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 proofs and programs 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 properties of Agda programs. 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 linear programs.

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 in terms of logical relations 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.

For this purpose, a fruitful strategy is to consider ways of *internalizing* parametricity into the internal logic of a dependent type theory. Parametricity proofs are typically conducted in the metatheory of a particular type system, but if that type system is expressive enough to encode mathematical propositions in its types – as is the case for dependent type theory – there exists the possibility of instead defining and proving parametricity properties within the system itself. By considering the sorts of constructs a dependent type theory must contain in order to afford such internal parametric reasoning, we may arrive at an axiomatic specification of parametricity itself, as desired.

However, such internalization of parametricity into dependent type theory poses challenges of its own, owing to the fact that internal parametricity may be applied to itself, generating a complex, higher-dimensional structure of relations on types (c.f. Bernardy & Moulin). Various type theories for internal parametricity have thus been proposed, each posing its own solution to taming this complexity. So far, however, the majority of these type theories have targeted only specific 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 suspiciously 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 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 in terms of 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.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:

$$\begin{tikzcd}
\mathcal{E} \arrow[d, bend right=70, "\Pi"'] \arrow[d, bend left=25, "\Gamma"] \\
\mathcal{S} \arrow[u, bend left=25, "\Delta"] \arrow[u, bend right=70, "\nabla"']
\end{tikzcd}$$

$$\Pi \dashv \Delta \dashv \Gamma \dashv \nabla$$

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, $\flat = \Delta \circ \Gamma$ is an idempotent comonad, and both $\flat$ and $\sharp$ preserve finite limits.

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

[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.

homotopy type theory – suitably extended with constructs for the above-described modalities (the ♭ 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, 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 operation of *transport*, which embodies the principle of *indiscernability of identicals* (i.e. if `a0 = a1` and `B a1` then `B a0`):

```
    transp : ∀ {ℓ κ} {A : Set ℓ} {a0 a1 : A}
             → (B : A → Set κ) → (a0 ≡ a1) → B a1 → B a0
    transp B refl b = b
```

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 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 commuting diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\eta"] \arrow[d, "f_\flat"] \arrow[dr, "f"] & \int A \arrow[d, "f^\int"] \\
\flat B \arrow[r, "\epsilon"] & B
\end{tikzcd}$$

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

I begin with an exposition of 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 $B : A \to \text{Type}$, 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 : (i1 ≡ i0) → ⊥
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

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

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

In addition to unary graph types, we also have *binary* graphs types for representing graphs of binary relations. That is, given types A, B, a type family $C : A \to B \to \mathsf{Type}$, and elements $i, j : I$, there is a type $\mathsf{Gph}^2\ i\ j\ A\ B\ C$. Intuitively, $\mathsf{Gph}^2\ i\ j\ A\ B\ C$ is a type of dependent triples whose first element (of type A) exists only under the assumption that $i = i_0$, whose second element exists only under the assumption that $j = i_1$, and whose third element, which may depend upon the first and second, exists only under the conjunction of these assumptions. Hence $\mathsf{Gph}^2\ i_0\ i_0\ A\ B\ C$ is equal to A while $\mathsf{Gph}^2\ i_1\ i_1\ A\ B\ C$ is equal to B. This intuitive description is formalized in the following rules, which are entirely analogous to those for unary graph types:

```
postulate
    Gph2 : ∀ {ℓ} (i j : I) (A : Set ℓ) (B : Set ℓ)
           → (C : A → B → Set ℓ) → Set ℓ
    g2rw00 : ∀ {ℓ} (A : Set ℓ) (B : Set ℓ) (C : A → B → Set ℓ)
             → Gph2 i0 i0 A B C ≡ A
    {-# REWRITE g2rw00 #-}
    g2rw11 : ∀ {ℓ} (A : Set ℓ) (B : Set ℓ) (C : A → B → Set ℓ)
             → Gph2 i1 i1 A B C ≡ B
    {-# REWRITE g2rw11 #-}

    g2triple : ∀ {ℓ} (i j : I) {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
               → (a : (i ≡ i0) → A) → (b : (j ≡ i1) → B)
               → (c : (p : i ≡ i0) (q : j ≡ i1) → C (a p) (b q))
               → Gph2 i j A B C
    g2triple00 : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
                 → (a : (i0 ≡ i0) → A) → (b : (i0 ≡ i1) → B)
                 → (c : (p : i0 ≡ i0) (q : i0 ≡ i1) → C (a p) (b q))
                 → g2triple i0 i0 {C = C} a b c ≡ a refl
    {-# REWRITE g2triple00 #-}
    g2triple11 : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
                 → (a : (i1 ≡ i0) → A) → (b : (i1 ≡ i1) → B)
                 → (c : (p : i1 ≡ i0) (q : i1 ≡ i1) → C (a p) (b q))
                 → g2triple i1 i1 {C = C} a b c ≡ b refl
    {-# REWRITE g2triple11 #-}

    g2fst : ∀ {ℓ} (j : I) {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
            → Gph2 i0 j A B C → A
```

```
g2beta1 : ∀ {ℓ} (j : I) {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
          → (a : (i0 ≡ i0) → A) → (b : (j ≡ i1) → B)
          → (c : (p : i0 ≡ i0) (q : j ≡ i1) → C (a p) (b q))
          → g2fst j (g2triple i0 j {C = C} a b c) ≡ a refl
{-# REWRITE g2beta1 #-}
g2fst00 : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
          → (g : Gph2 i0 i0 A B C) → g2fst i0 {B = B} {C = C} g ≡ g
{-# REWRITE g2fst00 #-}

g2snd : ∀ {ℓ} (i : I) {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
        → Gph2 i i1 A B C → B
g2beta2 : ∀ {ℓ} (i : I) {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
          → (a : (i ≡ i0) → A) → (b : (i1 ≡ i1) → B)
          → (c : (p : i ≡ i0) (q : i1 ≡ i1) → C (a p) (b q))
          → g2snd i (g2triple i i1 {C = C} a b c) ≡ b refl
{-# REWRITE g2beta2 #-}
g2snd11 : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
          → (g : Gph2 i1 i1 A B C) → g2snd i1 {A = A} {C = C} g ≡ g
{-# REWRITE g2snd11 #-}

g2thrd : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
         → (g : Gph2 i0 i1 A B C) → C (g2fst i1 g) (g2snd i0 g)
g2beta3 : ∀ {ℓ} {A : Set ℓ} {B : Set ℓ} {C : A → B → Set ℓ}
          → (a : (i0 ≡ i0) → A) → (b : (i1 ≡ i1) → B)
          → (c : (p : i0 ≡ i0) (q : i1 ≡ i1) → C (a p) (b q))
          → g2thrd (g2triple i0 i1 {C = C} a b c) ≡ c refl refl
{-# REWRITE g2beta3 #-}
```

In principle, we could continue in this manner, and define graph types for relations of any arity. However, for present purposes, unary and binary graph types are sufficient. The former are useful for proving theorems to do with *unary parametricity*, while the latter play a similar role in theorems based on *binary parametricity*. 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 finally 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.

The overall strategy of this proof is to

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

    lemma1 : B (g1fst i1 (α (Gph1 i1 A B) (g1pair i1 a (λ _ → b))))
    lemma1 = g1snd (α (Gph1 i1 A B) (g1pair i1 a (λ _ → b)))

    lemma2 : Edge (λ _ → A)
                  (α A a)
                  (g1fst i1 (α (Gph1 i1 A B) (g1pair i1 a (λ _ → b))))
    lemma2 = eabs (λ i → g1fst i (α (Gph1 i A B) (g1pair i a (λ _ → b))))

    lemma3 : B (α A a)
    lemma3 = transp B (mkInv idToEdge edA lemma2) lemma1

polyId : ∀ {ℓ} (A : Set ℓ) (edA : isEdgeDiscrete A) (a : A)
        → (α : PolyId ℓ) → α A a ≡ a
polyId A edA a α = paramId.lemma3 A edA (λ b → b ≡ a) a refl α
```

Some Applications

Impredicative Encodings

Modularity & Coinduction

Toward a synthetic theory of parametricity