Certified Programming with Dependent Types

Inductive Predicates

Niels van der Weide

March 7, 2017

Last Time

We discussed inductive types

```
Print nat.
(* Inductive nat : Set :=
| O : nat
| S : nat → nat*)
```

Recursion principle

```
Check nat_rect.
(* nat_rect
: forall P : nat → Type ,
P O →
(forall n : nat, P n → P (S n))
→ forall n : nat, P n*)
```

Last Time

We discussed inductive types

```
Print nat.
(* Inductive nat : Set :=
| O : nat
| S : nat → nat*)
```

Induction principle

```
Check nat_ind.
(* nat_ind
: forall P : nat → Prop ,
P O →
(forall n : nat, P n → P (S n))
→ forall n : nat, P n*)
```

Last Time

We discussed inductive types

```
Print nat.
(* Inductive nat : Set :=
| O : nat
| S : nat → nat*)
```

Recursion principle

```
Check nat_rec.
(* nat_rec
: forall P : nat → Set ,
P O →
(forall n : nat, P n → P (S n))
→ forall n : nat, P n*)
```

This raises several questions:

- Induction is for proving, recursion for programming. What's the difference between Prop and Set ?
- Can we do logic in the language?
- Can we define more complicated propositions on types?

Prop vs Set

Let's look at some examples.

```
Inductive True : Prop :=
| I : True.

True is defined
True_rect is defined
True_ind is defined
True_rec is defined
```

```
Inductive unit : Set :=
| tt : unit.

unit is defined
unit_rect is defined
unit_ind is defined
unit_rec is defined
```

Prop vs Set

We can prove that these two are isomorphic

```
Definition f := fun (_ : unit) ⇒ I.

Definition g := fun (_ : True) ⇒ tt.

Theorem eq1 : forall x : unit, x = g (f x).
Proof.
intro x.
induction x.
(* compute. *) reflexivity.
Qed.

Theorem eq2 : forall x : True, x = f (g x).
Proof.
intro x.
destruct x.
(* compute. *) reflexivity.
Qed.
```

Prop vs Set

But for the following example they are different!

```
Inductive boolP : Prop :=
| trueP : boolP
| falseP : boolP.

boolP is defined

boolP_ind is defined
```

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

bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
```

Prop vs Set

This means the following is **not** allowed

```
Definition h (x : boolP) : bool :=
match x with
| trueP ⇒ true
| falseP ⇒ false
end.
```

```
Definition h : boolP → bool.
Proof.
intro x.
induction x.
```

Error: Cannot find the elimination combinator boolP_rec, the elimination of the inductive definition boolP on sort Set is probably not allowed.

Prop vs Set

Inhabitants of the type Prop are propositions. These are **proof-irrelevant**: all inhabitants are equal.

Prop vs Set

Inhabitants of the type Prop are propositions. These are **proof-irrelevant**: all inhabitants are equal.
Inhabitants of the type Set are sets. These are **proof-relevant**: inhabitants might be equal, but do not have to.

Prop vs Set

Inhabitants of the type Prop are propositions. These are **proof-irrelevant**: all inhabitants are equal.
Inhabitants of the type Set are sets. These are **proof-relevant**: inhabitants might be equal, but do not have to.
Also, Prop is ignored during code extraction.

Logic in Type Theory (or Coq)

If inhabitants of Prop pretend to be propositions, can we treat them as such?

Logic in Type Theory (or Coq)

If inhabitants of Prop pretend to be propositions, can we treat them as such?

Yes, we can! Inductive types come to the rescue.

Logic in Type Theory (or Coq)

Conjunctions.

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

```
and is defined
and_rect is defined
and_ind is defined
and_rec is defined
```

```
Inductive prod
  (A : Set) (B : Set)
  : Set :=
| pair : A → B → prod A B.
```

```
prod is defined
prod_rect is defined
prod_ind is defined
prod_rec is defined
```

Logic in Type Theory (or Coq)

Disjunctions.

```
Inductive or
  (A : Prop) (B: Prop)
  : Prop :=
| orl : A → or A B
| orr : B → or A B.
```

or is defined

or_ind is defined

```
Inductive sum
  (A : Set) (B: Set)
  : Set :=
| inl : A → sum A B
| inr : B → sum A B.
```

sum is defined
sum_rect is defined
sum_ind is defined
sum_rec is defined

Proposition Logic in Coq

Coq got all these types natively. A nice table can be found on
http://andrej.com/coq/cheatsheet.pdf

Proposition Logic in Coq

Short demonstration of these tactics

```
Theorem and_com : forall P Q : Prop, P ∧ Q → Q ∧ P.
Proof.
intros.
destruct H.
split ; assumption.
Qed.
```

We can also prove it by programming.

```
Definition and_com' (P Q : Prop) (x : and P Q) : and Q P :=
match x with
| conj _ _ p q ⇒ conj Q P q p
end.
```

Proposition Logic in Coq

But it is much better!

```
Theorem complicatedProp :
  forall P Q : Prop, ¬ (P ∧ Q) ↔ ¬¬(¬Q ∨ ¬P).
Proof.
tauto. (* also possible: intuition. *)
Qed.
```

Note this also works for types:

```
Theorem complicatedType :
forall P Q : Type,
(P * Q) → False
↔
((( Q → False) + (P → False)) → False) → False.
Proof.
tauto. (* also possible: intuition *)
Qed.
```

Proposition Logic in Coq

The logic is constructive.

```
Theorem unprovable : forall P : Prop, P ∨ ¬ P.
Proof.
intuition.
(*
Hypothesis: P : Prop
Remaining goal: P ∨ (P → False)
*)
```

First-order Logic in Coq

Existential quantifier:

```
Inductive ex
  (A : Type) (P : A → Prop)
  : Prop :=
| ex_intro : forall (x : A),
  P x → ex P
```

```
Inductive sig
  (A : Type) (P : A → Type)
  : Type :=
| sig_intro : forall (x : A),
  P x → sig P
```

First-order Logic in Coq

Example with ∃:

```
Definition smaller : { n : nat & 0 <= n }.
Proof.
exists 3.
auto.
Defined.

Theorem muchSmaller : exists n : nat, 0 <= n.
Proof.
exists 37.
auto. (* does not automatically solve 0 <= 37.
        Searches to some fixed depth *)
auto 38. (* this solves the goal.
           We do le_S 37 times and le_n 1 time.
           So, we need depth 38 *)
Qed.
```

Short intermezzo: Defined vs Qed

Qed makes an *opaque definition* (no unfolding).

```
Eval compute in muchSmaller.
(* = muchSmaller
   : exists n : nat, 0 <= n
*)
```

Defined makes a *transparent definition* (with unfolding).

```
Eval compute in smaller.
(* = existT
  (fun n : nat ⇒ 0 <= n)
    3
    (le_S 0 2
      (le_S 0 1
        (le_S 0 0 (le_n 0)
      )
    )
   )
: {n : nat & 0 <= n}
*)
```

Now we can finally do the real work: make recursive predicates.
How to do this? The constructors tell how to prove the predicate.

Getting started: equality

How can we prove $x = y$? We can use reflexivity.

```
Print eq.
(* Inductive eq (A : Type) (x : A) : A → Prop :=
  eq_refl : x = x *)
```

Another Simple Predicate

We can define $n < 2$ as follows.

```
Inductive lessThanTwo : nat → Prop :=
| zero : lessThanTwo 0
| one : lessThanTwo 1.
```

Then we can easily prove:

```
Theorem zeroOrOne : forall n : nat, lessThanTwo n ↔ n = 0 ∨ n = 1.
Proof.
intro n.
split.
induction 1 ; auto.
intro H.
destruct H ; rewrite H ; constructor.
Qed.
```

Another Simple Predicate

We can define $n < 2$ as follows.

```
Inductive lessThanTwo : nat → Prop :=
| zero : lessThanTwo 0
| one : lessThanTwo 1.
```

Then we can easily prove:

```
Theorem twoNotLessThanTwo : lessThanTwo 2 → False.
Proof.
intro H.
inversion H.
Qed.
```

A More Complicated Predicate: Even Numbers

We define a predicate for the even numbers.

```
Inductive even : nat → Prop :=
| evenZ : even 0
| evenSS : forall n : nat, even n → even (S (S n)).
```

```
Hint Constructors even.
```

We need to give a hint, so that the auto tactic also considers the constructors of even.

A More Complicated Predicate: Even Numbers

Adding two even numbers: an automated proof.

```
Theorem evenAdd :
  forall (n m : nat),
  even n →
  even m →
  even (n + m).
Proof.
induction 1 ; induction 1 ; simpl ; auto.
Qed.
```

(In the book he is screwing around with inversion)

A More Complicated Predicate: Even Numbers

Without automation.

```
Theorem evenAdd' : forall (n m : nat),
  even n →
  even m →
  even (n + m).
Proof.
induction 1
; induction 1
; simpl
; constructor
; apply IHeven
; constructor
; apply H0.
Qed.
```

A More Complicated Predicate: Even Numbers

```
Theorem oddSuccessor :
  forall (n : nat),
  even n
  → even (S n)
  → False.
Proof.
intro n.
induction 1 ; intro H0.
 – inversion H0.
 – apply IHeven.
   inversion H0.
   apply H2.
Qed.
```

A More Complicated Predicate: Even Numbers

```
Theorem evenTwice : forall (n : nat), even (n + n).
Proof.
induction n ; simpl.
 - auto.
 - rewrite <- plus_n_Sm.
   constructor.
   apply IHn.
Qed.
```

A More Complicated Predicate: Even Numbers

```
Theorem evenContra :
  forall (n : nat),
  even (S (n + n))
  → False.
Proof.
intro n.
apply oddSuccessor.
apply evenTwice.
Qed.
```