (a , h) , λ d' → f a)
    , λ (a , h) (d' , d)
        → f♯ a d , g♯ (a , h) d' d)
```

Hence to show that the above-given diagrams commute for the candidate distributive law `distrLaw? u π` given above, it suffices to show that the distributors required to commute by these diagrams themselves arise – under the above-defined transformation – from Cartesian morphisms of the form `p ↑↑[ q ][ f ] r ⇆ u`, which, if `u` is a polynomial universe, are necessarily equal.

First of all, any distributor $(p \triangleleft r) \leftrightarrows (s \triangleleft q)$ may be extended along morphisms `p' ⇆ p`, `q ⇆ q'`, `r' ⇆ r`, `s ⇆ s'` to a distributor $(p' \triangleleft r') \leftrightarrows (s' \triangleleft q')$ by forming the composite

$$p' \triangleleft r' \to p \triangleleft r \to s \triangleleft q \to s' \triangleleft q'$$

```
module DistributorLens {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 ℓ5 ℓ6 ℓ7
                        κ0 κ1 κ2 κ3 κ4 κ5 κ6 κ7}
```

```
                    {p : Poly ℓ0 κ0} {p' : Poly ℓ4 κ4}
                    {q : Poly ℓ1 κ1} (q' : Poly ℓ5 κ5)
                    (r : Poly ℓ2 κ2) {r' : Poly ℓ6 κ6}
                    {s : Poly ℓ3 κ3} (s' : Poly ℓ7 κ7)
                    (g : p' ⇆ p) (h : q ⇆ q')
                    (k : r' ⇆ r) (l : s ⇆ s') where

  distrLens : (p ◃ r) ⇆ (s ◃ q) → (p' ◃ r') ⇆ (s' ◃ q')
  distrLens j =
      comp (s' ◃ q') (g ◃◃[ r ] k)
           (comp ((s' ◃ q')) j
                 (l ◃◃[ q' ] h))
```

The corresponding construction on morphisms out of _⇈[_][_]_ is given by forming the composite

$$p' \Uparrow[q'][h \circ f \circ g]\, r' \leftrightarrows p \Uparrow[q][f]\, r \leftrightarrows s \leftrightarrows s'$$

```
  ⇈→DistributorLens :
      {f : p ⇆ q} → (p ⇈[ q ][ f ] r) ⇆ s
      → (p' ⇈[ q' ][ comp q' g (comp q' f h) ] r') ⇆ s'
  ⇈→DistributorLens {f = f} j =
      comp s' (⇈[]Lens q' r (comp q' g (comp q' f h)) f
                       g h k ((λ a → refl) , (λ a d → refl)))
           (comp s' j l)

  ⇈→DistributorLens≡ : {f : p ⇆ q} (j : (p ⇈[ q ][ f ] r) ⇆ s)
                     → distrLens (⇈→Distributor q r j)
                       ≡ ⇈→Distributor q' r' (⇈→DistributorLens j)
  ⇈→DistributorLens≡ j = refl

open DistributorLens public
```

Similarly, there are two distinct ways of composing distributors:

1. Given distributors `p ◃ s ⇆ t ◃ q` and `q ◃ u ⇆ v ◃ r$`, we obtain a distributorp ◃ (s ◃ u) ⇆ (t ◃ v) ◃ r′ as the composite

$$p \triangleleft (s \triangleleft u) \simeq (p \triangleleft s) \triangleleft u \longrightarrow (t \triangleleft q) \triangleleft u \simeq t \triangleleft (q \triangleleft u) \longrightarrow t \triangleleft (v \triangleleft r) \simeq (t \triangleleft v) \triangleleft r$$

```
module DistributorComp1 {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 ℓ5 ℓ6 κ0 κ1 κ2 κ3 κ4 κ5 κ6}
                        {p : Poly ℓ0 κ0} {q : Poly ℓ1 κ1} (r : Poly ℓ2 κ2)
                        {s : Poly ℓ3 κ3} {t : Poly ℓ4 κ4}
                        (u : Poly ℓ5 κ5) {v : Poly ℓ6 κ6} where

  distrComp1 : (p ◃ s) ⇆ (t ◃ q) → (q ◃ u) ⇆ (v ◃ r)
               → (p ◃ (s ◃ u)) ⇆ ((t ◃ v) ◃ r)
  distrComp1 h k =
```

```
        comp ((t ◃ v) ◃ r) (◃assoc⁻¹ p s u)
             (comp ((t ◃ v) ◃ r) (h ◃◃[ u ] (id u))
                   (comp ((t ◃ v) ◃ r) (◃assoc t q u)
                         (comp ((t ◃ v) ◃ r) ((id t) ◃◃[ (v ◃ r) ] k)
                               (◃assoc⁻¹ t v r))))
```

The corresponding construction on morphisms `(p ↑↑[ q ][ f ] s) ⇆ t` and `(q ↑↑[ r ][ g ] u) ⇆ v` is to form the following composite with the colaxator of `_↑↑[_][_]_`:

$$p \uparrow\uparrow [r][g \circ f](s \triangleleft u) \leftrightarrows (p \uparrow\uparrow [q][f]s) \triangleleft (q \uparrow\uparrow [r][g]u) \leftrightarrows t \triangleleft v$$

```
    ↑↑→DistributorComp1 :
        {f : p ⇆ q} {g : q ⇆ r}
        → (p ↑↑[ q ][ f ] s) ⇆ t
        → (q ↑↑[ r ][ g ] u) ⇆ v
        → (p ↑↑[ r ][ comp r f g ] (s ◃ u)) ⇆ (t ◃ v)
    ↑↑→DistributorComp1 {f = f} {g = g} h k =
        comp (t ◃ v) (↑↑[]Distr p q r s u f g)
             (h ◃◃[ v ] k)

    ↑↑→DistributorComp1≡ :
        {f : p ⇆ q} {g : q ⇆ r}
        (h : (p ↑↑[ q ][ f ] s) ⇆ t)
        (k : (q ↑↑[ r ][ g ] u) ⇆ v)
        → distrComp1 (↑↑→Distributor q s h) (↑↑→Distributor r u k)
          ≡ ↑↑→Distributor r (s ◃ u) (↑↑→DistributorComp1 h k)
    ↑↑→DistributorComp1≡ h k = refl

open DistributorComp1 public
```

2. Given distributors $p \triangleleft u \leftrightarrows v \triangleleft q$ and $r \triangleleft t \leftrightarrows u \triangleleft s$, we obtain a distributor $(p \triangleleft r) \triangleleft t \leftrightarrows v \triangleleft (q \triangleleft s)$ as the composite

$$(p \triangleleft r) \triangleleft t \simeq p \triangleleft (r \triangleleft t) \leftrightarrows p \triangleleft (u \triangleleft s) \simeq (p \triangleleft u) \triangleleft s \leftrightarrows (v \triangleleft q) \triangleleft s \simeq v \triangleleft (q \triangleleft s)$$

```
module DistributorComp2
           {ℓ0 ℓ1 ℓ2 ℓ3 ℓ4 ℓ5 ℓ6 κ0 κ1 κ2 κ3 κ4 κ5 κ6}
           {p : Poly ℓ0 κ0} {q : Poly ℓ1 κ1}
           {r : Poly ℓ2 κ2} (s : Poly ℓ3 κ3)
           (t : Poly ℓ4 κ4) {u : Poly ℓ5 κ5}
           {v : Poly ℓ6 κ6} where

    distrComp2 : (r ◃ t) ⇆ (u ◃ s) → (p ◃ u) ⇆ (v ◃ q)
                 → ((p ◃ r) ◃ t) ⇆ (v ◃ (q ◃ s))
    distrComp2 h k =
```

```
    comp (v ◃ (q ◃ s)) (◃assoc p r t)
         (comp (v ◃ (q ◃ s))  ((id p) ◃◃[ u ◃ s ] h)
           (comp (v ◃ (q ◃ s)) (◃assoc⁻¹ p u s)
                 (comp (v ◃ (q ◃ s)) (k ◃◃[ s ] (id s))
                       (◃assoc v q s))))
```

The corresponding construction on morphisms `(p ⇈[ q ][ f ] u) ⇆ v` and `(r ⇈[ s ][ g ] t) ⇆ u` is to form the following composite with the morphism `⇈[]Curry` defined above:

$$(p \triangleleft r) \upuparrows [q \triangleleft s][f \triangleleft g]t \leftrightarrows p \upuparrows [q][f](r \upuparrows [s][g]t) \leftrightarrows p \upuparrows [q][f]u \leftrightarrows v$$

```
    ⇈→DistributorComp2 :
        {f : p ⇆ q} {g : r ⇆ s}
        → (r ⇈[ s ][ g ] t) ⇆ u
        → (p ⇈[ q ][ f ] u) ⇆ v
        → ((p ◃ r) ⇈[ (q ◃ s) ][ f ◃◃[ s ] g ] t) ⇆ v
    ⇈→DistributorComp2 {f = f} {g = g} h k =
        comp v (⇈[]Curry p q r s t f g)
             (comp v (⇈[]Lens q u f f
                              (id p) (id q) h
                              ( (λ a → refl)
                              , (λ a d → refl)))
                  k)

    ⇈→DistributorComp2≡ :
        {f : p ⇆ q} {g : r ⇆ s}
        → (h : (r ⇈[ s ][ g ] t) ⇆ u)
        → (k : (p ⇈[ q ][ f ] u) ⇆ v)
        → (distrComp2 (⇈→Distributor s t h)
                      (⇈→Distributor q u k))
          ≡ ⇈→Distributor (q ◃ s) t
                          (⇈→DistributorComp2 h k)
    ⇈→DistributorComp2≡ h k = refl
```

```
open DistributorComp2 public
```

Likewise, there are two corresponding notions of "identity distributor" on a polynomial `p`, the first of which is given by the following composition of unitors for `◃`:

$$p \triangleleft y \simeq p \simeq y \triangleleft p$$

and the second of which is given by the inverse such composition

$$y \triangleleft p \simeq p \simeq p \triangleleft y$$

```
module DistributorId {ℓ κ} (p : Poly ℓ κ) where

    distrId1 : (p ◃ y) ⇆ (y ◃ p)
    distrId1 = comp (y ◃ p) (◃unitr p) (◃unitl⁻¹ p)

    distrId2 : (y ◃ p) ⇆ (p ◃ y)
    distrId2 = comp (p ◃ y) (◃unitl p) (◃unitr⁻¹ p)
```

The corresponding morphisms `p ↑↑[ p ][ id p ] y ⇆ y` and `y ↑↑[ y ][ id y ] p ⇆ p` are precisely the maps `↑↑[]y` and `y↑↑[]` defined above, respectively:

```
    ↑↑→DistributorId1≡ : distrId1 ≡ ↑↑→Distributor p y (↑↑[]y p p (id p))
    ↑↑→DistributorId1≡ = refl

    ↑↑→DistributorId2≡ : distrId2 ≡ ↑↑→Distributor y p (y↑↑[] p)
    ↑↑→DistributorId2≡ = refl

open DistributorId public
```

It can thus be seen that the above operations defined on distributors are preicsely those occurring in the diagrams for a distributive law given above, and moreover, these all have corresponding constructions on morphisms out of `_↑↑[_][_]_`, all of which preserve Cartesian morphisms. Hence if π : $u \uparrow\uparrow u \leftrightarrows u$ is Cartesian, all of the morphisms involving `_↑↑[_][_]_` corresponding to those required to commute in order for `distrLaw? u π` to be a distributive law will be Cartesian, and so if u is a polynomial universe, these will all automatically be equal to one another.

```
ap↑↑→Distributor : ∀ {ℓ0 ℓ1 ℓ2 ℓ3 κ0 κ1 κ2 κ3}
                  → (p : Poly ℓ0 κ0) (q : Poly ℓ1 κ1)
                  → (r : Poly ℓ2 κ2) (s : Poly ℓ3 κ3) (f : p ⇆ q)
                  → (h k : (p ↑↑[ q ][ f ] r) ⇆ s) → EqLens s h k
                  → EqDistributor p q r s (↑↑→Distributor q r h)
                                          (↑↑→Distributor q r k)
ap↑↑→Distributor p q r s f h k (e , e♯) a γ =
    ( e (a , γ)
    , λ x → ( refl , (λ y → pairEq refl (coAp (e♯ (a , γ) x) y)) ) )

module DistrLaw {ℓ κ} (u : Poly ℓ κ) (univ : isUnivalent u)
                (η : y ⇆ u) (cη : isCartesian u η)
                (σ : (u ◃ u) ⇆ u) (cσ : isCartesian u σ)
                (π : (u ↑↑ u) ⇆ u) (cπ : isCartesian u π) where

    distrLaw1 : EqDistributor u u (u ◃ u) u
                    (distrLens u (u ◃ u) u (id u) (id u) (id (u ◃ u)) σ
                              (distrComp1 u u (distrLaw? u π)
```

```
                                          (distrLaw? u π)))
                (distrLens u u u (id u) (id u) σ (id u)
                           (distrLaw? u π))
distrLaw1 =
    ap↑↑→Distributor u u (u ◃ u) u (id u)
        (comp u (comp (u ◃ u) (↑↑Distr u u u) (π ◃◃[ u ] π)) σ)
        (comp u (↑↑[]Lens u u (id u) (id u) (id u) (id u) σ
                          ((λ a → refl) , (λ a d → refl)))
              π)
        (univ (compCartesian u
                    (compCartesian (u ◃ u)
                        (↑↑DistrCart u u u)
                        (◃◃Cart u u cπ cπ))
                    cσ)
              (compCartesian u
                (↑↑[]LensCart u u (id u) (id u) (id u) (id u) σ
                    ((λ a → refl) , (λ a d → refl))
                    (idCart u) cσ)
                cπ))

distrLaw2 : EqDistributor (u ◃ u) u u u
                (distrLens u u u (id (u ◃ u)) σ (id u) (id u)
                           (distrComp2 u u (distrLaw? u π)
                                           (distrLaw? u π)))
                (distrLens u u u σ (id u) (id u) (id u)
                           (distrLaw? u π))
distrLaw2 =
    ap↑↑→Distributor (u ◃ u) u u u σ
        (comp u
            (comp (u ↑↑ u)
                (comp (u ↑↑ (u ↑↑ u))
                    (↑↑[]Lens u u σ (id (u ◃ u))
                        (id (u ◃ u)) σ (id u)
                        ((λ a → refl) , (λ a d → refl)))
                    (↑↑Curry u u u))
                (↑↑Lens u u (id u) (id u)
                       ((λ a → refl) , (λ a d → refl))
                       π))
            π)
        (comp u (↑↑[]Lens u u σ (id u) σ (id u) (id u)
                          ((λ a → refl) , (λ a d → refl)))
              π)
        (univ (compCartesian u
                (compCartesian (u ↑↑ u)
```

```
                    (compCartesian (u ↑↑ (u ↑↑ u))
                        (↑↑[]LensCart u u σ (id (u ◃ u))
                            (id (u ◃ u)) σ (id u)
                            ((λ a → refl) , (λ a d → refl))
                            cσ (idCart u))
                        (↑↑CurryCart u u u))
                    (↑↑[]LensCart u u (id u) (id u) (id u) (id u) π
                                ((λ a → refl) , (λ a d → refl))
                                (idCart u) cπ))
                cπ)
              (compCartesian u
                (↑↑[]LensCart u u σ (id u) σ (id u) (id u)
                    ((λ a → refl) , (λ a d → refl))
                    (idCart u) (idCart u))
                cπ))

distrLaw3 : EqDistributor u u y u
              (distrLens u y u (id u) (id u) (id y) η
                         (distrId1 u))
              (distrLens u u u (id u) (id u) η (id u)
                         (distrLaw? u π))
distrLaw3 =
    ap↑↑→Distributor u u y u (id u)
        (comp u (↑↑y u) η)
        (comp u (↑↑Lens u u (id u) (id u)
                        ((λ a → refl) , (λ a d → refl)) η) π)
        (univ (compCartesian u (↑↑yCart u) cη)
              (compCartesian u
                (↑↑[]LensCart u u (id u) (id u) (id u) (id u) η
                             ((λ a → refl) , (λ a d → refl))
                             (idCart u) cη)
                cπ))

distrLaw4 : EqDistributor y u u u
              (distrLens u u u (id y) η (id u) (id u)
                         (distrId2 u))
              (distrLens u u u η (id u) (id u) (id u)
                         (distrLaw? u π))
distrLaw4 =
    ap↑↑→Distributor y u u u η
        (comp u (↑↑[]Lens u u η (id y) (id y) η (id u)
                          ((λ a → refl) , (λ a d → refl)))
                (y↑↑ u))
        (comp u (↑↑[]Lens u u η (id u) η (id u) (id u)
```

```
                    ((λ a → refl) , (λ a d → refl)))
          π)
(univ (compCartesian u
          (⇈[]LensCart u u η (id y) (id y) η (id u)
                       ((λ a → refl) , (λ a d → refl))
                       cη (idCart u))
          (y⇈Cart u))
        (compCartesian u
          (⇈[]LensCart u u η (id u) η (id u) (id u)
                       ((λ a → refl) , (λ a d → refl))
                       (idCart u) (idCart u))
          cπ))
```

Hence `distrLaw? u π` is a distributive law, as desired (and moreover, all of the higher coherences of an ∞-distributive law could be demonstrated, following this same method.)

6 Further Structures on Polynomial Universes

In closing, we turn to briefly consider whether and how some additional type-theoretic constructs may be defined for natural models / polynomial universes in the language of polynomial functors, starting with the concept of a universe itself.

6.1 The Shift Operator & Universes

Throughout this paper, we have made extensive use of universes of types. A natural question to ask, then, is when the type theory presented by a polynomial universe itself contains another such universe as a type within itself.

For this purpose, let v , u be polynomial universes with v = (V , ElV) and u = (U , ElU). If there is a (necessarily unique) Cartesian morphism $v \leftrightarrows u$, then it follows that every type family classified by v is also classified by u, by composition of Cartesian morphisms. However, what we want in this case is the stronger property that v is somehow represented as a type within u.

For this purpose, we define the following *shift* operation that takes a polynomial `p = (A , B)` to the polynomial `shift p = (⊤ , λ _ → A)`n':

```
shift : ∀ {ℓ κ} → Poly ℓ κ → Poly lzero ℓ
shift (A , _) = (⊤ , λ _ → A)
```

By construction, then, if there is a Cartesian morphism `(v , v♯) : shift (V , El`V`) ⇆ (`U `, El`U`)`, it follows that:

- There is a type `v tt :` U; type theoretically, this corresponds to a type formation rule of the form

$$\frac{}{\Gamma \vdash \mathcal{V}\ \mathsf{Type}}$$

We think of V as a type whose elements are "codes" for other types.

- There is a function $\mathtt{v\sharp\ tt : El}U\ \mathtt{(v\ tt)} \to V$, corresponding to the rule

$$\frac{\Gamma \vdash e : V}{\Gamma \vdash \lceil e \rceil\ \mathsf{Type}}$$

  which decodes a code contained in V to its corresponding type.
- There is a function $\mathtt{v\sharp^{-1}\ tt} : V \to \mathtt{El}U\ \mathtt{(v\ tt)}$, corresponding to the rule

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad T \text{ is classifed by } \mathfrak{v}}{\Gamma \vdash \lfloor A \rfloor : \mathcal{V}}$$

  that assigns a code to each type classified by v (note that this restriction to types classified by v is necessary to avoid the paradoxes that would arise from having a type universe that contained itself.)
- Such that the following equations hold

$$\lceil \lfloor A \rfloor \rceil = A \qquad e = \lfloor \lceil e \rceil \rfloor$$

6.2 The $(-)^{=}$ Operator & Extensional Identity Types

Another key construct of dependent type theory which has figured prominently in the foregoing development of polynomial universes, but which we have not yet shown how to internalize in such universes, is the construction of *identity types*. To some extent, this choice has been deliberate, as the theory of identity types is arguably one of the most complex aspects of dependent type theory, as evidenced by the fact that research into this topic ultimately precipitated the development of homotopy type theory. For this very reason, however, an account of the semantics of dependent type theory without at least some indication of its application to the theory of identity types would be incomplete.

Readers familiar with dependent type theory may be aware that an initial complication posed by the theory of identity types is that these types come in two flavors: extensional and intensional. Extensional identity types reflect propositional equality (i.e. the existence of an inhabitant for the type $\mathtt{a} \equiv \mathtt{b}$) into judgmental equality (i.e. the metatheoretic proposition that $\mathtt{a = b}$) and additionally regard all such proofs of identity as themselves identical. It follows that these identity types carry none of the homotopical information afforded by the alternative – intensional identity types, which are the sort which we have so far used in this paper. However, when working within such a homotopical framework, wherein metatheoretic equality need not be a mere proposition, there exists the possibility of defining extensional identity types in a polynomial universe so as to enable the aforementioned reflection while still allowing proofs of identity to carry higher-dimensional data.

For this purpose, let $u = (U\ ,\ \mathtt{El})$ be a polynomial universe. We wish to establish under what conditions u would be closed under the formation of "identity types" for the types classified by it. Solving this problem in essentially the same manner as led to the definiiton of the $\uparrow\uparrow$ functor in the previous section yields the following construction,

that maps `p = (A , B)` to the polynomial `p⁼ = (Σ A (λ a → B a × B a) , λ (_ , (b1 , b2)) → b1 ≡ b2)`.

```
_⁼ : ∀ {ℓ κ} → Poly ℓ κ → Poly (ℓ ⊔ κ) κ
(A , B)⁼ = (Σ A (λ a → B a × B a) , λ (_ , (b1 , b2)) → b1 ≡ b2)
```

If there is a Cartesian morphism `(ϵ , ϵ♯) : u⁼ ⇆ u` then:

- For each type `A : U` with elements `a₀ , a₁ : El A`, there is a type `Eq(A, a₀, a₁) : U`. Type theoretically, this corresponds to the type formation rule

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma \vdash a_0 : A \qquad \Gamma \vdash a_1 : A}{\Gamma \vdash a_0 \equiv_A a_1\ \mathsf{Type}}$$

- For each `A : U` and `a₀ , a₁ : El A` as above, there is a function `El Eq(A, a₀, a₁) → a₀ ≡ a₁` corresponding to the *reflection rule*

$$\frac{\Gamma \vdash e : a_0 \equiv_A a_1}{\Gamma \vdash a_0 = a_1}$$

  that converts an inhabitant of the propositional equality into a proof of the corresponding judgmental equality.
- Likewise, there is a function `a₀ ≡ a₁ → El Eq(A, a₀, a₁)` corresponding to the *reflexivity rule*

$$\frac{\Gamma \vdash a_0 = a_1}{\Gamma \vdash \mathsf{refl} : a_0 \equiv_A a_1}$$

  that produces an inhabitant of the propositional equality given a proof of the corresponding judgmental equality.
- Such that the above two functions/rules are mutually inverse.

What is missing from the above description of extensional identity types is the following rule

$$\frac{\Gamma \vdash e_1 : a_0 \equiv_A a_1 \qquad \Gamma \vdash e_2 : a_0 \equiv_A a_1}{\Gamma \vdash e_1 = e_2}$$

which says that all inhabitants of the identity type are themselves identical (i.e. the identity type is a *mere proposition*). This rule would be validated if we additionally required u to have the property that, for all types `A : U`, the type `El A` is a set (hence for any `a b : El A`, the type `a ≡ b` is a mere proposition.) However, if we do not make this requirement, this opens the possibility of having a model of extensional type theory, modulo the above rule, wherein proofs of equality still carry homotopical information – a potentially new direction in research on the semantics of identity types.

6.2.1 A Note on Intensional Identity Types & Inductive Types

Attempting to account for *intensional* rather than *extensional* identity types in the language of polynomial functors is rather more complicated, however. As mentioned in Section 2, the inhabitants of intensional identity types are inductively generated from

the constructor `refl`, corresponding to reflexivity. The problem with such inductive generation of data from constructors – from the point of view taken in this paper – is that it characterizes types in terms of their introduction forms, rather than their elimination forms. In type theoretic jargon, we say that intensional identity types, and inductive types more generally are *positive,* whereas all of the types we have considered so far are *negative,* in that they are characterized by their elimination forms. The universal properties of such negative types are therefore *mapping-in* properties, which are naturally described in terms of presheaves, which we have taken as our intended model for the development of this paper. By contrast, however, the universal properties of positive types are given by *mapping-out* properties, which are rather described in terms of (the opposite category of) *co-presheaves.*

As an illustrative example, let us consider the rather simpler case of (binary) coproducts, which are naturally regarded as positive types characterized by the left and right injections `A → A + B` and `B → A + B`. one might think to define binary coproducts on a polynomial universe u in the following way:

The *product* of two polynomial functors $p = \sum_{a:A} y^{B[a]}$ and $q = \sum_{c:C} y^{D[c]}$ can be calculated as follows:

$$\begin{array}{rl} & \left(\sum_{a:A} y^{B[a]}\right) \times \left(\sum_{c:C} y^{D[c]}\right) \\ \simeq & \sum_{(a,c):A\times C} y^{B[a]} \times y^{D[c]} \\ \simeq & \sum_{(a,c):A\times C} y^{B[a]+D[c]} \end{array}$$

Hence one might think to define binary coproducts on a polynomial universe u = `(`U `, El)` by asking there to be a Cartesian morphism $u \times u \leftrightarrows u$, since this would mean that for every pair of types `(A , B)` : $U \times U$, there is a type `plus(A , B)` : U such that `El(plus(A , B))` $\simeq$ `El A + El B`.

However, from the perspective of natural models, this condition is too strong. Given a category of contexts $\mathcal{C}$, the category $\mathbf{Set}^{\mathcal{C}^{op}}$ of presheaves on $\mathcal{C}$ is the free cocompletion of $\mathcal{C}$, which means that requiring $\mathcal{C}$ to be closed under taking binary coproducts of representables in $\mathbf{Set}^{\mathcal{C}^{op}}$ means not only that $\mathcal{C}$ has all binary coproducts, but that in fact all such coproducts in $\mathcal{C}$ are *free.*

Hence it remains to be seen if there can be found a general way of correctly expressing such "positive" type-theoretic concepts as for polynomial universes and natural models in the language of polynomial functors. We hope to continue investigations into these and related questions in future work.

6.3 Conclusion

In this paper, we have advanced a simplified and unified account of the categorical semantics of dependent type theory by expressing the core concepts of natural models entirely within the framework of polynomial functors in HoTT. By utilizing HoTT, we have been able strike an ideal balance between issues of strictness and higher-dimensional coherence that have bedeviled previous accounts. This shift not only streamlines the presentation of the semantics of dependent type theory, but also reveals additional structures thereof, such as the self-distributive law governing the interaction between dependent products and sums.

However, there remain many open questions regarding the further development of this framework, particularly with respect to *positive* type-theoretic constructs such as coproducts, inductive types, and intensional identity types. Further work is needed to explore whether polynomial functors can provide a fully general account of these concepts. We look forward to continuing these investigations, with the aim of extending the unification presented here to encompass a wider range of type-theoretic phenomena.

References

[AN18] Steve Awodey and Clive Newstead. *Polynomial pseudomonads and dependent type theory*. 2018. arXiv: 1802.00997 [math.CT] (cit. on p. 2).

[Awo14] Steve Awodey. "Natural models of homotopy type theory". In: (2014). eprint: arXiv:1406.3219 (cit. on p. 2).

[Mar75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08719451 (cit. on p. 1).

7 Appendix A

```
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

•invr : ∀ {ℓ} {A : Type ℓ} {a b : A}
        → (e : a ≡ b) → (sym e) • e ≡ refl
•invr refl = 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 (pairEquiv1 f ef)
                (pairEquiv2 g eg)

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

transpPre : ∀ {ℓ0 ℓ1 κ0 κ1} {A : Type ℓ0} {a a' : A} {B : A → Type κ0}
              {C : Type ℓ1} {D : C → Type κ1} {f : A → C}
```

```
              (mf : isMono f) (g : (x : A) → B x → D (f x))
              (e : f a ≡ f a') {b : B a}
              → transp D e (g a b) ≡ g a' (transp B (inv mf e) b)
transpPre {a = a} {a' = a'} {B = B} {D = D} {f = f} mf g e {b = b} =
    transp D e (g a b)
        ≡⟨ ap (λ e' → transp D e' (g a b)) (sym (snd (snd mf) e)) ⟩
    ( _ ≡⟨ (J (λ x e' → transp D (ap f e') (g a b) ≡ g x (transp B e' b))
               (inv mf e) refl) ⟩
    ((g a' (transp B (inv mf e) b)) □))

postulate
    funext : ∀ {ℓ κ} {A : Type ℓ}
             → {B : A → Type κ} {f g : (x : A) → B x}
             → ((x : A) → f x ≡ g x) → f ≡ g
    funextr : ∀ {ℓ κ} {A : Type ℓ}
              → {B : A → Type κ} {f g : (x : A) → B x}
              → (e : (x : A) → f x ≡ g x) → coAp (funext e) ≡ e
    funextl : ∀ {ℓ κ} {A : Type ℓ}
              → {B : A → Type κ} {f g : (x : A) → B x}
              → (e : f ≡ g) → funext (coAp e) ≡ e

transpD : ∀ {ℓ κ} {A : Type ℓ} {B : A → Type κ} {a a' : A}
          → (f : (x : A) → B x) (e : a ≡ a')
          → transp B e (f a) ≡ f a'
transpD f refl = refl

transpHAdj : ∀ {ℓ ℓ' κ} {A : Type ℓ} {B : Type ℓ'}
             → {C : B → Type κ} {a : A}
             → {g : A → B} {h : B → A}
             → (f : (x : A) → C (g x))
             → (e : (y : B) → g (h y) ≡ y)
             → (e' : (x : A) → h (g x) ≡ x)
             → (e'' : (x : A) → e (g x) ≡ ap g (e' x))
             → transp C (e (g a)) (f (h (g a))) ≡ f a
transpHAdj {C = C} {a = a} {g = g} {h = h} f e e' e'' =
    transp C (e (g a)) (f (h (g a)))
        ≡⟨ ap (λ ee → transp C ee (f (h (g a)))) (e'' a) ⟩
    (transp C (ap g (e' a)) (f (h (g a)))
        ≡⟨ sym (transpAp C g (e' a) (f (h (g a)))) ⟩
    ((transp (λ x → C (g x)) (e' a) (f (h (g a))))
        ≡⟨ transpD f (e' a) ⟩
    ((f a) □)))

PreCompEquiv : ∀ {ℓ ℓ' κ} {A : Type ℓ}
```

```
                → {B : Type ℓ'} {C : B → Type κ}
                → (f : A → B) → isEquiv f
                → isEquiv {A = (b : B) → C b}
                          {B = (a : A) → C (f a)}
                          (λ g → λ a → g (f a))
PreCompEquiv {C = C} f ef =
    let (f⁻¹ , l , r , e) = Iso→HAdj (isEquiv→Iso ef)
    in Iso→isEquiv ( (λ g b → transp C (r b) (g (f⁻¹ b)))
                   , ( (λ g → funext (λ b → transpD g (r b)))
                     , λ g → funext (λ a → transpHAdj g r l
                                           (λ x → sym (e x)))))

PostCompEquiv : ∀ {ℓ κ κ'} {A : Type ℓ}
                → {B : A → Type κ} {C : A → Type κ'}
                → (f : (x : A) → B x → C x) → ((x : A) → isEquiv (f x))
                → isEquiv {A = (x : A) → B x}
                          {B = (x : A) → C x}
                          (λ g x → f x (g x))
PostCompEquiv f ef =
    ( ( (λ g x → fst (fst (ef x)) (g x))
      , λ g → funext (λ x → snd (fst (ef x)) (g x))))
    , ( (λ g x → fst (snd (ef x)) (g x))
      , λ g → funext (λ x → snd (snd (ef x)) (g x)))
```