Predicates and Quantifications

Yves Bertot

August 2010

Predicates

- Functions from a given type to the type of propositions, `Prop`
- Can be used to build propositional values
- A few pre-existing ones : comparison between numbers, etc.
- important pre-existing predicate : equality.

Equality predicate

- If a and b are elements of the same type, then `a = b` is a proposition
- Proof by reflexivity when the two members are *visibly* equal
  - not always obvious for beginners

```
Lemma foureq : 4 = 4 ∧ 1+3 = 4.
Proof.
split.
  reflexivity.
reflexivity.
Qed.
```

Defining predicates using equality

- Equality can be used to define other predicates
- If A is a type and `f` is a function `f : ` A `-> bool`
  - `Definition Pf ( x : ` A`) := f x = true.`
  - `Pf` is a predicate
  - When reasoning about this predicate, use `unfold Pf`

Using equalities in proofs

- When $a = b$, replace a with b
- Tactic `rewrite`

```
Lemma ex_rewrite : (2 + 1) = 3 -> (2 + 1) + 4 = 3 + 4.
intros H.
...
  H : 2 + 1 = 3
  ==================
   2+1 + 4 = 3 + 4
rewrite H.
...
  H : 2 + 1 = 3
  ==================
   3 + 4 = 3 + 4
```

Equality and negation

- Special notation for `not (a=b)`
- `a <> b`
- Can also be used to define predicates
- `Definition nonzero (x : nat) : Prop := x <> 0.`

Universal quantification

- Expresses that a formula is satisfied for every member of a type
- Keyword `forall` followed by a variable and its type, then the formula
- The formula can also be a predicate applied to the variable
  - `forall x :` A`, Pf x`
  - `forall x: nat, x + 3 = 2 + x + 1`
- Possible to quantify over several variables and to omit types
  - `forall x y, x + y = y + x + 1`
- Universal quantification can help writing false formulas !

Higher-order logic

- Possible to quantify over functions and predicates
- functions and predicates can also take other functions as arguments
  - ```
    forall P : nat -> Prop,
        (forall x, P (x+1)) ->
      forall y, y <> 0 -> P y
    ```
    ```

Proving universal quantifications

- Prove the formula for an arbitrary fixed constant
- Keep the constant in the goal's context
  - with its type information
- Tactic `intros`

Tactic intros at work

```
Lemma ex_forall1 :
  forall P : nat -> Prop, (forall x, P (x + 1)) ->
  forall y, y <> 0 -> P y.
intros P.
...
  P : nat -> Prop
  ===============
   (forall x, P (x + 1)) ->
    forall y : nat, y <> 0 -> P y
```

- Combine several `intros` together

```
...
  P : nat -> Prop
  ===============
  (forall x, P (x + 1)) ->
    forall y : nat, y <> 0 -> P y
intros Hp y ynz.
...
  P : nat -> Prop
  Hp : forall x, P (x + 1)
  y : nat
  ynz : y <> 0
  ===============
   P y
```

Using universally quantified hypotheses

- Most theorems in Coq's database are universally quantified
- Special case for universally quantified equalities
- Uniform treatement of universal quantification and implication
- Main tactic `apply`
- Guesses values for quantified variables
  - match the head of the hypothesis with the goal
  - Help sometimes required : use a `with` directive

Tactic apply at work

```
1 subgoal
...
  H : forall x y:nat, Q x -> R x (y + 1) -> P x y
  ===============
   P 3 6
apply H.
2 subgoals
...
 ===============
   Q 3

subgoal 2 is:
   R 3 (6+1)
```

Tactic apply failure and repair : directive with

```
...
  H : forall x y z: nat, P x y  -> P y z -> P x z
  ===============
   P 1 3
apply H.
Error: Unable to find an instance for the variable y
apply H with (y := 2).
...
  ===============
   P 1 2

subgoal 2 is:
 P 2 3
```

Using universally quantified hypotheses as functions

- Case where `H : forall x y, P x y -> Q x y`
- `apply H with (x := a)` can be replaced by `apply (H a)`
- Actually `(H a)` has type `forall y, P a y -> Q a y`
- Case where `H' : forall x, P x -> forall y, Q x y` and `H1 : P a`
- `apply H' with (1 := H1)` is possible
- `apply (H' a H1)`
- actually `H' a H1` has type `forall y, Q a y`

Universally quantified equalities

- When a theorem's conclusion is an equality, use `rewrite` directly
- `rewrite` finds an instance, often the right one
- Use the `with` directive to choose the instance, when needed
- Use `pattern` to choose an occurrence and instance

Tactic rewrite at work

```
...
  H : forall x, f x = x + 3
  ===================
    f a * f b = a * b + 3 * f a
rewrite H.
  ===================
   (a + 3) * f b = a * b + 3 * (a + 3)
```

Tactic rewrite at work (2)

```
...
  H : forall x, f x = x + 3
  ===================
    f a * f b = a * b + 3 * f a
rewrite H with (x := b).
  ===================
  f a * (b + 3) = a * b + 3 * f a
```

Tactic rewrite at work (3)

```
...
  H : forall x, f x = x + 3
  ===================
    f a * f b = a * b + 3 * f a
pattern (f a) at 2; rewrite H.
  ===================
  f a * f b = a * b + 3 * (a + 3)
```

Existential quantification

- Expresses that a predicate is satisfied by at least one element of the type
- Keyword `exists`, then the variable and its type, then a comma and the formula
  - the type of the variable may sometimes be omitted
- `exists x, x*x = 25`
- `exists f:nat->nat, forall x, f x <= x`
- `Definition even (x : nat) := exists y, x = 2 * y.`

Proving existential quantifications

- Find a witness, then prove that it satisfies the formula
- Prove the formula by showing that a value indeed exists with the good property.
- Tactic `exists`

```
Lemma ex_exists : exists x, x * x = 25.
exists 5.
...
  =================
    5 * 5 = 25
reflexivity.
Qed.
```

Use existentially quantified hypotheses

- Add in the context a variable and the hypothesis that it satisfies the formula
- Move to a context where there visibly exits a value with the good property
- Tactic `destruct as [x Hx]`

```
  H : exists y : nat, x + 1 = 2 * y
  ===================
   exists z : nat, x + 2 = 2 * z + 1
destruct H as [y Hy].
  y : nat
  Hy : x + 1 = 2 * y
  ===================
   exists z : nat, x + 2 = 2 * z + 1
```

Automatic proofs

- `firstorder` : first order formulas, no domain knowledge
- `ring` : equalities between polynomial formulas, no use of the hypotheses
    - execute `Require Import Arith.` to have this tactic.
- `omega` : inequalities between linear formulas on integers, uses the hypotheses
    - execute `Require Import Omega.` or `Require Import ZArith.`