Primera práctica (1)[1]

1 Arrancar con Coq

- El comando `coqtop` lanza el sistema.
- Todos los comandos comienzan con mayúscula y se terminan con un punto.
- El comando para salir de Coq es `Quit`.
- En Coq se puede cambiar de directorio de trabajo con `Cd <directorio>`. El directorio actual se obtiene con `Pwd`.
- Se puede utilizar cualquier editor disponible para escribir las definiciones, theoremas y pruebas. Despues, cortar y pegar en la ventana de Coq para evaluar. Con LaTeX se puede utilizar el siguiente ejemplo de archivo `ejemplo_coq_tex.tex`:

```
\documentclass[11pt]{article}
\setlength{\textheight}{25cm}
\setlength{\textwidth}{15cm}
\setlength{\oddsidemargin}{.5cm}
\setlength{\evensidemargin}{.5cm}
\setlength{\topmargin}{0cm}
\begin{document}
\thispagestyle{empty}
Veamos un ejemplo de codigo coq:
\begin{coq_example}
Print nat.
Print le.
\end{coq_example}
\end{document}
```

que, mediante el comando `coq-tex` produce el archivo `ejemplo_coq_tex.v.tex` en el cual, el código Coq ha sido evaluado y el resultado incluido en el texto. Este nuevo archivo ya puede ser compilado con `latex`.

2 El entorno Coq

Algunos comandos útiles:

- `Inspect` *num* : imprime el tipo de los últimos m objetos del entorno.
- `Search` *nombre*: imprime el tipo de las declaraciones tales que *nombre* sea la constante de la "cabeza" de ese tipo.

Ejercicio 1 *Buscar todas las propiedades conocidas sobre la conjunción* `and`*, la disyunción* `or`*, la igualdad* `eq` *y el orden en los naturales* `le`*.*

[1] Deberá enviarse por *email* a `freire@fi.udc.es` con el subject (`PMD2`), antes del 20 de Noviembre, un trabajo personal que contenga las soluciones a los ejercicios explicadas con detalle y con todos los comentarios que se consideren necesarios. NO SE ADMITEN EN PAPEL. Se recomiendan los formatos ps, pdf, o texto simple.

- `Print` *nombre*: imprime el cuerpo de la constante *nombre* y su tipo.

  Ejercicio 2 *Buscar la definición del tipo* `nat`*, de los órdenes* `le, lt` *y de las pruebas de los lemas* `eq_S` *y* `f_equal` *dando una explicación detallada de su significado.*

- `Check` *término*: verifica que *término* es tipable e imprime su tipo. Si *término* es un identificador este comando permite imprimir el tipo de esta constante.

  Ejercicio 3 *¿Qué prueba el lema* `nat_case`*?*

  Ejercicio 4 *¿Están bien formados los siguientes términos:*
  `(plus O (S O), (plus true false)`*?*

- `Require` *nombre*: busca el módulo compilado `nombre.vo` en el camino de búsqueda actual y lo añade al entorno a menos que ya hubiera sido cargado anteriormente. Los módulos ya cargados y el camino de búsqueda pueden imprimirse con los comandos `Print Modules` y `Print LoadPath`.

  Librerías útiles para la artimética son `Le, Lt, Plus, Mult` y `Between`

  Ejercicio 5 *Cargar uno de los módulos del directorio* `..../theories/Arith` *y buscar de nuevo los teoremas conocidos sobre, por ejemplo,* `le`*.*

- `Reset` *nombre*: devuelve el sistema al estado en el que estaba antes de la declaración de *nombre*. `Reset Initial` devuelve el sistema al estado inicial.

3 Definiciones en Coq

3.1 Parámetros y definiciones

Para introducir los parámetros de la teoría que se quiere desarrollar la sintaxis es:

$$\{\ \texttt{Hypothesis} \mid \texttt{Axiom} \mid \texttt{Parameter}\ \}\ \textit{nombre} : \textit{tipo}^2$$

donde *nombre* es el nombre de la hipótesis o de la variable a introducir y *tipo* es su tipo.

Se pueden añadir varias variables con el mismo tipo utilizando la palabra clave `Variables` o `Hypothesis` y como *nombre* una lista de nombres separados por comas.

Por ejemplo, para introducir un tipo para los elementos de una lista y dos variables de este tipo se escribe:

```
Parameter A : Set.
Paramenter x,y : A.
```

La manera usual de introducir una definición es:

$$\texttt{Definition}\ \textit{nombre} := \textit{término} : \textit{tipo}$$

donde el tipo de la definición es opcional.

[2] esta notación significa que se debe utilizar una de estas palabras clave.

3.2 Definición de estructuras de datos : `Inductive Set`

La sintaxis es:

$$\texttt{Inductive Set}\ \textit{nombre} := c_1 : C_1 | \ldots | c_n : C_n$$

o también:

$$\texttt{Inductive}\ \textit{nombre} : \texttt{Set} := c_1 : C_1 | \ldots | c_n : C_n$$

donde *nombre* es el nombre del tipo a definir, c_i son los nombres de los constructores del tipo y los C_i son los tipos de los constructores.

Ejemplo. El tipo de los naturales se declara con el comando siguiente:

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

Nótese que los nombres de los constructores deben ser identificadores. En particular O es la letra mayúscula y no la cifra cero.

Ejercicio 6 *El tipo de los enteros puede representarse mediante el tipo siguiente:*

- *Enteros relativos (Zr) pueden ser representados como una estructura con cero, y dos inyecciones* `pos` *y* `neg` *de* `nat` *en Zr. Así,* `(pos n)` *representa el entero positivo* $n+1$ *y* `(neg n)` *al entero negativo* $-n-1$.
- *o bien con el que se implementa en la librería ZArith.*

Construye un isomorfismo entre ambos que respete la semántica.

Ejercicio 7 *Dar una interpretación del tipo:*

```
Inductive tree012:Set:=
   c0:tree012
  |c1:tree012 -> tree012
  |c2:tree012 -> tree012 -> tree012.
```

y compararla cpon la siguiente:

```
Inductive tree12:Set:=
  |c1':tree012 -> tree012
  |c2':tree012 -> tree012 -> tree012.
```

3.3 Definición de relaciones inductivas

Las definiciones inductivas de relaciones definen propiedades especificadas por cláusulas a la Prolog. Cada cláusula se traduce como el tipo de un constructor (axioma).

La sintaxis de este comando es:

$$\texttt{Inductive}\ \textit{nombre} :\ \textit{aridad} := c_1 : C_1 | \ldots | c_n : C_n$$

donde *nombre* es le nombre de la relación a definir, *aridad* es su tipo (por ejemplo `nat->nat->Prop` para una relación binaria sobre los naturales) y como antes c_i y C_i son, respectivamente, el nombre y el tipo de los constructores.

Ejemplo. La definición de la relación de orden sobre los naturales puede ser especificada por dos cláusulas: `(LE O n)` y `(LE n m) -> (LE (S n) (S n))` que se traducen en Coq :

```
Inductive LE : nat->nat->Prop :=
   LE_0 : (n:nat)(LE 0 n)
 | LE_S : (n,m:nat)(LE n m) -> (LE (S n) (S m))
```

De hecho la definición del orden en las librerías básicas es un poco diferente:

```
Inductive le [n : nat]  : nat->Prop :=
      le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m))
```

Ejercicio 8 *Demostrar que* $\forall n, m \cdot (LE\ n\ m) \Leftrightarrow (le\ n\ m)$.

Ejercicio 9 *Definir el predicado* Natural *sobre Z que caracterice al cero y a los enteros positivos.*

Ejercicio 10 *Definir una relación* Diff *tal que* (Diff *n m z*) *signifique, con* n *y* m *naturales y* z *un entero relativo, que el valor de* $n - m$ *es* z.

Ejercicio 11 *Definir una relación* SubZ *que especifique la diferencia de dos enteros de Z.*

4 Definir una función por análisis de casos

En un patrón ($ident_1$ ... $ident_n$), el primer nombre se supone que es un constructor.

Se pueden definir funciones por casos sobre los constructores de un tipo inductivo (*pattern matching*):

$$<tipo>\ \texttt{Cases}\ term_0\ \texttt{of}\ pattern_1 \Rightarrow term_1 | \ldots | pattern_n \Rightarrow term_n\ \texttt{end}$$

donde *tipo* es el tipo del resultado de la expresión[3], $term_0$ es un término cuyo tipo es una definición inductiva con n constructores y $term_i$ representa el resultado en el caso de que $pattern_i$ comience con el i–ésimo constructor.

```
Coq <  Check [x:nat]Cases x of 0 => 0 | (S x) => x end.
[x:nat]Cases x of
         0 => 0
       | (S x0) => x0
       end
     : nat->nat

Coq < Check [H:False] <(le 0 0)>Case H of   end.
[H:False]<[_:False](le 0 0)>Cases H of end
     : False->(le 0 0)
```

También se puede escribir equivalentemente (aunque resulta menos legible):

$$<tipo>\ \texttt{Case}\ term_0\ \texttt{of}\ term_1 \ldots term_n\ \texttt{end}$$

en cuyo caso $term_i$ representa el cálculo que hay que hacer en el caso del i–ésimo constructor.

Ejercicio 12 *Definir la función predecesor de tipo* `nat->nat` *así como una función de tipo* `nat->bool` *que compruebe si su argumento es cero o no.*

Ejercicio 13 *Sea* `A : Set` *y* `P : A -> Set`. *Definir la suma dependiente con un constructor* `intro : (x:A)(P x) -> sum`. *Después, definir las dos proyecciones* `p1 : sum-> A` *y* `p2 : (s:sum)(P (p1 s))`.

[3] cuando el tipo es el mismo que los de todos los términos $term_i$, esta anotación no es necesaria

4.1 El caso general

En `Coq` la construcción `Case` se utiliza uniformemente en todas las definiciones inductivas. Pero estas construcciones no son a menudo directamente necesarias sino que son utilizadas implicitamente por las tácticas de eliminación u otros términos generados automáticamente.

4.2 Definir las funciones por punto fijo

Para codificar las funciones interesanrtes sobre las estructuras inductivas es necesario utilizar una cierta forma de recursión. La recursión general no puede ser utilizada pues conduce a incoherencias lógicas.

La recurrencia autorizada es estructural. Una función puede definirse por punto fijo si uno de sus argumentos formales x tiene por tipo una definición inductiva y en cada llamada recursiva el argumento correspondiente puede ser reconocido, sintácticamente, como estructuralmente más pequeño que x.

La idea básica es que x es el argumento principal de una expresión `Case` y la llamada recursiva puede hacerse en las ramas sobre variables del patrón.

4.2.1 La construcción `Fixpoint`

La forma de definir un programa mediante la técnica del punto fijo es:

$$\texttt{Fixpoint}\ \mathit{nombre}\ [x_1 : \mathit{tipo}_1; \ldots; x_p : \mathit{tipo}_p; x : \mathit{tipo}_I] : \mathit{tipo}_f := \mathit{term}$$

La variable x (la última variable en el contexto después de *nombre*) es la variable de recursión. Su tipo tipo_I debe tener un tipo inductivo.

El tipo de *nombre* es $(x_1 : \mathit{tipo}_1) \ldots (x_p : \mathit{tipo}_I)\mathit{tipo}_f$. Las apariciones de *nombre* en *term* deben aparecer aplicadas a $p + 1$ argumentos y el $p + 1$–ésimo debe ser reconocido como más pequeño que x.

4.2.2 Ejemplos

Las dos definiciones siguientes de `plus` por recursión sobre el primer y segundo argumento son correctas:

```
Fixpoint plus1 [n:nat]:nat->nat :=
      [m:nat]<nat>Case n of m [p:nat](S (plus1 p m)) end.
```

O equivalentemente, con una notación más legible:

```
Fixpoint plus1 [n:nat]:nat->nat :=
      [m:nat]Cases n of
              O => m
        | (S p) => (S (plus1 p m))
             end.
```

Analogmente:

```
Fixpoint plus2 [n,m:nat]:nat :=
      <nat>Case m of n [p:nat](S (plus2 n p)) end.
```

```
Fixpoint plus2 [n,m:nat]:nat :=
```

```
           Cases m of
               O => n
           | (S P)=> (S (plus2 n p))
           end.
```

pero

```
Coq < Fixpoint plus_incorrecta [n,m:nat]:nat :=
      <nat>Case n of m [p:nat](S (plus_incorrecta  p m)) end.

Coq < Error:
Recursive call applied to an illegal term
The recursive definition plus_incorrecta :=
[n,m:nat]Cases n of
           O => m
         | (S p) => (S (plus_incorrecta p m))
         end is not well-formed
```

4.3 Comprobar igualdades

Se puede imprimir el resultado de una evaluación de una función con el comando `Eval Compute in`.

Por ejemplo, el comand `Eval Compute in [x:nat](plus O x)` nos dará el resultado `[x:nat]x : nat->nat`

Ejercicio 14 *Escribir la función* `or` *sobre los booleanos.*
Verificar la especificación `(or true _)=true, (or false b) = b.`

Ejercicio 15 *Escribir una función de Z en Z que calcule el opuesto de un entero.*

Ejercicio 16 *Escribir una inyección de* nat *de tipo* nat -> Z *que preserve la "semántica".*

Ejercicio 17 *Escribir una función de tipo* nat -> nat -> Z *que calcule la diferencia entre dos entero naturales. La especificación es:*

(diff 0 0)=cero, (diff (O (S *n*)) = (neg n), (diff (S *n*) O) = (pos *n*), (diff (S *n*) (S *m*)) = (diff *n m*)

Ejercicio 18 *Utilizar la función* diff *para definir la suma de dos enteros relativos como una función de tipo Z -> Z -> Z.*

Ejercicio 19 *Definir una función que asocie a cada espresión de tipo* expr *su "semántica" como elemento de Z.*

5 Pruebas en Coq

Para probar un teorema en Coq, es necesario enunciar la propiedad a probar. Esto se hace con la declaración:

{ `Theorem` | `Lemma` } *nombre : tipo*

donde *nombre* es el nombre de la prueba del teorema y *tipo* su enunciado.
Los comandos siguientes permiten construir una prueba de un teorema:

- `Proof` *term*: da explicitamente *term* como un término que es una prueba del teorema.
- `Save` : guarda en el entorno una prueba terminada.

Los comandos siguientes se utilizan en el caso de una tentativa de prueba errónea:

- `Undo` : anula la última táctica.
- `Restart` : anula toda la prueba.
- `Abort` : anula toda la prueba así como el enunciado del teorema.

La manera usual de combinar las tácticas es `tact1 ; tact2` que aplica `tact2` a todos los subobjetivos obtenidos por `tact1`.

5.1 Razonamiento de primer orden

En la tabla siguiente se dan las correspondencias entre la sitaxis de Coq, los conectores usuales y los nombres de las tácticas de introducción y de eliminación.

Proposición (P)	Sintaxis Coq	Introducción	Eliminación (H de tipo P)
$\bot$	`False`		`Elim` H, `Absurd`
$\neg A$	`~A`	`Red; Intro`	`Apply` H
$A \wedge B$	`A/\B`	`Split`	`Elim` H
$A \Rightarrow B$	`A -> B`	`Intro`	`Apply` H
$A \vee B$	`A\/B`	`Left, Right`	`Elim` H
$\forall x : A.P$	`(x:A)P`	`Intro`	`Apply` H
$\exists x : A.P$	`(Ex [x:A]P)`	`Exists` *testigo*	`Elim` H
$x =_A y$	`x = y`	`Reflexivity, Trivial`	`Elim` H, `Rewrite` H

`Assumption`. Busca un elemento del contexto que tenga asociado como tipo el objetivo actual.

`Apply with, Elim with`. En las reglas de eliminación, el argumento de las tácticas `Apply` o `Elim` un término que prueba la premisa principal.

En general el argumento H de las tácticas `Apply` o `Elim` no prueba exactamente el objetivo principal P sino una generalización `(x1:A1)...(xn:An)P'`. Las tácticas intentan encontrar una generalización adecuada de H que al ser eliminada engendrará, si es necesario, nuevos subobjetivos a probar. Si algunos argumentos no pueden ser inferidos, las tácticas `Apply` y `Elim` nos lo indican. En estos casos las versiones derivadas pueden ser utilizadas { `Apply` | `Elim` } H `with` $t_1 \ldots t_k$.

Ejercicio 20 *Probar las tautologías siguientes:*

$$A \wedge (B \vee C) \Rightarrow (A \wedge B) \vee (A \wedge C) \quad \neg\neg\neg A \Rightarrow \neg A$$

$$A \vee (\forall x.(Px)) \Rightarrow \forall x.(A \vee (Px)) \quad \exists x.\forall y.(Qx\,y) \Rightarrow \forall y.\exists x.(Qx\,y)$$

5.2 Razonamiento ecuacional

Probar igualdades. Si dos términos t y u son convertibles en Coq entonces la táctica `Reflexivity` prueba el objetivo $t = u$. Para reemplazar el objetivo $t = u$ por el objetivo $u = t$, se puede utilizar la táctica `Symmetry`. Para utilizar una etapa de transitividad, la táctica `Transitivity` v puede ser utilizada para generar los dos subobjetivos $t = v$ y $v = u$.

Utilizar las igualdades para reescribir La regla de eliminación para la igualdad dice que si existe una prueba de $t = u$ entonces para toda propiedad P si $P(t)$ es verdadera entonces $P(u)$ también lo es. La táctica de eliminación sobre una prueba de $t = u$ reemplaza el objetivo $P(u)$ por un nuevo objetivo $P(t)$.

Algunas variantes:

- Si H prueba $t = u$ se puede reemplazar t por u mediante la táctica `Rewrite` H.
- Para reemplazar t por u y generar el subobjetivo $u = t$ se utiliza la tactica `Replace` t `with` u.
- Para reemplazar sólamente ciertas apariciones del término u se utiliza la táctica `Pattern` *occ* u, donde *occ* es una lista de las apariciones a reescribir antes de utilizar las tácticas `Elim`, `Rewrite` o `Replace`.

Convertibilidad. Las tácticas `Simpl` o `Unfold` *nombre* reducen el objetivo transformando un término objetivo en otro calculatoriamente equivalente.

Ejercicio 21 *Probar el lema siguiente:*

```
Section induccion1.
Variables C:Prop;c:C;H:nat->C->C.
Fixpoint natind [n:nat]: C :=
          Cases n of O => c
                |(S p) => (H p (natind p))
          end.
Lemma equiv1: (n:nat)(nat_ind [x:nat]C c H n)==(natind n).
```

Ejercicio 22 *Probar el lema siguiente:*

```
Section induction2.

Variables P:nat->Prop;c:(P O);H:(n:nat)(P n)->(P (S n)).

Fixpoint natindd [n:nat]: (P n):=
<P>Cases n of   (* ESTA P ES IMPRESCINDIBLE *)
     O => c
|(S p) => (H p (natindd p))
   end.

End induction2.

Lemma equiv2:(P:nat->Prop;c:(P O);H:(n:nat)(P n)->(P (S n));n:nat)
(nat_ind P c H n)==(natindd P c H n).
```

Ejercicio 23 *Probar la estabilidad de los constructores del tipo Z por la igualdad.*

Ejercicio 24 *Probar la dirección opuesta. Por ejemplo* $(\mathsf{neg}\ n) = (\mathsf{neg}\ n) \Rightarrow n = m$.

Ejercicio 25 *Probar una de las igualdades definicionales de la función* diff. *Por ejemplo* $(\mathsf{diff}\ (S\,n)\ (S\,m)) = (\mathsf{diff}\ n\,m)$.

5.3 Prueba por recurrencia

La recurrencia corresponde a la eliminación de un objeto de tipo inductivo. La táctica general para una prueba por recurrencia es `Elim` *term*. Más precisamente, se trata de la aplicación de un esquema generado en el momento de la definición inductiva.

La táctica `Induction`. A menudo la recurrencia se utiliza sobre variables o hipótesis presentes desde el comienzo. Las tácticas `Induction` *nombre* e `Induction` *num* realizan primero las introducciones justo hasta la variable ligada *nombre* o la *num*–ésima hipótesis. A continuación procede a la eliminación del último objeto introducido.

Recurrencia sobre los objetos. Si *term* tiene tipo inductivo la recurrencia utilizada es la generalización natural de la recurrencia sobre los naturales. La dificultad principal es indicar al sistema la propiedad a demostrar por recurrencia.

Por defecto la propiedad inferida es la abstracción del objetivo respecto a todas las apariciones de *term*.

Si sólamente ciertas apariciones deben ser abstraidas, entonces la táctica `Pattern` *occ term* puede ser preferible.

A veces es necesario generalizar el objetivo antes de la recurrencia. Esto se puede hacer con la táctica `Cut` *tipo* que cambia el objetivo G por el *tipo* $\Rightarrow G$ y genera el nuevo objetivo *tipo*. Si la generalización se refiere a hipótesis en el contexto, se puede utilizar la táctica `Generalize`. Si `x` es una variable de tipo `A`, la táctica `Generalize x` cambia el objetivo `G` por el `(x:A)G`.

Recurrencia sobre las pruebas. Si *term* pertenece a una relación inductiva entonces la táctica de eliminación corresponde a la utilización de la minimalidad de la relación inductiva respecto de las cláusulas que la definen.

En general la propiedad a probar es $(I\, x_1 \ldots x_n) \Rightarrow G$ donde I es la relación inductiva. El objetivo G es abstraido respecto a $x_1 \ldots x_n$ para formar la relación utilizada en la recurrencia. Esto funciona correctamente cuando $x_1 \ldots x_n$ son variables. En caso de que $x_1 \ldots x_n$ sean términos estructurados, el sistema puede que no encuentre ninguna abstracción correctamente tipada o que encuentre una propiedad no demostrable. En este caso la táctica `Inversion` *term* en lugar de `Elim` *term* combina tácticas elementales para extraer el máximo de información de *term*.

Ejercicio 26 *Probar las propiedades (*Diff n n cero*), (*Diff $(S\, n)$ n (pos *0)).*

Ejercicio 27 *Probar que para cualesquiera naturales n y m, si $n \leq m$ entonces existe un entero z tal que (*Natural z*) y (*Diff n m z*).*

Ejercicio 28 *Probar que para todos los naturales n y m la propiedad (*Diff n m (diff n m)*) es cierta.*

Ejercicio 29 *En la siguiente definición:*

```
Section Max.
Require Arith.

Definition max :=
 (nat_rec [_:nat]nat->nat [n1:nat]n1
   [n1:nat]
    [Rec:nat->nat]
     [H0:nat]
```

```
 Cases H0 of
       O => (S n1)
| (S n2) => (S (Rec n2))
        end )
```

pretende definir la función que nos calcula el mínimo de dos números. Probar las siguientes propiedades:

```
max_0_r: (a:nat)(max a O)=a.
max_0_l: (a:nat)(max O a)=a.
max_sym: (a, b:nat)(max a b)=(max b a).
max_intro_l: (a, b:nat)(le a (max a b)).
max_intro_r: (a, b:nat)(le b (max a b)).
max_id: (a:nat)(max a a)=a.
max_le: (a, b:nat) (le a b) ->(max a b)=b.
max_is_lub: (a, b, c:nat) (le a c) -> (le b c) ->(le (max a b) c).
```

Ejercicio 30 *Construir la* prueba *adecuada de* `Definition max: nat -> nat ->nat.` *para que coincida con la función* `max` *del ejercicio anterior.*