Computación y Estructuras Discretas I

Andrés A. Aristizábal P.
aaaristizabal@icesi.edu.co

Departamento de Computación y Sistemas Inteligentes

2024-2

Agenda del día

1. **Presentación del tema**
   - Introducción a Coq

2. **Presentación del siguiente tema**
   - Demostraciones en Coq

Agenda del día

1. **Presentación del tema**
   - Introducción a Coq
2. **Presentación del siguiente tema**
   - Demostraciones en Coq

¿Qué es Coq?

¿Qué es Coq?

- Se trata de un asistente de demostraciones.
- Un programa que ayuda a llevar a cabo pruebas matemáticas e indica con seguridad que lo que se ha verificado sea cierto.
- Coq tiene diversas aplicaciones.
  - Formalizar pruebas ya existentes para comprobar con seguridad que son ciertas.
  - Ayudar a demostrar nuevos teoremas.
  - Extraer programas correctos por construcción.
  - Probar que determinado programa, lenguaje, función, o sistema de tipos preserva una invariante.
  - (semi)automatizar todas estas tareas.

¿Algo adicional?

¿Algo adicional?

- A partir de especificaciones en Coq se pueden extraer programas ejecutables en Objective Caml o Haskell.
- En Coq todo se formaliza en una lenguaje que se denomina CIC (Calculus of Inductive Constructions).
  - CoC (Calculus of Constructions) es una teoría de tipos que puede servir como un lenguaje funcional tipado o una base de construcción matemática.
  - CIC es una variante de CoC que añade tipos inductivos.
  - La importancia de los tipos inductivos es que permite la creación de nuevo tipos a partir de constantes y funciones.
  - Características que permiten bridar un rol similar a las estructuras de datos.
  - Permite una teoría de tipos que adiciona conceptos como números, relaciones y árboles.

Ejemplo de definición inductiva de los números naturales en coq

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

- Este es el ejemplo estándar de como codificar números naturales usado la codificación de Peano.
- Un número natural se crea a partir de la constante “O”
- La función “S” a otro número natural.
- “S” es la función sucesor que representa sumar 1 a un número
- “O” es cero, “S O” es uno, “S (S O)” es dos, “S (S (S O))” es tres, etc.

Además...

- Todos los juicios lógicos en Coq son juicios tipados.
- El eje de Coq es de hecho un algoritmo de verificación de tipos.
- Algoritmos ejecutables

(Paréntesis - ¿Qué es la programación funcional?)

(Paréntesis - ¿Qué es la programación funcional?)

- Tanto Objective Caml como Haskell son ejemplos de lenguajes funcionales.
- Otros lenguajes como C++, Python, Scala o el mismo Java soportan elementos de este paradigma.
- La programación funcional es un paradigma de programación en donde los programas se construyen aplicando y componiendo funciones.
- Es un paradigma de programación declarativo (declara lo que el programa debe cumplir más no como lo ha de hacer, a diferencia del imperativo que se basa en una secuencia de pasos detallados que modifican el estado del programa).
- Se basa en funciones que no modifican variables pero si generan nuevas como salida.
- La salida de una función pura solamente depende de sus parámetros de entrada (no hay impactos externos y se evitan efectos secundarios)

(Paréntesis - ¿Por qué es importante la programación funcional?)

(Paréntesis - ¿Por qué es importante la programación funcional?)

- Funciones puras garantizan que el estado fuera del programa no se altere.
- Más fáciles de probar.
- Conducen a menos bugs.
- Código funcional tiende a ser más fácil de comprender.
- La recursión es más simple.
- Entre otros.

Función imperativa de potencia

```
function pow(x, n) {
    let res = 1;
    for (let i = 0; i < n; i++) {
        res *= x;
    }
    return res;
}
```

Función funcional de potencia

```
function pow(x, n) {
    if (n == 1) {
        return x;
    }
    return x * pow(x, n - 1);
}
```

¿Por qué es relevante Coq?

¿Por qué es relevante Coq?

- En la práctica, hay muchos programas con errores que tienen un coste tremendo, desde fallos de seguridad, hasta errores de cálculo.
- En general, las metodologías de desarrollo intentan evitar estos problemas siguiendo una serie de técnicas como el uso de tests.
- Desafortunadamente estas técnicas no pueden asegurar que el programa sea correcto, sólo que no tenga fallos conocidos.
- Mediante los métodos formales podemos asegurarnos de que un programa cumpla con determinadas propiedades matemáticamente demostrables:
  - Que una función acabe en tiempo finito
  - Que converja correctamente a un resultado.
  - Que no traspase información que ha de permanecer secreta.

Vistazo general

- Coq actúa como todo REPL (read-eval-print loop).
- Lee las declaraciones que introducimos, las evalúa e imprime el resultado.

Ejemplos sobre verificación de tipos

```
Coq < Check 0.
0
     : nat

Coq < Check Nat.add.
Nat.add
     : nat -> nat -> nat
```

Verificación de tipos

- Check es un comando que nos da el tipo de una expresión.
- En el primer caso, el 0 es un número natural.
- El tipo de Nat.add, viene dado como nat -> nat -> nat.
- Esto indica que add toma un número natural (nat), devuelve algo que toma otro natural y devuelve un natural (nat -> nat).

Funciones y parámetros

- En Coq se utiliza la noción de función de modo más formal que en otros lenguajes.
- Estamos acostumbrados a tener funciones que toman varios parámetros, como `int suma(int a, int b).`
- En Coq las funciones toman un parámetro y devuelven un parámetro.
- Estas funciones se trabajan mediante el uso de funciones parciales y la técnica denominada “currying”.
- Una función como suma, en Coq, toma un número natural, devuelve una función que toma un sólo número natural, le suma el anterior, y devuelve el resultado, que es otro natural.
- A las funciones que toman o devuelven otras funciones como parámetros se las denomina funciones de orden superior.
- Simplemente, `add` es una función que, a efectos prácticos, toma dos naturales y nos devuelve otro, mediante una función intermedia de orden superior.

Ejemplo sobre el cómputo de funciones

```
Coq < Compute Nat.add 2 3.
      = 5
      : nat
```

¿Qué nos presenta el anterior ejemplo?

¿Qué nos presenta el anterior ejemplo?

- `add` se aplica sobre el parámetro 2 y retorna otra función que se aplica sobre 3 para retornarnos el valor 5.
- Adicionalmente nos indica el tipo del valor de retorno, un natural.

¿Qué nos presenta el anterior ejemplo?

- `add` se aplica sobre el parámetro 2 y retorna otra función que se aplica sobre 3 para retornarnos el valor 5.
- Adicionalmente nos indica el tipo del valor de retorno, un natural.

¿Pero qué es un tipo en Coq?

¿Qué nos presenta el anterior ejemplo?

- `add` se aplica sobre el parámetro 2 y retorna otra función que se aplica sobre 3 para retornarnos el valor 5.
- Adicionalmente nos indica el tipo del valor de retorno, un natural.

¿Pero qué es un tipo en Coq?

- Función de primera clase (se puede tratar como otro valor del lenguaje).
- Se construye de manera inductiva.

Ejemplo sobre definición inductiva en Coq

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

¿Qué nos presenta el anterior ejemplo?

¿Qué nos presenta el anterior ejemplo?

- nat es un tipo inductivo (perteneciente a Set (conjunto)) que se puede formar mediante dos constructores:
  1. O que representa el cero.
  2. Una función S (función sucesor) que toma un nat y devuelve otro nat.

Ejemplo de verificación de tipo de sucesor

```
Coq < Check S (S (S O)).
3
     : nat
```

¿Qué aprendemos del ejemplo anterior?

¿Qué aprendemos del ejemplo anterior?

- 3 es simplemente azúcar sintáctico para S (S (S O))
- Es la codificación de Peano que vimos previamente.
- Utiliza un sólo dígito y aunque es poco eficiente permite realizar pruebas inductivas con mayor facilidad.

Ejemplo de verificación de tipo de add

```
Coq < Print Nat.add.
Nat.add =
fix add (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | S p => S (add p m)
  end
     : nat -> nat -> nat
```

- fix es un operador para definir funciones recursivas (se llama a sí misma).
- (n m : nat) {struct n} : nat. indica que toma 2 parámetros, n y m, de tipo nat.
- struct n indica que la recursión es estructural en n, es decir, que cuando add se llama a sí misma, n va a ser siempre inferior.
- Esto garantiza que add siempre devuelva un resultado en tiempo finito.
- El último nat es el tipo del valor de retorno de la función.
- Decimos que toma dos parámetros porque el currying es automático.

- Luego del := viene el cuerpo en de la función.
- match es un operador para encajar patrones.
- Se indica la variable para hacer el match junto a with y los patrones respectivos.
- En este caso se encaja con n
- En el primer caso si n encaja con 0, la función devuelve m pues 0 + m = m.
- Si n encaja con S p, es decir, cuando n es el sucesor de otro nat p, la función devuelve S (add p m), el sucesor de añadir p a m.
- Aquí se percibe la recursión estructural en n, porque al llamarse a sí misma, el valor de n siempre tendrá que disminuir para llegar al caso base.
- Finalmente, end cierra la función, y se presenta el tipo, nat -> nat -> nat.

Ejercicio

1. *Defina la función recursiva del factorial en Coq*
2. *Defina la función recursiva de la multiplicación de dos números n y m en Coq*

Solución

```
Fixpoint factorial (n:nat) : nat :=
match n with
| O => (S O)
| S p => (S p) * (factorial p)
end.
```

Solución

```
Fixpoint multiplicar (n m:nat) : nat :=
match n with
| O => O
| S p => m + (multiplicar p m)
end.
```

Primer ejemplo de demostración

```
Coq < Theorem add_0_m: forall m:nat, 0 + m = m.
```

Primer ejemplo de demostración

```
1 subgoal
  ============================
  forall m : nat, 0 + m = m
add_0_m <
```

Primer ejemplo de demostración

```
add_0_m < intros.
  m : nat
  ============================
  0 + m = m
```

Primer ejemplo de demostración

```
add_0_m < simpl.
  m : nat
  ============================
  m = m
```

Primer ejemplo de demostración

```
add_0_m < reflexivity.
No more subgoals.
```

Primer ejemplo de demostración

```
add_0_m < Qed.
intros.
simpl.
reflexivity.
Qed.
add_0_m is defined
Coq <
```

Agenda del día

1. Presentación del tema
   - Introducción a Coq

2. **Presentación del siguiente tema**
   - Demostraciones en Coq

¿Qué es una declaración en Coq?

¿Qué es una declaración en Coq?

Una declaración asocia un nombre con una especificación.

¿Qué es una declaración en Coq?

Una declaración asocia un nombre con una especificación.

¿Qué es una especificación en Coq?

¿Qué es una declaración en Coq?

Una declaración asocia un nombre con una especificación.

¿Qué es una especificación en Coq?

Es una expresión formal que clasifica la noción que está declarando.

- Hay tres clases de especificaciones: proposiciones lógicas, colecciones matemáticas y tipos abrstractos.
- Estas se clasifican en tres tipos: *Prop, Set* y *Type.*

¿Cuáles son algunas definiciones?

¿Cuáles son algunas definiciones?

- `nat` es una definición aritmética, una colección matemática.
- Las constantes `O, S` y `plus` se definen como objetos de tipo `nat, nat -> nat,` y `nat -> nat -> nat.`
- También se pueden introducir nuevas definiciones con un nombre y un tipo bien definido

Ejemplo

Definición de uno

```
Coq < Definition one := (S O).
one is defined
```

Ejemplo

Definición de dos

```
Coq < Definition two : nat := S one.
two is defined
```

Ejemplo

Definición de doblar un número

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

Introducción al motor de demostraciones

- Vamos a verificar la veracidad de una tautología.
- $((A \to (B \to C)) \to (A \to B) \to (A \to C)$
- `Lemma tautology1 : forall A B C : Prop, (A->(B->C))->(A->B)->(A->C). -`

Introducción al motor de demostraciones

- Vamos a verificar la veracidad de una tautología.
- $((A \to (B \to C)) \to (A \to B) \to (A \to C)$
- `Lemma tautology1 : forall A B C : Prop, (A->(B->C))->(A->B)->(A->C). -`

```
1 goal

============================
\forall A B C : Prop, (A -> B -> C) -> (A -> B) -> A -> C
```

¿Qué ocurre en este momento?

¿Qué ocurre en este momento?

- El sistema muestra la meta u objetivo debajo de la doble línea (lo que queremos demostrar).
- Las hipótesis locales se presentan arriba de dicha línea (en este caso aún no hay).
- Esta combinación de hipótesis locales y metas se denomina juicio.
- En este momento nos encontramos en un ciclo interno del sistema que se denomina modo de demostración.
- En este modo existen comandos denominados tácticas que nos ayudaran a alcanzar nuestra meta.

¿Qué son las tácticas?

¿Qué son las tácticas?

- Estrategias de demostración.
- Operan en la meta actual buscando construir una prueba para el juicio correspondiente.
- Pueden partir de pruebas de juicios hipotéticos.
- Pueden generar conjeturas que se añaden a la lista de elementos útiles para resolver el juicio original.
- Por ejemplo, la táctica `intro` se puede aplicar a cualquier juicio en el cual su meta es una implicación.
- Mueve la proposición a la izquierda de la aplicación a la lista de hipótesis locales

Introducción al motor de demostraciones

- Aplicamos la táctica intros

```
intros.
1 goal

  A, B, C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
  C
```

Introducción al motor de demostraciones

- Nos damos cuenta que la meta es C
- Para obtenerla simplemente aplicamos la hipótesis H.

```
apply H.
2 goals

  A, B, C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
  A
subgoal 2 is:
 B
```

Introducción al motor de demostraciones

- Tenemos dos submetas A y B
- La primera ya la tenemos con la hipótesis H1
- Por tal aplicamos la táctica `exact H1`

```
exact H1.
1 goal

  A, B, C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
  B
```

Introducción al motor de demostraciones

- Si aplicamos *H*0 ya necesitamos solamente A

```
apply H0.
1 subgoal

  A, B, C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
  A
```

Introducción al motor de demostraciones

- Utilizamos `exact H1`
- Para terminar se usa `Qed`

¿Cuáles son algunas de estas tácticas útiles para resolver metas sencillas?

¿Cuáles son algunas de estas tácticas útiles para resolver metas sencillas?

- `assumption`: resuelve una meta que ya se supone en el contexto
- `reflexivity`: resuelve una meta si existe una igualdad trivial
- `trivial`: resuelve una serie de metas sencillas
- `auto`: resuelve una gran variedad de metas sencillas
- `discriminate`: resuelve una meta si es una desigualdad trivial o una falsa igualdad
- `exact`: resuelve una meta si se conoce previamente el término que demuestra la meta
- `contradiction`: resuelve una meta si el contexto contiene `False` o una hipótesis contradictoria

¿Cuáles son algunas tácticas útiles para transformar metas?

¿Cuáles son algunas tácticas útiles para transformar metas?

- `intro / intros`: introducen variables definidas o términos que aparecen a la izquierda de las implicaciones
- `simpl`: simplifica una meta o una hipótesis dentro del contexto
- `apply`: utiliza implicaciones para transformar metas e hipótesis
- `rewrite`: reemplaza un término con otro equivalente si dicha equivalencia ya ha sido demostrada
- `left / right`: reemplaza una meta que consiste de una disyunción P \/ Q con solamente P o Q

¿Cuáles son algunas tácticas útiles para dividir metas e hipótesis?

¿Cuáles son algunas tácticas útiles para dividir metas e hipótesis?

- `split`: reemplaza una meta que consiste de una conjunción $P \wedge Q$ con dos submetas P y Q
- `destruct (and/or)`: reemplaza una hipótesis $P \wedge Q$ con dos hipótesis P y Q. De manera alternativa si la hipótesis es una disyunción $P \vee Q$, genera dos submetas, una donde P se mantiene y otra donde Q se mantiene.
- `destruct (análisis por casos)`: genera una submeta para cada constructor de tipo inductivo
- `induction`: genera una submeta para cada constructor de tipo inductivo y provee una hipótesis inductiva para constructores definidos recursivamente

¿Cuáles son algunas tácticas útiles para resolver metas?

¿Cuáles son algunas tácticas útiles para resolver metas?

- `ring:` resuelve metas que consisten de operaciones de adición y multiplicación
- `tauto:` resuelve metas que consisten de tautologías que se mantienen bajo una lógica constructiva
- `field:` resuelve metas que consisten de adicción, substracción, multiplicación y división

¿Cómo realizar una demostración?

¿Cómo realizar una demostración?

$P \to P$

```
Lemma id_P : forall P : Prop, P -> P.
```

¿Cómo realizar una demostración?

$P \to P$

```
Lemma id_P : forall P : Prop, P -> P.
intros.
```

¿Cómo realizar una demostración?

$P \to P$

```
Lemma id_P : forall P : Prop, P -> P.
intros.
exact p.
```

¿Cómo realizar una demostración?

$P \to P$

```
Lemma id_P : forall P : Prop, P -> P.
intros.
exact p.
Qed.
```

Ejemplo

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
```

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
intros.
```

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
intros.
apply H0.
```

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
intros.
apply H0.
apply H.
```

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
intros.
apply H0.
apply H.
exact H1.
```

Ejemplo

$(P \to Q) \to (Q \to R) \to P \to R$

```
Lemma imp_trans : forall P Q R : Prop,
(P->Q)->(Q->R)->P->R.
intros.
apply H0.
apply H.
exact H1.
Qed.
```

Ejemplo

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.
P : “Juliana es una gran matemática”
Q : “Martina es una excelente nadadora”

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
destruct H.
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
destruct H.
split.
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
destruct H.
split.
exact H0.
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.

P : “Juliana es una gran matemática”

Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
destruct H.
split.
exact H0.
exact H.
```

Ejemplo

Demuestre que las premisas “Juliana es una gran matemática” y “Martina es una excelente nadadora” permiten concluir que “Martina es una excelente nadadora y Juliana es una gran matemática”.
P : “Juliana es una gran matemática”
Q : “Martina es una excelente nadadora”

```
Lemma and_comm : forall P Q : Prop, (P/ \Q)->(Q /
\P).
intros.
destruct H.
split.
exact H0.
exact H.
Qed.
```

Ejemplo

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
right.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
right.
exact H.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
right.
exact H.
left.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
right.
exact H.
left.
exact H.
```

Ejemplo

Demuestre que la premisa “Carlos va a salir de compras o se va a quedar viendo televisión” permite concluir que “Carlos se va a quedar viendo televisión o va a salir de compras”.

P : “Carlos va a salir de compras”

Q : “Carlos se va a quedar viendo televisión”

```
Lemma or_comm : forall P Q : Prop, (P \/ Q)->(Q \/
P).
intros.
destruct H.
right.
exact H.
left.
exact H.
Qed.
```

Ejemplo

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
```

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
intros.
```

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
intros.
destruct H.
```

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
intros.
destruct H.
left.
```

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
intros.
destruct H.
left.
exact H.
```

Ejemplo

Demuestre que las premisas “A Andrés le gusta jugar tenis” y “A Andrés le gusta jugar fútbol” permiten concluir que “A Andrés le gusta jugar tenis o fútbol”.

P : “A Andrés le gusta jugar tenis”

Q : “A Andrés le gusta jugar fútbol”

```
Lemma example1 : forall P Q : Prop, (P/ \Q)->(P\/Q).
intros.
destruct H.
left.
exact H.
Qed.
```