Making all equalities equal

Guilherme Horta Alvares da Silva

November 2, 2021

Abstract

1 Equalities

There are multiple ways of defining equalities in a theorem prover. In the next sections, they will be defined.

1.1 Imports

First, it will be necessary to give some agda arguments:

```
{-# OPTIONS --cubical --cumulativity #-}
module paper where
```

The cubical flag is necessary because we are using cubical equality, and the cumulativiy flag is also necessary for level subtyping,

```
open import Agda.Primitive.Cubical using (I; i0; i1)
```

This library loads Cubical Agda Primitives as the equality interval.

1.2 Martin-Löf Equality

At the begin of Agda and in most theorems proves, equality is given by Martin-Löf's definition:

```
module Martin-Löf {ℓ} {A : Set ℓ} where

  data _≡_ (x : A) : A → Set ℓ where
    refl : x ≡ x
```

This equality is very convenient in proof assistances like Agda because it is possible to pattern match using them:

```
private variable
  x y z : A

sym : x ≡ y → y ≡ x
sym refl = refl

trans : x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

But the problem of this equality is that it does not handle extensionality and other axioms very well.

```
module FunExt {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} where
  open Martin-Löf

  funExt-Type = {f g : A → B}
    → ((x : A) → f x ≡ g x) → f ≡ g
```

1.3 Cubical Equality

To solve this problem, Agda adopted cubical type theory that equality is a function from the path to type:

```
module CubicalEquality {ℓ} {A : Set ℓ} where
  postulate
    PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

  _≡_ : A → A → Set ℓ
  _≡_ = PathP λ _ → A
```

From this equality, I will define reflection, symmetry and extensionality:

```
module CubicalResults {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} where
  open import Cubical.Core.Primitives

  private variable
    x y z : A

  refl : x ≡ x
  refl {x = x} = λ _ → x

  sym : x ≡ y → y ≡ x
  sym p i = p (˜ i)

  funExt : {f g : A → B}
    → ((x : A) → f x ≡ g x) → f ≡ g
  funExt p i x = p x i
```

The operator ∼ invert the interval. If the interval i goes from *i0* to *i1*, the interval ∼ i goes from *i1* to *i0*.

1.4 Leibniz equality

Leibniz equality is defined in this way: If a is equal to b, then for every propositional P, if $P\ a$, then $P\ b$. The main idea is that if both values are equal, then they are seen equal for every angle.

```
module LeibnizEquality {A : Set} where
  _≐_ : A → A → Set₁
  a ≐ b = (P : A → Set) → P a → P b
```

2 Joining all equalities

All equalities have something in common. They are all equal to each other. So it will be defined as a common record that all equalities should have. In the next definition, all equalities are equal to cubical equality:

```
open import Cubical.Foundations.Prelude
open import Cubical.Foundations.Isomorphism
open import Cubical.Foundations.Equiv
open import Cubical.Foundations.Univalence
open import Cubical.Foundations.Function
open import Cubical.Data.Equality

module _ {a ℓ} {A : Set a} where
  ≜-Type = A → A → Set ℓ
  private
    ℓ₁ = ℓ-max a ℓ

  record IsEquality (_≜_ : ≜-Type) : Set (ℓ-suc (ℓ-max a ℓ)) where
    constructor eq
    field
      ≜-≡-≡ : ∀ {x y} → let
        x≡y : Type ℓ₁
        x≡y = x ≡ y

        x≜y : Type ℓ₁
        x≜y = x ≜ y

        in _≡_ {ℓ-suc ℓ₁} x≜y x≡y

    ≡-≡-≜ : {x y : A} → let
      x≡y : Type (ℓ-max a ℓ)
      x≡y = x ≡ y
      x≜y = x ≜ y
      in x≡y ≡ x≜y
```

```
    ≡-≡-≙ = sym ≙-≡-≡

  module _ {_≙_ : ≙-Type} where
    sym-Equality : (≡-≡-≙ : {x y : A} → let
        x≡y : Type (ℓ-max a ℓ)
        x≡y = x ≡ y
        x≙y = x ≙ y
        in x≡y ≡ x≙y)
        → IsEquality _≙_
    sym-Equality ≡-≡-≙ = eq (sym ≡-≡-≙)

  record Equality : Set (ℓ-suc (ℓ-max a ℓ)) where
    constructor eqC
    field
        _≙_ : ≙-Type
        { isEquality } : IsEquality _≙_

  EqFromInstance : {≙ : ≙-Type} → IsEquality ≙ → Equality
  EqFromInstance inst = eqC _ { inst }

  eqsEqual : (_≙₁_ _≙₂_ : ≙-Type)
    { ≙₁-eq : IsEquality _≙₁_ }
    { ≙₂-eq : IsEquality _≙₂_ }
    → ∀ {x y} → let
        x≙₁y : Type ℓ₁
        x≙₁y = x ≙₁ y

        x≙₂y : Type ℓ₁
        x≙₂y = x ≙₂ y

        in _≡_ {ℓ-suc ℓ₁} x≙₁y x≙₂y
  eqsEqual _ _ { eq ≙-≡-≡₁ } { eq ≙-≡-≡₂ } = ≙-≡-≡₁ ∙ sym ≙-≡-≡₂
```

It will be defined for each equality, its instance:

2.1 Cubical Equality

The simplest example is the cubical equality hence this equality is already equal to itself.

```
module _ {a} {A : Set a} where
  instance
    ≡-IsEquality : IsEquality {A = A} _≡_
    ≡-IsEquality = eq refl
  ≡-Equality : Equality {ℓ = a}
  ≡-Equality = eqC _≡_
```

2.2 Martin-Löf equality

The proof of Martin-Löf equality is more difficult, but it is already in Cubical library as p-c.

```
instance
  ≡p-IsEquality : IsEquality {A = A} _≡p_
  ≡p-IsEquality = sym-Equality p-c
≡p-Equality : Equality {ℓ = a}
≡p-Equality = eqC _≡p_
```

2.3 Isomorphism

The isomorphism is an equality between types.

```
module _ {ℓ} where
  univalencePath' : {A B : Type ℓ} → (A ≡ B) ≡ (A ≃ B)
  univalencePath' {A} {B} =
    ua {ℓ-suc ℓ} {A ≡ B} {A ≃ B} (compEquiv (univalence {ℓ} {A} {B})
    (isoToEquiv (iso {ℓ} {ℓ-suc ℓ}
    (λ x → x) (λ x → x) (λ b i → b) λ a i → a)))
```

univalencePath is already defined in Agda library, but with $A \simeq B$ instead of *Lifted* $(A \simeq B)$. This change can be done because of the cumulativity flag.

```
instance
  ≃-IsEquality : IsEquality
    {A = Type ℓ} _≃_
  ≃-IsEquality = sym-Equality univalencePath'
≃-Equality : Equality {ℓ = ℓ}
≃-Equality = eqC _≃_
```

2.4 Leibniz Equality

The hardest equality to proof that is equalty is the Leibniz Equality.

```
liftIso : ∀ {a b} {A : Type a} {B : Type b}
  → Iso {a} {b} A B → Iso {ℓ-max a b} {ℓ-max a b} A B
liftIso {a} f = iso fun inv
  (λ x i → rightInv x i) (λ x i → leftInv x i)
```

This liftIso will be used to lift the Isomorphism to types of the same maximum level of both.

```
where open Iso f
```

```
open import leibniz
open Leibniz
```

It is importing the definition of Leibniz equality made by [?]. In this work, there is already a proof of the isomorphism between the Leibniz and the Martin-Löf equality.

```
module FinalEquality {A : Set} where
  open MainResult A

  ≐≅≡ : ∀ {a b} → Iso (a ≐ b) (a ≡p b)
  ≐≅≡ = iso j i (ptoc ∘ ji) (ptoc ∘ ij)
```

In Cubical Library, the definition of isomorphism uses cubical equality instead of Martin-Löf equality when we have to proof that ∀ *x* → *from (to x)* ≡ *x* and ∀ *x* → *to (from x)* ≡ *x*. ptoc is necessary to do this convertion from these equalities.

```
≐≡≡ : ∀ {a b} → (a ≐ b) ≡c (a ≡p b)
≐≡≡ = let lifted = liftIso ≐≅≡ in isoToPath lifted
```

Using the univalence and liftIso defined previously, it is possible to transform the isormorphism into an equality.

```
open IsEquality

instance
  ≐-IsEquality : IsEquality {A = A} _≐_
  ≐-IsEquality = eq λ {x} {y} → ≐≡≡ •
    λ i → ≡p-IsEquality {ℓ-zero} .≙-≡-≡ {x} {y} i

≐-Equality : Equality {ℓ = ℓ-suc ℓ-zero}
≐-Equality = eqC _≐_
```

The last pass is to join the three equalities between equalities: Leibniz to Martin-Löf to cubical equality.

3 New Equalities types

The equalities used previously were defined using the cubical equality. Now I will define them using other equalities.

```
module Equalities {a ℓ} {A : Set a} where
  private
    ≙-Type' = ≙-Type {a} {ℓ} {A}
    ℓ₁ = ℓ-max a ℓ
```

Loaded the modules using the levels to be more generic.

```
module _
  (Eq₁ : Equality {_} {ℓ} {A})
```

```
  where

  open Equality Eq₁ renaming (_≜_ to _≡₁_; isEquality to eq₁)
```

I am importing a generic equality to use it to define a more generic equality.

```
record IsEquality₂ (_≜_ : ≜-Type') : Set (ℓ-suc ℓ₁) where
  constructor eq
  field
    ≜-≡-≡ : ∀ {x y} → let
      x≡y : Type ℓ₁
      x≡y = x ≡₁ y
```

Different from previously definition of IsEquality, the cubical equality defined in the line above was substituted by the more generic equality ≡₁.

```
x≜y : Type ℓ₁
x≜y = x ≜ y

in _≡_ {ℓ-suc ℓ₁} x≜y x≡y

≡-≡-≜ : {x y : A} → let
  x≡y : Type ℓ₁
  x≡y = x ≡₁ y
  x≜y = x ≜ y
  in x≡y ≡ x≜y
≡-≡-≜ = sym ≜-≡-≡
```

The rest of the definition is the same.

```
instance
  ≜-isEquality : IsEquality _≜_
  ≜-isEquality = eq (≜-≡-≡ • IsEquality.≜-≡-≡ eq₁)
```

From a more generic defintion of equality, it is easily possible to return to the less generic definition.

```
module _
  (Eq₂ : Equality {_} {ℓ} {A})
  where

  open Equality Eq₂ renaming (_≜_ to _≡₂_; isEquality to eq₂)
```

I am defining a new generic equality to prove that it is an equality of type 2:

```
eqsEqual₂ : ∀ {x y} → let
  x≜₁y : Type ℓ₁
  x≜₁y = x ≡₁ y
```

```
    x≙ 2y : Type ℓ 1
    x≙ 2y = x ≡ 2 y

    in _≡_ {ℓ-suc ℓ 1} x≙ 1y x≙ 2y
  eqsEqual 2 = eqsEqual _≡ 1_ _≡ 2_

  instance
    ≡ 2-Equality 2 : IsEquality 2 _≡ 2_
    ≡ 2-Equality 2 = eq (sym eqsEqual 2)
      where open IsEquality

  module _ {_≙_ : ≙-Type} where
    sym-Equality 2 : (≡-≡-≙ : {x y : A} → let
      x≡y : Type ℓ 1
      x≡y = x ≡ 1 y
      x≙y = x ≙ y
      in x≡y ≡ x≙y)
      → IsEquality 2 _≙_
    sym-Equality 2 ≡-≡-≙ = eq (sym ≡-≡-≙)
```

Given a symmetric definition of the previous equality, it is easy to prove that it is also an equality of type 2.

```
module _
  (Eq 3 : Equality {A = Set ℓ 1})
  where

  open Equality Eq 3 renaming (_≙_ to _≡ 3_; isEquality to eq 3)

  record IsEquality 3 (_≙_ : ≙-Type') : Set (ℓ-suc ℓ 1) where
    constructor eq
    field
      ≙-≡-≡ : ∀ {x y} → let
        x≡y : Type ℓ 1
        x≡y = x ≡ 1 y

        x≙y : Type ℓ 1
        x≙y = x ≙ y

        in x≙y ≡ 3 x≡y

    instance
      ≙-isEquality 2 : IsEquality 2 _≙_
      ≙-isEquality 2 = eq (transport (IsEquality.≙-≡-≡ eq 3) ≙-≡-≡)
```

```
    ≙-isEquality : IsEquality _≙_
    ≙-isEquality = eq (IsEquality ₂.≙-≡-≡ ≙-isEquality ₂ • IsEquality.≙-≡-≡ eq ₁)

  ≡-≡-≙ : {x y : A} → let
    x≡y : Type ℓ ₁
    x≡y = x ≡ ₁ y
    x≙y = x ≙ y
    in x≡y ≡ ₃ x≙y
  ≡-≡-≙ = let
    α ₁ = IsEquality.≡-≡-≙ eq ₃
    α ₂ = IsEquality.≙-≡-≡ eq ₁
    α ₃ = IsEquality.≡-≡-≙ ≙-isEquality
    in transport α ₁ (α ₂ • α ₃)

module _
  (Eq ₂ : Equality {_} {ℓ} {A})
  where

  open Equality Eq ₂ renaming (_≙_ to _≡ ₂_; isEquality to eq ₂)

  instance
    ≡ ₂-Equality ₃ : IsEquality ₃ _≡ ₂_
    ≡ ₂-Equality ₃ = eq α
      where
        open IsEquality eq ₃
        α : ∀ {x y} → (x ≡ ₂ y) ≡ ₃ (x ≡ ₁ y)
        α = transport ≡-≡-≙ (IsEquality ₂.≙-≡-≡ (≡ ₂-Equality ₂ (eqC _≡ ₂_)))

module _ {_≙_ : ≙-Type'} where
  sym-Equality ₃ : (≡-≡-≙ : {x y : A} → let
    x≡y : Type (ℓ-max a ℓ)
    x≡y = x ≡ ₁ y
    x≙y = x ≙ y
    in x≡y ≡ ₃ x≙y)
    → IsEquality ₃ _≙_
  sym-Equality ₃ ≡-≡-≙ = eq (let
    α ₁ = IsEquality.≡-≡-≙ eq ₃
    α ₂ = transport (sym α ₁) ≡-≡-≙
    in transport α ₁ (sym α ₂))

module LeibnizFromPEquality {A : Set} where
  open Equalities {ℓ-zero} {ℓ-suc ℓ-zero}

  _≡p ₁_ : A → A → Set ₁
  x ≡p ₁ y = x ≡p y

  instance
    ≡p ₁-isEquality : IsEquality _≡p ₁_
```

```
  ≡p ₁-isEquality = eq λ {x y} → (sym λ i → let
    α : Type ₁
    α = p-c {ℓ-zero} {x = x} {y = y} i
    in α)

leibniz : IsEquality {A = A} _≐_
leibniz =
  IsEquality ₂.≜-isEquality {Eq ₁ = eqC _≡p_} (eq FinalEquality.≐≡≡)
```

Acknowledgements