Administrivia

Project proposal (out today) is due on Friday by 11:59PM.

There is a GitHub Codespace configuration for the repository **CS400-Lib.**

Agda: The Proof Assistant

Type Theory and Mechanized Reasoning
Lecture 14

CAS CS 400

Objectives

See how the **Curry–Howard Isomorphism** plays out in Agda.

See how to **translate mathematics** into Agda.

See how (dependent) **inductive data types** can be used to define better structures for proving in Agda.

Recap

Recall: Types are First-Class Values

```
Int : Set
Int = Nat & Nat

IsZero : Nat -> Set
IsZero zero = Unit
IsZero _ = Empty
```

Types have the type **Set.** They can appear as the arguments and return values of functions.

Recall: Function types "take arguments"

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x

head-test : Nat
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x

head-test : Nat
head-test = head (1 :: 2 :: 3 :: []) unit
```

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x

head-test : Nat
head-test = head (1 :: 2 :: 3 :: []) unit
```

When we pass an argument to a function, **we also pass that argument to the type.**

Recall: Function types "take arguments"

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit

head : {A : Set} -> (l : List A) -> (l-has-head : IsNonEmpty l) -> A
head (x :: xs) l-has-head = x

head-test : Nat
head-test = head (1 :: 2 :: 3 :: []) unit
```

When we pass an argument to a function, **we also pass that argument to the type.**

We can vary the behavior of the function based on the input.

Recall: Curry-Howard Isomorphism

Recall: Curry-Howard Isomorphism

$$x : A, y : B \vdash \lambda z\,.\,\text{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

Recall: Curry-Howard Isomorphism

$$x : A, y : B \vdash \lambda z\,.\,\text{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

$$A, B \vdash ((A \to \bot) \vee (B \to \bot)) \to \bot$$

Recall: Curry-Howard Isomorphism

$$x : A, y : B \vdash \lambda z\,.\,\mathsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

$$A, B \vdash ((A \to \bot) \vee (B \to \bot)) \to \bot$$

Given A and B, it is not the case that either A or B is false.

Recall: Curry-Howard Isomorphism

$$x : A, y : B \vdash \lambda z\,.\,\textsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

$$A, B \vdash ((A \to \bot) \vee (B \to \bot)) \to \bot$$

$$\boxed{\text{Given } A \text{ and } B\text{, it is not the case that either } A \text{ or } B \text{ is false.}}$$

When we derive a term to have a given type, we prove a theorem.

Recall: Curry-Howard Isomorphism

$$x : A, y : B \vdash \lambda z\,.\,\mathsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

$$A, B \vdash ((A \to \bot) \vee (B \to \bot)) \to \bot$$

$$\boxed{\begin{array}{c}\texttt{Given } A \texttt{ and } B\texttt{, it is not the case} \\ \texttt{that either } A \texttt{ or } B \texttt{ is false.}\end{array}}$$

When we derive a term to have a given type, we prove a theorem.

Types are Theorems. Programs are Proofs.

Curry-Howard Isomorphism in Agda

Interpreting Agda as Mathematics

```
m : A
m = ...
```

Interpreting Agda as Mathematics

```
m : A
m = ...
```

If m is of type **A** then **m** is a proof of A (read as a theorem).

Interpreting Agda as Mathematics

```
m : A
m = ...
```

If m is of type **A** then **m** is a proof of A (read as a theorem).

We can use this to reason about what it should mean to prove ***more complex mathematical statements***.

Interpreting Agda as Mathematics

```
m : A
m = ...
```

If m is of type **A** then **m** is a proof of A (read as a theorem).

We can use this to reason about what it should mean to prove ***more complex mathematical statements***.

Aside. *Is **2** a proof of **Nat**?*

Conjunction

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

```
data And A B : Set where
  _,_ : A -> B -> And A B
```

Conjunction

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

```
data And A B : Set where
  _,_ : A -> B -> And A B
```

To prove $A \wedge B$, I need to prove A and B.

Conjunction

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

```
data And A B : Set where
  _,_ : A -> B -> And A B
```

To prove $A \wedge B$, I need to prove A and B.

In Agda: a proof of $A \wedge B$ is a term m of type A together with a term n of type B, i.e., **a pair** (m, n).

Implication

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

```
f : A -> B
f x = M
```

Implication

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

```
f : A -> B
f x = M
```

To prove $A \to B$, I need to prove that, assuming A, I can prove B.

Implication

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

```
f : A -> B
f x = M
```

To prove $A \to B$, I need to prove that, assuming A, I can prove B.

In Agda: A proof of $A \to B$ is a term which, given m of type A, converts m into a proof of B, i.e., **a function from A's to B's.**

Disjunction

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl} M : A + B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

```
data Or A B : Set where
  left : A -> Or A B
  right : B -> Or A B
```

Disjunction

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl} M : A + B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

```
data Or A B : Set where
  left : A -> Or A B
  right : B -> Or A B
```

To prove $A \vee B$, I need to prove A or prove B.

Disjunction

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl} M : A + B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

```
data Or A B : Set where
  left : A -> Or A B
  right : B -> Or A B
```

To prove $A \vee B$, I need to prove A or prove B.

In Agda. A term of type $A \vee B$ is either a term of type A or a term of type B, i.e., **a element of the union of A and B.**

Truth

$$\frac{}{\vdash \text{unit} : \top}$$

```
data Unit : Set where
  unit : Unit
```

⊤ *has a trivial proof.*

In Agda: Unit.

Falsity

$$\frac{}{\varnothing \vdash \bot : \mathrm{Type}}$$

```
data Empty : Set where
```

⊥ has no proof.

In Agda: Empty.

Negation

```
Not : Set -> Set
Not A = A -> Empty
```

Negation

```
Not : Set -> Set
Not A = A -> Empty
```

To prove $\neg A$, I need to prove that, assuming A, I can prove a contradiction.

Negation

```
Not : Set -> Set
Not A = A -> Empty
```

To prove $\neg A$, I need to prove that, assuming A, I can prove a contradiction.

In Agda: A term of type $\neg A$ is a term of type $A \to \text{Empty}$.

CH-Isomorphism for Propositions

Logic	**Agda (CS400-Lib)**	**Type Theory**
proposition	`A`	Type
proof	`m : A`	Term
conjunction	`And A B, A & B, A /\ B`	Prod. type
implication	`A -> B`	Func. type
disjunction	`Or A B, Either A B, A \/ B`	Union type
truth	`Unit, True`	Unit type
falsity	`Empty, False`	Empty type
negation	`Not, A -> Empty`	

Aside: BHK Interpretation

Aside: BHK Interpretation

These are the **Brouwer–Heyting–Kolmogorov interpretations** of logical operators.

Aside: BHK Interpretation

These are the **Brouwer–Heyting–Kolmogorov interpretations** of logical operators.

What do logical operators require for their proofs.

Aside: BHK Interpretation

These are the **Brouwer–Heyting–Kolmogorov interpretations** of logical operators.

What do logical operators require for their proofs.

The case of disjunction departs from classical propositional logic.

Let's do a demo.

(De Morgan)

Agda and STLC

```
de-morgan-again : {A B : Set} ->
  A /\ B -> ((A -> Empty) \/ (B -> Empty)) -> Empty
de-morgan-again = \p -> \q ->
  case q (\f -> f (fst p)) (\g -> g (snd p))
```

Agda contains STLC as a fragment.

We have a bit more power with pattern matching.

An Aside: Constructive Mathematics

```
de-morgan-2 : {A B : Set} ->
  (((A -> Empty) \/ (B -> Empty)) -> Empty) -> A /\ B
de-morgan-2 prf = ?
```

It is much less clear how to prove this.

This has to do with the fact that Agda is used for ***constructive proofs***.

Translating Mathematics

Predicates and Properties

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit
```

Predicates and Properties

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit
```

If theorems are types, then ***properties*** are *indexed* types.

Predicates and Properties

```
IsNonEmpty : {A : Set} -> (l : List A) -> Set
IsNonEmpty [] = Empty
IsNonEmpty (x :: xs) = Unit
```

If theorems are types, then ***properties*** are *indexed* types.

Indexed types are types which differ according to a value.

Indexed Types as Inductive Data Types

```
data NonEmpty A : List A -> Set where
  has-head :
    (x : A) -> (xs : List A) -> NonEmpty A (x :: xs)
```

An ***indexed*** ADT is one which is a function type with return type **Set** (as opposed to just **Set**).

Recall: Equality

```
data _=P_ {A : Set} (x : A) : A -> Set where
  instance refl : x =P x

cong : {A B : Set} {x y : A}
  (f : A -> B) -> x =P y -> f x =P f y
cong f refl = refl
```

When we define indexed sets, the constructors define those cases in which the property holds.

Equality is indexed by two values, and only holds when those two values are the same.

Let's do a demo.

(Evenness)

Universal Quantification

```
even-or-odd : (n : Nat) -> even n \/ odd n
even-or-odd = ?
```

Universal Quantification

```
even-or-odd : (n : Nat) -> even n \/ odd n
even-or-odd = ?
```

To prove $\forall x : A(P(x))$, I need to prove $P(m)$ for any choice of m.

Universal Quantification

```
even-or-odd : (n : Nat) -> even n \/ odd n
even-or-odd = ?
```

To prove $\forall x : A (P(x))$*, I need to prove* $P(m)$ *for any choice of* m*.*

In Agda: A proof of $\forall x : A \,.\, P(x)$ is a term which, given m for type A, converts it to a proof of $P(m)$, i.e., **a function of type** $(x : A) \to P(x)$.

Example: In English

Example: In English

Theorem. For any natural number n,

Example: In English

Theorem. For any natural number n,

$$n + 0 = n$$

Example: In English

Theorem. For any natural number n,

$$n + 0 = n$$

Proof. By induction on n. If $n = 0$ then $0 + 0 = 0$. Suppose $n = k$ and that $k + 0 = k$. Then

Example: In English

Theorem. For any natural number n,

$$n + 0 = n$$

Proof. By induction on n. If $n = 0$ then $0 + 0 = 0$. Suppose $n = k$ and that $k + 0 = k$. Then

$$(Sk) + 0 = S(k + 0) = Sk$$

Example: In English

Theorem. For any natural number n,

$$n + 0 = n$$

Proof. By induction on n. If $n = 0$ then $0 + 0 = 0$. Suppose $n = k$ and that $k + 0 = k$. Then

$$(Sk) + 0 = S(k + 0) = Sk$$

Where the first equality is the definition of addition and the second is replacement of $k + 0$ by k according to the inductive hypothesis.

Example: In "Formal" Math

Example: In "Formal" Math

$P(k) = \text{"}k + 0\text{"}$

Example: In "Formal" Math

$P(k) = \text{"}k + 0\text{"}$

base case: $P(0) = \text{"}0 + 0 = 0\text{"}$ which holds by reflexivity of equality.

Example: In "Formal" Math

$P(k) = $ "$k + 0$"

base case: $P(0) = $ "$0 + 0 = 0$" which holds by reflexivity of equality.

inductive step: If $P(k)$ holds, then $k + 0 = k$. Apply S to both sides to get $S(k + 0) = Sk$. Then $S(k + 0) = (Sk) + 0$ by definition of addition.

Example: In "Formal" Math

$P(k) = "k + 0"$

base case: $P(0) = "0 + 0 = 0"$ which holds by reflexivity of equality.

inductive step: If $P(k)$ holds, then $k + 0 = k$. Apply S to both sides to get $S(k + 0) = Sk$. Then $S(k + 0) = (Sk) + 0$ by definition of addition.

Apply induction on natural numbers with P, **base case,** and **inductive step.**

Induction Principle on Natural Numbers

For any property P of natural numbers, if $P(0)$ holds, and $P(k)$ implies $P(k+1)$ for any choice of k, then $P(n)$ holds for any n.

Induction Principle on Natural Numbers

For any **P : Nat -> Set**, if **(P 0)** holds, and **(P k)** implies **(P (suc k))** for any choice of k, then **(P n)** holds for any **n**.

Induction Principle on Natural Numbers

For any **P : Nat -> Set**, if
(P 0) holds, and
(P k) -> (P (suc k)) for any choice of k, then
(P n) holds for any **n.**

Induction Principle on Natural Numbers

For any **P : Nat -> Set,**
(base : P 0) ->
(ind-hyp: P k -> P (suc k) for any choice of k**) ->**
P n holds for any **n.**

Induction Principle on Natural Numbers

```
(P : Nat -> Set) ->
(base: P 0) ->
(ind-hype: (k : Nat) -> P k -> P (suc k)) ->
(n : Nat) -> (P n)
```

`This is just a type in Agda.`

Proof in Agda

```
n+0=n : (n : Nat) -> n + 0 =P n
n+0=n = qed where
  P : Nat -> Set
  P k = (k + 0) =P k

  base-case : (0 + 0) =P 0
  base-case = refl

  ind-step : (k : Nat) -> (pk : k + 0 =P k)-> (suc k) + 0 =P suc k
  ind-step k pk = cong suc pk

  qed = ind-Nat P base-case ind-step
```

Proof of Induction

```
ind-Nat : (P : Nat -> Set) (base : P zero)
  (ind-hyp : (k : Nat) -> P k -> P (suc k))
  (n : Nat) -> P n
ind-Nat P p0 ind-hyp zero = p0
ind-Nat P p0 ind-hyp (suc n) =
  ind-hyp n (ind-Nat P p0 ind-hyp n)
```

We can prove induction within Agda.

It's just pattern matching.

Simplified Proof in Agda

```
n+0=n' : (n : Nat) -> (n + 0) =P n
n+0=n' zero = refl
n+0=n' (suc n) = cong suc (n+0=n n)
```

We can just pattern match on the input.

Pattern matching is ***proof by cases.***

The recursive call is ***the inductive hypothesis.***

Let's do a demo

(snoc)

Existential Quantification

```
data Exists {A : Set} (P : A -> Set) : Set where
  wit : (x : A) -> (prf : P x) -> Exists P

syntax Exists (\x -> B) = [ x ] B
```

Existential Quantification

```
data Exists {A : Set} (P : A -> Set) : Set where
  wit : (x : A) -> (prf : P x) -> Exists P

syntax Exists (\x -> B) = [ x ] B
```

To prove $\exists x : A(P(x))$, I need to find a value m and prove that $P(m)$.

Existential Quantification

```
data Exists {A : Set} (P : A -> Set) : Set where
  wit : (x : A) -> (prf : P x) -> Exists P

syntax Exists (\x -> B) = [ x ] B
```

To prove $\exists x : A(P(x))$*, I need to find a value* m *and prove that* $P(m)$*.*

In Agda: A proof of type $\exists x : A.(P(x))$ is a term which consists of a term m of type A and a term of type $P(m)$, i.e., **a dependent pair.**

Let's do a demo

(evenness again)

Extra demo (if there's time)

(reflection)