Parametricity via Cohesion

C.B. Aberlé

Abstract: Parametricity is a key metatheoretic property of type systems, from which one is able to derive strong uniformity/modularity properties of the structure of types within these systems. In recent years, various systems of dependent type theory have emerged with the aim of internalizing such parametric reasoning into their internal logic, employing various syntactic and semantic methods and concepts toward this end. This paper presents a first step toward the unification, simplification, and extension of these various methods for internalizing parametricity. Specifically, I argue that there is an essentially modal aspect of parametricity, which is intimately connected with the category-theoretic concept of cohesion.

On this basis, I describe a general categorical semantics for modal parametricity, develop a corresponding framework of axioms (with computational interpretations) in dependent type theory that can be used to internally represent and reason about such parametricity, and show this in practice by implementing these axioms in Agda and using them to verify parametricity theorems therein. In closing, I sketch the outlines of a more general synthetic theory of parametricity, and consider potential applications in domains ranging from homotopy type theory to the analysis of program modules.

The past, present & future of parametricity in type theory

Reynolds (1983) began his seminal introduction of the concept of *parametricity* with the declaration that "type structure is a syntactic discipline for enforcing levels of abstraction." In the ensuing decades, this idea of Reynolds' has been overwhelmingly vindicated by the success of type systems in achieving modularity and abstraction in domains ranging from programming to interactive theorem proving. Yet the fate of Reynolds' particular strategy for formalizing this idea is rather more ambiguous.

Reynolds' original analysis of parametricity targeted the polymorphic λ-calculus (aka System F). Intuitively, polymorphic programs in System F cannot inspect the types over which they are defined and so must behave *essentially the same* for all types at which they are instantiated. To make this intuitive idea precise, Reynolds posed an ingenious solution in terms of logical relations, whereby every System F type is equipped with a suitable binary relation on its inhabitants, such that all terms constructible in System F must preserve the relations defined on their component types. On this basis, Reynolds was able to establish many significant properties of the abstraction afforded by System F's type structure, e.g: all closed terms of type $\forall X.X \to X$ are extensionally equivalent to the identity function.

Reynolds' results have been extended to many type systems, programming languages, etc. However, as the complexity and expressivity of these systems increases, so too does the difficulty of defining parametricity relations for their types, and proving that the inhabitants of these types respect their corresponding relations. Moreover, the abstract, mathematical patterns underlying the definitions of such relations have not always been clear. The problem – from a certain point of view, at any rate – is that the usual theory of parametricity relations for type systems is "analytic," i.e. defined in terms of the particular constructs afforded by those systems, when what would be more desirable is a "synthetic" theory that boils the requirements for parametricity down to a few axioms, such that any type theory satisfying these axioms would necessarily enjoy parametricity and the expected consequences thereof.

The need for such an axiomatic specification of parametricity is made all the more apparent by several recent developments in *Homotopy Type Theory* and related fields, wherein *dependent* type theories capable of proving parametricity theorems *internally* for their own type structures have been developed and applied by various authors to solve some major open problems in these fields. Each such type theory has posed its own solution to the problem of internalizing parametricity, and although some commonalities exist between these systems, there is as yet no one framework that subsumes them all. In particular, the majority of these type theories have targeted only specific semantic models (usually some appropriately-chosen presheaf categories) rather than an axiomatically-defined *class* of models, and in this sense the analysis of parametricity offered by these type theories remains *analytic,* rather than *synthetic*. This in particular limits any insight into whether and how these appraoches to internal parametricity may be related to one another, generalized, or simplified.

As a first step toward the unification, simplification, and generalization of these approaches to internal parametricity, I propose an analysis of parametricity in terms of the category-theoretic concept of *cohesion*. Cohesion here refers to a particular relation between categories whereby one is equipped with an adjoint string of modalities that together make its objects behave like abstract spaces, whose points are bound together by some sort of *cohesion* that serves to constrain the structure-preserving maps that exist between these spaces. Such a notion of cohesion is reminiscent of the informal idea behind Reynolds' formulation of parametricity outlined above, particularly if one interprets the *cohesion* that binds types together as the structure of relations that inhere between them. Moreover, by inspection of the categorical models of extant type theories for internal parametricity, along with classical models of parametricity, one finds that many (if not quite all) of these exhibit cohesion, in this sense. The main contribution of this paper is thus to show that this basic setup of cohesion is essentially *all* that is needed to recover classical parametricity results internally in dependent type theory. As both an illustration and verification of this idea, this paper is also a literate Agda document wherein the axioms for and theorems resulting from such parametricity via cohesion have been fully formalized and checked for validity.

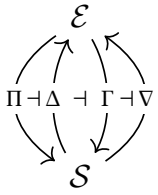
```
{-# OPTIONS --rewriting --cohesion --flat-split --without-K #-}
module parametricity-via-cohesion where

open import Agda.Primitive
open import Data.Empty
open import Agda.Builtin.Unit
open import Agda.Builtin.Bool
open import Agda.Builtin.Sigma
```

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

Cohesion and Parametricity

The notion of *cohesion* as an abstract characterization of when one category (specifically a topos) behaves like a category of spaces defined over the objects of another, is due primarily to Lawvere. The central concept of axiomatic cohesion is an arrangement of four adjoint functors as in the following diagram:



where $\mathcal{E}, \mathcal{S}$ are both topoi, Δ, ∇ are both fully faithful, and Π preserves finite products. Given such an arrangement, we think of the objects of $\mathcal{E}$ as *spaces* and those of $\mathcal{S}$ as *sets* (even if $\mathcal{S}$ is not the category of sets), where Γ is the functor that sends a space to its set of points, Δ sends a set to the corresponding *discrete space*, ∇ sends a set to the corresponding *codiscrete space*, and Π sends a space to its set of connected components. These in turn induce a string of adjoint modalities on $\mathcal{E}$:

$$\int \dashv \flat \dashv \sharp$$

where $\int = \Delta \circ \Pi$ and $\sharp = \nabla \circ \Gamma$ are idempotent monads, and $\flat = \Delta \circ \Gamma$ is an idempotent comonad.

A concrete example of cohesion comes from the category of reflexive graphs **RGph**, which is cohesive over the category of sets **Set**. Here, Γ is the functor that sends a reflexive graph to its set of vertices, Δ sends a set V to the "discrete" reflexive graph on V whose only edges are self-loops, ∇ sends V to the "codiscrete" graph where there is a unique edge between any pair of vertices, and Π sends a reflexive graph to its set of (weakly) connected components. It is worth noting, at this point, that many classical models of parametricity are based upon semantic interpretations of types as reflexive graphs, and this, I wish to argue is no accident, and the key property of reflexive graphs underlying such interpretations is precisely their cohesive structure.[1] More generally, for any base topos $\mathcal{S}$, we may construct its corresponding topos $\mathbf{RGph}(\mathcal{S})$ of *internal reflexive graphs*, which will similarly be cohesive over $\mathcal{S}$, so we can in fact use the language of such internal reflexive graphs to derive parametricity results for *any* topos (or indeed, any ∞-topos, as described below).

In fact, this same setup of cohesion is interpretable, *mutatis mutandis*, in the case where $\mathcal{E}, \mathcal{S}$ are not topoi, but rather ∞-*topoi*, i.e. models of homotopy type theory. This allows us to use the language of homotopy type theory – suitably extended with constructs for the above-described modalities (the $\flat$ modality in particular, which, for technical reasons, cannot be axiomatized directly in ordinary HoTT) – to work *synthetically* with the structure of such a cohesive ∞-topos. For present purposes, we accomplish this by working in Agda with the `--cohesion` and `--flat-split` flags enabled,

[1] Similarly, the category of bicubical sets is cohesive over that of cubical sets, a fact exploited by Nuyts, Devriese & Vezzossi in the semantics for their type theory for parametric quantifiers.

along with `--without-K`, which ensures compatibility with the treatment of propositional equality in HoTT.

I therefore begin by recalling some standard definitions from HoTT, which shall be essential in defining much of the structure to follow. Essentially all of these definitions have to do with the *identity type* former `_≡_` and its associated constructor `refl : ∀ {ℓ} {A : Set ℓ} {a : A} → a ≡ a`, as defined in the module `Agda.Builtin.Equality`:

```
module hott where
```

First of all, we have the induction principle for the identity type, aka the J rule:

```
J : ∀ {ℓ κ} {A : Set ℓ} {a : A}
    → (B : (b : A) → a ≡ b → Set κ)
    → {b : A} → (p : a ≡ b) → B a refl → B b p
J B refl b = b
```

As an immediate corollary of this, we obtain the operation of *tranport* as the recursor for the identity type:

```
transp : ∀ {ℓ κ} {A : Set ℓ} {a b : A}
         → (B : A → Set κ) → (a ≡ b) → B a → B b
transp B p b = J (λ a _ → B a) p b
```

Additionally, both `J` and `transp` are symmetric, and so can be applied "in the opposite direction":

```
J⁻¹ : ∀ {ℓ κ} {A : Set ℓ} {a : A}
      → (B : (b : A) → a ≡ b → Set κ)
      → {b : A} → (p : a ≡ b) → B b p → B a refl
J⁻¹ B refl b = b

transp⁻¹ : ∀ {ℓ κ} {A : Set ℓ} {a b : A}
           → (B : A → Set κ) → (a ≡ b) → B b → B a
transp⁻¹ B p b = J⁻¹ (λ a _ → B a) p b
```

Since all functions must preserve relations of identity, we may apply a function to both sides of an identification as follows:

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

The notion of *contractibility* then expresses the idea that a type is essentially uniquely inhabited.

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

Similarly, the notion of *equivalence* expresses the idea that a function between types has an *essentially unique* inverse. (Those familiar with HoTT may note that I use the *contractible fibres* definition of equivalence, as this shall be the most convenient to work with, for present purposes).

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

    mkInv : ∀ {ℓ κ} {A : Set ℓ} {B : Set κ}
            → (f : A → B) → isEquiv f → B → A
    mkInv f e b = fst (fst (e b))
```

```
open hott
```

The reader familiar with HoTT may note that, so far, we have not included anything relating to the *univalence axiom*, arguably the characteristic axiom of HoTT. In fact, this is by design, as a goal of the current formalization is to assume only axioms that can be given straightforward computational interpretations that preserve the property that every closed term of the ambient type theory evaluates to a *canonical normal form* (canonicity), so that these axioms give a *constructive* and *computationally sound* interpretation of parametricity. While the univalence axiom *can* be given a computational interpretation compatible with canonicity, as in Cubical Type Theory, doing so is decidedly *not* a straightforward matter. Moreover, it turns out that the univalence axiom is largely unneeded in what follows, save for demonstrating admissibility of some additional axioms which admit much more striaghtforward computational interpretations that are (conjecturally) compatible with canonicity. I thus shall have need for univalence only as a metatheoretic assumption. Nonetheless, for expositional purposes, it is useful to define univalence formally, for which purpose I include the following submodule:

```
module univalence where
```

To state univalence, we first define the operation that takes an identity to an equivalence between types:

```
    idToEquiv : ∀ {ℓ} (A B : Set ℓ)
                → A ≡ B → Σ (A → B) (λ f → isEquiv f)
    idToEquiv A B refl =
        ((λ a → a) , λ b → ((b , refl) , λ {(c , refl) → refl}))
```

The univalence axiom asserts that this map is itself an equivalence:

```
    postulate
        axiom : ∀ {ℓ} {A B : Set ℓ} → isEquiv (idToEquiv A B)
```

Intuitively, univalence allows us to convert equivalences between types into *identifications* of those types, which may then be transported over accordingly.

Having defined some essential structures of the language of HoTT, we may now proceed to similarly define some essential structures of the language of HoTT *with cohesive modalities*.

```
module cohesion where
```

Of principal importance here is the ♭ modality, which (intuitively) takes a type A to its corresponding *discretization* $\flat A$, wherein all cohesion between points has been eliminated. However, in order for this

operation to be well-behaved, A must not depend upon any variables whose types are not themselves *discrete*, in this sense. To solve this problem, the `--cohesion` flag introduces a new form of variable binding `@♭ x : X`, which metatheoretically asserts that x is an element of the discretization of X. In this case, we say that x is a *crisp* element of X.

With this notion in hand, we can define the ♭ modality as an operation on *crisp* types:

```
data ♭ {@♭ ℓ : Level} (@♭ A : Set ℓ) : Set ℓ where
    con : (@♭ x : A) → ♭ A
```

As expected, the ♭ modality is a comonad with the following counit operation ϵ:

```
ϵ : {@♭ l : Level} {@♭ A : Set l} → ♭ A → A
ϵ (con x) = x
```

A crisp type is then *discrete* precisely when this map is an equivalence:

```
isDiscrete : ∀ {@♭ ℓ : Level} → (@♭ A : Set ℓ) → Set ℓ
isDiscrete {ℓ = ℓ} A = isEquiv (ϵ {ℓ} {A})
```

```
open cohesion
```

Perhaps surprisingly, with these few definitions alone, we are already nearly in a position to derive parametricity theorems in Agda (and more generally, in any type theory supporting the above constructions). What more is needed is merely a way of detecting when the elements of a given type are *related,* or somehow bound together, by the cohesive structure of that type.

For this purpose, it is useful to take a geometric perspective upon cohesion, and correspondingly, parametricity. What we are after is essentially the *shape* of an abstract relation between points, and an object I in our cohesive topos $\mathcal{E}$ (correspondingly, a type in our type theory) which *classifies* this shape in other objects (types) in that maps $I \to A$ correspond to such abstract relations between points in A. For this purpose, I propose that the *shape* of an abstract relation is nothing other than a *line segment,* i.e. two distinct points which are somehow *connected*. By way of concrete example, in the topos of reflexive graphs **RGph**, the role of a classifier for this shape is played by the "walking edge" graph $\bullet \to \bullet$, consisting of two points and a single non-identity (directed) edge. More generally, using the language of cohesion, we can capture this notion of an abstract line segment in the following axiomatic characterization of I:

> I is an object of $\mathcal{E}$ that is *strictly bipointed* and *weakly connected*.

Unpacking the terms used in this characterization, we have the following:

- *Strictly bipointed* means that I is equipped with a choice of two elements $i_0, i_1 : I$, such that the proposition $(i_0 = i_1) \to \bot$ (i.e. $i_0 \neq i_1$) holds in the internal language of $\mathcal{E}$.
- *Weakly connected* means that the unit map $\eta : I \to \int I$ is essentially constant, in that it factors through a contractible object/type. Intuitively, this says that the image of I in $\int I$ essentially consists of a single connected component.

Note that the above-given example of the walking edge graph straightforwardly satisfies both of these requirements, as it consists of two distinct vertices belonging to a single (weakly) connected component. I also note in passing that, If the assumption of weak connectedness is strengthened to

strong connectedness – i.e. the object/type $\int I$ is itself contractible – then the existence of such an object I as above is equivalent to Lawvere's axiom of *sufficient cohesion,* as proved by him.

We can begin to formalize this idea in Agda by postulating a type I with two given elements i_0, i_1:

```
postulate
    I : Set₀
    i0 i1 : I
```

We could also, in principle, directly postulate strict bipointedness of I, as in the following module:

```
module strictBpt where
    postulate
        axiom : i0 ≡ i1 → ⊥
```

However, this is in fact unnecessary, as this axiom will instead follow from an equivalent formulation introduced in the following subsection – hence why this axiom is postulated in a separate module.

On the other hand, we do not yet have the capability to postulate the axiom of connectedness as written above, since we have not yet formalized the $\int$ modality. We could do so, but again, it is in fact better for present purposes to rephrase this axiom in an equivalent form involving only the $\flat$ modality, which can be done as follows:

> A crisp type A is connected if and only if, for every *discrete* type B, any function $A \to B$ is essentially constant, in the sense of factoring through a contractible type.

To see that this equivalence holds: in one direction, assume that A is weakly connected. Then for any map $f : A \to B$, by the adjunction $\int \dashv \flat$ and discreteness of B, there exist maps $f_\flat : A \to \flat B$ and $f^\int : \int A \to B$, such that following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{\eta} & \int A \\ {\scriptstyle f_\flat}\downarrow & \searrow^{f} & \downarrow{\scriptstyle f^\int} \\ \flat B & \xrightarrow{\epsilon} & B \end{array}$$

Then since by assumption η factors through a contractible type, so does f.

In the other direction, assume that every map $f : A \to B$ is essentially constant, for every discrete type B. Then in particular, the map $\eta : A \to \int A$ is essentially constant, since $\int A$ is discrete (as it lies in the image of the *discretization* functor Δ).

Hence the property of I being weakly connected is just as much a particular *relation* between I and all discrete types as it is a property of I itself. Specifically, if we think of maps $I \to A$ as abstract *relations* or *edges* between elements of A, then weak connectedness of I equivalently says that *all edges between elements of discrete types are constant*.

In order to conveniently express this property in Agda, it shall therefore be prudent first to introduce some additional constructs for ergonomically handling edges, analogous to the definition of *path* types in Cubical type theory. For this purpose, I now introduce a corresponding notion of *edge types*.

Edge Types

In principle, given $a, b : A$, we could define the type of edges from a to b in A as the type $\Sigma f : I \to A.(f\ i_0 = a) \times (f\ i_1 = b)$. However, experience with such a naïve formalization shows that it incurs a high number of laborious transportations along equalities that should be easy enough to infer automatically. Hence I instead follow the approach taken by cubical type theory and related systems, such as simplicial type theory, and give an explicit axiomatization for *edge types*, with corresponding rewrite rules to apply the associated equalities automatically:

```
postulate
  Edge : ∀ {ℓ} (A : I → Set ℓ) (a0 : A i0) (a1 : A i1) → Set ℓ
```

The introduction rule for edge types corresponds to *function abstraction*

```
  eabs : ∀ {ℓ} {A : I → Set ℓ}
         → (f : (i : I) → A i) → Edge A (f i0) (f i1)
```

and likewise, the elimination rule corresponds to *function application*.

```
  eapp : ∀ {ℓ} {A : I → Set ℓ} {a0 : A i0} {a1 : A i1}
         → (e : Edge A a0 a1) → (i : I) → A i
```

We may then postulate the usual β-law as an identity for this type, along with special identities for application to $i0$ and $i1$. All of these are made into rewrite rules, allowing Agda to apply them automatically, and thus obviating the need for excessive use of transport:

```
  ebeta : ∀ {ℓ} {A : I → Set ℓ} (f : (i : I) → A i)
          → (i : I) → eapp (eabs f) i ≡ f i
  {-# REWRITE ebeta #-}
  eapp0 : ∀ {ℓ} {A : I → Set ℓ} {a0 : A i0} {a1 : A i1}
          → (e : Edge A a0 a1) → eapp e i0 ≡ a0
  {-# REWRITE eapp0 #-}
  eapp1 : ∀ {ℓ} {A : I → Set ℓ} {a0 : A i0} {a1 : A i1}
          → (e : Edge A a0 a1) → eapp e i1 ≡ a1
  {-# REWRITE eapp1 #-}
```

(We could additionally postulate an η-law for edge types, analogous to the usual η-law for function types; however, this is unnecessary for what follows, and so I omit this assumption.)

With this formalization of edge types in hand, we can straightforwardly formalize the equivalent formulation of weak connectedness of I given above. For this purpose, we first define the map `idToEdge` that takes an identification $a{\equiv}b$ to an edge from a to b:

```
idToEdge : ∀ {ℓ} {A : Set ℓ} {a b : A}
           → a ≡ b → Edge (λ _ → A) a b
idToEdge {a = a} refl = eabs (λ _ → a)
```

A type A is *edge-discrete* if for all $a, b : A$ the mape `idToEdge` is an equivalence:

```
isEdgeDiscrete : ∀ {ℓ} (A : Set ℓ) → Set ℓ
isEdgeDiscrete {ℓ = ℓ} A =
```

```
    {a b : A} → isEquiv (idToEdge {ℓ} {A} {a} {b})
```

We then postulate the following axioms:

```
postulate
    edgeConst1 : ∀ {@♭ ℓ : Level} {@♭ A : Set ℓ} {a b : A}
                 → isDiscrete A → (e : Edge (λ _ → A) a b)
                 → Σ (a ≡ b) (λ p → idToEdge p ≡ e)
    edgeConst2 : ∀ {@♭ ℓ : Level} {@♭ A : Set ℓ} {a b : A}
                 → (dA : isDiscrete A) → (e : Edge (λ _ → A) a b)
                 → (q : a ≡ b) → (r : idToEdge q ≡ e)
                 → edgeConst1 dA e ≡ (q , r)
```

which together imply that, if A is discrete, then it is *edge-discrete*:

```
isDisc→isEDisc : ∀ {@♭ ℓ : Level} {@♭ A : Set ℓ}
                 → isDiscrete A → isEdgeDiscrete A
isDisc→isEDisc dA e =
    (edgeConst1 dA e , λ (p , q) → edgeConst2 dA e p q)
```

As it stands, we have not yet given a procedure for evaluating the axioms `edgeConst1` and `edgeConst2` when they are applied to canonical forms, which means that computation on these terms will generally get stuck and thus violate canonicity. Toward rectifying this, I prove a key identity regarding these axioms, add a further postulate asserting that this identity is equal to `refl`, and convert both of these to rewrite rules:

```
rwEdgeConst1 : ∀ {@♭ ℓ : Level} {@♭ A : Set ℓ} {a : A}
               → (dA : isDiscrete A)
               → edgeConst1 dA (eabs (λ _ → a)) ≡ (refl , refl)
rwEdgeConst1 {a = a} dA = edgeConst2 dA (eabs (λ _ → a)) refl refl
{-# REWRITE rwEdgeConst1 #-}

postulate
    rwEdgeConst2 : ∀ {@♭ ℓ : Level} {@♭ A : Set ℓ} {a : A}
                   → (dA : isDiscrete A)
                   → edgeConst2 dA (eabs (λ _ → a)) refl refl ≡ refl
    {-# REWRITE rwEdgeConst2 #-}
```

Although a full proof of canonicity is beyond the scope of this paper, I conjecture that adding these rules suffices to preserve canonicity, and I verify a few concrete cases of this conjecture later in the paper.

So much for the (weak) connectedness of I; let us now turn our attention to the other property we had previously stipulated of I, namely its *strict bipointedness*. As mentioned previously, we could simply postulate this stipulation directly as an axiom – however, for the purpose of proving parametricity theorems, a more prudent strategy is to instead formalize a class of I-indexed type families, whose computational behavior follows from this assumption (and which, in turn, implies it). Because these type families essentially correspond to the *graphs* of predicates and relations on arbitrary types, I refer to them as *graph types*.

Graph Types

For present purposes, we need only concern ourselves with the simplest class of graph types: *unary graph types*, which, as the name would imply, correspond to graphs of unary predicates. Given a type A, a type family $B : A \to \mathsf{Type}$, and an element $i : I$, the *graph type* $\mathsf{Gph}^1\ i\ A\ B$ is defined to be equal to A when i is i_0, and equivalent to $\Sigma x : A.Bx$ when i is i_1. Intuitively, an element of $\mathsf{Gph}^1\ i\ A\ B$ is a dependent pair whose second element *only exists when i is equal to i_1*. We may formalize this in Agda as follows, by postulating a rewrite rule that evaluates $\mathsf{Gph}^1\ i_0\ A\ B$ to A:

```
postulate
    Gph1 : ∀ {ℓ} (i : I) (A : Set ℓ) (B : A → Set ℓ) → Set (ℓ)

    g1rw0 : ∀ {ℓ} (A : Set ℓ) (B : A → Set ℓ)
          → Gph1 i0 A B ≡ A
    {-# REWRITE g1rw0 #-}
```

We then have the following introduction rule for elements of $\mathsf{Gph}^1\ i\ A\ B$, which are pairs where the second element of the pair only exists under the assumption that $i = i_1$. When $i = i_0$ instead, the pair collapses to its first element:

```
    g1pair : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ} (i : I)
           → (a : A) → (b : (i ≡ i1) → B a) → Gph1 i A B

    g1pair0 : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
            → (a : A) → (b : (i0 ≡ i1) → B a)
            → g1pair {B = B} i0 a b ≡ a
    {-# REWRITE g1pair0 #-}
```

The first projection from such a pair may then be taken no matter what i is, and reduces to the identity function when i is i_0:

```
    g1fst : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ} (i : I)
          → (g : Gph1 i A B) → A

    g1beta1 : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ} (i : I)
            → (a : A) → (b : (i ≡ i1) → B a)
            → g1fst i (g1pair {B = B} i a b) ≡ a
    {-# REWRITE g1beta1 #-}

    g1fst0 : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
           → (g : Gph1 i0 A B)
           → g1fst {B = B} i0 g ≡ g
    {-# REWRITE g1fst0 #-}
```

The second projection, meanwhile, may only be taken when i is equal to i_1:

```
    g1snd : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
          → (g : Gph1 i1 A B) → B (g1fst i1 g)
```

```
    g1beta2 : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
              → (a : A) → (b : (i1 ≡ i1) → B a)
              → g1snd (g1pair {B = B} i1 a b) ≡ b refl
    {-# REWRITE g1beta2 #-}
```

It is straightforward to see that the inclusion of graph types makes strict bipointedness of the interval provable, as follows:

```
strBpt : (i0 ≡ i1) → ⊥
strBpt p = g1snd (transp (λ i → Gph1 i ⊤ (λ _ → ⊥)) p tt)
```

And in fact, the converse holds under the assumption of univalence. Specifically, in the presence of univalence and the assumption of strict bipointedness for I, the type $\mathsf{Gph}^1\ i\ A\ B$ may be regarded as a computationally convenient shorthand for the type $\Sigma x : A.(i = i_1) \to Bx$, in much the same way as the type $\mathsf{Edge}\ A\ a_0\ a_1$ serves as shorthand for the type $\Sigma f : (\Pi i : I.Ai).(f\ i_0 = a_0) \times (f\ i_1 = a_1)$. This fact is due to the following equivalence

$$\begin{aligned} & \Sigma x : A.(i_0 = i_1) \to Bx \\ \simeq\ & \Sigma x : A.\bot \to Bx \\ \simeq\ & \Sigma x : A.\top \\ \simeq\ & A \end{aligned}$$

which, under univalence, becomes an identity between $\Sigma x : A.(i_0 = i_1) \to Bx$ and A, thereby justifying the use of this and associated identities as rewrite rules which, conjecturally, are fully compatible with canonictiy.

A few additional theorems, concerning identities between elements of graph types, are as follows, the latter of which I make into a rewrite rule for convenience:

```
apg1pair : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
           → {a b : A} {aB : B a} {bB : B b}
           → (p : a ≡ b) → aB ≡ transp⁻¹ B p bB
           → (i : I) → g1pair i a (λ _ → aB) ≡ g1pair i b (λ _ → bB)
apg1pair refl refl i = refl

apg1pair0 : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
            → {a b : A} {aB : B a} {bB : B b}
            → (p : a ≡ b) → (q : aB ≡ transp⁻¹ B p bB)
            → apg1pair p q i0 ≡ p
apg1pair0 refl refl = refl
{-# REWRITE apg1pair0 #-}
```

In principle, we could continue in this manner, and define graph types for relations of arities greater than 1 as well. However, unary graph types are sufficient for what is to follow. Speaking of which, having given appropriate axioms (and corresponding computation rules) to capture the desiderata that I be strictly bipointed and weakly connected, we are now in a position to prove some classical parametricity theorems using this structure.

Parametricity via Sufficient Cohesion

I begin this section with an old chestnut of parametricity theorems – a proof that any *polymorphic function* of type `(X : Set) → X → X` must be equivalent to the polymorphic identity function `λ X → λ x → x`.

```
PolyId : (ℓ : Level) → Set (lsuc ℓ)
PolyId ℓ = (X : Set ℓ) → X → X
```

Before proceeding with this proof, however, it will be prudent to consider the *meaning* of this theorem in the context of the cohesive type theory we have so-far developed. Specifically, I wish to ask: over which types should the type variable `X` in the type `(X : Set) → X → X` be considered as ranging in the statement of this theorem?

Although it is tempting to think that the answer to this question should be "all types" (or as close to this as one can get predicatively), if one considers the relation between our cohesive setup and Reynolds' original setup of parametricity, a subtler picture emerges. A type, in our framework, corresponds not to a type in the object language of e.g. bare sets, but rather to an object of the cohesive topos used to interpret the parametric structure of this object language, e.g. the category of reflexive graphs. In this sense, we should expect the parametricity result for the type `(X : Set) → X → X` to generally hold only for those types corresponding to those in the object language, i.e. the *discrete types*. Indeed, the discrete types by construction are those which cannot distinguish elements of other types belonging to the same connected component, which intuitively captures the essential idea of parametricity – that functions defined over these types must behave essentially the same for related inputs.

However, we cannot state this formulation of the theorem directly, since it would require us to bind X as `♭ X : Set`, which would kill all of the cohesive structure on `Set` and pose no restriction on the functions inhabiting this type. The solution, in this case, is to restrict the range of `X` to types which are *edge-discrete*, since this requirement can be stated even for `X` not crisp.

To prove the desired parametricity theorem for the type `PolyId` as above, then, we first prove a *substitution lemma* as follows:

> For any edge-discrete type `A` and any type family `B : A → Set` with `a : A`, there is a function `(α : PolyId) → B a → B(α A a)`

```
module paramId {ℓ} (A : Set ℓ) (edA : isEdgeDiscrete A) (B : A → Set ℓ)
                  (a : A) (b : B a) (α : PolyId ℓ) where
```

The key step in the proof of this lemma is to construct a "dependent edge" over the type family `Gph1 i A B : I → Set` as follows:

```
  lemma0 : (i : I) → Gph1 i A B
  lemma0 i = α (Gph1 i A B) (g1pair i a (λ _ → b))
```

Then taking the second projection of `lemma0` evaluated at `i1` yields an element of B evaluated at the first projection of `lemma0` evaluated at `i1`:

```
  lemma1 : B (g1fst i1 (lemma0 i1))
  lemma1 = g1snd (lemma0 i1)
```

And we can use `lemma0` to construct an edge from α `A a` to `g1fst i1 (lemma0 i1)` as follows:

```
lemma2 : Edge (λ _ → A) (α A a)
              (g1fst i1 (lemma0 i1))
lemma2 = eabs (λ i → g1fst i (lemma0 i))
```

And then since `A` is edge-discrete, this edge becomes an equality along which we can transport `lemma1`:

```
substLemma : B (α A a)
substLemma = transp⁻¹ B (mkInv idToEdge edA lemma2) lemma1
```

From this substitution lemma, it straightforwardly follows that any element of `PolyId` must be extensionally equivalent to the polymorphic identity function when evaluated at an edge-discrete type:

```
polyId : ∀ {ℓ} (A : Set ℓ) (edA : isEdgeDiscrete A) (a : A)
         → (α : PolyId ℓ) → α A a ≡ a
polyId A edA a α = paramId.substLemma A edA (λ b → b ≡ a) a refl α
```

Before we congratulate ourselves for proving this theorem, however, we ought to reflect on the significance of what we have proved. For we have proved *only* that the restrictions of elements of `PolyId` to edge-discrete types are equivalent to that of the polymorphic identity function. The theorem would then after all be trivial if it turned out that the only edge-discrete types were (e.g.) those containing at most one element (i.e. the *mere propositions* in the terminology of HoTT). To show that this is not the case, we make use of our assumption of *connectedness* for I, which we have already seen implies that every discrete type is edge-discrete. To give a concrete (non-trivial) example of this, I now show that the type of Booleans is discrete (hence edge-discrete) and use this to test the canonicity conjecture on a few simple cases:

```
module BoolDiscrete where
```

Showing that `Bool` is discrete is a simple matter of pattern matching

```
boolIsDisc : isDiscrete Bool
boolIsDisc false = (con false , refl) , λ { (con false , refl) → refl}
boolIsDisc true  = (con true  , refl) , λ { (con true , refl) → refl}
```

It follows that `Bool` is also edge-discrete and so the above parametricity theorem may be applied to it.

```
boolIsEDisc : isEdgeDiscrete Bool
boolIsEDisc = isDisc→isEDisc boolIsDisc

polyIdBool : (α : PolyId lzero) → (b : Bool) → α Bool b ≡ b
polyIdBool α b = polyId Bool boolIsEDisc b α
```

We can use this to check some specific cases of that the proof of `polyId` yields a canonical form (namely `refl`) when all of its inputs are themselves canonical forms:

```
shouldBeRefl1 : true ≡ true
shouldBeRefl1 = polyIdBool (λ X → λ x → x) true
```

```
shouldBeRefl2 : false ≡ false
shouldBeRefl2 = polyIdBool (λ X → λ x → x) false
```

Running Agda's normalization procedure on both of these terms shows that they do indeed evaluate to `refl`.

Parametricity & (Higher) Inductive Types

The foregoing proof of parametricity for the type of the polymorphic identity function – although theoretically significant, and illustrative of powerful techniques for proving parametricity – remains ultimately a toy example. To demonstrate the true power of this approach to parametricity, I turn now to some more intricate examples of its use, in proving universal properties for simplified presentations of inductive types and higher inductive types.

In general, it is easy to write down what should be the *recursion* principle for an inductive type generated by some set of constructors, but harder (though feasible) to derive the corresponding *induction* principle. When one begins to consider more complex generalizations of inductive types, such as higher inductive types, inductive-inductive types, etc, however, these difficulties begin to multiply. What would be ideal would be a way of deriving the induction principle for an inductive type from its *recursor,* hence requiring only the latter to be specified as part of the primitive data of the inductive type. However, in most systems of ordinary dependent type theory this is generally not possible (c.f. Geuvers). In HoTT, there is one way around this issue, due to Awodey, Frey & Speight, whereby additional *naturality* constraints are imposed upon the would-be inductive type, that serve to make the corresponding induction principle derivable from the recursor. However, when one goes on to apply this technique to *higher-inductive types,* which may specify constructors not only for *elements* of a type, but also for instances of its (higher) identity types, the complexity of these naturality conditions renders them impractical to work with. The ballooning complexity of these conditions is an instance of the infamous *coherence problem* in HoTT, whereby the complexity of coherence conditions for higher-categorical structures seemingly escapes any simple inductive definition.

As an alternative, let us consider ways of using *parametricity* to derive induction principles for inductive and higher inductive types, starting with the prototypical inductive type, the natural numbers $\mathbb{N}$.

First, we define the type of the recursor for $\mathbb{N}$:

```
RecN : Setω
RecN = ∀ {ℓ} (A : Set ℓ) → (A → A) → A → A
```

We may then postulate the usual constructors and identities for $\mathbb{N}$

```
postulate
    N : Set₀
    zero : N
    succ : N → N
    recN : N → RecN
    zeroβ : ∀ {ℓ} (A : Set ℓ) (f : A → A) (a : A)
          → recN zero A f a ≡ a
    {-# REWRITE zeroβ #-}
```

```
    succβ : ∀ {ℓ} (n : N) (A : Set ℓ) (f : A → A) (a : A)
          → recN (succ n) A f a ≡ f (recN n A f a)
    {-# REWRITE succβ #-}
    Nη : (n : N) → recN n N succ zero ≡ n
    {-# REWRITE Nη #-}
```

Note that among the identities postulated as rewrite rules for $\mathbb{N}$ is its η-law, i.e. $\mathsf{rec}\mathbb{N}\ n\ \mathbb{N}\ \mathsf{succ}\ \mathsf{zero} = n$ for all $n : \mathbb{N}$, as this will be important for deriving the induction principle for $\mathbb{N}$.

From here, we may proceed essentially as in the proof of parametricity for `PolyId`, by proving an analogous *substitution lemma* for `RecN`, following essentially the same steps:

```
module paramN {ℓ} (A : Set ℓ) (edA : isEdgeDiscrete A) (B : A → Set ℓ)
                  (f : A → A) (ff : (x : A) → B x → B (f x))
                  (a : A) (b : B a) (α : RecN) where

    lemma0 : (i : I) → Gph1 i A B
    lemma0 i = α (Gph1 i A B)
                 (λ g →
                    let g' j q = transp (λ k → Gph1 k A B) q g in
                    g1pair i (f (g1fst i g))
                           (λ p →
                              J⁻¹ (λ j q → B (f (g1fst j (g' j q)))) p
                                  (ff (g1fst i1 (g' i1 p)) (g1snd (g' i1 p)))))
                 (g1pair i a (λ _ → b))

    lemma1 : B (g1fst i1 (lemma0 i1))
    lemma1 = g1snd (lemma0 i1)

    lemma2 : Edge (λ _ → A) (α A f a)
                  (g1fst i1 (lemma0 i1))
    lemma2 = eabs (λ i → g1fst i (lemma0 i))

    substLemma : B (α A f a)
    substLemma =
        transp⁻¹ B (mkInv idToEdge edA lemma2) lemma1
```

In order to apply this lemma to $\mathbb{N}$ itself, we must further postulate that $\mathbb{N}$ is edge-discrete (in fact, one could show by induction that N is *discrete*, hence edge-discrete by the assumption of connectedness for I; however, since we have not yet proven induction for $\mathbb{N}$, we must instead take this result as an additional axiom, from which induction on $\mathbb{N}$ will follow).

```
postulate
    edN1 : ∀ {m n : N} (e : Edge (λ _ → N) m n)
         → Σ (m ≡ n) (λ p → idToEdge p ≡ e)
    edN2 : ∀ {m n : N} (e : Edge (λ _ → N) m n)
         → (q : m ≡ n) (r : idToEdge q ≡ e)
```

```
          → edN1 e ≡ (q , r)

edN : isEdgeDiscrete N
edN e = (edN1 e , λ (q , r) → edN2 e q r)

rwEDN1 : (n : N) → edN1 (eabs (λ _ → n)) ≡ (refl , refl)
rwEDN1 n = edN2 (eabs (λ _ → n)) refl refl
{-# REWRITE rwEDN1 #-}

postulate
    rwEDN2 : (n : N) → edN2 (eabs (λ _ → n)) refl refl ≡ refl
    {-# REWRITE rwEDN2 #-}
```

Induction for $\mathbb{N}$ then follows as a straightforward consequence of the substitution lemma:

```
indN : (P : N → Set)
       → ((n : N) → P n → P (succ n))
       → P zero → (n : N) → P n
indN P ps pz n =
    paramN.substLemma N edN P succ ps zero pz (recN n)
```

Note that our use of the η-law for $\mathbb{N}$ as a rewrite rule is critical to the above proof, since otherwise in the last step, we would obtain not a proof of `P n`, but rather `P (recN n N succ zero)`.

As in the case of the parametricity theorem for `PolyId` we may test that the derived induction principle for $\mathbb{N}$ evaluates canonical forms to canonical forms, as in the following example, based on the usual inductive proof that `zero` is an identity element for addition on the right:

```
module Nexample where
    _plus_ : N → N → N
    m plus n = recN m N succ n

    zeroIdR : (n : N) → n plus zero ≡ n
    zeroIdR n = indN (λ m → m plus zero ≡ m)
                     (λ m p → ap succ p) refl n

    shouldBeRefl3 : succ (succ zero) ≡ succ (succ zero)
    shouldBeRefl3 = zeroIdR (succ (succ zero))
```

Running Agda's normalization procedure on `shouldBeRefl3` again confirms that it evaluates to `refl`.

Moving on, then, from inductive types to *higher* inductive types, I now consider deriving the induction principle for the *circle* S^1, defined to be the type freely generated by a single basepoint $\mathsf{base} : S^1$, with a nontrivial identification $\mathsf{loop} : \mathsf{base} = \mathsf{base}$. The recursor for S^1 thus has the following type:

```
RecS¹ : Setω
RecS¹ = ∀ {ℓ} (A : Set ℓ) → (a : A) → a ≡ a → A
```

Then, just as before, we may postulate the corresponding constructors and $\beta\eta$-laws for S^1:

```
postulate
    S¹ : Set₀
    base : S¹
    loop : base ≡ base
    recS¹ : S¹ → RecS¹
    baseβ : ∀ {ℓ} (A : Set ℓ) (a : A) (l : a ≡ a)
          → recS¹ base A a l ≡ a
    {-# REWRITE baseβ #-}
    loopβ : ∀ {ℓ} (A : Set ℓ) (a : A) (l : a ≡ a)
          → ap (λ s → recS¹ s A a l) loop ≡ l
    {-# REWRITE loopβ #-}
    S¹η : (s : S¹) → recS¹ s S¹ base loop ≡ s
    {-# REWRITE S¹η #-}
```

The proof of induction for S^1 is then in essentials the same as the one given above for $\mathbb{N}$. We begin by proving a *substitution lemma* for $\mathtt{RecS}^1$, following exactly the same steps as in the proof of the corresponding theorem for $\mathtt{Rec}N$:

```
module paramS¹ {ℓ} (A : Set ℓ) (edA : isEdgeDiscrete A) (B : A → Set ℓ)
                   (a : A) (b : B a) (l : a ≡ a)
                   (lB : b ≡ transp⁻¹ B l b) (α : RecS¹) where

    lemma0 : (i : I) → Gph1 i A B
    lemma0 i = α (Gph1 i A B)
                 (g1pair i a (λ _ → b))
                 (apg1pair l lB i)

    lemma1 : B (g1fst i1 (lemma0 i1))
    lemma1 = g1snd (lemma0 i1)

    lemma2 : Edge (λ _ → A) (α A a l)
                  (g1fst i1 (lemma0 i1))
    lemma2 = eabs (λ i → g1fst i (lemma0 i))

    substLemma : B (α A a l)
    substLemma =
        transp⁻¹ B (mkInv idToEdge edA lemma2) lemma1
```

We then postulate that S^1 is edge-discrete, as before, in order to apply this lemma to S^1 itself:

```
postulate
    edS¹1 : ∀ {s t : S¹} (e : Edge (λ _ → S¹) s t)
          → Σ (s ≡ t) (λ p → idToEdge p ≡ e)
    edS¹2 : ∀ {s t : S¹} (e : Edge (λ _ → S¹) s t)
          → (q : s ≡ t) (r : idToEdge q ≡ e)
```

```
            → edS¹1 e ≡ (q , r)

edS¹ : isEdgeDiscrete S¹
edS¹ e = (edS¹1 e , λ (q , r) → edS¹2 e q r)

rwEDS¹1 : (s : S¹) → edS¹1 (eabs (λ _ → s)) ≡ (refl , refl)
rwEDS¹1 s = edS¹2 (eabs (λ _ → s)) refl refl
{-# REWRITE rwEDS¹1 #-}

postulate
    rwEDS¹2 : (s : S¹) → edS¹2 (eabs (λ _ → s)) refl refl ≡ refl
    {-# REWRITE rwEDS¹2 #-}
```

And then the desired induction principle for S^1 follows straightforwardly:

```
indS¹ : (P : S¹ → Set)
        → (pb : P base) → pb ≡ transp⁻¹ P loop pb
        → (s : S¹) → P s
indS¹ P pb pl s =
    paramS¹.substLemma S¹ edS¹ P base pb loop pl (recS¹ s)
```

Although it is not in general possible to verify that this same construction is capable of deriving induction principles for *all* higher inductive types – essentially because there is as yet no well-established definition of what higher inductive types are *in general* – there appears to be no difficulty in extending this method of proof to all known classes of higher inductive types. Moreover, that the proof of induction for S^1 is essentially no more complex than that for $\mathbb{N}$ suggests that this method is capable of taming the complexity of coherences for such higher inductive types, and in this sense provides a solution to this instance of the coherence problem.

Toward a synthetic theory of parametricity

In closing, I consider the significance of the system of axioms outlined above, both as a sketch of a more general *synthetic* theory of parametricity, and in relation to other such theories.

The theory so-far developed essentially gives a synthetic framework for working in the internal language of a (weakly) *sufficiently cohesive* ∞*-topos*. This framework in turn proves capable of deriving significant parametricity results internally, with immediate applications in e.g. resolving coherence problems having to do with higher inductive types. It remains to be seen what further applications can be developed for this theory and its particular approach to parametricity. From this perspective, it is profitable to survey what other approaches there are to internalizing parametricity theorems in dependent type theory, and how they might be related to the one given in this paper.

Cohesion & Gluing

In recent years, there has been some work toward a synthetic theory of parametricity in terms of the topos-theoretic construction of *Artin Gluing*. This approach, outlined iniitally by Sterling in his

thesis and spearheaded by Sterling and his collaborators as part of his more general programme of *Synthetic Tait Computability* (STC), works in the internal language of a topos equipped with two (mere) propositions L and R, that are *mutually exclusive* in that $L \wedge R \to \bot$. The central idea of this approach is to synthetically reconstruct the usual account of parametricity in terms of logical relations by careful use of the *open* and *closed* modalities induced by these propositions. In this context, the *open* modality corresponding to a proposition ϕ defines a subuniverse of types which becomes trivial (i.e. all types in this universe become contractible) in any context where ϕ is false, while, dually, the *closed* modality corresponding to ϕ defines a subuniverse which becomes trivial in any context where ϕ is true. These modalities thus play a similar role in STC to the *graph types* introduced above. Using this approach, one can e.g. prove representation independence theorems for module signatures as in Sterling & Harper.

Clearly, there is some affinity between the approach to parametricity in terms of gluing and that in terms of cohesion presented above, not least of which having to do with the fact that these two approaches both make extensive use of *modalities*. Yet the modalities used in STC are of a rather different sort than the *cohesive* modalities used in the above, since the former arise from (mere) propositions, while the latter generally do not. However, there is a deeper sense in which these two approaches are related, which is that every sufficiently cohesive topos contains a model of Sterling's setup of STC for parametricity, as follows: given a sufficiently topos $\mathcal{E}$ over some base topos $\mathcal{S}$, for any strictly bipointed object $I \in \mathcal{E}$ with distinguished points $i_0, i_1 : I$, the *slice topos* $\mathcal{E}/I$, whose internal language corresponds to that of $\mathcal{E}$ extended with an arbitrary element $i : I$, contains two mutually exclusive propositions, namely $i = i_0$ and $i = i_1$. Hence all of the parametricity theorems available in STC can be recovered in the above-given framework (in particular, instances of closed modalities can be encoded as higher inductive types, which, as we have just seen, are easily added to the above framework).

On the other hand, there is seemingly no analogue in STC of the axiom of connectedness for the interval, and its consequent parametricity theorems, most notably the above-mentioned derivability of induction principles for higher inductive types. This suggests that in these latter results, which are intimately connected with the *coherence problem* of HoTT, the structure of *cohesion* and its relation to parametricity plays an essential role. Nonetheless, it remains interesting to consider how parametricity via cohesion and parametricity via gluing may yet prove to be related, and one may hope for a fruitful cross-pollination between these two theories.

Cohesion & Coherence

As just mentioned, there appears to be an intimate link between *parametricity*, *cohesion*, and (higher) *coherence*, as demonstrated e.g. by the above-given proof of induction for S^1. The existence of such a link is further supported by other recent developments in HoTT and elsewhere. E.g. Cavallo & Harper have used their system of internal parametricity for cubical type theory to derive coherences for the smash product. More recently still, Kolomatskaia & Shulman have introduced their system of *displayed* type theory, which utilizes yet another form of internal parametricity to solve the previously-open problem of representing *semi-simplicial types* in HoTT.

In outlining the above framework for parametricity in terms of cohesion, I hope to have taken a first step toward the unification of these various systems that use parametricity to tackle instances of the coherence problem. Cavallo & Harper's system is readily assimilated to this framework, as are

other related systems that take a *cubical* approach to internal parametricity, such as that of Nuyts, Devriese & Vezzossi, since these all take their semantics in various topoi of bicubical sets, which all are sufficiently cohesive over corresponding topoi of cubical sets. On the other hand, Kolomatskaia & Shulman's system cannot be assimilated in quite the same way, since their system takes its semantics in the ∞-topos of augmented semisimplicial spaces, which is *not* cohesive over spaces. It thus remains to be seen if the above framework can be generalized so as to be inclusive of this example as well. Alternatively, one might seek to generalize or modify Kolomatskaia & Shulman's system so as to be interpretable in an ∞-topos that *is* cohesive over spaces, e.g. the ∞-topos of simplicial spaces. If this latter proves feasible, then this in turn would reveal yet further connections between parametricity via cohesion and another prominent attempt at a solution to the coherence problem, namely Riehl & Shulman's *simplicial type theory*, which takes its semantics in simplicial spaces.

It appears, in all these cases, that parametricity is *the* tool for the job of taming the complexity of higher coherences in HoTT and elsewhere. In this sense, Reynolds was right in thinking that parametricity captures a fundamental property of abstraction, for, as any type theorist worth their salt knows, abstraction is ultimately the best tool we have for managing complexity.

Acknowledgements

The origins of this paper trace to research I did as an undergraduate at Merton College, Oxford, in the Summer of 2020, as part of the Merton College Summer Projects Scheme. Naturally, this work was conducted at a time of considerable stress for myself and the world at large, and I am massively indebted to the academic support staff at Merton, who were immensely helpful in keeping me afloat at that time. I am particularly grateful to Katy Fifield, Jemma Underdown, and Jane Gover for the support they provided to me in the course of my undergraduate studies. More recently, I am grateful to Frank Pfenning and Steve Awodey for their encouragement in continuing to pursue this line of research.