Asistente de pruebas Coq

Matemáticas Discretas II

November 9, 2009

Prueba de teoremas

- Una prueba es un argumento irrefutable sobre la veracidad de una aserción matemática.
- Cómo podemos probar teoremas con un ordenador?
  - **Probadores automáticos de teoremas:** Encuentran las pruebas por sí mismos.
    - Éxito en dominios limitados (prueba de hardware).
    - Enorme complejidad computacional.
  - **Comprobadores de pruebas:** verifican una prueba dada.
  - **Asistentes de pruebas:**
    - La "parte difícil" de las pruebas viene dada por la interacción con los usuarios.
    - La "parte sencilla" de las pruebas se realiza automáticamente.
    - Ejemplos: PVS (SRI), Isabelle (Cambridge / Munich), Coq (INRIA), ...

Qué es Coq?

- Sistema de ayuda para la demostración de teoremas.
- Trabaja con aserciones matemáticas.
- Permite extraer programas certificados (correctos) a través de la prueba constructiva de sus especificaciones.
- No es un sistema automático, pero proporciona tácticas para la prueba de teoremas.
  - Las tácticas son comandos utilizados en la prueba de teoremas para indicarle a Coq como debe actuar para probar la corrección de un término.

Formas de ejecución

1. Ejecución interactiva:
   1.1 El *top-level*:
      - Entrar en el top-level: coqtop
      - Salir del top-level: Quit.
   1.2 Interfaz gráfica: coqide
2. Compilador no interactivo: coqc

Declaraciones

- Asocian un nombre a una especificación.
- Tipos de especificaciones:
  1. Proposiciones lógicas: `Prop`.
  2. Colecciones matemáticas: `Set`.
  3. Tipos Abstractos: `Type`.
- Todas las expresiones tienen un tipo asociado (`Check`).

```
Coq < Check 3.
3 : nat
Coq < Check nat.
nat : Set
Coq < Check True.
True : Prop
Coq < Check (3+4).
3 + 4 : nat
```

```
Coq < Check (3=5).
3=5 : Prop
Coq < Check (3,4).
(3,4) : nat * nat
Coq < Check nat -> Prop.
nat -> Prop : Type
Coq < Check (3 <= 6).
3 <= 6 : Prop
```

Declaraciones

- Secciones: Estructurar la modelización. Limitar el ámbito de parámetros, hipótesis, etc.

```
Coq < Section Declaration.
```

- Variables:

```
Coq < Variable n : nat.
n is assumed
```

- Hipótesis:

```
Coq < Hypothesis PosN : (gt n 0).
PosN is assumed
```

Definiciones

- Coq contiene varias definiciones aritméticas: `O, S, plus`.
- Las definiciones asignan un identificador a un valor.

```
Coq < Definition one := (S O).
one is defined
Coq < Definition two : nat := S one.
two is defined
Coq < Definition double (m:nat) := plus m m.
double is defined
```

- Podemos mezclar variables libres y acotadas:

```
Coq < Definition addN (m:nat) := plus m n.
addN is defined
```

- Limpiar el entorno. Borrar contenidos de la sección:

```
Coq < Reset Declaration.
```

Convenciones sintácticas

- $a \to b \to c$ equivale a $a \to (b \to c)$.
- $f\ a\ b$ equivale a $(f\ a)\ b$.
- Cuantificadores del lenguaje `forall` y `exists`, `fun`, `let ... in`.

Introducción al motor de pruebas

- A $\rightarrow$ B se lee como "A implica B". Sobrecarga operador.

```
Coq < Section MinimalLogic.
Coq < Variables A B C : Prop.
Coq < Check (A -> B).
   A -> B : Prop
Coq < Goal (A -> B -> C) -> (A -> B) -> A -> C.
   1 subgoal
   A : Prop
   B : Prop
   C : Prop
   ============================
   (A -> B -> C) -> (A -> B) -> A -> C
Coq < intro H.
   1 subgoal
   A : Prop
   B : Prop
   C : Prop
   H : A -> B -> C
   ============================
   (A -> B) -> A -> C
```

Introducción al motor de pruebas

- Táctica `intros`: Transforma premisas o cuantificadores del objetivo al contexto de la prueba.

```
Coq < intros H' HA.
   1 subgoal
   A B C: Prop
   H : A -> B -> C
   H' : A -> B
   HA : A
======================
C
Coq < apply H.
   2 subgoals
   A B C: Prop
   H : A -> B -> C
   H' : A -> B
   HA : A
======================
A
subgoal 2 is: B
```

```
Coq < exact HA.
   1 subgoal
   A : Prop
   B : Prop
   C : Prop
   H : A -> B -> C
   H' : A -> B
   HA : A
======================
B
```

- Táctica `apply`: intenta mapear el objetivo con su conclusión. Añade tantos sub-objetivos como premisas.

Introducción al motor de pruebas

- Táctica `exact`: se aplica cuando dos expresiones son convertibles.

```
Coq < apply H'.
   1 subgoal
   A : Prop
   B : Prop
   C : Prop
   H : A -> B -> C
   H' : A -> B
   HA : A
   ============================
   A
Coq < exact HA.
   Proof completed.
Coq < Save trivialLemma.
```

- `Abort` desecha el lema.
- `Save` lo guarda para futuro uso.

Introducción al motor de pruebas

- `Lemma` o `Theorem` se definen igual que `Goal`.
- Posibilidad de concatenar varias tácticas:
  - T1;T2: aplica T1 a objetivo actual y T2 a todos los sub-objetivos generados por T1.
  - T;[T1|T2|...|Tn]: aplica T al objetivo actual, T1 al primer sub-objetivo generado, T2 al segundo, etc.
- Táctica `auto`: intenta probar el problema combinando una serie de tácticas a las hipótesis y a ciertos lemas de forma automática.
  - Podemos enriquecer `auto` con teoremas ya probados: `Hint Resolve TheoremName`.
- `Restart`: recomenzar la prueba del teorema.
- `Undo`: retroceder un paso en la prueba del teorema.

Algunas pruebas sencillas

- Prueba de la conmutatividad de la intersección:

```
Coq < Lemma andCommutative : A /\ B -> B /\ A.
   1 subgoal
   A B C: Prop
   ============================
   A /\ B -> B /\ A

Coq < intro.
   1 subgoal
   A B C: Prop
   H : A /\ B
   ============================
   B /\ A
```

Algunas pruebas sencillas

- Táctica `split`: divide la conjunción en dos sub-objetivos.

```
Coq < split.
   2 subgoals
   A B C: Prop
   H : A /\ B
   ============================
   B
subgoal 2 is:
   A
Coq < apply H. apply H.
```

- Ojo! Podría probarse utilizando `auto`.

Algunas pruebas sencillas

- Prueba alternativa de la conmutatividad de la intersección:
- Táctica `elim`: divide un término en sus componentes.

```
Coq < elim H.
   1 subgoal
   A B C: Prop
   H : A /\ B
   ============================
   A -> B -> B / A
Coq < split.
Coq < apply H. apply H.
```

Prueba de teoremas sencillos

- Táctica `clear`: borra hipótesis innecesarias.
- Táctica `trivial`: equivalente a `auto` pero opera en un único paso.
- Táctica `tauto`: intenta aplicar una tautología.

```
Coq < Lemma orCommutative : A \/ B -> B \/ A.
   1 subgoal
   A B C: Prop
============================
   A \/ B -> B \/ A
Coq < tauto.
   Proof completed.
Coq < Qed.
   tauto.
   orCommutative is defined
```

- `Qed`: valida la prueba y la introduce en el entorno de forma opaca.

Prueba de teoremas sencillos

- La táctica `generalize` permite recuperar una de las hipótesis locales e introducirlas en el objetivo.

```
...
H : termH
...
============================
termGoal
Coq < generalize H.
...
H : termH
...
============================
termH -> termGoal
```

- Expresión del tipo `x = y`. `x` e `y` mismo tipo. Como reemplazar `x` con `y`?
  - Tácticas `rewrite` y `replace`.

Prueba de teoremas sencillos

```
Coq < Variable f : nat -> nat.
Coq < Hypothesis foo : f 0 = 0.
Coq < Lemma L1 : forall k:nat, k = 0 -> f k = k.
1 subgoal
   f: nat -> nat.
   foo: f 0 = 0.
   ============================
   forall k : nat, k = 0 -> f k = k
Coq < intros k E.
1 subgoal
   f: nat -> nat.    foo: f 0 = 0.
   k : nat
   E : k = 0
   ============================
   f k = k
Coq < rewrite E.
1 subgoal
   ...
   ============================
   f 0 = 0
```

Prueba de teoremas sencillos

```
Coq < Hypothesis f10 : f 1 = f 0.
Coq < Lemma L2 : f (f 1) = 0.
1 subgoal
   f : nat -> nat
   foo : f 0 = 0
   f10 : f 1 = f 0
   ============================
   f (f 1) = 0
Coq < rewrite f10.
   f : nat -> nat
   foo : f 0 = 0
   f10 : f 1 = f 0
   ============================
   f (f 0) = 0
Coq < rewrite foo. exact foo.
```

Prueba de teoremas sencillos

- Táctica `unfold`: sustituye una definición por la fórmula que la define.
  - `Coq < Definition one := (S O).`
  - `unfold term in H`: en la hipótesis en lugar del objetivo.
- Táctica `symmetry`: sustituye el término `t = u` por `u = t`.
- Táctica `reflexivity`: se aplica a un objetivo de la forma `t = u`, comprueba si `t` y `u` son convertibles.
- Muchas otras tácticas!

Tipos de datos inductivos

- Se definen indicando nombre, tipo y constructores.

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

- Definición automática de redes de eliminación. `bool_ind` permite el razonamiento por casos.
- Demostrar que un booleano es `true` o `false`.

```
Coq < Lemma duality : forall b:bool, b = true \/ b = false.
1 subgoal
   ============================
   forall b : bool, b = true \/ b = false
```

- Táctica `elim`: divide un término en sus componentes

Tipos de datos inductivos

```
Coq < intro b.
1 subgoal
   b : bool
   ============================
   b = true \/ b = false
Coq < elim b.
2 subgoals
   b : bool
   ============================
   true = true \/ true = false
   subgoal 2 is:
   false = true \/ false = false
Coq < left; trivial. |    Coq < left; trivial.
```

- La prueba entera podría haberse reducido a la siguiente instrucción:

  `Coq < simple induction b; auto.`

Tipos de datos inductivos

- Números naturales:

```
Coq < Inductive nat : Set :=
      | O : nat
      | S : nat -> nat.
```

- Una vez definido el tipo podemos realizar operaciones:

```
Coq < Fixpoint plus (n m:nat) struct n : nat :=
      match n with
      | O => m
      | S p => S (plus p m)
      end.
```

- Construcción `match ... with`.

Tipos de datos inductivos

- Táctica `destruct`: divide o "destruye" un tipo inductivo en sus posibles casos

```
Theorem negb_involutive : forall b : bool,
   negb (negb b) = b.
Proof.
   intros b. destruct b.
   reflexivity.
   reflexivity.
Qed.
```

Tipos de datos inductivos

- También podemos probar teoremas por inducción:

```
Coq < Lemma plus_n_O : forall n:nat, n = n + 0.
1 subgoal
   ============================
   forall n : nat, n = n + 0
Coq < intro n; elim n.
2 subgoals
   n : nat
   ============================
   0 = 0 + 0
subgoal 2 is:
   forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
Coq < simpl; auto.
Coq < simpl; auto.
```

- auto utiliza como hint:

```
Coq < Check eq_S.
eq_S : forall x y : nat, x = y -> S x = S y
```

- simpl utiliza la definición de la función para simplificar la expresión a probar.

Tipos de datos inductivos

- Declarar `plus_n_0` para poder ser utilizado como hint por `auto`:

```
Coq < Hint Resolve plus_n_O.
```

- Vamos a probar ahora la conmutatividad de la suma:

```
Coq < Lemma plus_com : forall n m:nat, n + m = m + n.
1 subgoal
   ============================
   forall n m : nat, n + m = m + n
Coq < simple induction m; simpl; auto.
1 subgoal
   n m: nat
   ============================
   forall n0 : nat, n + n0 = n0 + n -> n + S n0 = S (n0 + n)
Coq < intros m' E; rewrite <- E; auto.
Coq < Qed.
```

- `auto` funciona gracias a nuestro hint `plus_n_0`.

Tipos de datos inductivos

- Táctica `induction`: divide el objetivo en dos sub-objetivos, el caso base y el de inducción.
- Al probar el paso base, la hipótesis de inducción aparece en el contexto de la prueba.

```
Theorem plus0_r : forall n:nat, plus n 0 = n.
Proof.
   intros n. induction n as [| n ].
   reflexivity.
   simpl. rewrite -> IHn . reflexivity.
Qed.
```

Tipos de datos inductivos

- Función que devuelve `true` si el número es 0 y `false` en cualquier otro caso.

```
Coq < Definition Is_S (n:nat) := match n with
      | O => False
      | S p => True
      end.
```

- Prueba de teoremas:

```
Coq < Lemma S_is_S : forall n:nat, IsS (S n).
1 subgoal
   ============================
   forall n : nat, Is_S (S n)
Coq < simpl; trivial.
Coq < Qed.
```

Tipos de datos inductivos

- Nuevos teoremas:

```
Coq < Lemma no_confusion : forall n:nat, 0 <> S n.
1 subgoal
   ============================
   forall n : nat, 0 <> S n.
Coq < intro n; discriminate.
Coq < Qed.
```

- Táctica `discriminate`: prueba de objetivos asumiendo que dos términos estructuralmente diferentes de un conjunto inductivo son iguales (`S (S 0) = S 0`).

Módulos

- Por defecto, Coq carga varias librerías (lógica y aritmética).
- Abrir nuevos módulos: `Require Import Arith.`
- Creación de módulos propios:
  - Edición de fichero mi_modulo.v. `Module ModName. ... End ModName.`.
  - Compilación con coqc. Librería en fichero mi_modulo.vo.
  - Carga con `Require Import ModName.`
- Módulo N depende del módulo M. Queremos que M sea visible cuando se importe N. `Require Export M.`