Proofs are Programs

Basics (continued) and Induction

Short Recap

Enumerated Types

```
Inductive bool : Type :=
  | true
  | false.
```

Function Evaluation

```
Compute (orb true false).
(* ==> true : bool *)
```

Functions with pattern matching

```
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

Short Recap

Enumerated Types

```
Inductive bool : Type :=
  | true
  | false.
```

Function Evaluation

```
Compute (orb true false).
(* ==> true : bool *)
```

Specific way of function application

Functions with pattern matching

```
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

Types

Types

```
Check true.
(* ===> true : bool *)
```

Gives the type of an expression

Types

```
Check true.
(* ===> true : bool *)
```

Gives the type of an expression

```
Check true
  : bool.
```

Only goes through if the correct type is provided

Types

```
Check true.
(* ===> true : bool *)
```

Gives the type of an expression

```
Check true
  : bool.
```

Only goes through if the correct type is provided

```
Check (negb true)
  : bool.
```

```
Check negb
  : bool -> bool.
```

Types

```
Check true.
(* ===> true : bool *)
```

Gives the type of an expression

```
Check true
  : bool.
```

Only goes through if the correct type is provided

```
Check (negb true)
  : bool.
```

```
Check negb
  : bool -> bool.
```

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

New Types from Old

```
Inductive rgb : Type :=
  | red
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

New Types from Old

```
Inductive rgb : Type :=
  | red
  | green
  | blue.
```

```
Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

```
Check primary.
(* ==> primary: rgb -> color *)
```

New Types from Old

```
Inductive rgb : Type :=
  | red
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

```
Definition monochrome (c : color) : bool :=
  match c with
  | black => true
  | white => true
  | primary p => false
  end.
```

```
Check primary.
(* ==> primary: rgb -> color *)
```

New Types from Old

```
Inductive rgb : Type :=
  | red
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

```
Check primary.
(* ==> primary: rgb -> color *)
```

```
Definition monochrome (c : color) : bool :=
  match c with
  | black => true
  | white => true
  | primary p => false
  end.

Definition isred (c : color) : bool :=
  match c with
  | black => false
  | white => false
  | primary red => true
  | primary _ => false
  end.
```

Tuples

```
Inductive bit : Type :=
  | B0
  | B1.
```

```
Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).
```

Tuples

```
Inductive bit : Type :=
  | B0
  | B1.
```

```
Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).
```

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
  | (bits B0 B0 B0 B0) => true
  | (bits _ _ _ _) => false
  end.
```

Tuples

```
Inductive bit : Type :=
  | B0
  | B1.
```

```
Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).
```

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
  | (bits B0 B0 B0 B0) => true
  | (bits _ _ _ _) => false
  end.
```

```
Compute (all_zero (bits B1 B0 B1 B0)).
(* ===> false : bool *)
```

```
Compute (all_zero (bits B0 B0 B0 B0)).
(* ===> true : bool *)
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

O ⟷ 0

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

```
  O      <-----> 0
  S O    <-----> 1
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

	O	0
|	S O	1
||	S (S O)	2

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

```
    O              <-------->  0
|   S O            <-------->  1
||  S (S O)        <-------->  2
||| S (S (S O))    <-------->  3
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

```
     O               ↔  0
|    S O             ↔  1
||   S (S O)         ↔  2
|||  S (S (S O))     ↔  3
```

```
Inductive nat' : Type :=
  | stop
  | tick (foo : nat').
```

Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Check S.
(* ==> S: nat -> nat *)
```

```
|    O              <------->  0  <------->   stop
||   S O            <------->  1  <------->   tick stop
|||  S (S O)        <------->  2  <------->   tick (tick stop)
     S (S (S O))    <------->  3  <------->   tick (tick (tick stop))
```

```
Inductive nat' : Type :=
  | stop
  | tick (foo : nat').
```

Computing with Natural Numbers

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

```
Definition minustwo (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

Computing with Natural Numbers

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

```
Compute (match 2 with
         | (S (S O)) => true
         | _ => false end).
(* ==> true: bool *)
```

```
Definition minustwo (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

Computing with Natural Numbers

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

```
Compute (match 2 with
         | (S (S O)) => true
         | _ => false end).
(* ==> true: bool *)
```

```
Definition minustwo (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

```
Compute (minustwo 4).
(* ===> 2 : nat *)
```

Recursive Functions

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O          => true
  | S O        => false
  | S (S n') => even n'
  end.
```

Recursive Functions

Keyword for recursive function

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => even n'
  end.
```

Recursive Functions

Keyword for recursive function

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => even n'
  end.
```

Recursive Call

Recursive Functions

Keyword for recursive function

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => even n'
  end.
```

Recursive Call

```
Definition odd (n:nat) : bool :=
  negb (even n).
```

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$n + m = S^n(m)$$

$$n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}$$

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$\boxed{n + m = S^n(m)}$$

$$\boxed{n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}}$$

```
plus (S (S (S O))) (S (S O))
```

$3 + 2$

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$n + m = S^n(m)$$

$$n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}$$

```
plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O)))
```

$$3 + 2$$

$$= \boxed{1 +} (2 + 2)$$

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$n + m = S^n(m)$$

$$n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}$$

```
plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O)))
==> S (S (plus (S O) (S (S O))))
```

$$3 + 2$$

$$= 1 + (2 + 2)$$

$$= 1 + (1 + (1 + 2))$$

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$n + m = S^n(m)$$

$$n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}$$

```
plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O)))
==> S (S (plus (S O) (S (S O))))
==> S (S (S (plus O (S (S O)))))
```

$$\begin{aligned} & 3 + 2 \\ &= \boxed{1 +} (2 + 2) \\ &= 1 + (\boxed{1 +} (1 + 2)) \\ &= 1 + (1 + (\boxed{1 +} (0 + 2))) \end{aligned}$$

Multi-argument Recursive Functions

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

$$n + m = S^n(m)$$

$$n + m = \begin{cases} m & n = 0 \\ (n' + m) + 1 & n = n' + 1 \end{cases}$$

```
plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O)))
==> S (S (plus (S O) (S (S O))))
==> S (S (S (plus O (S (S O)))))
==> S (S (S (S (S O))))
```

$$\begin{aligned} & 3 + 2 \\ &= \boxed{1 +} (2 + 2) \\ &= 1 + (\boxed{1 +} (1 + 2)) \\ &= 1 + (1 + (\boxed{1 +} (0 + 2))) \\ &= 1 + (1 + (1 + 2)) \end{aligned}$$

More Arithmetic Functions

```
Fixpoint mult (n m : nat) : nat :=
  match n with
  | O => O
  | S n' => plus m (mult n' m)
  end.
```

$$n * m = \begin{cases} 0 & n = 0 \\ m + (n' * m) & n = n' + 1 \end{cases}$$

More Arithmetic Functions

```
Fixpoint mult (n m : nat) : nat :=
  match n with
  | O => O
  | S n' => plus m (mult n' m)
  end.
```

$$n * m = \begin{cases} 0 & n = 0 \\ m + (n' * m) & n = n' + 1 \end{cases}$$

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O   , _    => O
  | S _ , O    => n
  | S n', S m' => minus n' m'
  end.
```

$$n - m = \begin{cases} 0 & n = 0 \\ n & n = n' + 1 \ \wedge \ m = 0 \\ n' - m' & n = n' + 1 \ \wedge \ m = m' + 1 \end{cases}$$

Equality

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
```

Equality

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

```
Compute (4 =? 2).
(* ==> false: bool *)
```

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
```

Equality

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

```
Compute (4 =? 2).
(* ==> false: bool *)
```

```
Compute (4 = 2).
(* ==> 4 = 2: Prop *)
```

Statement (proposition) to be proven that 4=2 (more details on that soon)

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
```

Proofs by Simplification

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
```

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
```

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.

  intros n.
```

Let n be a natural number. We show that 0 + n = n

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.

  intros n.

  simpl.
```

Let n be a natural number. We show that 0 + n = n

From the definition of + we know that 0 + n computes to n. So it is left to show that n = n

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.
  reflexivity.
```

Let n be a natural number. We show that 0 + n = n

From the definition of + we know that 0 + n computes to n. So it is left to show that n = n

By the reflexivity of equality, it holds that n = n is true

Proofs by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.

  intros n.

  simpl.

  reflexivity.

  Qed.
```

Let n be a natural number. We show that 0 + n = n

From the definition of + we know that 0 + n computes to n. So it is left to show that n = n

By the reflexivity of equality, it holds that n = n is true

Proofs by Rewriting

Proofs by Rewriting

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.
```

Proofs by Rewriting

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.

  intros n m.
```

Let n and m be natural numbers. We show that if n = m then n+n = m+m

Proofs by Rewriting

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.

  intros n m.

  intros H.
```

Let n and m be natural numbers. We show that if n = m then n+n = m+m

Assume that n = m (refer to this assumption as H)

Proofs by Rewriting

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.

  intros n m.

  intros H.

  rewrite -> H.
```

Let n and m be natural numbers. We show that if n = m then n+n = m+m

Assume that n = m (refer to this assumption as H)

Use assumption H to replace n with m. We are left to show that m+m = m+m

Proofs by Rewriting

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.

  intros n m.

  intros H.

  rewrite -> H.

  reflexivity.  Qed.
```

- `intros n m.` — Let n and m be natural numbers. We show that if n = m then n+n = m+m
- `intros H.` — Assume that n = m (refer to this assumption as H)
- `rewrite -> H.` — Use assumption H to replace n with m. We are left to show that m+m = m+m
- `reflexivity. Qed.` — By the reflexivity of equality, it holds that m+m = m+m is true

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.
```

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.
```

```
Theorem plus_1_neq_0 : forall n : nat,
  (eqb (plus n (S O)) 0) = false.
```

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.
```

```
Theorem plus_1_neq_0 : forall n : nat,
  (eqb (plus n (S O)) 0) = false.
```

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.
```

```
Theorem plus_1_neq_0 : forall n : nat,
  (eqb (plus n (S O)) 0) = false.
```

Does not simplify because plus is defined by recursion on first argument

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

Proofs by Case Analysis

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
```

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.

  destruct n as [| n'] eqn:E.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

Case analysis on the natural number n

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.

  destruct n as [| n'] eqn:E.

  - reflexivity.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

Case analysis on the natural number n

Case 1) Let n = 0. Then by the definition of plus and eqb, ((0 + 1) =? 0) evaluates to false.

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.

  destruct n as [| n'] eqn:E.

  - reflexivity.

  - simpl.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

Case analysis on the natural number n

Case 1) Let n = 0. Then by the definition of plus and eqb, ((0 + 1) =? 0) evaluates to false.

Case 2) Let n = S n’ (E). Then by the definition of plus and eqb,
((S n’ + 1) =? 0) ==> (S (n’ + 1) =? 0) ==> false

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

```
  destruct n as [| n'] eqn:E.
```

Case analysis on the natural number n

```
  - reflexivity.
```

Case 1) Let n = 0. Then by the definition of plus and eqb, ((0 + 1) =? 0) evaluates to false.

```
  - simpl.
```

Case 2) Let n = S n' (E). Then by the definition of plus and eqb,
((S n' + 1) =? 0) ==> (S (n' + 1) =? 0) ==> false

```
    reflexivity.
```

By the reflexivity of equality, it holds that m+m = m+m is true

Proofs by Case Analysis

```
Theorem plus_1_neq_0 : forall n : nat,
  ((n + 1) =? 0) = false.

Proof.
  intros n.

  destruct n as [| n'] eqn:E.

  - reflexivity.

  - simpl.


    reflexivity.
  Qed.
```

Let n be a natural number. We show that ((n + 1) =? 0) = false

Case analysis on the natural number n

Case 1) Let n = 0. Then by the definition of plus and eqb, ((0 + 1) =? 0) evaluates to false.

Case 2) Let n = S n' (E). Then by the definition of plus and eqb,
((S n' + 1) =? 0) ==> (S (n' + 1) =? 0) ==> false

By the reflexivity of equality, it holds that m+m = m+m is true