Dependent inductive types and logical connectives

Yves Bertot

April 15, 2008

Flashback: inductive datatypes

- Monomorphic: only define a single type,
- Each contructor gives a way to build elements in the type,
- Arguments in the same type are allowed for the constructors,
- Example:
  Inductive tree : Type := L : tree | B : tree → tree → tree.
- Alternative notation
  Inductive tree : Type := L | B (t1 t2 : tree).

Flashback: inductive datatypes

- Monomorphic: only define a single type,
- Each contructor gives a way to build elements in the type,
- Arguments in the same type are allowed for the constructors,
- Example:
  Inductive tree : Type := L : tree | B : tree → tree → tree.
- Alternative notation
  Inductive tree : Type := L | B (t1 t2 : tree).
- Generalize to *parameterized* types, where the parameter does not change.
  Inductive list (A:Type) : Type :=
  nil : list A | cons : A → list A → list A.

Families of datatypes

- constructors may have arguments in other types of the same family,
- Known example vectors:
  Inductive vector (A:Type) : nat → Type :=
    VNil : vector A 0
  | VCons : ∀n : nat, A → vector A n → vector A (S n).

Case-by-case reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → vector A n → vector A (S n).
```

Case-by-case reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → vector A n → vector A (S n).
vector_cases :
```

Case-by-case reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → vector A n → vector A (S n).
vector_cases :
∀A : Type ∀P : ∀n : nat, vector n A → Prop,
```

Case-by-case reasoning for a family of inductive types

Inductive vector (A:Type) : nat $\rightarrow$ Type :=
VNil : vector A 0
| VCons : $\forall$n : nat, A $\rightarrow$ vector A n $\rightarrow$ vector A (S n).

vector_cases :
$\forall$A : Type $\forall$P : $\forall$n : nat, vector n A $\rightarrow$ Prop,
P 0 (VNil A) $\rightarrow$

Case-by-case reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → vector A n → vector A (S n).
vector_cases :
∀A : Type ∀P : ∀n : nat, vector n A → Prop,
P 0 (VNil A) →
(∀(n:nat) (a:A) (v:vector n), P (S n) (Vcons A n a v)) →
```

Case-by-case reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → vector A n → vector A (S n).
vector_cases :
∀A : Type ∀P : ∀n : nat, vector n A → Prop,
P 0 (VNil A) →
(∀(n:nat) (a:A) (v:vector n), P (S n) (Vcons A n a v)) →
∀(n:nat) (v:vector A n), P n v
```

Induction reasoning for a family of inductive types

```
Inductive vector (A:Type) : nat → Type :=
  VNil : vector A 0
| VCons : ∀n : nat, A → list A → list A.
vector_ind :
∀A : Type ∀P : ∀n : nat, vector n A → Prop,
P 0 (VNil A) →
(∀(n:nat) (a:A) (v:vector n), P n v →P (S n) (Vcons A n a v)) →
∀(n:nat) (v:vector A n), P n v
```

Curry Howard Isomorphism on Inductive Families

- For vectors, for every n in nat, the type vector A n contains at least one element,
- Why not design inductive families where only some indices have an *inhabited* type in the family,
- Example
  Inductive ev : nat → Type :=
  ev0 : ev 0 | ev2 : ∀n, ev n → ev (S (S n)).
- The constructors make it possible to build elements in ev 0, ev 2, ev 4, . . .
- To prove that ev 1 is not inhabited, we need an induction principle.

Induction principle for ev

- Induction principle
  ev_ind: $\forall$P: nat $\rightarrow$ Prop,
  P 0 ev0 $\rightarrow$ ($\forall$(n:nat) (e:ev n), P n e $\rightarrow$ P (S (S n)) (ev2 n e))
  $\rightarrow$ $\forall$(n:nat) (e:ev n), P n (ev n)
- Let's prove that ev 1 is empty.
  Lemma ex_ev1 : $\forall$(n:nat) (e:ev n), n <> 1.

Induction principle for ev

- Induction principle
  ev_ind: ∀P: nat → Prop,
  P 0 ev0 → (∀(n:nat) (e:ev n), P n e → P (S (S n)) (ev2 n e))
  → ∀(n:nat) (e:ev n), P n (ev n)
- Let's prove that ev 1 is empty.
  Lemma ex_ev1 : ∀(n:nat) (e:ev n), n <> 1.
  intros n e; elim e.
- At this point, try to apply ev_ind, with its 5th argument matching e,
- P is chosen so that P n e ≡ n <> 1.
  ====================
  0 <> 1
  discriminate.

ev 1 is not inhabited

- Second subgoal:

  ```
  ====================
  ∀n', ev n' → n' <> 1 → S (S n') <> 1
  ```

ev 1 is not inhabited

- Second subgoal:

  ====================

  ∀n', ev n' → n' <> 1 → S (S n') <> 1

  intros n' _ _; discriminate. Qed.

ev 1 is not inhabited

- Second subgoal:
  ====================
  ∀n', ev n' → n' <> 1 → S (S n') <> 1
  intros n' _ _; discriminate. Qed.
- Exercice, prove that ∀x, ev x → ∃ y, x = y+y

Inductive predicates

- Make a systematic use of inductive families where all members may not be inhabited,
- Declare explicitely that elements are irrelevant,
- Adapt the induction principle.

Inductive predicates

- Make a systematic use of inductive families where all members may not be inhabited,
- Declare explicitly that elements are irrelevant,
- Adapt the induction principle.
- Inductive even : nat → Prop :=
  even0 : even 0
  | even2 : ∀n:nat, even n → even (S (S n)).
- Simpler induction principle:
  ∀P: n → Prop,
  P 0 →
  (∀n, even n → P n → P (S (S n))) →
  ∀n:nat, even n → P n

Designing inductive predicates

- Always remember that the constructors should state theorems that you want to be true,
- Do not forget that the arrow is not a “rewriting” step,
- Always test that you can prove a few basic facts.

Example of wrong design

- What happens with the following definition:
  Inductive wev : nat → Prop :=
  wev0 : wev 0
  | wev2 : ∀n, wev (S (S n)) → wev n.
- Why would you write this? To reduce the problem of proving that a large number is even to a simpler problem?
- Remember that the proof process reads implications backward.
- Here you would never be able to prove wev 2,
- Exercise: prove ~wev 2.
- Exercise: define divides inductively, prove th1 and th2 from the first lecture.

Choosing proofs by induction

- When proving a property on an object that satisfies an inductive predicate,
- Two solutions
  - Either prove by induction on the object (if possible),
  - Or prove by induction on the inductive predicate.

Example of wrong choice

- Lemma even_plus : forall x y, even x $\rightarrow$ even y $\rightarrow$ even (x+y).
  intros x; elim x.
  intros y _ evy; exact evy.
- The first case was easy!

Example of wrong choice

- Lemma even_plus : forall x y, even x $\rightarrow$ even y $\rightarrow$ even (x+y).
  intros x; elim x.
  intros y _ evy; exact evy.
- The first case was easy!
  ===========
  $\forall$n, ($\forall$y, even n $\rightarrow$ even y $\rightarrow$ even (n+y)) $\rightarrow$
  $\forall$y, even (S n) $\rightarrow$ even y $\rightarrow$ even((S n)+y)
- Here even (S n) cannot be used to fill the premise of the induction hypothesis.

Example of good choice

- Lemma even_plus : forall x y, even x → even y → even (x+y).
  intros x y evx evy; elim evx.
  evy : even y
  ==========
  even (0 + y)

Example of good choice

- Lemma even_plus : forall x y, even x → even y → even (x+y).
  intros x y evx evy; elim evx.
  evy : even y
  ==========
  even (0 + y)
  exact evy.
- The first case is also easy!

Example of good choice

- Lemma even_plus : forall x y, even x → even y → even (x+y).
  intros x y evx evy; elim evx.
  evy : even y
  ============
  even (0 + y)
  exact evy.
- The first case is also easy!
  ============
  ∀n, even n → even (n+y) → even (S (S n) + y)
  intros n _ evny; simpl.
- Force addition to compute,
  ============
  even (S (S (n + y)))
  apply even2; exact evny.
  Qed.

Inversion

- Some instances of an inductive predicate can be proved by a single constructor,
  - for instance even (S x) can only be proved by even2,
- In this case, the premises of this constructor must hold,
  - for even, if we know even (S (S x)) we can deduce even x
- In general, this means some implications can be read the other way round,
- This is done by a tactic called inversion.

Things that can be described using Inductive predicates

- order relations

  ```
  Inductive le (n:nat) : nat → Prop :=
    le_n : le n n | le_S : ∀m, le n m → le n (S m).
  ```

- Partial functions, viewed as a relation

  ```
  Inductive rsyracuse : nat → nat → Prop :=
    ps_1 : rsyracuse 1 0
  | rs_p : ∀x n, rsyracuse x n → rsyracuse (2*x) (n+1)
  | rs_o : ∀x n, ~even x → rsyracuse (3*x+1) n →
          rsyracuse x (n+1).
  ```

- Also many-to-many relations,
- Very useful for programming language semantics.

Logical connectives as Inductive predicates

- Conjunction and Disjunction

  Inductive and (A B:Prop) : Prop :=
   conj : A → B → and A B.

Logical connectives as Inductive predicates

- Conjunction and Disjunction

  ```
  Inductive and (A B:Prop) : Prop :=
    conj : A → B → and A B.
  and_ind: ∀A B P:Prop,
  ```

Logical connectives as Inductive predicates

- Conjunction and Disjunction

  Inductive and (A B:Prop) : Prop :=
   conj : A → B → and A B.
  and_ind: ∀A B P:Prop,(A → B → P)

Logical connectives as Inductive predicates

- Conjunction and Disjunction

  Inductive and (A B:Prop) : Prop :=
  conj : A → B → and A B.

  and_ind: ∀A B P:Prop,(A → B → P) → A /\ B → P

  Inductive or (A B:Prop) : Prop :=
  or_introl : $\boxed{\text{A} \rightarrow \text{or A B}}$
  | or_intror : B → or A B.

Logical connectives as Inductive predicates

- Conjunction and Disjunction

  Inductive and (A B:Prop) : Prop :=
  conj : A → B → and A B.
  and_ind: ∀A B P:Prop,(A → B → P) → A /\ B → P

  Inductive or (A B:Prop) : Prop :=
  or_introl : $\boxed{\text{A} \rightarrow \text{or A B}}$
  | or_intror : B → or A B.
  or_ind: ∀A B P:Prop, $\boxed{\text{(A} \rightarrow \text{P)}}$ → (B → P) → A \/ B → P
- The tactic elim uses and_ind and or_ind,
- The tactic case or destruct use a more primitive but equivalent mechanism (pattern-matching).

Equality as an inductive predicate

- Inductive eq (A : Type) (x : A) : A → Prop :=
  refl_equal : eq A x x.

Equality as an inductive predicate

- Inductive eq (A : Type) (x : A) : A $\rightarrow$ Prop :=
  refl_equal : eq A x x.
  eq_ind: $\forall$(A:Type) (x:A) (P : A $\rightarrow$ Prop),

Equality as an inductive predicate

- Inductive eq (A : Type) (x : A) : A → Prop :=
  refl_equal : eq A x x.

  eq_ind: ∀(A:Type) (x:A) (P : A → Prop),
    P x

Equality as an inductive predicate

- Inductive eq (A : Type) (x : A) : A → Prop :=
  refl_equal : eq A x x.

  eq_ind: ∀(A:Type) (x:A) (P : A → Prop),
  P x → ∀y:A, x = y → P y

Equality as an inductive predicate

- Inductive eq (A : Type) (x : A) : A → Prop :=
  refl_equal : eq A x x.
  eq_ind: ∀(A:Type) (x:A) (P : A → Prop),
  P x → ∀y:A, x = y → P y
- The tactic elim applies eq_ind,
- In practice, we start with a goal of the form P y, and we end up with a goal of the form P x,
- Instances of y have been replaced by instances of x: rewriting to the left,
- The tactic rewrite ← uses eq_ind, rewrite → uses a symmetric eq_ind_r.

Existential quantification as an inductive predicate

- Inductive ex (A : Prop)(P : A $\rightarrow$ Prop) : Prop :=
  ex_intro : $\forall$x, P x $\rightarrow$ ex A P.

Existential quantification as an inductive predicate

- Inductive ex (A : Prop)(P : A → Prop) : Prop :=
  ex_intro : ∀x, P x → ex A P.
  ex_ind : ∀(A : Type)(P : A → Prop)(Q:Prop),

Existential quantification as an inductive predicate

- Inductive ex (A : Prop)(P : A → Prop) : Prop :=
  ex_intro : ∀x, P x → ex A P.
  ex_ind : ∀(A : Type)(P : A → Prop)(Q:Prop),
  (∀x :A, P x → Q) → ex A P → Q

Existential quantification as an inductive predicate

- Inductive ex (A : Prop)(P : A → Prop) : Prop :=
  ex_intro : ∀x, P x → ex A P.
  ex_ind : ∀(A : Type)(P : A → Prop)(Q:Prop),
  (∀x :A, P x → Q) → ex A P → Q
- The tactic elim uses ex_ind,
- Using elim produces an arbitrary element that satisfies the property,
- The notation exists x:A, P stands for ex A (fun x:A => P),
- In this course, I usually write ∃ x:A, P.

Tactics elim, destruct, intro on inductive types

- The tactic elim produces hypotheses,
- You usually need intro right away,
- The tactic destruct combines several elim then intros together.
- The tactic intro with a pattern also combines several elim and intro,
- The idea is to follow the structure of terms.

Example of destructuring intro

- Lemma exdi : forall P Q R, P /\ (Q \/ (∃ x:nat, R x)) →
  (P /\ Q) \/(exists x:nat, P /\ R x).
  intros P Q R [hP [hQ | [w hR]]].
  hP : P
  hQ : Q
  ====================
  P /\ Q \/ ∃ x : nat, P /\ R x
  left; split; [exact hP | exact hQ].
  hP : P
  w : nat
  hR : R w
  ====================
  P /\ Q \/ ∃ x : nat, P /\ R x
  right; exists w; split;[exact hP | exact hR]. Qed.