Electronic Notes in Theoretical Informatics And Computer Science

ENTICS
https://entics.episciences.org

Volume 5
Proceedings of
MFPS 2025

Polynomial Universes in Homotopy Type Theory

C.B. Aberlé [a,1] David I. Spivak[b,2]

[a] *Department of Computer Science*
Carnegie Mellon University
Pittsburgh, PA, USA

[b] *Topos Institute, Berkeley, CA, USA*

Abstract

Awodey, later with Newstead, showed how polynomial functors with extra structure (termed "natural models") hold within them the categorical semantics for dependent type theory. Their work presented these ideas clearly but ultimately led them outside of the usual category of polynomial functors to a particular *tricategory* of polynomials in order to explain all of the structure possessed by such models. This paper builds off that work—explicating the categorical semantics of dependent type theory by axiomatizing them entirely in terms of the usual category of polynomial functors. In order to handle the higher-categorical coherences required for such an explanation, we work with polynomial functors in the language of Homotopy Type Theory (HoTT), which allows for higher-dimensional structures to be expressed purely within this category. The move to HoTT moreover enables us to express a key additional condition on polynomial functors—*univalence*—which is sufficient to guarantee that models of type theory expressed as univalent polynomials satisfy all higher coherences of their corresponding algebraic structures, purely in virtue of being closed under the usual constructors of dependent type theory. We call polynomial functors satisfying this condition *polynomial universes*. As an example of the simplification to the theory of natural models this enables, we highlight the fact that a polynomial universe being closed under dependent product types implies the existence of a distributive law of monads, which witnesses the usual distributivity of dependent products over dependent sums.

Keywords: Dependent Type Theory, Category Theory, Homotopy Type Theory, Polynomial Functors, Natural Models

1 Introduction

The *dependency* of types-of-things on values-of-things is fundamental to our ability to express complex mathematical ideas and build up sophisticated abstractions. By taking this essential idea to heart, dependent type theory [1] provides both of the following:

- An elegant **syntax** for expressing mathematical ideas, that can moreover be **computably** realized.
- A robust **categorical semantics** that allow type theoretic syntax to be used in order to work in the internal languages of well-structured categories, and even more complex structures, such as ∞-categories.

⋆ This material is based upon work supported by the Air Force Office of Scientific Research under award numbers FA9550-20-1-0348 and FA9550-23-1-0376.

[1] Email: caberle@andrew.cmu.edu
[2] Email: dspivak@gmail.com

Rec'd Apr 3, 2025; Pub'd Dec 15, 2025
10.46298/entics.16885

Proceedings Available Online at
ENTICS MFPS 41 Proceedings

Of these, it is the **categorical semantics** of dependent type theory that shall be our focus in this paper. Specifically, although these categorical semantics are evidently quite powerful, they are also notoriously subtle, owing to the issue of *strictness*, whereby various identities that typically hold only up to isomorphism in arbitrary categories must hold *strictly* in order to soundly model the type-theoretic syntax.

A key device in resolving this difficulty turns out to be the categorical machinery of *polynomial functors*. Awodey [3]—and later Newstead [4]—have shown that there is a strong connection between dependent type theory and polynomial functors, via the concept of *natural models,* which cleanly solves the strictness problem via the type-theoretic concept of a *type universe,* and such universes turn out to naturally be regarded as certain polynomial endofunctors on a suitably-chosen category of presheaves.

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 to construct a rather complicated *tricategory* of polynomial functors in order to make sense of 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 resolve this impasse by showing how polynomial universes can be fully axiomatized in terms of the ordinary category of polynomial functors, by defining this category internally in the language of *Homotopy Type Theory* (HoTT) [2].

HoTT thus provides a *synthetic* setting in which to work with polynomial functors and their higher-categorical coherences without leaving the usual category of polynomial functors. As we shall see, this 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 closed under the usual type formers of dependent type theory, regarded as a polynomial pseudomonad, gives rise to distributive law of u over itself, which in particular witnesses the usual distributive law of dependent products over dependent sums. [3]

Additionally, the move from set theory to HoTT as a setting for developing the theory of polynomial universes makes it well-suited to formalization in a proof assistant. We have formalized the main results of this paper in Agda, with the code of this formalization given in the appendix.

2 The Trouble with Dependent Types

In what follows, we work in an informal setting of Martin-Löf type theory [1] and its categorical semantics [13], which we assume familiarity with on the part of the reader.

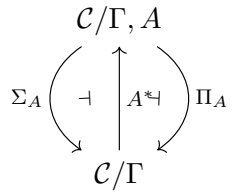
In the typical (naïve) categorical semantics of dependent type theory [13], one considers a category $\mathcal{C}$ whose objects are considered as corresponding to *contexts* Γ, such that morphisms $f : \Gamma \to \Delta$ correspond to *substitutions between contexts*, with a type $\Gamma \vdash A$ type dependent upon Γ being represented as the substitution $\Gamma, A \to \Gamma$ that forgets the type A from the extended context Γ, A (commonly called a *display map*). Application of a substitution $f : \Gamma \to \Delta$ to a type $\Delta \vdash A$ is then represented as the pullback

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

In particular, any display map $A : \Gamma, A \to \Gamma$ induces a functor $\mathcal{C}/\Gamma \to \mathcal{C}/\Gamma, A$ by substitution along A, which corresponds to *weakening* a context by adding a variable of type A. The left and right adjoints to

[3] A precursor to this story was attempted by the second-named author in a blog post, which may be useful for readers seeking intuition. There, a proposed self-distributive law of the list monad was claimed to witness dependent products. However, the author only later noticed that one of the four equations for distributive laws was doomed to fail [5] due to the 1-dimensionality of types in the category of sets. The present project was inspired by this.

the weakening functor (if they exist) then correspond to Σ and Π types, respectively.



In order for the operation of forming Σ and Π types to be stable under substitution (i.e. pullback), these must additionally satisfy the *Beck-Chevalley* condition:

$$\begin{array}{ccc} \Gamma, x : A[f] & \xrightarrow{g} & \Delta, x : A \\ {\scriptstyle A[f]}\downarrow & \lrcorner & \downarrow{\scriptstyle A} \\ \Gamma & \xrightarrow[f]{} & \Delta \end{array} \implies (\Sigma_A(B))[f] \xleftarrow{\simeq} \Sigma_{A[f]}(B[g])$$

Unfortunately, this pleasingly simple story of categorical dependent types 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, and *strictly* satisfies the Beck-Chevalley condition. However, in the above categorical semantics, substitution is computed by pullback, which is in general only associative up to isomorphism, and likewise for the Beck-Chevalley condition. It is precisely this problem which natural models exist to solve.

2.1 Natural Models

As mentioned previously, the key insight (due originally to Voevodsky) 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 a *type universe*, i.e. a type $\mathcal{U}$ whose elements can themselves be regarded as types. In categorical semantics, we interpret such a universe as a display map $u : \mathcal{U}_\bullet \to \mathcal{U}$. A type in context Γ may then be represented as a morphism $A : \Gamma \to \mathcal{U}$, rather than a display map $\Gamma, A \to \Gamma$, which we may recover as the pullback:

$$\begin{array}{ccc} \Gamma, A & \longrightarrow & \mathcal{U}_\bullet \\ \downarrow & \lrcorner & \downarrow{\scriptstyle u} \\ \Gamma & \xrightarrow[A]{} & \mathcal{U} \end{array}$$

We say that a display map $\Gamma, A \to \Gamma$ is *classified* by $\mathcal{U}$ if there is a pullback square as above.

Hence given a universe of types $\mathcal{U}$, rather than representing substitution as pullback, we can simply represent the action of applying a substitution $f : \Gamma \to \Delta$ to a family of types $A : \Delta \to \mathcal{U}$ as the precomposition $A \circ f : \Gamma \to \mathcal{U}$, which is automatically strictly associative, and strictly satisfies the Beck-Chevalley condition for Σ types / Π types if $\mathcal{U}$ is closed under Σ types / Π types, respectively (although what it means for $\mathcal{U}$ to be closed under Σ and Π types is rather nontrivial—indeed, this topic forms the main subject of this paper).

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 possesses such a *universe* whose types correspond to those of $\mathcal{C}$. A natural candidate for such an embedding is the *Yoneda embedding* $\mathbf{y} : \mathcal{C} \to \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$.

Hence we can work in the category of presheaves $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ to study the type-theoretic language of $\mathcal{C}$. The universe $\mathcal{U}$ is then given by:

(i) an object of $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$, i.e. a contravariant functorial assignment, to each context Γ, of a set $\mathcal{U}(\Gamma)$ of *types in context* Γ, together with

(ii) an object $u \in \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}/\mathcal{U}$, i.e. a natural transformation $u : \mathcal{U}_\bullet \to \mathcal{U}$, where for each context Γ, $\mathcal{U}_\bullet(\Gamma)$ is the set of terms in context Γ, and $u_\Gamma : \mathcal{U}_\bullet(\Gamma) \to \mathcal{U}(\Gamma)$ assigns each term to its type.

The condition that all types in $\mathcal{U}$ "belong to $\mathcal{C}$" can then be expressed by requiring u to be *representable* in the following sense: for any representable $\gamma \in \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ with $\alpha : \gamma \to \mathcal{U}$, the pullback

$$\begin{array}{ccc} \gamma.\alpha & \longrightarrow & \mathcal{U}_\bullet \\ \downarrow & \lrcorner & \downarrow u \\ \gamma & \xrightarrow[\alpha]{} & \mathcal{U} \end{array}$$

is representable. In particular, this says that, given a context Γ and a type $A \in \mathcal{U}[\Gamma]$, there is a context Γ, A together with a substitution $\Gamma, A \to A$ that corresponds to the above pullback under the Yoneda embedding. Type-theoretically, this corresponds to the operation of *context extension*.

The question, then, is how to express that $\mathcal{C}$ has Σ types, Π types, etc., in terms of the structure of u. Toward answering this question, we may note that u gives rise to an endofunctor (indeed, a *polynomial endofunctor*) $P_u : \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}} \to \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$, defined by

$$P_u(X) = \sum_{A:\mathcal{U}} X^{\mathcal{U}_\bullet[A]}$$

(This notation will be made precise momentarily). As it turns out, much of the type-theoretic structure of u (and by extension $\mathcal{C}$) can be accounted for in terms of this functor. For instance, u is closed under unit and Σ types if and only if P_u carries the structure of a *Cartesian pseudomonad* on $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ (c.f. Theorem 2.3 of [4]). To see why this is the case, we could, on the one hand, proceed as in past developments of the theory of natural models and define a certain tricategory of polynomial functors in which such pseudomonads can be defined. However, here we diverge from these past approaches, and instead develop the theory of polynomial functors in the language of *Homotopy Type Theory*.

3 Polynomial Functors in HoTT

In order to understand the higher-dimensional identities and coherences of type formers in natural models, we now change our setting from the *extensional* type-theoretic language of 1-topoi such as $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ to the *intensional* type-theoretic language of ∞-topoi, which is a form of HoTT [14]. All the constructions on natural models we considered previously carry over to this setting—mutatis mutandis—by replacing the category of presheaves $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ with the ∞-category of presheaves $\infty\mathbf{Grpd}^{\mathcal{C}^{\mathrm{op}}}$.[4]

Working internally in the language of $\infty\mathbf{Grpd}^{\mathcal{C}^{\mathrm{op}}}$, let `Type` denote the *type* of types [5] and let **Type** be the *category* of types and functions between them. For any type A, let y^A denote the corresponding co-representable functor **Type** $\to$ **Type** that maps a type y to the function type `A → y`. A *polynomial endofunctor* P : **Type** $\to$ **Type** is an endofunctor on **Type** equivalent to a sum of such co-representables

$$P(y) = \sum_{x:A} y^{B[x]}$$

for some type A and a family of types $B[x]$ indexed by $x : A$. The data of a polynomial functor is thus

[4] Since essentially all of the categorical structures considered henceforth in this paper shall be infinite dimensional, we shall generally omit the prefix "∞" from our descriptions of these structures. Hence hereafter "category" means ∞-category, "functor" means ∞-functor, "limit" means homotopy limit, and so on, unless otherwise specified.

[5] Technically, we assume an infinite hierarchy of stratified type universes in order to avoid inconsistency. However, for the purpose of high-level explanation, we shall generally omit universe levels and speak generically of the "type" of all types and similarly the category of all such, even though, strictly speaking, neither of these exist.

uniquely determined by the choice of A and B. [6] Hence we may represent the *type* of such functors as that of pairs (A, B) of this form. In Agda, this type can be expressed as follows:

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

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. 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, allowing for an elegant treatment of the theory of natural models.

For $p = \sum_{a:A} y^{B(a)}$, $q = \sum_{c:C} y^{D(c)} \in \mathbf{Poly}$, a natural transformation $p \to q$ is equivalently given by:

- a *forward* map `f : A → B`, and
- a *backward* map `g : (a : A) → D (f a) → B a`

as can be seen from the following argument via Yoneda:

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

In Agda, we use the notation $p \leftrightarrows q$ to the latter type (AKA the type of *dependent lenses*—or just *lenses*—from p to q), which may be written 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)
```

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 of particular importance to us.

We say that a lens `(f , f♯) : (A , B) ⇆ (C , D)` is *Cartesian* if for every `a : A`, the map `f♯ a : D[f a] → B a` is an equivalence. [7]

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

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

open Cart public
```

In what follows, Cartesian lenses shall be of special interest to us, owing to the fact that the existence of a Cartesian lens $p \to u$ in a certain sense expresses that u is closed under all the types encoded by p. Specifically, if we view `p = (A , B)` as an `A`-indexed family of types, given by `B`, then the existence of a Cartesian lens `(f , f♯) : p ⇆ u` essentially shows that, for each `a : A` there is an element `f a : U` such that $B[a] \simeq u[f\ a]$, i.e. every type in the family $B[a]$ is equivalent to one for which there exists a

[6] A similar treatment to ours of universes via polynomial functors, there referred to as "containers," is given in [15], including the treatment of Σ types via monadic structure. In our case, we leverage some additional properties of the category of polynomial enfodunctors, such as the Vertical-Cartesian factorization system in particular, to additionally characterize Π types in terms of distributive laws.

[7] For a proof that this notion of Cartesian morphism between polynomials is equivalent to the ordinary definition of Cartesian natural transformations between polynomial functors, see Chapter 5.5 of [8]

"code" in U. To show that u is closed under Σ types, Π types, etc., we therefore need only find polynomials that suitably represent these types, and ask that there be Cartesian morphisms from these to u.

Cartesian lenses are also closed under composition, so there is a wide subcategory $\mathbf{Poly}^{\mathbf{Cart}}$ of the category of polynomial functors, whose morphisms are Cartesian lenses. By studying which categorical properties of **Poly** are inherited by $\mathbf{Poly}^{\mathbf{Cart}}$, we can then deduce the corresponding properties of polynomial universes/natural models. This shall be our method of choice in what follows.

3.1 Composition of Polynomial Functors and Σ Types

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

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

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

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

_◁_ : ∀ {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)) )
```

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 classifies 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 [3,4], 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.

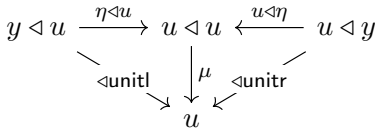
```
module ◁Assoc {ℓ0 ℓ1 ℓ2 κ0 κ1 κ2} (p : Poly ℓ0 κ0)
              (q : Poly ℓ1 κ1) (r : Poly ℓ2 κ2) where

    ◁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)) )

open ◁Assoc public
```

while the left and right unitors of $\triangleleft$ correspond to the fact that $\top$ is both a left and a right unit for Σ.

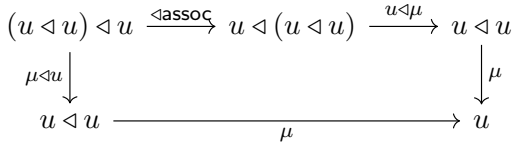
```
module ◁LRUnit {ℓ κ} (p : Poly ℓ κ) where

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

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

open ◁LRUnit public
```

In fact, $\triangleleft$ restricts to a monoidal product on $\mathbf{Poly}^{\mathbf{Cart}}$, since the functorial action of $\triangleleft$ on lenses preserves Cartesian lenses, and all of the above-defined structure morphisms for $\triangleleft$ are Cartesian. We should expect, then, for these equivalences to be somehow reflected in the structure of Cartesian lenses $\eta : y \leftrightarrows u$ and $\mu : u \triangleleft u \leftrightarrows u$ witnessing the closure of u under $\top$ and Σ. This would be the case, e.g.,

if the following diagrams were to commute:



$(M1)$



$(M2)$

One may recognize these as the usual diagrams for a monoid in a monoidal category, hence (since $\triangleleft$ corresponds to endofunctor composition) 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 monad, but rather of an ∞-*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, as we shall show in the following section. For now, however, we turn to the other main type former of dependent type theory which we have not yet considered: the dependent function type, or Π type.

3.2 The $\uparrow\uparrow$ Functor & Π Types

We have so far considered how polynomial universes may be equipped with structure to interpret the unit type and dependent pair types. We have not yet, however, said much in the way of *dependent function types.* In order to rectify this omission, it will first be prudent to consider some additional structure on the category of polynomial functors – specifically a new functor $\uparrow\uparrow$ 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**.

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

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

and the following definition in Agda:

```
_↑↑_ : ∀ {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 $p \uparrow\uparrow q \to p' \uparrow\uparrow q$ of $\uparrow\uparrow$ on a lens $f : p \to p'$ equipped with a left inverse $f' : p' \to p$, i.e. such that $f' \circ f = \mathrm{id}_p$. [8]

[8] To see why this is the right choice of morphism for which $\uparrow\uparrow$ 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

```
⇈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) ↱ x →
         g♯ (γ x)
            (transp (λ y → snd s (g (γ y)))
                    (sym (e♯ a x))
                    (↱ (f'♯ (f a) (transp (snd p) (e a) x))))) )
```

A Cartesian lens `(π , π♯) : u ⇈ u ⇆ u` then effectively shows that u is closed under Π-types, since:

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

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

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

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

- Moreover, for all `(A , B)`, the inverse map $\pi\sharp^{-1}$ `(A , B) : (x : u(A)) → u(B(x)) → u(`π`(A,B))` to $\pi\sharp$ `(A , B)` corresponds to λ-abstraction:

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

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

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

Although it is clear enough that ⇈ serves its intended purpose of characterizing Π types polynomially, 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 ⇈ 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 ⇈.

---

twisted arrow category of **Poly**, i.e. diagrams of the following form:

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

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 $\upuparrows$ can be viewed as a functor $\mathbf{Poly}_R \times \mathbf{Poly} \to \mathbf{Poly}$. Then $\upuparrows$ 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 equivalences

$$y \upuparrows p \simeq p \quad \text{and} \quad (p \triangleleft q) \upuparrows r \simeq p \upuparrows (q \upuparrows r)$$

defined as follows:

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

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

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) )

open ◁⇈ public
```

The fact that `⇈Curry` is an equivalence corresponds to the usual currying isomorphism 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, $\upuparrows$ is colax with respect to $\triangleleft$ in its second argument in that there are natural transformations

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

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

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

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)) )

open ⇈◁ public
```

Moreover, this colax structure on $\uparrow\uparrow$ descends to $\mathbf{Poly}^{\mathbf{Cart}}$ in that the natural transformations defined above are *Cartesian*. In particular, the fact that $\uparrow\uparrow$`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 whether this distributive law is related to a distributive law of the monad structure on a universe u given by Σ types (as discussed in the previous subsection) over itself, i.e. a morphism

$$u \triangleleft u \to u \triangleleft u$$

The answer to this question is affirmative, but in order to see why, we must first explain the machinery that allows us to straightforwardly derive the coherence data for such structures from the mere existence of cartesian morphisms as described above, namely *univalence*.

4 Polynomial Universes

For any polynomial u, we say that u is *univalent* if it 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 mere proposition, or in other words, 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
      → f ≡ g
```

We call this property of polynomials univalence in analogy with the usual univalence axiom of HoTT, since the univalence axiom for a universe $\mathcal{U}$ is equivalent to the statement that the polynomial

$$\sum_{A:\mathcal{U}} y^A$$

is univalent in the above sense. Indeed, the statement that there are *enough* univalent families in $\infty\mathbf{Grpd}^{\mathcal{C}^{\mathrm{op}}}$ is equivalent to the existence of a *terminal family* in $\mathbf{Poly}^{\mathbf{Cart}}$, i.e. a collection of subterminal objects $u_i \in \mathbf{Poly}^{\mathbf{Cart}}$ such that every $p \in \mathbf{Poly}^{\mathbf{Cart}}$ has a Cartesian morphism to *some* u_i. In what follows, we therefore assume the existence of such a terminal family u_i, which, although not strictly necessary for the main theorems of this paper to do with semantics of dependent type theory, [9] is useful for constructing illustrative examples.

We refer to univalent polynomial functors as *polynomial universes*. If we think of a polynomial `p` as representing a family of types, then what this tells us is that if u is a polynomial universe, there is essentially at most one way for u to contain the types represented by `p`, where containment is here understood as existence of a Cartesian lens `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 we have the following:

Theorem 4.1 *If u is a polynomial universe with Cartesian lenses $\eta : y \leftrightarrows u$ and $\mu : u \triangleleft u \leftrightarrows u$, then the diagrams M1, M2 given in §3.1 commute.*

[9] which are in fact all valid in ordinary intensional Martin-Löf type theory without univalence

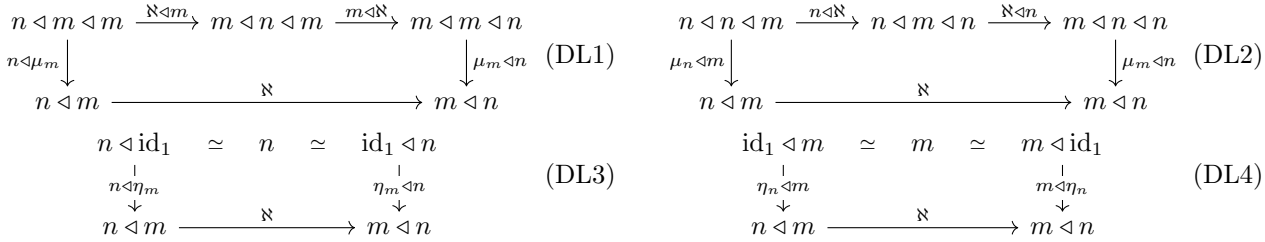
The proof is essentially by observation that all paths through the diagrams M1 and M2 are Cartesian morphisms that terminate in u and hence, by univalence of u, must be equal. Moreover, one can in principle reason similarly to derive all the higher coherence data of an ∞-monad, although, since it is not currently known how to express the totality of such data as a type in HoTT, this cannot be expressed directly as a theorem in HoTT at present.

For Π types, the situation is more complex. In the remainder of this section, we will concern ourselves with showing that the closure of a univalent universe u under Π types gives rise to a *distributive law* of the monad structure on u over itself.

Recall that a (1-dimensional) distributive law of a monad m over another monad n is essentially an answer to the question "when is the composite $m \triangleleft n$ also a monad?" given by a morphism:

$$\aleph : n \triangleleft m \to m \triangleleft n$$

such that the following diagrams commute:



Note that, in the case where $m = n = u$, the morphisms identified by these diagrams are *not* generally Cartesian morphisms into u, so we cannot immediately apply univalence as we did for the monad laws.

The solution to this problem proceeds in several steps:

(i) We first generalize from distributive laws $\aleph$, as above, to *distributors*, [10] which are morphisms of the form

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

(ii) We likewise generalize the definition of the $\uparrow\uparrow$ functor given previously to an action on **Poly** of the *twisted arrow category* $\mathsf{Tw}(\mathbf{Poly})$ of **Poly**. In particular, since $\mathbf{Poly}^R$ as defined previously is equivalent to the full (monoidal) subcategory of $\mathsf{Tw}(\mathbf{Poly})$ spanned by identity morphisms on polynomial functors, our previous definition of $\uparrow\uparrow$ turns out to be the restriction of this generalized definition from an action of $\mathsf{Tw}(\mathbf{Poly})$ to one of $\mathbf{Poly}^R$ along the embedding

$$\mathbf{Poly}^R \hookrightarrow \mathsf{Tw}(\mathbf{Poly})$$

In what follows, we therefore continue to write $p \uparrow\uparrow q$ as a shorthand for $\mathrm{id}_p \uparrow\uparrow q$.

(iii) So-generalized, $\uparrow\uparrow$ acquires a key property: given $\phi : p \to s$, every morphism $f : \phi \uparrow\uparrow q \to r$ gives rise to a distributor $\nabla_f : p \triangleleft q \to r \triangleleft s$.

(iv) We show that the operations on distributors appearing in the equations for a distributive law correspond to compositions of morphisms of the form $\phi \uparrow\uparrow q \to r$ under the above translation between morphisms $\phi \uparrow\uparrow q \to r$ and distributors $p \triangleleft q \to r \triangleleft s$, making use of the structure of $\uparrow\uparrow$ as a colax monoidal action. This allows us to convert the diagrams for a distributive law of u over itself into diagrams involving a (Cartesian) morphism $\mathrm{id}_u \uparrow\uparrow u \to u$, whence we conclude as before in the case

[10] Distributors as above arise in $\mathbf{EM}\text{-}\mathbb{C}\mathbf{at}^\sharp$ the Eilenberg-Moore completion [10] of the double category $\mathbb{C}\mathbf{at}^\sharp = \mathbb{C}\mathbf{omod}(\mathbf{Poly})$ of polynomial comonads and bicomodules [11]. Indeed, there we find morphisms of the form $m \triangleleft p \to p \triangleleft n$ for polynomial monads m, n, and distributive laws between monads are the formal monads in $\mathbf{EM}\text{-}\mathbb{C}\mathbf{at}^\sharp$. Distributors also arise in what Lynch et al. call *effects handlers* [12], where morphisms are of the form $s \triangleleft c \to d \triangleleft s$ for polynomial comonads $c, d \in \mathbb{C}\mathbf{at}^\sharp$.

of the monad laws for Σ types that, if u is univalent, then these diagrams (and all higher-dimensional diagrams involving them) must commute.

Hence we have the following:

Theorem 4.2 *If u is a polynomial universe with Cartesian lenses $\eta : y \leftrightarrows u$, $\mu : u \triangleleft u \leftrightarrows u$, along with $\pi : u \upuparrows u \leftrightarrows u$, then letting $m = n = u$ and $\aleph = \nabla_\pi$, the diagrams DL1, DL2, DL3, and DL4 commute.*

A full proof, formalized in Agda, is given in the appendix.[11] For the remainder of this section, we point out some high-level details of particular note in the proof.

4.1 Distributors & Jump Structures

Not every distributor $\nabla : p \triangleleft q \to r \triangleleft s$ is of the form ∇_f for some lens $f : \phi \upuparrows q \to s$. We define a *jump structure* on a distributor ∇ to be a witness to the fact that $\nabla = \nabla_f$ for some f. Expressed this way, the type of jump structures on a distributor ∇ may be written as

$$\sum_{\phi : p \to s} \sum_{f : \phi \upuparrows q \to r} \nabla = \nabla_f$$

However, there is another description of jump structures purely in terms of the structure of polynomial functors. Intuitively, a jump structure witnesses that ∇ essentially acts as ϕ on its components in p, s while "jumping over" q, r. To express this directly in terms of polynomials, we need a notion of "independence" of the components of a distributor. For this, we make use of an additional monoidal product on **Poly**, the *tensor product* $\otimes$, defined as follows: given polynomials $p = \sum_{a:A} y^{B[a]}, q = \sum_{c:C} y^{D[c]}$, their tensor product $p \otimes q$ is defined as

$$p \otimes q = \sum_{(a,c):A \times C} y^{B[a] \times D[c]}$$

Unlike $\triangleleft$, the monoidal product $\otimes$ is symmetric on **Poly**. However, $\otimes$ and $\triangleleft$ are compatible with one another, in that they together form the structure of a normal duoidal category [8]. This in particular gives rise to a monoidal natural transformation

$$\mathsf{indep}_{p,q} : p \otimes q \to p \triangleleft q$$

between these two structures. Intuitively, thinking of $\otimes$ as independent and $\triangleleft$ as dependent combination of polynomials, indep exhibits independence as a trivial kind of dependence.

We further note that, although lenses give a canonical notion of morphism between polynomial functors, they are not the only such. If we think of polynomial functors as corresponding to their representative display maps, then another candidate notion of morphism between polynomials is given by commutative squares. I.e. for $p : B \to A$ and $q : D \to C$, such a morphism $p \not\to q$ consists of a commuting square:

$$\begin{array}{ccc} B & \xrightarrow{f_1} & D \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle q} \\ A & \xrightarrow{f^\flat} & C \end{array}$$

Following [9], we refer to natural transformations of polynomial functors as *lenses* and to commutative squares of their representative morphisms as *charts*. Notably, the category of polynomial functors and charts has the following properties:

[11] As in the case of Theorem 4.2, we could in principle continue in this same manner to derive the higher coherences of an ∞-distributive law of ∞-monads, but since the *type* of all such data is not currently definable, we cannot express this as a theorem in HoTT at present.

(i) It is equivalent to the *arrow category* $\mathbf{Type} \downarrow \mathbf{Type}$ of the category **Type** of types and functions.

(ii) It is the fibrewise opposite of **Poly**, viewed as a category fibred over **Type**.

Straightforwardly, the category of polynomial functors and charts between them inherits $\otimes$ as a monoidal product, which is moreover the Cartesian product for that category. Although $\triangleleft$ does not define a corresponding monoidal product on the category of polynomial functors and charts, it is the case that, for any p, q, there exists a chart $\pi_1 : p \triangleleft q \not\to p$ given by the first projection of both components of p.

Lenses and charts together form a *double category*, i.e. a category with two distinct kinds of morphisms, along with a notion of *commuting squares* formed by these morphisms (c.f. [9] for a full definition of this double category). When depicting this double category, we draw lenses horizontally, and charts vertically. Then a square

$$\begin{array}{ccc} p & \xrightarrow{\phi} & q \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ p' & \xrightarrow[\phi']{} & q' \end{array}$$

where $\phi = (\phi_1, \phi^\sharp),\ \phi' = (\phi'_1, \phi'^\sharp),\ f = (f_1, f_\flat),\ g = (g_1, g_\flat)$, is said to commute if the following diagrams commute

$$\begin{array}{ccc} A & \xrightarrow{\phi_1} & C \\ {\scriptstyle f_1}\downarrow & & \downarrow{\scriptstyle g_1} \\ A' & \xrightarrow[\phi'_1]{} & C' \end{array} \qquad \begin{array}{ccc} D[\phi_1] & \xrightarrow{\phi^\sharp} & B \\ {\scriptstyle g_\flat[\phi_1]}\downarrow & & \\ D'[g_1 \circ \phi_1] & & \downarrow{\scriptstyle f_\flat} \\ \simeq & & \\ D'[\phi'_1 \circ f_1] & \longrightarrow D'[\phi'_1] \xrightarrow[\phi'^\sharp]{} & B' \end{array}$$

Using these definitions, we can straightforwardly define a jump structure as follows:

Definition 4.3 A *jump structure* of a morphism $\phi : p \to s$ on a distributor $\nabla : p \triangleleft q \to r \triangleleft s$ consists of homotopies witnessing that:

(i) ∇ factors through indep, i.e. there exists

$$\nabla' : p \triangleleft q \to r \otimes s$$

such that $\nabla = \mathsf{indep}_{r,s} \circ \nabla'$

(ii) The following square commutes

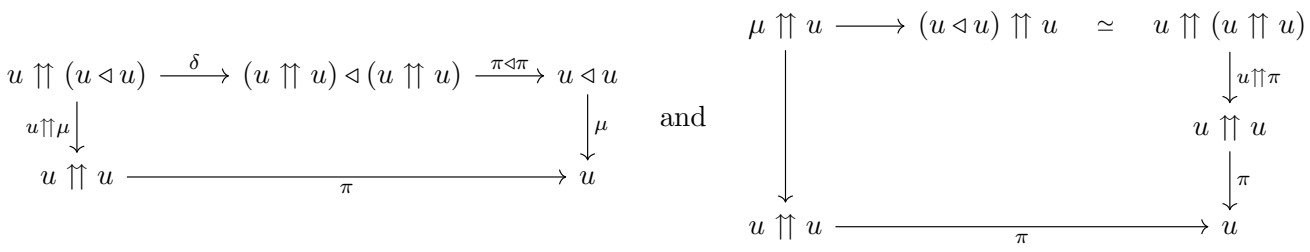
$$\begin{array}{ccc} p \triangleleft q & \xrightarrow{\nabla'} & r \otimes s \\ {\scriptstyle \pi_1}\downarrow & & \downarrow{\scriptstyle \pi_2} \\ p & \xrightarrow[\phi]{} & s \end{array}$$

We define a *Cartesian* jump structure to be a jump structure whose corresponding morphism $\phi \upuparrows r \to s$ is Cartesian. It then follows, by Theorem 4.2 that, for u a polynomial universe, the (mere) existence of a Cartesian jump structure on a distributive law $\aleph : u \triangleleft u \to u \triangleleft u$ is equivalent to u being closed under Π types, in that there exists a Cartesian morphism $u \upuparrows u \to u$.

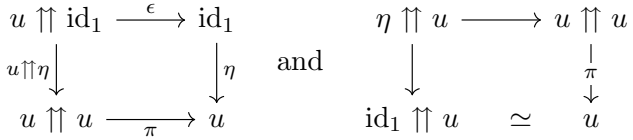
4.2 Induced Diagrams

Under the translation between morphisms $\phi \upuparrows q \to r$ and distributors $p \triangleleft q \to r \triangleleft s$, for a (Cartesian) morphism $\pi : u \upuparrows u \to u$, the following diagrams correspond to those for a distributive law.

For DL1 and DL2, respectively, the corresponding diagrams are:



(where the unlabeled morphisms arise from the structure of $\uparrow\uparrow$ as an action of the twisted arrow category on **Poly**). For DL3 and DL4, the corresponding diagrams are:



(where, as for DL2, the unlabeled morphisms arise from the structure of $\uparrow\uparrow$ as an action of the twisted arrow category on **Poly**). We note that the morphisms identified by these diagrams terminate in u and are Cartesian, assuming η, μ, π are Cartesian. Hence, if u is univalent, these diagrams must commute.

5 Examples of Polynomial Universes

5.1 Proof-Relevant Partiality Monads

For a univalent universe $\mathcal{U}$, the corresponding polynomial functor looks like

$$X \mapsto \sum_{A:\mathcal{U}} X^A$$

If we specialize this to the case where $\mathcal{U} = \mathbf{Prop}$, the type of propositions, this gives

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

This monad is well-known in type theory by another name – the *partiality* monad. Specifically, this is the monad M whose Kleisli morphisms $A \to M(B)$ correspond to partial functions $A \rightharpoonup B$, i.e. functions that associate to each element $a : A$ a proposition $\mathsf{def}_f(a)$ stating whether f is defined at input a, such that if $\mathsf{def}_f(a)$ is true, then one can obtain a value $f(a) : B$.

It follows that one can more generally consider the polynomial monads derived from polynomial universes as *proof-relevant partiality monads*.

5.2 Rezk Completion, Lists & Finite Sets

Additionally, we can show that *any* polynomial functor p can be quotiented to a corresponding univalent polynomial, using a familiar construct from the theory of categories in HoTT – the *Rezk Completion.* [6]

We obtain the Rezk completion of p as the image factorization in $\mathbf{Poly}^{\mathbf{Cart}}$ of the classifying Cartesian morphism $p \to u_i$ from p to a univalent polynomial u_i [12]

$$p \twoheadrightarrow \mathsf{Rezk}(p) \rightarrowtail u_i$$

[12] where the existence of such a classifying map follows from the assumption previously mentioned in §4 that the

Then, since $\mathsf{Rezk}(p)$ is a subobject of a subterminal object in $\mathbf{Poly}^{\mathbf{Cart}}$, it follows that it is itself also subterminal in $\mathbf{Poly}^{\mathbf{Cart}}$.

Example 5.1 The polynomial functor determined by the function $((m,n) \mapsto n) : \{m < n \in \mathbb{N}\} \to \mathbb{N}$ is

$$X \mapsto \sum_{n \in \mathbb{N}} X^{\{m \in \mathbb{N} | m < n\}} \cong \mathsf{List}(X)$$

This polynomial isn't univalent, because $\mathbb{N}$ is a set (i.e. there is at most one path/identity between any two elements of $\mathbb{N}$), whereas the types $\{m \in \mathbb{N} \mid m < n\}$ form a groupoid (in general, there are $n!$ automorphisms of the path $\{m \in \mathbb{N} \mid m < n\}$, corresponding to permutations of finite sets). However, we can upgrade this to a univalent polynomial using the Rezk completion. If we write out an explicit description of $\mathsf{Rezk}(\mathsf{List})$, we see that it is the subuniverse of types X that are merely equivalent to some finite type $\{m \in \mathbb{N} \mid m < n\}$. In constructive mathematics, these types (they are necessarily sets) are known as *Bishop finite sets*. Hence the Rezk completion of the list monad is precisely the subuniverse of types spanned by (Bishop) finite sets.

Both List and $\mathsf{Rezk}(\mathsf{List})$ are Cartesian monads, hence closed under unit and Σ. However, although List does not have a self-distributive law corresponding to Π types, $\mathsf{Rezk}(\mathsf{List})$ *does* by Theorem 4.2, since finite sets are closed under finite products. Moreover, although List doesn't *quite* have such a distributive law, it *almost* does, where the pertinent morphism $\mathsf{List} \triangleleft \mathsf{List} \to \mathsf{List} \triangleleft \mathsf{List} \in \mathbf{Poly}$ is the "Cartesian Product" operation on lists that maps a list-of-lists

$$xss = [[x_{11}, \dots, x_{1i_1}], \dots, [x_{j1}, \dots, x_{ji_j}]]$$

to the list of all j-element lists consisting of one element from each list in xss, ordered lexicographically by their occurrence in xss. This operation fails to satisfy some of the equational laws of a distributive law, namely DL1 [5]. However, passing to the Rezk completion of List essentially *forces* this operation to satisfy these equations – up to homotopy. It is therefore an interesting question, although beyond the scope of this paper, to consider what structure on a base (non-univalent) polynomial suffices to guarantee that its Rezk completion will be closed under Σ types / Π types.

Additionally, the fact that $\mathsf{Rezk}(\mathsf{List})$ posesses such a distributive law over itself has intriguing consequences for higher-categorical algebra. Since identities between elements of $\mathsf{Rezk}(\mathsf{List})$ correspond to permutations of finite sets, it follows that an *algebra* for $\mathsf{Rezk}(\mathsf{List})$ should be a type equipped with an associative and unital operation for combining any finite number of elements of that type, that is invariant under all pemutations of finite tuples of elements of the type, i.e. (up to coherent homotopy) a commutative monoid. Hence $\mathsf{Rezk}(\mathsf{List})$ may be regarded as a higher-categorical analogue of the free commutative monoid monad. Moreover, this monad is both polynomial and Cartesian, unlike the ordinary free commutative monoid monad on the category of sets.

More strikingly still, because $\mathsf{Rezk}(\mathsf{List})$ possesses a distributive law over itself, $\mathsf{Rezk}(\mathsf{List}) \triangleleft \mathsf{Rezk}(\mathsf{List})$ is *also* a monad, and we conjecture that this monad forms a higher-categorical analogue of the free commutative ring monad. Note that unlike $\mathsf{Rezk}(\mathsf{List})$, this monad is not Cartesian, although it is (by construction) still polynomial, again in contrast to the ordinary free commutative ring monad on the category of sets. We leave a further consideration of this monad and its algebras, hopefully along with a proof of this conjecture, to future work.

---

ambient type theory has univalent families, which is equivalent to the existence of a *terminal family* in $\mathbf{Poly}^{\mathbf{Cart}}$, i.e. a family of polynomials $\{u_i\}_{i \in I}$ such that every other polynomial p has a unique Cartesian morphism to some u_i.

References

[1] 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
https://doi.org/10.1016/S0049-237X(08)71945-1

[2] The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study (2013).
https://homotopytypetheory.org/book.

[3] Steve Awodey. Natural models of homotopy type theory. Mathematical Structures in Computer Science. 28 2 (2016). pp. 241-286
https://doi.org/10.48550/arXiv.1406.3219

[4] Steve Awodey and Clive Newstead. "Polynomial pseudomonads and dependent type theory". In: arXiv (2018), eprint.
https://doi.org/10.48550/arXiv.1802.00997

[5] Maaike Zwart and Dan Marsden. "No-go theorems for distributive laws". In: Logical Methods in Computer Science1 8 (2022)
https://doi.org/10.48550/arXiv.2003.12531

[6] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. "Univalent Categories and the Rezk Completion". In: Extended Abstracts Fall 2013. Ed. by Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock. Cham: Springer International Publishing, 2015, pp.75–76.
Journal article available at: https://doi.org/10.1017/S0960129514000486

[7] Nicola Gambino and Joachim Kock. "Polynomial functors and polynomial monads". 2012. In: Mathematical Proceedings of the Cambridge Philosophical Society 154. 1 (2013), pp. 153-192
https://doi.org/10.1017/S0305004112000394

[8] Nelson Niu and David I. Spivak. *Polynomial Functors: A Mathematical Theory of Interaction*. London Mathematical Society Lecture Notes, to appear. Cambridge University Press, 2024
Available online at: https://toposinstitute.github.io/poly/poly-book.pdf

[9] David Jaz Myers. *Categorical Systems Theory*. Cambridge University Press (in preparation), 2023
Preprint avilable at: https://www.davidjaz.com/Papers/DynamicalBook.pdf

[10] Stephen Lack and Ross Street. "The formal theory of monads II". In: Journal of pure and applied algebra 175. 1-3 (2002), pp. 243–265
https://doi.org/10.1016/S0022-4049(02)00137-8

[11] Brandon T. Shapiro and David I. Spivak. "A Polynomial Construction of Nerves for Higher Categories". 2024. arXiv: 2405.13157 [math.CT].
https://doi.org/10.48550/arXiv.2405.13157

[12] Owen Lynch, Brandon T. Shapiro, and David I. Spivak. "All Concepts are $\mathbf{Cat}^\sharp$". 2023. arXiv: 2305.02571 [math.CT]
https://doi.org/10.48550/arXiv.2305.02571

[13] R. A. G. Seely, "Locally cartesian closed categories and type theory," Mathematical Proceedings of the Cambridge Philosophical Society, vol. 95, no. 1, pp. 33–48, 1984.
DOI: https://doi.org/10.1017/S0305004100061284

[14] M. Shulman, "All $(\infty, 1)$-toposes have strict univalent universes," 2019. arXiv: 1904.07004 [math.AT]
https://doi.org/10.48550/arXiv.1904.07004

[15] T. Altenkirch, G. Pinyo, "Monadic containers and universes," 2018. Talk slides available at https://people.cs.nott.ac.uk/psztxa/talks/types-17-cont.pdf

A Homotopy Type Theory in Agda

The following Agda module is used to define key definitions from HoTT that will be used by subsequent modules in formalizing the results of this paper.

```
{-# OPTIONS --without-K --rewriting #-}
module hott where

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

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

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

Basic properties of the identity type, including reflexivity, transitivity, symmetry, and congruence, and some convenient notation for equality proofs.

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

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

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

_•_ : ∀ {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 #-}

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

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

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

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
```

Equality for pairs:

```
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

    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
```

The bi-invertible maps definition of equivalence, and closure of equivalences under identity and composition:

```
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))

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                    □)))
```

Isomorphisms and equi-inhabitation of isomorphism and equivalence:

```
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)))

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
```

Proof that the inverse of an equivalence is 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
```

Proof that transport along a path is an equivalence:

```
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
```

Some additional facts about the identity type that will be used throughout this formalization, are as follows:

```
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) □)))
```

Half-adjoint equivalences and converting an isomorphism to a half-adjoint equivalence:

```
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))) □)))))
```

Equivalences of dependent pair types:

```
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)
```

The J rule, i.e. the induction principle for the identity type:

```
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
```

Function extensionality and derived results:

```
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)))
```

B Polynomial Functors in Agda

This module gives basic definitions involving polynomial functors, lenses, Cartesian lenses, and univalence.

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

open import Agda.Primitive
open import Agda.Builtin.Sigma
open import Agda.Builtin.Unit
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
open import hott
```

Definition of polynomial functors:

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

Lenses:

```
_⇆_ : ∀ {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)
```

Type of equality proofs for 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))
```

Identity and composition of lenses:

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

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) )
```

Cartesian lenses:

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

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

open Cart public
```

Identity and composition of Cartesian lenses:

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

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))
```

Univalent polynomials:

```
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
```

C Composition of Polynomials and Monads in Agda

```
{-# OPTIONS --without-K --rewriting #-}
module sum where

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

```
open import hott
open import poly
```

Composition and identity of polynomial endofunctors, and functoriality of composition:

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

_◁_ : ∀ {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'))
```

Associativity of ◁:

```
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
```

Left and right unit laws for ◁:

```
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
```

Restriction of ◁ to a monoidal product on $\mathbf{Poly}^{\mathbf{Cart}}$:

```
◁◁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)))

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
```

Proof of Theorem 4.1:

```
module PolyMonad {ℓ κ} (u : Poly ℓ κ) (univ : isUnivalent u)
                 (η : y ⇆ u) (ηcart : isCartesian u η)
                 (μ : (u ◃ u) ⇆ u) (μcart : isCartesian u μ) where

    idl : EqLens u (◃unitl u) (comp u (η ◃◃[ u ] (id u)) μ)
    idl = univ (◃unitlCart u) (compCartesian u (◃◃Cart u u ηcart (idCart u)) μcart)

    idr : EqLens u (◃unitr u) (comp u (id u ◃◃[ u ] η) μ)
    idr = univ (◃unitrCart u) (compCartesian u (◃◃Cart u u (idCart u) ηcart) μcart)

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

```
                (compCartesian u (◁◁Cart u u μcart (idCart u)) μcart)

open PolyMonad public
```

D The ⇈ Functor and Distributive Laws in Agda

This module sets up the necessary definitions and lemmas for the proof of Theorem 4.2.

```
{-# OPTIONS --without-K --rewriting #-}
module prod where

open import Agda.Primitive
open import Agda.Builtin.Sigma
open import Agda.Builtin.Unit
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
open import hott
open import poly
open import sum
```

Definition of the ⇈ functor:

```
_⇈_ : ∀ {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)))
```

Functoriality of ⇈:

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

Interaction of ⇈ with ◁ in its first argument, exhibiting ⇈ as a monoidal action, which moreover descends to a monoidal action on $\mathbf{Poly}^{\mathbf{Cart}}$:

```
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
```

Interaction of ⇈ with ◁ in its second argument, exhibiting ⇈ as a colax monoidal functor, which moreover descends to a colax monoidal functor on $\mathbf{Poly}^{\mathbf{Cart}}$:

```
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 putative distributive law induced by ⇈:

```
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)) )
```

Generalizing ⇈ to an action of the twisted arrow category of **Poly** on **Poly** and $\mathbf{Poly}^{\mathbf{Cart}}$:

```
_⇈[_][_]_ : ∀ {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) Γ x →
            snd k (γ (snd f a x))
               (transp (λ y → snd r' (fst k (γ y)))
                       (sym (snd e a x))
                       (Γ (snd h (fst f' (fst g a))
                                 (transp (snd q) (fst e a) x)))) )

    ⇈[]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
```

Extension of properties noted above for ⇈ to the generalized ⇈:

```
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)

⇈[]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 )
```

Equality of 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))))))
```

Converting from lenses out of _⇈[_][_]_ to distributors:

```
⇈→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)
```

Functoriality of distributors:

```
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))

    ↑↑→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
```

There are two distinct ways of composing distributors:

1. Given distributors $j1 : p \triangleleft s \leftrightarrows t \triangleleft q$ and $j2 : q \triangleleft u \leftrightarrows v \triangleleft r$, we obtain a distributor $p \triangleleft (s \triangleleft u) \leftrightarrows (t \triangleleft v) \triangleleft r$ as the composite

$$p \triangleleft (s \triangleleft u) \simeq (p \triangleleft s) \triangleleft u \xrightarrow{j1 \triangleleft u} (t \triangleleft q) \triangleleft u \simeq t \triangleleft (q \triangleleft u) \xrightarrow{j2} t \triangleleft (v \triangleleft r) \simeq (t \triangleleft v) \triangleleft r$$

```
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:

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

```
    ↑↑→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 $\triangleleft$:

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

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
```

Proof of Theorem 4.2:

```
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))
```