Práctico 4

Definiciones Inductivas

- Primera Parte -

Objetivos: Trabajar con tipos inductivos simples, paramétricos y mutuamente recursivos. Definir funciones recursivas y demostrar propiedades utilizando recursión.

Principales tácticas a utilizar en estos ejercicios:

`Inductive`

`Inductive with`

`Fixpoint`

`match` **term,** `destruct` **term**

`elim` **term,** `induction` **id**

`constructor` **[n]**

`discriminate , simplify_eq, …`

Ejercicio 4.1. Defina en Coq los siguientes tipos inductivos:

1. Listas y árboles binarios paramétricos (`list` y `bintree`).
2. Arreglos y matrices paramétricos de cierto largo (`Array` y `Matrix`).
3. La relación `leq` (*menor o igual*) entre naturales.
4. La relación `eq_list` de igualdad entre listas de cualquier tipo A a partir de la igualdad en A.
5. El predicado `sorted` sobre listas que especifique que una lista esta ordenada crecientemente según una relación de orden R.
6. La relación binaria `mirror` entre árboles que especifique que un árbol es “espejo” del otro (o sea, que tengan la misma estructura, pero con todos sus sub-árboles invertidos).
7. La relación binaria `isomorfo` entre árboles que especifique que un árbol es igual estructuralmente a otro. Esto es, los dos árboles poseen igual estructura aunque puede diferir el contenido de los nodos en ambos árboles.
8. El tipo inductivo `Gtree` de los árboles generales (finitarios) cuyos nodos internos son de un tipo y las hojas de otro (hay que definirlo mutuamente con las forestas).

Ejercicio 4.2. Escriba en Coq utilizando pattern matching (`match`):

1. Un programa que implemente las funciones booleanas `Or:bool→bool→bool`, `And:bool→bool→bool`, `Not:bool→bool` y `Xor:bool→bool→bool` (O exclusivo).
2. Un programa `is_nil` que decida si una lista es vacía.

Ejercicio 4.3. Defina recursivamente las siguientes funciones aritméticas en Coq:

1. `sum:nat->nat-> nat`
2. `prod:nat->nat->nat`
3. `pot:nat->nat->nat`
4. `leBool:nat->nat->bool`

Ejercicio 4.4. Defina recursivamente las siguientes funciones sobre listas en Coq utilizando el tipo de las listas definido en el ejercicio 4.1:

1. `length: (forall A: Set, list A -> nat)`, que calcula la longitud de una lista.
2. `append: (forall A: Set, list A -> list A -> list A)`, que concatena dos listas.
3. `reverse: (forall A: Set, list A -> list A)`, que invierte una lista.
4. `filter: (forall A: Set, (A -> bool) -> list A -> list A)`, que deja en una lista los elementos que cumplen cierta propiedad.
5. `map: (forall A B: Set, (A -> B) -> list A -> list B)`, que aplica una función a todos los elementos de una lista.
6. `exists_: (forall A: Set, (A -> bool) -> list A -> bool)`, que devuelve `true` si y sólo si hay por lo menos un elemento en la lista que cumple cierta propiedad.

Ejercicio 4.5.

1. Defina una función `inverse:bintree->bintree` sobre el tipo de árboles binarios del ejercicio 4.1 que calcule el árbol espejo del dado.
2. Defina recursivamente una función booleana sobre los árboles genéricos del ejercicio 4.1 que devuelva `true` si y sólo si la cantidad de nodos internos es mayor que la cantidad de nodos externos.

Ejercicio 4.6. Defina en Coq el tipo de las listas de naturales `ListN` como una instancia del tipo de listas paramétricas del ejercicio 4.1 y defina recursivamente las siguientes funciones sobre listas de naturales:

1. `member:nat->listN->bool`
2. `delete:listN->nat->listN`, que borra todas las ocurrencias de un elemento en una lista.

3. `insert_sort:listN->listN` que ordena una lista de naturales por el método de inserción.

Generalice las funciones anteriores para que puedan ser aplicadas a listas de cualquier tipo.

Ejercicio 4.7.

1. Defina el tipo de las expresiones aritméticas `Exp:Set->Set` formadas a partir de átomos de un conjunto `A` (de tipo `Set`), con las operaciones `+`, `*` y `-` (unario).
2. Defina una función que evalúe un elemento de `(Exp nat)` y devuelva el natural correspondiente de interpretar a `+` como la suma, `*` como el producto y `-` como la identidad.
3. Defina una función que evalúe un elemento de `(Exp bool)` y devuelva el booleano correspondiente de interpretar `+` como el Or, `*` como el And y `-` como el Not.

Ejercicio 4.8. Demuestre que sus programas del ejercicio 4.2 son correctos probando los siguientes lemas:

1. Asociatividad y conmutatividad del And y del Or
2. `LAnd : forall a b : bool, Andd a b = true <-> a = true /\ b = true`
3. `LOr1 : forall a b : bool, Or a b = false <-> a = false /\ b = false`
4. `LOr2 : forall a b : bool, Or a b = true <-> a = true \/ b = true`
5. `LXor : forall a b : bool, Xor a b = true <-> a <> b`
6. `LNot : forall b : bool, Not (Not b) = b`

Ejercicio 4.9. Demuestre las siguientes propiedades de la suma y el producto, definidos en el ejercicio 4.3:

1. `SumO : forall n : nat, sum n 0 = n`
2. `SumS : SumS : forall n m : nat, sum n (S m) = sum (S n) m`
3. `SumConm : forall n m : nat, sum n m = sum m n`
4. `SumAsoc : forall n m p : nat, sum n (sum m p) = sum (sum n m) p`
5. `ProdConm : forall n m : nat, prod n m = prod m n`
6. `ProdAsoc : forall n m p : nat, prod n (prod m p) = prod (prod n m) p`
7. `ProdDistr : forall n m p : nat,`

   `prod n (sum m p) = sum (prod n m) (prod n p)`

Ejercicio 4.10. Demuestre las siguientes propiedades de las funciones definidas en el ejercicio 4.4 (no escribimos el parámetro A para hacer más fácil la lectura):

1. `L1 : forall (A : Set) (l : list A), append A l (nil A) = l`

2. `L2 : forall (A : Set) (l : list A) (a : A), ~(cons A a l) = nil A`

3. ```
   L3 : forall (A : Set) (l m : list A) (a : A),
          cons A a (append A l m) = append A (cons A a l) m
   ```

4. ```
   L4 : forall (A : Set) (l m : list A),
          length A (append A l m) = sum (length A l) (length A m)
   ```

5. `L5 : forall (A : Set) (l : list A), length A (reverse A l) = length A l`

6. ```
   L6 : forall (A : Set) (l m : list A),
           reverse A (append A l m) = append A (reverse A m) (reverse A l)
   ```

Ejercicio 4.11. Demuestre las siguientes propiedades de las funciones definidas en el ejercicio 4.4:

1. ```
   L7 : forall (A B : Set) (l m : list A) (f : A -> B),
        map A B f (append A l m) = append B (map A B f l) (map A B f m).
   ```

2. ```
   L8 : forall (A : Set) (l m : list A) (P : A -> bool),
        filter A P (append A l m) = append A (filter A P l) (filter A P m)
   ```

3. ```
   L9 : forall (A : Set) (l m n : list A),
        append A l (append A m n) = append A (append A l m) n
   ```

4. `L10 : forall (A : Set) (l : list A), reverse A (reverse A l) = l`

Ejercicio 4.12. Considere la siguiente función definida sobre listas de elementos de un tipo genérico:

```
Fixpoint filterMap (A B : Set) (P : B -> bool) (f : A -> B)
      (l : list A) {struct l} : list B :=
      match l with
      | nil => nil B
      | cons a l1 =>
            match P (f a) with
            | true => cons B (f a) (filterMap A B P f l1)
            | false => filterMap A B P f l1
      end
end.
```

Pruebe el siguiente lema:

```
Lemma FusionFilterMap :
      forall (A B : Set) (P : B -> bool) (f : A -> B) (l : list A),
      filter B P (map A B f l) = filterMap A B P f l.
```

Donde las funciones `filter` y `map` son las definidas en el ejercicio 4.4.

Ejercicio 4.13. Demuestre que la función `inverse` del ejercicio 4.5 da como resultado un árbol que cumple con la propiedad de ser espejo del dado (utilice la definición de espejo del ejercicio 4.1).

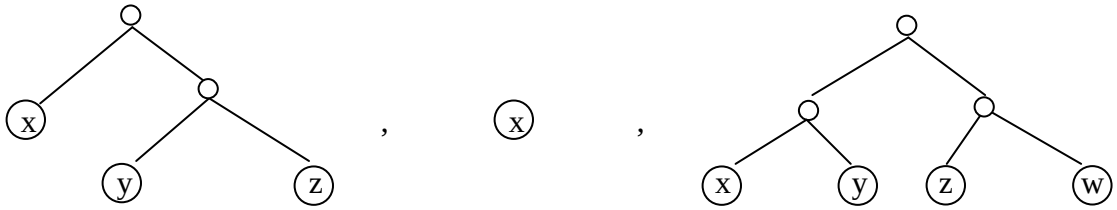
Ejercicio 4.14.

1. Demuestre que la función identidad para árboles binarios de elementos de un tipo genérico da como resultado un árbol que cumple con la propiedad de ser isomorfo del dado.

2. Pruebe que la relación de isomorfismo entre árboles binarios es reflexiva y simétrica.

Utilice la definición de isomorfo del ejercicio 4.1.

Ejercicio 4.15. Defina el tipo de datos `Tree` que representa árboles binarios de elementos de un tipo genérico que sólo guarda información en los nodos hojas (nodos externos). Los nodos internos no guardan información. El árbol más pequeño es una hoja. Ejemplos:



1. Defina una función `mapTree` que dado un árbol de tipo (`Tree A`), para un tipo genérico `A`, y una función `f:A->B`, con `B` un Set dado, retorne un árbol de tipo (`Tree B`) obtenido por la aplicación de la función `f` a cada uno de los nodos hojas del árbol parámetro.

2. Defina una función que cuente la cantidad de nodos hojas que posee un árbol de tipo (`Tree A`), para un tipo genérico `A`.

3. Pruebe que la función `MapTree` preserva la cantidad de nodos hojas del árbol parámetro.

4. Defina una función `hojas` que retorne una lista con las hojas de un árbol de tipo (`Tree A`), para un tipo genérico `A`. Pruebe luego que la cantidad de nodos hojas que posee un árbol de tipo (`Tree A`), para un tipo genérico `A`, es igual a la longitud de la lista resultante de aplicarle la función `hojas` al árbol.

Ejercicio 4.16.

1. Defina ***inductivamente*** una relación binaria *posfijo* entre listas de elementos de un tipo genérico, tal que una lista l1 se relaciona con una lista l2 si y sólo si l1 es un posfijo de l2 (notación: l1 « l2). Más formalmente, l1 « l2 sii l2 = l3++l1, para alguna lista l3, donde ++ representa la concatenación de listas (append).

2. Defina la función ++ entre listas de elementos de un tipo genérico y pruebe la corrección de la relación *posfijo* del ejercicio anterior demostrando que:

   - para todas listas l1, l2 y l3: Si l2 = l3++l1 entonces l1 « l2.
   - para todas listas l1, l2: Si l1 « l2 entonces existe una lista l3 tal que l2 = l3++l1.

3. *Defina* una función *último* que dada una lista de elementos de un tipo genérico retorne la lista que contiene a su último elemento. Si la lista parámetro es vacía, la función *último*

retorna la lista vacía.

4. Pruebe que la función *último* de una lista es un posfijo de dicha lista.

Ejercicio 4.17.

1. Defina el tipo de datos **`ABin`** de árboles binarios con nodos internos de un tipo genérico **`A`** y nodos externos (hojas) de un tipo genérico **`B`**. El árbol más pequeño es una hoja.
2. Defina una función que cuente la cantidad de nodos externos de un árbol binario de tipo **`ABin`**.
3. Defina una función que cuente la cantidad de nodos internos de un árbol binario de tipo **`ABin`**.
4. Pruebe que la cantidad de nodos externos en un árbol árbol binario de tipo **`ABin`** es igual a la cantidad de nodos internos del árbol más 1. Puede utilizar tácticas automáticas.

Ejercicio 4.18.

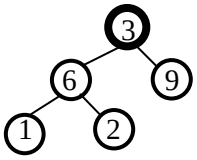
Considere las siguientes definiciones:

```
Section Ej_18.

   Variable A : Set.
   Inductive Tree_ : Set :=
         | nullT : Tree_
         | consT : A -> Tree_ -> Tree_ -> Tree_.
```

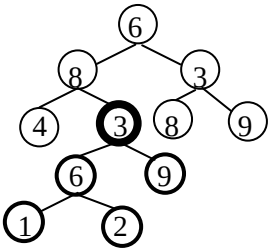
Considere además la siguiente definición de la relación binaria subárbol entre árboles binarios:

A* es subárbol de *B si *A* es un árbol hijo de algún nodo de *B*, o *A* es igual a *B*.

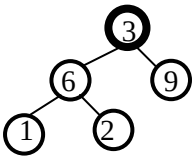
Por ejemplo,



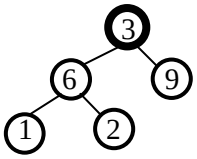
es un subárbol de



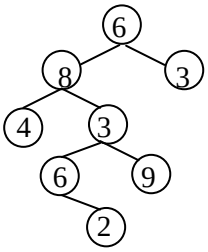
y de



Pero,

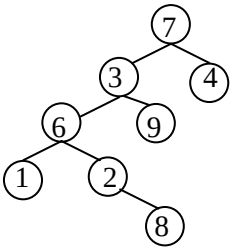


NO es un subárbol de



[pues falta el (1)]

NI de



[pues sobra el (8)]

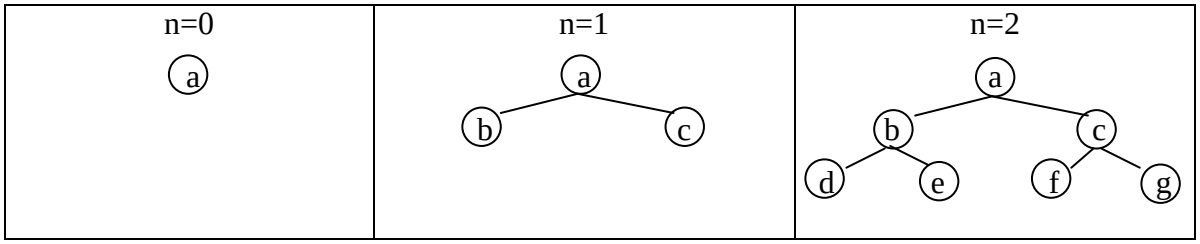
1. Defina indutivamente la relación `isSubTree` entre árboles binarios de tipo `Tree_`.
2. Pruebe que la relación `isSubTree` es reflexiva.
3. Pruebe que la relación `isSubTree` es transitiva.

```
End Ej_18.
```

Ejercicio 4.19.

1. Defina inductivamente la familia de conjuntos sobre los naturales `ACom` de árboles no vacíos de altura n y completos (o sea, que tienen todos sus niveles llenos), con nodos internos y externos de un tipo genérico `A`.

Ejemplos:



2. Defina una función `h` que, para todo `n:nat`, calcule la cantidad de hojas de un árbol de tipo `(ACom n)`.
3. Pruebe que para todo `n:nat` y para todo árbol de tipo `(ACom n)` se cumple que la cantidad de hojas del árbol es igual a 2^n. Puede utilizar tácticas automáticas.

   Asuma las siguientes declaraciones para la función potencia de números naturales:

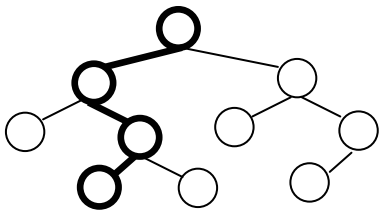
   ```
   (* Parameter pot: nat -> nat -> nat. *)

   Axiom pot0 : forall n : nat, pot (S n) 0 = 1.

   (* n^0 = 1   ∀n>0 *)

   Axiom potS : forall m: nat, pot 2 (S m) = sum (pot 2 m) (pot 2 m).

   (* 2^(m+1) = 2^m + 2^m *)
   ```

Ejercicio 4.20.

1. Defina inductivamente la familia de conjuntos sobre los naturales `AB` de árboles binarios de altura `n` y elementos de un tipo genérico `A` (la altura del árbol vacío es 0).
2. Defina una función `camino` que, para todo `n:nat` retorne una lista con el camino más largo de un árbol de tipo `(AB A n)`, con `A` el tipo genérico de los elementos del árbol. Si hay más de un camino más largo, la función debe retornar el camino de más a la izquierda de ellos. El camino debe comenzar con el nodo de la raíz del árbol y finalizar

con una hoja. Si el árbol es vacío, `camino` debe retornar la lista vacía. La función `camino` debe recorrer, en el cómputo, sólo un camino del árbol.

Ejemplo: árbol de altura 4



3. Pruebe que la función `camino`, dado un árbol de tipo `(AB A n)` retorna una lista de largo igual a `n`.

Ejercicios a entregar: 4.9, 4.10, 4.11, 4.12, 4.16, 4.18, 4.19, 4.20.

Ver fecha de envío en el calendario de entregas, en el sitio web del curso.

El archivo a entregar (NombreApellido.v) debe compilar correctamente en Coq.