Introduction to Type Theory in Agda

Lecture 3 – Dependent types and equality

Todd Waugh Ambridge

10 April 2024

Lecture Outline

1. Recap – Propositional Logic via Types
2. Predicate Logic via Dependent Types
   - Π and Σ
3. Type universes
   - Type, Type_1, Type_2, ...
4. Decidable equality
5. Propositional equality
   - $\equiv$

Lecture Outline

1. **Recap – Propositional Logic via Types**
2. Predicate Logic via Dependent Types
   - Π and Σ
3. Type universes
   - Type, Type_1, Type_2, ...
4. Decidable equality
5. Propositional equality
   - $\equiv$

Recap

In the last lecture, we introduced the propositions-as-types interpretation in Agda by showing that function types interpret implication, and further defining:

1. $\mathbb{1}$ for interpreting truth,
2. $\mathbb{0}$ for interpreting falsity,
3. $\neg$-types for interpreting negation,
4. $+$-types for interpreting disjunction,
5. $\times$-types for interpreting conjunction.

Lecture Outline

1. Recap – Propositional Logic via Types
2. **Predicate Logic via Dependent Types**
   - Π and Σ
3. Type universes
   - Type, Type_1, Type_2, ...
4. Decidable equality
5. Propositional equality
   - $\equiv$

Dependent Types

To complete our propositions-as-types interpretation of constructive logic, we need to interpret the two quantifier connectives of predicate logic:

- Universal quantification $\forall x : X, Px$,
- Existential quantification $\exists x : X, Px$.

These quantifiers are interpreted by Martin-Lof's dependent types:

- Π-types interpret "for all" statements,
- Σ-types interpret "there exists" statements.

Dependent Types

To complete our propositions-as-types interpretation of constructive logic, we need to interpret the two quantifier connectives of predicate logic:

- Universal quantification $\forall x : X, Px$,
- Existential quantification $\exists x : X, Px$.

These quantifiers are interpreted by Martin-Lof's dependent types:

- Π-types interpret "for all" statements,
- Σ-types interpret "there exists" statements.

MLTT in Agda

(a) Function types $\to$,
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types,
(d) Disjoint union types $+$,
(e) Binary product types $\times$,
(f) **Dependent function types Π**,
(g) Dependent pair types Σ,
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

Dependent Types – Π-types

Earlier, we introduced the dependent type family, `is-odd` : ℕ → Type. But this isn't the first type family we've seen:

- _ + _, _ × _ : Type → Type → Type are binary type families,
- Each of the `induction` principles featured functions $P : X \to \text{Type}$ – these are also type families.

These induction principles also featured dependent functions $p : (x : X) \to P(x)$.

```
ℕ-induction : -- Type family
              (P : ℕ → Type)
            → (p₀ : P zero)
              -- Dependent function
            → (pₛ : (n : ℕ) → P n → P (succ n))
...
```

Dependent Types – Π-types

Earlier, we introduced the dependent type family,
`is-odd` : $\mathbb{N} \to \text{Type}$. But this isn't the first type family we've seen:

- $\_ + \_, \_ \times \_ : \text{Type} \to \text{Type} \to \text{Type}$ are binary type families,
- Each of the `induction` principles featured functions $P : X \to \text{Type}$ – these are also type families.

These induction principles also featured dependent functions $p : (x : X) \to P(x)$.

```
ℕ-induction : -- Type family
              (P : ℕ → Type)
            → (p₀ : P zero)
              -- Dependent function
            → (pₛ : (n : ℕ) → P n → P (succ n))
...
```

Dependent Types – Π-types

Earlier, we introduced the dependent type family, `is-odd` : $\mathbb{N} \to \text{Type}$. But this isn't the first type family we've seen:

- $\_ + \_, \_ \times \_ : \text{Type} \to \text{Type} \to \text{Type}$ are binary type families,
- Each of the `induction` principles featured functions $P : X \to \text{Type}$ – these are also type families.

These induction principles also featured dependent functions $p : (x : X) \to P(x)$.

```
ℕ-induction : -- Type family
              (P : ℕ → Type)
            → (p₀ : P zero)
              -- Dependent function
            → (pₛ : (n : ℕ) → P n → P (succ n))
...
```

Dependent Types – Π-types

Earlier, we introduced the dependent type family,
`is-odd` : $\mathbb{N} \to \text{Type}$. But this isn't the first type family we've seen:

- $\_ + \_, \_ \times \_ : \text{Type} \to \text{Type} \to \text{Type}$ are binary type families,
- Each of the `induction` principles featured functions $P : X \to \text{Type}$ – these are also type families.

These induction principles also featured dependent functions $p : (x : X) \to P(x)$.

```
ℕ-induction : -- Type family
                (P : ℕ → Type)
              → (p₀ : P zero)
                -- Dependent function
              → (pₛ : (n : ℕ) → P n → P (succ n))
...
```

Dependent Types – Π-types

Given a type family $P : X \to \text{Type}$, a *dependent function* $p : (x : X) \to P(x)$ is a function whose domain type $P(x) : \text{Type}$ *depends* on the value of the given argument $x : X$.

Clearly, as we have already seen a fair few dependent functions, they are built-in to Agda, just like non-dependent functions.

While non-dependent functions $f : A \to B$ are terms of function types, dependent functions $f : (x : X) \to Y\ x$ are terms of *Π-types*.

Dependent Types – Π-types

Given a type family $P : X \to \text{Type}$, a *dependent function* $p : (x : X) \to P(x)$ is a function whose domain type $P(x) : \text{Type}$ *depends* on the value of the given argument $x : X$.

Clearly, as we have already seen a fair few dependent functions, they are built-in to Agda, just like non-dependent functions.

While non-dependent functions $f : A \to B$ are terms of function types, dependent functions $f : (x : X) \to Y\ x$ are terms of *Π-types*.

Dependent Types – Π-types

$$\frac{\Gamma \vdash X : \mathsf{Type} \quad \Gamma x : X \vdash Y(x) : \mathsf{Type}}{\Gamma \vdash \Pi_{(x:X)} Y : \mathsf{Type}} \; (\Pi\text{-Form})$$

$$\frac{\Gamma, x : X \vdash y : Y(x)}{\Gamma \vdash \lambda(x : X).y : \Pi_{(x:X)} Y} \; (\Pi\text{-Intro})$$

$$\frac{\Gamma \vdash f : \Pi_{(x:X)} Y \quad \Gamma \vdash a : X}{\Gamma \vdash f(a) : Y(a)} \; (\Pi\text{-Elim})$$

$$\frac{\Gamma, x : X \vdash y : Y(x) \quad \Gamma \vdash a : X}{\Gamma \vdash (\lambda(a : A).b)\,(a) = y[a/x] : B(a)} \; (\Pi\text{-Comp})$$

Dependent Types – Π-types

We can align Agda's syntax for Π-types with MLTT's.

Note that non-dependent functions are just special cases of dependent functions, where the type family $P : X \to \text{Type}$ is constant.

$$(X \to Y) = \Pi\ x : X, Y$$

Dependent Types – Π-types

While non-dependent functions interpret implication, dependent functions interpret universal quantification. That is, to prove $\forall x : X, P\ x$ holds, we need to define a dependent function $f : \Pi\ x : X, P\ x$.

As an example, let's prove that we can decide whether `is-odd` n : Type holds for every $n : \mathbb{N}$. This proof is inductive, following the definition of `is-odd` : $\mathbb{N} \to$ Type itself.

Dependent Types – Π-types

While non-dependent functions interpret implication, dependent functions interpret universal quantification. That is, to prove $\forall x : X, P\ x$ holds, we need to define a dependent function $f : \Pi\ x : X, P\ x$.

As an example, let's prove that we can decide whether `is-odd` n : Type holds for every $n : \mathbb{N}$. This proof is inductive, following the definition of `is-odd` : $\mathbb{N} \to$ Type itself.

Dependent Types – Π-types

While non-dependent functions interpret implication, dependent functions interpret universal quantification. That is, to prove $\forall x : X, P\ x$ holds, we need to define a dependent function $f : \Pi\ x : X, P\ x$.

As an example, let's prove that we can decide whether `is-odd` n : Type holds for every $n : \mathbb{N}$. This proof is inductive, following the definition of `is-odd` : $\mathbb{N} \to$ Type itself.

Dependent Types – Π-types

Let's see another example: first we define the binary type family that corresponds to the order on natural numbers.

Then, we prove that, for every $n : \mathbb{N}$, it is the case that $n < \text{succ } n$.

MLTT in Agda

(a) Function types $\to$,
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types,
(d) Disjoint union types $+$,
(e) Binary product types $\times$,
(f) Dependent function types Π,
(g) **Dependent pair types Σ**,
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

Dependent Types – Σ-types

Given a type family $Y : X \to \text{Type}$, how do we interpret the concept that there exists a term $x : X$ such that $Y\ x : \text{Type}$ is true?

The way existential quantification works is the second key difference between constructive and classical logic. Classically, we can show that there exists an $x : X$ that satisfies $Y\ x$ by showing that the lack of such an $x : X$ leads to a contradiction.

But this argument doesn't hold in constructive maths: constructively, to show that $Y\ x$ holds, we have to actually *specify* which $x : X$ is satisfactory.

Dependent Types – Σ-types

Given a type family $Y : X \to \text{Type}$, how do we interpret the concept that there exists a term $x : X$ such that $Y\ x : \text{Type}$ is true?

The way existential quantification works is the second key difference between constructive and classical logic. Classically, we can show that there exists an $x : X$ that satisfies $Y\ x$ by showing that the lack of such an $x : X$ leads to a contradiction.

But this argument doesn't hold in constructive maths: constructively, to show that $Y\ x$ holds, we have to actually *specify* which $x : X$ is satisfactory.

Dependent Types – Σ-types

Given a type family $Y : X \to \text{Type}$, how do we interpret the concept that there exists a term $x : X$ such that $Y\ x : \text{Type}$ is true?

The way existential quantification works is the second key difference between constructive and classical logic. Classically, we can show that there exists an $x : X$ that satisfies $Y\ x$ by showing that the lack of such an $x : X$ leads to a contradiction.

But this argument doesn't hold in constructive maths: constructively, to show that $Y\ x$ holds, we have to actually *specify* which $x : X$ is satisfactory.

Dependent Types – Σ-types

So, to show that $\exists\ x : X, Y\ x$, we need to provide a pair of terms:

1. A term $x : X$, called the *witness* of Y,
2. A proof term $Y\ x$: Type, which *depends* on the witness.

In MLTT, these dependent pairs are called *Σ-types*. As with non-dependent pairs (i.e. $\times$-types), we define them using `record`.

Dependent Types – Σ-types

$$\frac{\Gamma \vdash X : \mathsf{Type} \qquad \Gamma, x : X \vdash Y(x) : \mathsf{Type}}{\Gamma \vdash \Sigma_{(x:X)} Y : \mathsf{Type}} \ (\Sigma\text{-Form})$$

$$\frac{\Gamma, x : X \vdash Y(x) : \mathsf{Type} \qquad \Gamma \vdash w : X \qquad \Gamma \vdash y : Y(a)}{\Gamma \vdash (w, p) : \Sigma_{(w:X)} Y} \ (\Sigma\text{-Intro})$$

$$\frac{\begin{matrix}\Gamma, z{:}\Sigma_{(x:X)} Y \\ \vdash P(z){:}\mathsf{Type}\end{matrix} \qquad \Gamma, \begin{matrix} w{:}X, \\ y{:}Y(w)\end{matrix} \vdash p((w, y)) : P((w, y))}{\Gamma, z : \Sigma_{(x:X)} Y \vdash \Sigma\text{-induction}(P, p, z) : P(z)} \ (\Sigma\text{-Elim})$$

$$\frac{\begin{matrix}\Gamma, z{:}\Sigma_{(x:X)} Y \\ \vdash P(z){:}\mathsf{Type}\end{matrix} \qquad \Gamma, \begin{matrix} w{:}X, \\ y{:}Y(w)\end{matrix} \vdash p((w, y)) : P((w, y))}{\Gamma, \begin{matrix} a{:}X \\ b{:}Y(a)\end{matrix} \vdash \Sigma\text{-induction}(P, p, (a, b)) = p((a, b)) : P((a, b))} \ (\Sigma\text{-Comp})$$

Dependent Types – Σ-types

We can now re-define $\times$-types as the non-dependent case of Σ-types (as with non-dependent functions and Π-types).

Dependent Types – Σ-types

As an example of using Σ-types, let's prove that, for every $n : \mathbb{N}$, there exists an $m : \mathbb{N}$ larger than it.

Dependent Types – Σ-types

As an example of using Σ-types, let's prove that, for every $n : \mathbb{N}$, there exists an $m : \mathbb{N}$ larger than it.

In the above, we chose to specify succ n as the witness that there is a number bigger than n. But we could have chose succ(succ n) or `add` 1000 n...

The term that we choose as a witness changes the computational content of the resulting proof. Therefore, in constructive type theory, the method of proving something is relevant — not just the fact that we have proved it.

This is called *proof relevance*.

Dependent Types – Σ-types

As an example of using Σ-types, let's prove that, for every $n : \mathbb{N}$, there exists an $m : \mathbb{N}$ larger than it.

In the above, we chose to specify succ n as the witness that there is a number bigger than n. But we could have chose succ(succ n) or `add` 1000 n...

The term that we choose as a witness changes the computational content of the resulting proof. Therefore, in constructive type theory, the method of proving something is relevant — not just the fact that we have proved it.

This is called *proof relevance*.

Dependent Types – Σ-types

As an example of using Σ-types, let's prove that, for every $n : \mathbb{N}$, there exists an $m : \mathbb{N}$ larger than it.

In the above, we chose to specify succ n as the witness that there is a number bigger than n. But we could have chose succ(succ n) or `add` 1000 n...

The term that we choose as a witness changes the computational content of the resulting proof. Therefore, in constructive type theory, the method of proving something is relevant — not just the fact that we have proved it.

This is called *proof relevance*.

Dependent Types – Σ-types

Another important point about Σ-types is that they form collections in our type theory.

For example, the type $\Sigma_{(n:\mathbb{N})}\,\texttt{is-odd}\ n$ collects every possible pair of a number with a proof of its oddness. Therefore, this is the *type of odd numbers* itself.

Dependent Types – Σ-types

Another important point about Σ-types is that they form collections in our type theory.

For example, the type $\Sigma_{(n:\mathbb{N})}$ `is-odd` n collects every possible pair of a number with a proof of its oddness. Therefore, this is the *type of odd numbers* itself.

Dependent Types – Σ-types

But what if we want to collect *types* themselves?

For example, we could carve out a subset of our logic that relates to the Boolean-logic; i.e., we could define a Σ-type that collects all decidable types together.

Well, we could, if not for that we get a type error! What is going on here? And how do we fix it? What does `Set₁ != Set` mean???

Dependent Types – Σ-types

But what if we want to collect *types* themselves?

For example, we could carve out a subset of our logic that relates to the Boolean-logic; i.e., we could define a Σ-type that collects all decidable types together.

Well, we could, if not for that we get a type error! What is going on here? And how do we fix it? What does $\texttt{Set}_1$ `!= Set` mean???

Lecture Outline

1. Recap – Propositional Logic via Types
2. Predicate Logic via Dependent Types
   - Π and Σ
3. **Type universes**
   - Type, Type_1, Type_2, ...
4. Decidable equality
5. Propositional equality
   - $\equiv$

Type Universes

I've been deliberately vague about what Type itself is. In Martin-Lof's first type theory (which appeared in a 1971 preprint), there were *terms* and there were *types* — terms had types, but types were just types. For example, $a : A$, but $A : \text{Type}$.

This raises the interesting question: what is the type of Type? Well, the 1971 type theory had an axiom that said

$$\text{Type} : \text{Type}.$$

But, similarly to Russell with set theory, Girard showed that this axiom made the system inconsistent.

Type Universes

I've been deliberately vague about what Type itself is. In Martin-Lof's first type theory (which appeared in a 1971 preprint), there were *terms* and there were *types* — terms had types, but types were just types. For example, $a : A$, but $A : \text{Type}$.

This raises the interesting question: what is the type of Type? Well, the 1971 type theory had an axiom that said

$$\text{Type} : \text{Type}.$$

But, similarly to Russell with set theory, Girard showed that this axiom made the system inconsistent.

Type Universes

I've been deliberately vague about what Type itself is. In Martin-Lof's first type theory (which appeared in a 1971 preprint), there were *terms* and there were *types* — terms had types, but types were just types. For example, $a : A$, but $A : \text{Type}$.

This raises the interesting question: what is the type of Type? Well, the 1971 type theory had an axiom that said

$$\text{Type} : \text{Type}.$$

But, similarly to Russell with set theory, Girard showed that this axiom made the system inconsistent.

MLTT in Agda

(a) Function types $\to$,
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types,
(d) Disjoint union types $+$,
(e) Binary product types $\times$,
(f) Dependent function types Π,
(g) Dependent pair types Σ,
(h) **Type universes** $\mathsf{Type}_0, \mathsf{Type}_1, \ldots$,
(i) Identity types $\equiv$.

Type Universes

So Martin-Lof went back to the drawing board, and built his next type theory (1972's MLTT) around the idea of countably-many type universes:

$$\mathsf{Type} : \mathsf{Type}_1 : \mathsf{Type}_2 : \ldots$$

A *type universe* is a type whose terms are also types.

Agda also has type universes (but with the annoying name Set):

$$\mathtt{Set} : \mathtt{Set}_1 : \mathtt{Set}_2 : \ldots$$

Type Universes

So Martin-Lof went back to the drawing board, and built his next type theory (1972's MLTT) around the idea of countably-many type universes:

$$\mathsf{Type} : \mathsf{Type}_1 : \mathsf{Type}_2 : \ldots$$

A *type universe* is a type whose terms are also types.

Agda also has type universes (but with the annoying name Set):

$$\texttt{Set} : \texttt{Set}_1 : \texttt{Set}_2 : \ldots$$

Type Universes

So Martin-Lof went back to the drawing board, and built his next type theory (1972's MLTT) around the idea of countably-many type universes:

$$\mathsf{Type} : \mathsf{Type}_1 : \mathsf{Type}_2 : \dots$$

A *type universe* is a type whose terms are also types.

Agda also has type universes (but with the annoying name Set):

$$\texttt{Set} : \texttt{Set}_1 : \texttt{Set}_2 : \dots$$

Type Universes

So that we don't have to rename a countably infinite number of terms let's properly rename `Set` to `Type` using Agda's builtin file for type universes.

In that file, we can see a glimpse of how type universes are implemented in Agda. The idea is that `Type`, $\texttt{Type}_1$, $\texttt{Type}_2$, etc. are actually syntax sugars for the *type universes* `Type lzero`, `Type (lsuc lzero)`, `Type (lsuc (lsuc lzero))`; where these objects beginning with `l` are called *universe levels*.

Now, let's redefine Π and Σ to correctly use type universes.

Type Universes

So that we don't have to rename a countably infinite number of terms let's properly rename `Set` to `Type` using Agda's builtin file for type universes.

In that file, we can see a glimpse of how type universes are implemented in Agda. The idea is that `Type`, $\texttt{Type}_1$, $\texttt{Type}_2$, etc. are actually syntax sugars for the *type universes* `Type lzero`, `Type (lsuc lzero)`, `Type (lsuc (lsuc lzero))`; where these objects beginning with `l` are called *universe levels*.

Now, let's redefine Π and Σ to correctly use type universes.

Type Universes

So that we don't have to rename a countably infinite number of terms let's properly rename `Set` to `Type` using Agda's builtin file for type universes.

In that file, we can see a glimpse of how type universes are implemented in Agda. The idea is that `Type`, $\texttt{Type}_1$, $\texttt{Type}_2$, etc. are actually syntax sugars for the *type universes* `Type lzero`, `Type (lsuc lzero)`, `Type (lsuc (lsuc lzero))`; where these objects beginning with `l` are called *universe levels*.

Now, let's redefine Π and Σ to correctly use type universes.

Type Universes

Now that we have universes, we can define the type of decidable types.

Lecture Outline

1. Recap – Propositional Logic via Types
2. Predicate Logic via Dependent Types
   - Π and Σ
3. Type universes
   - Type, Type_1, Type_2, ...
4. **Decidable equality**
5. Propositional equality
   - $\equiv$

Decidable equality

We finally have a full, well defined, Type-valued first-order (predicate) logic. The final step is to have an interpretation of equality.

In the second exercise class, we played around with this Type-valued logic. For example, we defined an equality relation on the Booleans and another on the natural numbers.

It is important to realise here that, given any two terms $a, b : \mathrm{Bool}$ (respectively $n, m : \mathbb{N}$), $a == b$ (respectively $n \equiv m$) is a type.

Decidable equality

We finally have a full, well defined, Type-valued first-order (predicate) logic. The final step is to have an interpretation of equality.

In the second exercise class, we played around with this Type-valued logic. For example, we defined an equality relation on the Booleans and another on the natural numbers.

It is important to realise here that, given any two terms $a, b : \text{Bool}$ (respectively $n, m : \mathbb{N}$), $a == b$ (respectively $n \equiv m$) is a type.

Decidable equality

We finally have a full, well defined, Type-valued first-order (predicate) logic. The final step is to have an interpretation of equality.

In the second exercise class, we played around with this Type-valued logic. For example, we defined an equality relation on the Booleans and another on the natural numbers.

It is important to realise here that, given any two terms $a, b : \mathrm{Bool}$ (respectively $n, m : \mathbb{N}$), $a == b$ (respectively $n \equiv m$) is a type.

Decidable equality

Recall also that, in the exercise class, we showed the equality relation on the natural numbers is indeed an equality relation: it is reflexive, symmetric and transitive.

```
≣-is-reflexive : (n : ℕ) → n ≣ n
≣-is-reflexive zero = ⋆
≣-is-reflexive (succ n) = ≣-is-reflexive n

≣-is-symmetric : (n m : ℕ) → n ≣ m → m ≣ n
≣-is-symmetric zero zero p = ⋆
≣-is-symmetric (succ n) (succ m) p = ≣-is-symmetric n m p

≣-is-transitive : (n m k : ℕ) → n ≣ m → m ≣ k → n ≣ k
≣-is-transitive zero zero zero p q = ⋆
≣-is-transitive (succ n) (succ m) (succ k) p q
 = ≣-is-transitive n m k p q
```

Decidable equality

The Booleans and the natural numbers have what we call decidable equality; i.e., given any two terms $a, b : \mathrm{Bool}$ (respectively $n, m : \mathbb{N}$), the question of whether $a == b$ (respectively $n \equiv m$) is a decidable proposition.

This is the same as saying they are decidable types.

The proof of the former is by the fact that truth and falsity are decidable propositions; i.e. $\mathbb{1}$ and $\mathbb{0}$ are decidable types. The latter proof is also by these facts, and induction.

Decidable equality

The Booleans and the natural numbers have what we call decidable equality; i.e., given any two terms $a, b : \mathrm{Bool}$ (respectively $n, m : \mathbb{N}$), the question of whether $a == b$ (respectively $n \equiv m$) is a decidable proposition.

This is the same as saying they are decidable types.

The proof of the former is by the fact that truth and falsity are decidable propositions; i.e. $\mathbb{1}$ and $\mathbb{0}$ are decidable types. The latter proof is also by these facts, and induction.

Decidable equality

The Booleans and the natural numbers have what we call decidable equality; i.e., given any two terms a, b : Bool (respectively $n, m : \mathbb{N}$), the question of whether $a == b$ (respectively $n \equiv m$) is a decidable proposition.

This is the same as saying they are decidable types.

The proof of the former is by the fact that truth and falsity are decidable propositions; i.e. $\mathbb{1}$ and $\mathbb{0}$ are decidable types. The latter proof is also by these facts, and induction.

Decidable equality

But, as we discussed last lecture, the law of excluded middle[1] does not hold constructively – it is not the case that every proposition is decidable.

It is also not the case that every type has decidable equality; consider the example of functions from last lecture. We cannot decide whether or not two functions $f, g : \mathbb{N} \to \text{Bool}$ are equal, because any procedure that could do this cannot be guaranteed to halt.

[1]This is something that we cannot prove nor disprove in our type theory; we could, if we wanted to (which we don't), add it as an axiom and our theory would remain consistent.

Decidable equality

But, as we discussed last lecture, the law of excluded middle[1] does not hold constructively – it is not the case that every proposition is decidable.

It is also not the case that every type has decidable equality; consider the example of functions from last lecture. We cannot decide whether or not two functions $f, g : \mathbb{N} \to \text{Bool}$ are equal, because any procedure that could do this cannot be guaranteed to halt.

[1] This is something that we cannot prove nor disprove in our type theory; we could, if we wanted to (which we don't), add it as an axiom and our theory would remain consistent.

Decidable equality

So, have I lied to you? Because last lecture I entirely motivated the Type-valued logic by saying we could it to define equality on functions; i.e., I said we could define a type family

$$(\mathbb{N} \to \mathsf{Bool}) \to (\mathbb{N} \to \mathsf{Bool}) \to \mathsf{Type}.$$

But how do we actually go about defining this?

Lecture Outline

1. Recap – Propositional Logic via Types
2. Predicate Logic via Dependent Types
   - Π and Σ
3. Type universes
   - Type, Type_1, Type_2, ...
4. Decidable equality
5. **Propositional equality**
   - $\equiv$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \text{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \text{Type},$$

$$\text{or } (\mathbb{N} \to \text{Bool}) \to (\mathbb{N} \to \text{Bool}) \to \text{Type} \ldots$$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \mathrm{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \mathrm{Type},$$

$$\text{or } (\mathbb{N} \to \mathrm{Bool}) \to (\mathbb{N} \to \mathrm{Bool}) \to \mathrm{Type} \ldots$$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \mathrm{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \mathrm{Type},$$

$$\text{or } (\mathbb{N} \to \mathrm{Bool}) \to (\mathbb{N} \to \mathrm{Bool}) \to \mathrm{Type} \ldots$$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \mathrm{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \mathrm{Type},$$

$$\text{or } (\mathbb{N} \to \mathrm{Bool}) \to (\mathbb{N} \to \mathrm{Bool}) \to \mathrm{Type} \ldots$$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \mathrm{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \mathrm{Type},$$

$$\text{or } (\mathbb{N} \to \mathrm{Bool}) \to (\mathbb{N} \to \mathrm{Bool}) \to \mathrm{Type} \ldots$$

Propositional equality

We need to think about equality much more generally. In a first-order logic with equality, equality is itself considered to be a proposition.

This suggests that, as with the other connectives of predicate logic, it must be interpreted as a *type (family)*.

Thus far we have interpreted propositional equality differently for each type, but this is not necessary. Rather than thinking about equality as type families

$$Bool \to Bool \to \text{Type},$$

$$\text{or } \mathbb{N} \to \mathbb{N} \to \text{Type},$$

$$\text{or } (\mathbb{N} \to \text{Bool}) \to (\mathbb{N} \to \text{Bool}) \to \text{Type} \ldots$$

MLTT in Agda

(a) Function types $\to$,
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types,
(d) Disjoint union types $+$,
(e) Binary product types $\times$,
(f) Dependent function types Π,
(g) Dependent pair types Σ,
(h) Type universes $\mathsf{Type}_0, \mathsf{Type}_1, \ldots$,
(i) **Identity types $\equiv$.**

Propositional equality

...let's just think of it as a single type family.

This type family would have to depend on both the type and the two elements of that type that we are trying to show are equal.

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash x \equiv_A y : \mathsf{Type}_i} \ (\equiv\text{-Form})$$

Propositional equality

...let's just think of it as a single type family.

This type family would have to depend on both the type and the two elements of that type that we are trying to show are equal.

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash x \equiv_A y : \mathsf{Type}_i} \ (\equiv\text{-Form})$$

Propositional equality

But how would we introduce elements of these types?

When can we genuinely decide that two elements $x, y : X$ of any type $X : \text{Type}$ are equal?

Propositional equality

But how would we introduce elements of these types?

When can we genuinely decide that two elements $x, y : X$ of any type $X : \text{Type}$ are equal?

Propositional equality

But how would we introduce elements of these types?

When can we genuinely decide that two elements $x, y : X$ of any type $X : \text{Type}$ are equal?

Well... only when they are literally the same thing.

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash x : A}{\Gamma \vdash \text{refl } x : x \equiv_A x} \; (\equiv\text{-Intro})$$

Propositional equality

```
data _≡_ {i : Level} {X : Type i} : X → X → Type i where
 refl : (x : X) → x ≡ x
```

By the above, for any two elements $x, y : X$ of the same type X : Type there is a type $x \equiv y$: Type whose terms are identifications of x and y.

These types have one single constructor, which states the reflexivity, which states the reflexivity law of equality: every element $x : X$ is equal to itself.

Therefore, for now, the only way of introducing a term of these types is by writing refl $x : x \equiv x$; but we cannot *prove* that this is the *only* way of identifying two things[2].

[2]This is called Axiom K; it is by default on in Agda, but I have swithced it off for this course because it is not provable or disprovable in MLTT.

Propositional equality

```
data _≡_ {i : Level} {X : Type i} : X → X → Type i where
 refl : (x : X) → x ≡ x
```

By the above, for any two elements $x, y : X$ of the same type X : Type there is a type $x \equiv y$: Type whose terms are identifications of x and y.

These types have one single constructor, which states the reflexivity, which states the reflexivity law of equality: every element $x : X$ is equal to itself.

Therefore, for now, the only way of introducing a term of these types is by writing refl $x : x \equiv x$; but we cannot *prove* that this is the *only* way of identifying two things[2].

[2] This is called Axiom K; it is by default on in Agda, but I have swithced it off for this course because it is not provable or disprovable in MLTT.

Propositional equality

```
data _≡_ {i : Level} {X : Type i} : X → X → Type i where
 refl : (x : X) → x ≡ x
```

By the above, for any two elements $x, y : X$ of the same type X : Type there is a type $x \equiv y$: Type whose terms are identifications of x and y.

These types have one single constructor, which states the reflexivity, which states the reflexivity law of equality: every element $x : X$ is equal to itself.

Therefore, for now, the only way of introducing a term of these types is by writing refl $x : x \equiv x$; but we cannot *prove* that this is the *only* way of identifying two things[2].

[2]This is called Axiom K; it is by default on in Agda, but I have swithced it off for this course because it is not provable or disprovable in MLTT.

Propositional equality

All we can say for now is that the type $x \equiv y$: Type is definitely inhabited if x and y are genuinely (judgementally) equal.

Next time...

Next lecture, we will look at the identity type in more detail, exploring how this expands our idea of proof relevance in type theory. Finally, we will see how thinking about equality in MLTT leads us towards univalent type theory.

Please join me in the exercise classes, where you can get experience of programming Type Theory in Agda yourself!