Introduction to Type Theory in Agda

Lecture 2 – Propositions as types

Todd Waugh Ambridge

9 April 2024

Lecture Outline

1. Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$
2. Boolean Logic
   - `tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`
3. The Decidability Problem
4. Propositions-as-types Interpretation of Propositional Logic
   - $\mathbb{1}$, $\mathbb{0}$, $\rightarrow$, $\neg$, $+$, and $\times$
5. Predicate Logic via Dependent Types
   - Π and Σ

Lecture Outline

1. **Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$**
2. Boolean Logic
   - `tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`
3. The Decidability Problem
4. Propositions-as-types Interpretation of Propositional Logic
   - $\mathbb{1}$, $\mathbb{0}$, $\rightarrow$, $\neg$, $+$, and $\times$
5. Predicate Logic via Dependent Types
   - Π and Σ

Recap – Definition of $\mathbb{N}$

In the last lecture, we got started with type theory in Agda by defining the natural numbers $\mathbb{N}$ as an inductive type.

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}}\ (\mathbb{N}\text{-Form})$$

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_s)$$

```
data ℕ : Type where
  zero : ℕ
  succ : ℕ → ℕ
```

Recap – Definition of $\mathbb{N}$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, 0) = p_0 : P(0)} \ (\mathbb{N}\text{-Comp}_0)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash \substack{\mathbb{N}\text{-induction}(P, p_0, p_s, \mathsf{succ}\ n) \\ = p_s(n, \mathbb{N}\text{-}induction(P, p_0, p_s, n))} : P(\mathsf{succ}\ n)} \ (\mathbb{N}\text{-Comp}_s)$$

Recap – Definition of ℕ

```
ℕ-induction : (P : ℕ → Type)
            → (p₀ : P zero)
            → (pₛ : (n : ℕ) → P n → P (succ n))
            → (n : ℕ) → P n
ℕ-induction P p₀ pₛ zero = p₀
ℕ-induction P p₀ pₛ (succ n) = pₛ n IH
 where
  IH : P n
  IH = ℕ-induction P p₀ pₛ n
```

Recap – Definition of ℕ

In the exercise class, we defined some basic functions recursively by pattern matching...

```
add : ℕ → ℕ → ℕ
add zero m = m
add (succ n) m = succ (add n m)
```

...and also by using the induction principle, derived from the typing rules of natural numbers in MLTT.

```
add' : ℕ → ℕ → ℕ
add' n = ℕ-induction (λ _ → ℕ) n (λ _ r → succ r)
```

Recap – Definition of ℕ

Note that induction is *dependent*: it takes a type family $P : \mathbb{N} \to \text{Type}$ and returns a term $n : P\ n$.

```
ℕ-induction : (P : ℕ → Type)
            → (p₀ : P zero)
            → (pₛ : (n : ℕ) → P n → P (succ n))
            → (n : ℕ) → P n
```

But, when `P` is a constant function (as with `λ _ → ℕ` on the previous slide), we can instead think of this as a recursion principle.

```
ℕ-elim : {X : Type} → X → (ℕ → X → X) → ℕ → X
ℕ-elim {X} = ℕ-induction (λ _ → X)

add'' : ℕ → ℕ → ℕ
add'' n = ℕ-elim n (λ _ → succ)
```

Recap

Defining functions is all well and good, but Agda's true strength comes from the fact that it is a *proof assistant* as well as a programming language.

Recall that, last lecture, we learned that (via the Curry-Howard correspondence/propositions-as-types interpretation) MLTT corresponds to constructive predicate logic.

This means that, in our Agda setting, we can directly program proofs to mathematical statements.

Course Outline

Lecture 1: Introduction

Lecture 2: Propositions as types

Lecture 3: Equality and equivalence

Lecture 4: Towards univalent type theory

Course Outline

Lecture 1: Introduction

Lecture 2: Propositions as types

Lecture 3: Equality and equivalence

Lecture 4: Towards univalent type theory

Lecture Outline

1. Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$
2. **Boolean Logic**
   - **`tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`**
3. The Decidability Problem
4. Propositions-as-types Interpretation of Propositional Logic
   - $\mathbb{1}$, $\mathbb{0}$, $\rightarrow$, $\neg$, $+$, and $\times$
5. Predicate Logic via Dependent Types
   - Π and Σ

Boolean Logic – Type

Before we define a logic based on types, let's look at a logic we are more familiar with; i.e. logic as defined on the Booleans.

First, let's define a type of Boolean values.

```
data Bool : Type where
  tt ff : Bool
```

Boolean Logic – Type

We won't worry too much about this type's MLTT typing rules, but its induction (and, hence, recursion) principle is clear.

```
Bool-induction : (P : Bool → Type)
               → (pᵗ : P tt) (pᶠ : P ff)
               → (x : Bool) → P x
Bool-induction P pᵗ pᶠ tt = pᵗ
Bool-induction P pᵗ pᶠ ff = pᶠ

Bool-elim : A → A → Bool → A
Bool-elim {A} = Bool-induction (λ _ → A)
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- Disjunction,
- Conjunction.

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- **Truth,**
- Falsity,
- Implication,
- Negation,
- Disjunction,
- Conjunction.

```
tt : Bool
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- **Falsity,**
- Implication,
- Negation,
- Disjunction,
- Conjunction.

```
ff : Bool
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- **Implication,**
- Negation,
- Disjunction,
- Conjunction.

P	Q	$P \Rightarrow Q$
tt	tt	tt
tt	ff	ff
ff	tt	tt
ff	ff	tt

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- **Implication,**
- Negation,
- Disjunction,
- Conjunction.

```
_⇒_ : Bool → Bool → Bool
tt ⇒ Q = Q
ff ⇒ Q = tt
-- _⇒_ P = Bool-elim P tt
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- **Negation,**
- Disjunction,
- Conjunction.

P	$!P$
`tt`	`ff`
`ff`	`tt`

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- **Negation,**
- Disjunction,
- Conjunction.

```
!_ : Bool → Bool
! tt = ff
! ff = tt
-- !_ = Bool-elim ff tt
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- **Disjunction,**
- Conjunction.

P	Q	$P \mid\mid Q$
`tt`	`tt`	`tt`
`tt`	`ff`	`tt`
`ff`	`tt`	`tt`
`ff`	`ff`	`ff`

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- **Disjunction,**
- Conjunction.

```
_||_ : Bool → Bool → Bool
tt || Q = tt
ff || Q = Q
-- _||_ P = Bool-elim tt P
```

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- Disjunction,
- **Conjunction.**

P	Q	P `&&` Q
`tt`	`tt`	`tt`
`tt`	`ff`	`ff`
`ff`	`tt`	`ff`
`ff`	`ff`	`ff`

Boolean Logic – Propositional Connectives

Next, let's go as far as interpreting Boolean propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- Disjunction,
- **Conjunction.**

```
_&&_ : Bool → Bool → Bool
tt && Q = Q
ff && Q = ff
-- _&&_ Q = Bool-elim Q ff
```

Boolean Logic – Examples

Let's see some example computations of logical propositions:

Modus ponens says that if P and $P \Rightarrow Q$ hold, then Q holds.

```
modus-ponens' : Bool → Bool → Bool
modus-ponens' P Q = ((P ⇒ Q) && P) ⇒ Q
```

Modus tollendo ponens says that if P or Q hold, and not P holds, then Q holds.

```
modus-tollendo-ponens' : Bool → Bool → Bool
modus-tollendo-ponens' P Q = ((P || Q) && ! P) ⇒ Q
```

Boolean Logic – Examples

We can then check whether Agda correctly deduces these statements are true by computing the answer of their conjunction on all Boolean values.

```
run-ex : (Bool → Bool → Bool) → Bool
run-ex ex = ex tt tt
 && ex tt ff
 && ex ff tt
 && ex ff ff
```

We use (e.g.) `C-c C-n run-ex modus-ponens'` to run the computation.

Boolean Logic – Examples

We can also add numbers to our logic, and define propositions such as "Is n an odd number?".

```
is-odd? : ℕ → Bool
is-odd? 0 = ff
is-odd? 1 = tt
is-odd? (succ (succ n))
 = is-odd? n

is-odd?' : ℕ → Bool
is-odd?' 0 = ff
is-odd?' (succ n) = ! is-odd?' n
-- is-odd?' = ℕ-elim ff (λ _ r → ! r)
```

Lecture Outline

1. Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$
2. Boolean Logic
    - **`tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`**
3. **The Decidability Problem**
4. Propositions-as-types Interpretation of Propositional Logic
    - $\mathbb{1}$, $\mathbb{0}$, $\rightarrow$, $\neg$, $+$, and $\times$
5. Predicate Logic via Dependent Types
    - Π and Σ

The Decidability Problem

So far so good...

But the problem with Boolean logic is that it is not very expressive. There are many things we would like to reason about that cannot be captured by a `Bool`.

For example, let's say we want to prove that our two definitions of `is-odd?` are equal.

```
is-odd-functions-equal : Bool
is-odd-functions-equal = is-odd? === is-odd?'
```

We would first need to define equality `_===_` on functions $\mathbb{N}$ → `Bool`. But how can we do this?

The Decidability Problem

Perhaps something like...

```
_==_ : Bool → Bool → Bool
tt == Q = Q
ff == Q = ! Q
-- _==_ P = Bool-elim P (! P)

_===_ : (ℕ → Bool) → (ℕ → Bool) → Bool
f === g = (f 0 == g 0)
       && ((λ n → f (succ n)) === (λ n → g (succ n)))
```

But Agda cannot verify this will ever produce an answer.

The Decidability Problem

```
_===_ : (ℕ → Bool) → (ℕ → Bool) → Bool
f === g = (f 0 == g 0)
     && ((λ n → f (succ n)) === (λ n → g (succ n)))
```

This algorithm checks whether `f 0 == g 0`, and then `f 1 == g 1`, and then …

This procedure is clearly never going to terminate if each of these values reduces to `tt` — it will never return an answer.

Therefore, we cannot *decide* whether `f === g` should reduce to `tt`or `ff`.

The Decidability Problem

Propositions P that *can* be reduced to `tt` or `ff` are called *decidable propositions*.

The law of excluded middle, $P \vee \neg P$, holds for all decidable propositions.

Of course, if we take `Bool` as our type of propositions, then every definable proposition immediately satisfies the law of excluded middle by definition of being a Boolean.

```
is-decidable' : Bool → Bool
is-decidable' P = P || ! P

lem : Bool
lem = is-decidable' tt && is-decidable' ff
```

The Decidability Problem

In *classical mathematics* every proposition P is decidable, because the law of excluded middle holds in general.

But in *constructive mathematics*, like in programming, if we want to say that P is decidable, we have to actually compute the answer of whether or not P holds.

Therefore, the law of excluded middle does not hold in general, because there are questions that we cannot decide computationally: for example, the question of whether two programs are equal, or whether a program halts.

The Decidability Problem

So in constructive type theory, the Booleans will not do as our type of propositions. We need to use a more general type, that has an interpretation of truth and falsity, but that we cannot in general *decide* whether a given term of that type *is* true or false.

Using the propositions-as-types interpretation, the type of propositions is... Type itself.

A type A : Type can

- hold (i.e. if we can construct a term $a : A$),
- be shown to not hold (i.e. if we can prove a term of it cannot be constructed),

but we cannot decide in general whether a given type *has* a term that we can construct or not.

Lecture Outline

1. Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$
2. Boolean Logic
   - `tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`
3. The Decidability Problem
4. **Propositions-as-types Interpretation of Propositional Logic**
   - $\mathbb{1}$**,** $\mathbb{0}$**,** $\rightarrow$**,** $\neg$**,** $+$**, and** $\times$
5. Predicate Logic via Dependent Types
   - Π and Σ

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)*.

Propositions-as-types

Let's interpret Type-valued propositional logic with MLTT in Agda.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- Disjunction,
- Conjunction.

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- **Truth,**
- Falsity,
- Implication,
- Negation,
- Disjunction,
- Conjunction.

Propositions-as-types – Truth

When is a proposition true, constructively?

When a proof of the proposition can be yielded. If P is a type representing a proposition, then $p : P$ is a proof of that proposition.

So, if we can exhibit an element of a type, then the proposition that type interprets is true.

But which type should represent base truth itself?

Propositions-as-types – Truth

Technically, any type that we can exhibit a term of can represent truth, but it would feel strange to use (for example) $\mathbb{N}$.

This is because there are many elements of $\mathbb{N}$ and each has a different computational meaning. It would be better to choose a type that has a single element, so that once we see the element we know that this just means "true", with no additional computation content.

Truth is interpreted by a unit type $\mathbb{1}$: Type – a type with just one element – which we now define inductively.

Propositions-as-types – Truth

$$\frac{}{\Gamma \vdash \mathbb{1} : \mathsf{Type}} \ (\mathbb{1}\text{-Form}) \qquad \frac{}{\Gamma \vdash \star : \mathbb{1}} \ (\mathbb{1}\text{-Intro})$$

$$\frac{\Gamma, x : \mathbb{1} \vdash P(x) : \mathsf{Type} \quad \Gamma \vdash p : P(\star)}{\Gamma, x : \mathbb{1} \vdash \mathbb{1}\text{-induction}(P, p, x) : P(x)} \ (\mathbb{1}\text{-Elim})$$

$$\frac{\Gamma, x : \mathbb{1} \vdash P(x) : \mathsf{Type} \quad \Gamma \vdash p : P(\star)}{\Gamma \vdash \mathbb{1}\text{-induction}(P, p, \star) = p : P(\star)} \ (\mathbb{1}\text{-Comp})$$

Propositions-as-types – Truth

```
data 𝟙 : Type where
 ⋆ : 𝟙

𝟙-induction : (P : 𝟙 → Type)
            → (p⋆ : P ⋆)
            → (x : 𝟙) → P x
𝟙-induction P p⋆ ⋆ = p⋆

𝟙-elim : A → 𝟙 → A
𝟙-elim {A} = 𝟙-induction (λ _ → A)
```

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- **Falsity,**
- Implication,
- Negation,
- Disjunction,
- Conjunction.

Propositions-as-types – Falsity

If truth is an inhabited type, then falsity is an uninhabited type.

Falsity is interpreted by an empty type $\mathbb{0} : \mathsf{Type}$ – a type with no introduction rules.

Propositions-as-types – Falsity

$$\frac{}{\Gamma \vdash \mathbb{0} : \mathsf{Type}}\ (\mathbb{0}\text{-Form})$$

$$\frac{\Gamma, x : \mathbb{0} \vdash P(x) : \mathsf{Type}}{\Gamma, x : \mathbb{0} \vdash \mathbb{0}\text{-induction}(P, x) : P(x)}\ (\mathbb{0}\text{-Elim})$$

Propositions-as-types – Falsity

```
data 𝟘 : Type where

𝟘-induction : (P : 𝟘 → Type)
            → (x : 𝟘) → P x
𝟘-induction P ()

𝟘-elim : 𝟘 → A
𝟘-elim {A} = 𝟘-induction (λ _ → A)
```

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- **Implication,**
- Negation,
- Disjunction,
- Conjunction.

Propositions-as-types – Implication

Implication introduction says that $P \Rightarrow Q$ when an assumption of that P is true leads to the truth of Q.

With our interpretation of truth, this means that any term of P allows us to construct a term of Q.

So to interpret implication, we need a type whose terms are procedures that take terms of P and construct terms of Q.

This is just the function type $P \to Q$! We introduced the typing rules of these last lecture, and they are built-in to Agda.

Propositions-as-types – Implication

Implication is interpreted by function types. To illustrate this, the principle of modus ponens is just function application.

```
modus-ponens : (A → B) → A → B
modus-ponens f a = f a
```

Further, implication should be transitive – which is indeed the case for function application. From a programming point of view, this is just the composition of two functions.

```
→-transitive : (A → B) → (B → C) → (A → C)
→-transitive f g a = g (f a)

_∘_ : (B → C) → (A → B) → (A → C)
g ∘ f = λ a → g (f a)
```

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- **Negation,**
- Disjunction,
- Conjunction.

Propositions-as-types – Negation

Negation introduction says that a proposition P is false $\neg P$ if it implies a contradiction.

How can we interpret the idea that P leads to a contradiction as a type?
Hint: Have a think about what would constitute a contradiction to our theory.

Propositions-as-types – Negation

Constructing a term $x : \mathbb{0}$ would be contradictory! That would say that we have a way to introduce a term of a type that has no rules of introduction.

Negation $\neg P$ is interpreted as "P implies $\mathbb{0}$". This is a type that we do not define inductively – it arises from types we have already built (i.e. function types and the empty type).

```
¬_ : Type → Type
¬ X = X → 𝟘

modus-tollens : (A → B) → ¬ B → ¬ A
modus-tollens f ¬b = ¬b ∘ f
```

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- **Disjunction,**
- Conjunction.

Propositions-as-types – Disjunction

Disjunction introduction says that if we have P we have $P \vee Q$, and also if we have Q we have $P \vee Q$.

In imperative programming, this is like a `Union` class. There are two constructors: either by using an element of P or by using an element of Q.

In MLTT, these are called binary sum types, or tagged unions.

Propositions-as-types – Disjunction

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}} \ (+\text{-Form})$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}\ a : A + B} \ (+\text{-Intro}_{\mathsf{l}})$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type} \quad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}\ b : A + B} \ (+\text{-Intro}_{\mathsf{r}})$$

Propositions-as-types – Disjunction

```
data _+_ (A B : Type) : Type where
 inl : A → A + B
 inr : B → A + B
```

Propositions-as-types – Disjunction

$$\frac{\Gamma, x : A + B \vdash P(x) : \mathsf{Type} \quad \begin{array}{c}\Gamma, a{:}A \vdash p_a(a) : P(\mathsf{inl}\ a) \\ \Gamma, b{:}B \vdash p_b(b) : P(\mathsf{inr}\ b)\end{array}}{\Gamma, x : A + B \vdash +\text{-induction}(P, p_a, p_b, x) : P(x)} \ (+\text{-Elim})$$

$$\frac{\Gamma, x : A + B \vdash P(x) : \mathsf{Type} \quad \begin{array}{c}\Gamma, a{:}A \vdash p_a(a) : P(\mathsf{inl}\ a) \\ \Gamma, b{:}B \vdash p_b(b) : P(\mathsf{inr}\ b)\end{array}}{\Gamma, a : A \vdash +\text{-induction}(P, p_a, p_b, \mathsf{inl}\ a) = p_a(a) : P(\mathsf{inl}\ a)} \ (+\text{-Comp}_\mathsf{l})$$

$$\frac{\Gamma, x : A + B \vdash P(x) : \mathsf{Type} \quad \begin{array}{c}\Gamma, a{:}A \vdash p_a(a) : P(\mathsf{inl}\ a) \\ \Gamma, b{:}B \vdash p_b(b) : P(\mathsf{inr}\ b)\end{array}}{\Gamma, b : B \vdash +\text{-induction}(P, p_a, p_b, \mathsf{inr}\ b) = pb(b) : P(\mathsf{inr}\ b)} \ (+\text{-Comp}_\mathsf{r})$$

Propositions-as-types – Disjunction

```
+-induction : (P : A + B → Type)
            → (pa : (a : A) → P (inl a))
            → (pb : (b : B) → P (inr b))
            → (x : A + B) → P x
+-induction P pa pb (inl a) = pa a
+-induction P pa pb (inr b) = pb b

+-elim : A → A → B + C → A
+-elim {A} {B} {C} a_l a_r
 = +-induction (λ _ → A) (λ _ → a_l) (λ _ → a_r)
```

Propositions-as-types – Disjunction

Note that the `induction` and `elim` functions for $+$-types are similar to those for `Bool`, which makes sense given that both have two explicit ‘options’.

In fact, we can write the standard `if-then-else` function using $+$-types, rather than Booleans (which is the case in most programming contexts).

```
if_then_else_ : A + B → C → C → C
if x then y else z = +-elim y z x
```

Propositions-as-types – Disjunction

The interpretation of disjunction is the first key difference between classical and constructive logic.

In classical logic, knowing that $P \vee Q$ holds does not tell you which of the two cases holds. In order to decide a disjunction classically, you have to know that one of the two sides cannot hold.

```
modus-tollendo-ponens : A + B → ¬ A → B
modus-tollendo-ponens (inl a) ¬a = 𝟘-elim (¬a a)
modus-tollendo-ponens (inr b) ¬a = b
```

But, in constructive type theory, the 'tag' of the term (i.e. `inl` or `inr`) specifies explicitly whether the proof is "in the left" or "in the right" side of the disjunction.

Propositions-as-types

So, let's interpret Type-valued propositional logic.
In order to interpret propositional logic, we need interpretations of:

- Truth,
- Falsity,
- Implication,
- Negation,
- Disjunction,
- **Conjunction.**

Propositions-as-types – Conjunction

Finally, conjunction introduction says that if we have P and Q then we have $P \wedge Q$.

In imperative programming, this is like a `Pair` class:

```
public class Pair<K,V> {
    private K fst;
    private V snd;

    public K getFst() { return fst; }
    public V getSnd() { return snd; }
}
```

In MLTT, they are called binary product types.

Propositions-as-types – Conjunction

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}}\ (\times\text{-Form})$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ (\times\text{-Intro})$$

Propositions-as-types – Conjunction

Although the introduction rule for binary product types can be defined by giving a single *constructor* using `data`...

```
data _×_ (A B : Type) : Type where
 _,_ : A → B → A × B
```

...we can also define it by giving its two destructors using `record`; which are like classes in other programming languages.

```
record _×_ (A B : Type) : Type where
    field
     fst : A
     snd : B

    constructor _,_
```

Propositions-as-types – Conjunction

To compare the `data` and `record` definitions another way:

- The `data` definition defines $\times$-types by interpreting conjunction *introduction*,

$$A, B \vdash A \wedge B$$

- The `record` definition defines $\times$-types by interpreting conjunction *elimination*,

$$A \wedge B \vdash A, \qquad A \wedge B \vdash B.$$

Propositions-as-types – Conjunction

Also, optionally, records can contain methods, just like in other programming languages. The two below are just illustrative, as we will always prefer to directly use the eliminators.

```
record _×_ (A B : Type) : Type where
  field
   fst : A
   snd : B

  constructor _,_

  getFst : A
  getFst = fst

  getSnd : B
  getSnd = snd
```

Propositions-as-types – Conjunction

$$\frac{\Gamma, x : A \times B \vdash P(x) : \mathsf{Type} \qquad \Gamma, {\scriptstyle \begin{matrix} a:A, \\ b:B \end{matrix}} \vdash p((a,b)) : P((a,b))}{\Gamma, x : A \times B \vdash \times\text{-induction}(P, p, x) : P(x)} \ (\times\text{-Elim})$$

$$\frac{\Gamma, x : A \times B \vdash P(x) : \mathsf{Type} \qquad \Gamma, {\scriptstyle \begin{matrix} a:A, \\ b:B \end{matrix}} \vdash p((a,b)) : P((a,b))}{\Gamma, {\scriptstyle \begin{matrix} x:A, \\ y:B \end{matrix}} \vdash \times\text{-induction}(P, p, (x,y)) = p((x,y)) : P((x,y))} \ (\times\text{-Comp})$$

Propositions-as-types – Conjunction

```
×-induction : (P : A × B → Type)
            → (p× : (a : A) (b : B) → P (a , b))
            → (x : A × B) → P x
×-induction P p× (a , b) = p× a b

×-elim : (A → B → C) → (A × B → C)
×-elim {A} {B} {C} = ×-induction (λ _ → C)
```

The elimination principle for binary products is just the act of uncurrying the function. We can of course also curry the function.

```
uncurry = ×-elim

curry : (A × B → C) → (A → B → C)
curry f a b = f (a , b)
```

Propositions-as-types – Example & Type Families

As with our Boolean logic, let's add numbers to our propositions-as-types interpretation, so that we can reason about them.

```
is-odd : ℕ → Type
is-odd 0 = 𝟘
is-odd 1 = 𝟙
is-odd (succ (succ n)) = is-odd n
```

For any $n : \mathbb{N}$, the term `is-odd` n : Type is a dependent type, because it depends on the value of n : Type.

- If n is indeed odd, then `is-odd` $n = \mathbb{1}$: Type, and therefore the type is inhabited.
- Otherwise, `is-odd` $n = \mathbb{0}$: Type, and therefore it is empty.

Propositions-as-types – Example & Type Families

A term p : `is-odd` n is hence a *proof* that n is odd, because if it were even, it would have been impossible for us to have constructed such a term p.

```
5-is-odd : is-odd 5
5-is-odd = ⋆

84-is-not-odd : ¬ (is-odd 84)
84-is-not-odd ()
```

We call `is-odd` : $\mathbb{N} \to \text{Type}$ itself a (dependent) *type family*.

Lecture Outline

1. Recap – Definition of $\mathbb{N}$ and functions on $\mathbb{N}$
2. Boolean Logic
   - `tt`, `ff`, $\Rightarrow$, `!`, `||`, and `&&`
3. The Decidability Problem
4. Propositions-as-types Interpretation of Propositional Logic
   - $\mathbb{1}$, $\mathbb{0}$, $\rightarrow$, $\neg$, $+$, and $\times$
5. **Predicate Logic via Dependent Types**
   - **Π and Σ**

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` : $\mathbb{N} \to$ Type. But this isn't the first type family we've seen:

- $\_ + \_, \_ \times \_$: Type $\to$ Type $\to$ Type are binary type families,
- Each of the `induction` principles featured functions $P : X \to$ 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

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

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

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

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

Dependent Types – Π-types

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

```
Π : {X : Type} (Y : X → Type) → Type
Π {X} Y = (x : X) → Y x

syntax Π {X} (λ x → y) = Π x ꞉ X , y
```

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.

```
is-decidable : Type → Type
is-decidable X = X + ¬ X

odd-or-even : Π n : ℕ , is-decidable (is-odd n)
odd-or-even 0 = inr 𝟘-elim
odd-or-even 1 = inl ⋆
odd-or-even (succ (succ n)) = odd-or-even n
```

Dependent Types – Π-types

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

```
_<_ : ℕ → ℕ → Type
0      < 0      = 𝟘
0      < succ _ = 𝟙
succ _ < 0      = 𝟘
succ n < succ m = n < m
```

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

```
succ-is-bigger : Π n : ℕ , (n < succ n)
succ-is-bigger zero = ⋆
succ-is-bigger (succ n) = succ-is-bigger 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

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`.

```
record Σ {X : Type} (Y : X → Type) : Type where
  constructor _,_
  field
   fst : X
   snd : Y fst

syntax Σ {X} (λ x → y) = Σ x ꞉ X , y
```

Dependent Types – Σ-types

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

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

$$\frac{\substack{\Gamma, z:\Sigma_{(x:X)} Y \\ \vdash P(z):\mathsf{Type}} \quad \Gamma, \substack{w:X, \\ y:Y(w)} \vdash p((w,y)) : P((w,y))}{\Gamma, z : \Sigma_{(x:X)} Y \vdash \Sigma\text{-induction}(P, p, z) : P(z)} \; (\Sigma\text{-Elim})$$

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

Dependent Types – Σ-types

```
Σ-induction : {X : Type} {Y : X → Type}
            → (P : Σ Y → Type)
            → (p : (w : X) (y : Y w) → P (w , y))
            → (z : Σ Y) → P z
Σ-induction P p (x , y) = p x y
```

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

```
_×_ : Type → Type → Type
A × B = Σ a ꞉ A , B

×-induction = Σ-induction
```

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.

```
always-a-bigger-number : Π n : ℕ , Σ m : ℕ , n < m
always-a-bigger-number n = succ n , succ-is-bigger n
```

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*.

Next time...

Next lecture, we will look at formalising the identity type and type universes of MLTT in Agda, and see how this expands our idea of proof relevance in type theory.

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