# Introduction to Type Theory in Agda

## Lecture 4 – Exploring equality towards univalent type theory

Todd Waugh Ambridge

11 April 2024

# Lecture Outline

1. Recap – Dependent types and equality
2. Inductive equality proofs
3. Equality of functions
4. Equality of types
5. Equality of proofs

# Lecture Outline

1. **Recap – Dependent types and equality**
2. Inductive equality proofs
3. Equality of functions
4. Equality of types
5. Equality of proofs

# Recap

In the last lecture, we introduced dependent types, completing our propositions-as-types interpretation of constructive logic in Agda:

- $\Pi$ for interpreting universal quantification,
- $\Sigma$ for interpreting existential quantification,
- $\equiv$ for interpreting propositional equality.

Recall also that we started proving some *inductive equality proofs*:

```
!-is-involutive : (b : Bool) → ! (! b) ≡ b
!-is-involutive tt = refl tt
!-is-involutive ff = refl ff
```

# Recap

```
ap-succ : {n m : ℕ} → n ≡ m → succ n ≡ succ m
ap-succ {n} {.n} (refl .n) = refl (succ n)

adding-1-≡-succ' : (n : ℕ) → add n 1 ≡ succ n
adding-1-≡-succ' zero = refl 1
adding-1-≡-succ' (succ n)
  = ap-succ (adding-1-≡-succ' n)
```

# Lecture Outline

1. Recap – Dependent types and equality
2. **Inductive equality proofs**
3. Equality of functions
4. Equality of types
5. Equality of proofs

# Inductive equality proofs

As discussed at the end of the last lecture, there is a more general rule to prove here: i.e., that *functions respect equality*.

```
ap : {x y : A} (f : A → B) → x ≡ y → f x ≡ f y
ap f (refl x) = refl (f x)
```

Because succ is just a function in our type theory, ap – succ is just an instance of ap.

```
adding-1-≡-succ' : (n : ℕ) → add n 1 ≡ succ n
adding-1-≡-succ' zero = refl 1
adding-1-≡-succ' (succ n) = ap succ (adding-1-≡-succ' n)
```

Let's continue looking at proving equalities of terms inductively.

# Inductive equality proofs

Above, we saw a couple of weird looking proofs of equality. To understand what is going on here, let's take a look at the *elimination* and *computation* rules for the identity type.

$$\frac{\Gamma, x, y : A, e : (x \equiv y) \vdash P : \text{Type} \quad \Gamma, z : A \vdash p : P(z, z, \text{refl } z)}{\Gamma, x, y : A, e : (x \equiv y) \vdash \equiv\text{-}induction(P, p, x, y, e) : P(x, y, e)} \ (\equiv\text{-Elim})$$

$$\frac{\Gamma, x, y : A, e : (x \equiv y) \vdash P : \text{Type} \quad \Gamma, z : A \vdash p : P(z, z, \text{refl } z)}{\Gamma, x : A \vdash \equiv\text{-}induction(P, p, x, x, \text{refl } x) = p : P(x, x, \text{refl } x)} \ (\equiv\text{-Comp})$$

# Inductive equality proofs

```
≡-induction : {X : Type}
            → (P : (x y : X) → x ≡ y → Type)
            → (p : (x : X) → P x x (refl x))
            → (x y : X) (e : x ≡ y) → P x y e
≡-induction P p x .x (refl .x) = p x
```

# Inductive equality proofs

The elimination rule says that, given there is a type $P(x, y, e)$ : Type for any elements $x, y : A$ that are equal by a proof term $e : x \equiv y$, in order to construct an element of $P(x, y, e)$ for a given $x$, $y$ and $e$ we only have to consider that happens when $e = \text{refl}\ x : x \equiv y$.

In Agda, when we pattern match on the proof that $x \equiv y$, the *only pattern* (by the `data` definition of $\equiv$) is refl. Therefore, $x$ and $y$ must be judgementally equal, and the two terms are thus identified in Agda's type system.

# Inductive equality proofs

Put simply, if $e = \text{refl}\ x : x \equiv y$, then because $\text{refl}\ x : x \equiv x$, it *must be the case* that $x = y : A$.

This is why the proof of *ap* looked a bit strange above! The `.` then simply notes a copy of the same element in the same statement (Agda automatically puts them in, but they are optional).

# Inductive equality proofs

So, equality respects functions. It should also be *symmetric* and *transitive*.

```
sym : {x y : A} → x ≡ y → y ≡ x
sym (refl x) = refl x

trans : {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans (refl x) (refl x) = refl x
```

# Lecture Outline

1. Recap – Dependent types and equality
2. Inductive equality proofs
3. **Equality of functions**
4. Equality of types
5. Equality of proofs

# Equality of functions

Now let's go back to our motivating example of functions from the previous lectures – we can now finally form the type $\equiv \{\mathbb{N} \to \mathrm{Bool}\}$ for equalities on functions $\mathbb{N} \to \mathrm{Bool}$, but can we introduce elements of this type?

We can obviously introduce an element when the two functions are judgementally equal.

```
is-odd?-≡-is-odd? : is-odd? ≡ is-odd?
is-odd?-≡-is-odd? = refl is-odd?
```

# Equality of functions

But can we show that two *behaviourally equivalent* functions are equal, e.g. `is-odd?` $\equiv$ `is-odd?'`.

Short answer: no. There is no way to argue this in our current type theory.

All we can say for now is that these two objects are behaviourally equivalent; for functions, this means that they are *pointwise-equal*.

# Equality of functions

However, we can (if we want to) add an axiom to our theory that says behaviourally equivalent functions are equal – this is called *function extensionality*.

```
FunExt : Typeω
FunExt = {i j : Level} {X : Type i} {Y : X → Type j}
       → (f g : Π Y) → f ∼ g → f ≡ g
```

# Equality of functions

Function extensionality can be assumed locally, and gives us a way other than `refl` to introduce elements of the identity type.

```
is-odd?-≡-is-odd?' : FunExt → is-odd? ≡ is-odd?'
is-odd?-≡-is-odd?' fe
 = fe is-odd? is-odd?' is-odd?-∼-is-odd?'
```

# Equality of functions

However, as we have assumed this axiom without proving it (because it is independent of MLTT, it can be neither proved nor disproved), we have to be careful where we use it. Function extensionality has no computational interpretation in Agda, and so it can 'destroy' the computational content of our proofs.

There is a philosophical question here: clearly we do not accept the law of excluded middle as an axiom. Why do we accept function extensionality? Well, maybe you don't! And that's okay, but you have to then accept you won't be able to talk about the equality of functions in any meaningful way in basic Agda.

However, in Cubical Agda, function extensionality actually has a computational interpretation.

# Lecture Outline

1. Recap – Dependent types and equality
2. Inductive equality proofs
3. Equality of functions
4. **Equality of types**
5. Equality of proofs

# Equality of types

We can now discuss equality of basic elements of our theory, and of functions. What about *types* themselves?

We can of course say that two syntatically equal types are equal.

```
Baire-≡-Baire : (ℕ → ℕ) ≡ (ℕ → ℕ)
Baire-≡-Baire = refl (ℕ → ℕ)
```

But what about *behaviourally equivalent* types?

# Equality of types

The Bool type has two elements. We might refer to it as the $\mathbb{2}$ type.

```
𝟚 : Type
𝟚 = Bool
```

Meanwhile, the $\mathbb{1} + \mathbb{1}$ type also has two points (hence its name!). These two types are thus *isomorphic* – they could be swapped out for each other in any program with no loss of computational meaning.

But we can't say that they are equal using equality as it currently stands.

# Equality of functions

But we could employ an axiom (similarly to how we introduced function extensionality) which says that behaviourally equivalent (i.e. isomorphic) types are equal.

To do that, we first need to define the *concept* of two types being behaviourally equivalent as a type family. Any ideas?

# Equality of functions

```
_≅_ : {i : Level} (X Y : Type i) → Type i

_∘_ : {i j k : Level}
      {A : Type i} {B : Type j} {C : Type k}
    → (B → C) → (A → B) → (A → C)
g ∘ f = λ a → g (f a)

id : {i : Level} {X : Type i} → X → X
id x = x

X ≅ Y = Σ f : (X → Y) , Σ g : (Y → X)
      , ((f ∘ g ∼ id) × (g ∘ f ∼ id))
```

# Equality of functions

So, types $X$ and $Y$ are said to be (quasi-)equivalent[1] $X \cong Y$ if we have a function $f : X \to Y$ which is a *bijection*.

Let's show that $\mathbb{2}$ and $\mathbb{1} + \mathbb{1}$ are equivalent.

[1] I use the prefix 'quasi' here because equivalence is defined slightly differently in univalent type theory.

# Equality of functions

```
𝟚≅𝟙+𝟙 : 𝟚 ≅ 𝟙 + 𝟙
𝟚≅𝟙+𝟙 = f , (g , (η , ε))
 where
  f : 𝟚 → 𝟙 + 𝟙
  f tt = inl ⋆
  f ff = inr ⋆
  g : 𝟙 + 𝟙 → 𝟚
  g (inl x) = tt
  g (inr x) = ff
  η : (f ∘ g) ∼ id
  η (inl ⋆) = refl (inl ⋆)
  η (inr ⋆) = refl (inr ⋆)
  ε : (g ∘ f) ∼ id
  ε tt = refl tt
  ε ff = refl ff
```

# Equality of functions

There is of course another equivalence:

```
2≅𝟙+𝟙' : 2 ≅ 𝟙 + 𝟙
2≅𝟙+𝟙' = f , (g , (η , ε))
 where
  f : 2 → 𝟙 + 𝟙
  f tt = inr ⋆
  f ff = inl ⋆
  g : 𝟙 + 𝟙 → 2
  g (inl x) = ff
  g (inr x) = tt
  η : (f ∘ g) ∼ id
  η (inl ⋆) = refl (inl ⋆)
  η (inr ⋆) = refl (inr ⋆)
  ε : (g ∘ f) ∼ id
  ε tt = refl tt
  ε ff = refl ff
```

# Equality of functions

Using either equivalence, by the axiom of *weak univalence*, these types are identical.

```
WeakUniv : Typeω
WeakUniv = {i : Level} → (X Y : Type i) → X ≅ Y → X ≡ Y

2≡𝟙+𝟙 : WeakUniv → 𝟚 ≡ 𝟙 + 𝟙
2≡𝟙+𝟙 wu = wu 𝟚 (𝟙 + 𝟙) 2≅𝟙+𝟙

2≡𝟙+𝟙' : WeakUniv → 𝟚 ≡ 𝟙 + 𝟙
2≡𝟙+𝟙' wu = wu 𝟚 (𝟙 + 𝟙) 2≅𝟙+𝟙'
```

# Equality of functions

We are starting to tread our feet into univalent type theory now. Axiom K (which we touched on last lecture) says that every equality is just refl.

```
AxiomK : Typeω
AxiomK = {i : Level} {X : Type i} (x y : X) (p q : x ≡ y)
       → p ≡ q

AllEqsRefl : Typeω
AllEqsRefl = {i : Level} {X : Type i} (x : X) (p : x ≡ x)
           → p ≡ refl x

axiom-K-all-eqs-refl : AxiomK → AllEqsRefl
axiom-K-all-eqs-refl ak x p = ak x x p (refl x)
```

Meanwhile, the (weak) univalence axiom says that equality is something more: in particular, equivalences are equalities.

# Equality of functions

Both of these axioms are *independent* of MLTT – they are not provable nor disprovable, and can be separately added to our theory while remaining consistent.

However, each axiom contradicts the other, so we cannot have both.

Univalent type theory builds upon MLTT with the univalence axiom, and other concepts, in order to explore the equalities of a wide variety of structures that cannot be captured by just the identity type.

# Lecture Outline

1. Recap – Dependent types and equality
2. Inductive equality proofs
3. Equality of functions
4. Equality of types
5. **Equality of proofs**

# Equality of proofs

We have seen equalities of:

- Booleans and natural numbers,
- Lists (in the exercise class),
- Functions (by function extensionality),
- Types (by weak univalence).

But what about *proofs* themselves?

# Equality of proofs

Let's think back to our definition of `is-odd`. Are two proofs of `is-odd n` for a given $n : \mathbb{N}$ equal?

```
is-odd-proofs-unique? : (n : ℕ) → (p q : is-odd n)
                      → p ≡ q
is-odd-proofs-unique? zero ()
is-odd-proofs-unique? (succ zero) ⋆ ⋆ = refl ⋆
is-odd-proofs-unique? (succ (succ n))
 = is-odd-proofs-unique? n
```

# Equality of proofs

So `is-odd` $n$ always has a unique proof.

What about with our example of proof relevance? Are two proofs of $\Sigma_{(m:\mathbb{N})}(n < m)$ for a given $n : \mathbb{N}$ equal?

# Equality of proofs

```
succ-n-is-not-succ-succ-n
 : (n : ℕ) → ¬ (succ n ≡ succ (succ n))
succ-n-is-not-succ-succ-n n ()

<-is-transitive : (n m k : ℕ) → n < m → m < k → n < k
<-is-transitive zero (succ m) (succ k) n<m m<k = ⋆
<-is-transitive (succ n) (succ m) (succ k) n<m m<k
 = <-is-transitive n m k n<m m<k

bigger-number-proofs-unique? : (n : ℕ)
                             → ¬ ((p q : Σ m : ℕ , n < m) →
bigger-number-proofs-unique? n f
 = succ-n-is-not-succ-succ-n n
     (ap fst (f (succ n , succ-is-bigger n)
      (succ (succ n) , <-is-transitive
                       n (succ n) (succ (succ n))
                       (succ-is-bigger n)
                       (succ-is-bigger (succ n)))))
```

# Equality of proofs

So we have proved what we discussed in lecture two: there are many proofs of $\Sigma_{(m:\mathbb{N})}(n < m)$.

This course has discussed how propositions can be viewed as types via the *propositions as types* perspective.

But there is a slight alternative of this which is of interest to univalent type theory. The *propositions as some types* perspective only considers propositions to be types with at most one element – i.e. those propositions that can only be true in one way.

```
is-prop : {i : Level} → Type i → Type i
is-prop X = (x y : X) → x ≡ y
```

# Equality of proofs

Further to that, types whose identity types are propositions are in univalent type theory called... sets.

```
is-set : {i : Level} → Type i → Type i
is-set X = (x y : X) → is-prop (x ≡ y)
```

Continuing this line of thinking, along with the earlier consideration of the univalence axiom, leads us to *univalent type theory*.

If you'd like to read more about this, I strongly recommend the book 'Homotopy Type Theory'.

Thank you for taking this course!

# Acknowledgements


- Ingo Blechschmidt *(for Agdapad and ongoing support with it)*
- Stefania Damato *(for being a great TA!)*
- Tom de Jong *(for proof reading Lecture 1)*
- Thorsten Altenkirch *(for great feedback)*
- Bruno da Rocha Paiva *(for interesting and helpful discussions)*
- Roy Crole, Reiko Heckel and Adam Machowczyk *(for organising a fantastic MGS 2024)*
- Alice Menzel (*for helpful feedback and support*)
- Martín Escardó *(for teaching me all of this to begin with)*