Lógica Computacional 2017-2
Práctica 6:`Coq` como lenguaje de programación.

Lourdes del Carmen González Huesca
Roberto Monroy Argumedo
Fernando A. Galicia Mendoza

Facultad de Ciencias, UNAM

Fecha de entrega: Domingo, 14 de mayo del 2017

La práctica deberá ser realizada de forma individual.

Utilizando el archivo `IntroCoq.v` realiza los ejercicios indicados en las siguientes secciones.

1. Programación funcional

1.1. Listas y option

El tipo `option` representa el tipo `Maybe` de `Haskell`, es decir, es el tipo que no permite la finalización abrupta del programa para funciones parciales.
El tipo se encuentra definido de la siguiente forma:

```
Inductive option (A:Type) : Type :=
| None : option A
| Some : A → option A.
```

1. Define las versiones formales de las funciones `head` y `last` de listas, es decir, que su firma sea:

   ```
   head_error, last_error : lista A → option A
   ```

   Sugerencia: Hacer análisis de casos sobre el resultado del caso recursivo.

2. Define una función que obtenga el $n-$ésimo elemento de una lista, es decir, su firma deberá ser:

   ```
   nthElem : lista A → nat → option A
   ```

   Observación: Recuerda que las listas están indexadas desde 0 en adelante, por ejemplo la lista `[1,2,3]` el elemento 0 es el número 1, el elemento 1 es el número 2 y el elemento 2 es el número 3.

1.2. Números naturales con paridad

Recordemos que los números naturales se pueden ver como la unión de dos conjuntos: números pares y números impares. Esta idea nos permite definir una gramática alterna a la inspirada en los axiomas de Peano.
La gramática que define estos números es:

$$PNat \ ::= \ Cero \mid D\ PNat \mid I\ PNat$$

Donde:

- $Cero$ es el representante del número cero.
- $D\ x$ es el representante del número $2x$ con x un número $PNat$.
- $I\ x$ es el representante del número $2x+1$ con x un número $PNat$.

Algunos ejemplos:

- El número 7 con esta gramática sería $2(2(2(0)+1)+1)+1$.
- El número 4 con esta gramática sería $2(2(2(0)+1))$.
- El número 0 es un caso peculiar, ya que, no tiene representación única. Por ejemplo tiene la representación: 0 o bien $2(0)$ o bien $2(2(0))$ o bien $2(2(2(0)))$ y así sucesivamente.

Esta representación a pesar de parecer inconveniente nos permite realizar funciones recursivas cuyos casos se enfoquen en pares e impares, entonces le damos la vuelta a la recursión general y poder realizar recursión primitiva.
Y lo mejor de todo es: ¡Está representación es equivalente a la usual! Con un análisis especial sobre el 0.

1. Utilizando el vernáculo `Inductive` representa la gramática de los números con paridad, nombra al tipo `PNat`.

2. En la presentación `P7_LC172` se define la relación `mist` cuya especificación indica: `mist`(X, Y, Z) es válida syss $X^Y = Z$. Esta relación fue inspirada en la función de mismo nombre que es la versión eficiente de la exponenciación de números naturales. Si traduces a `Haskell` la relación, observarás que es necesario hacer recursión general, sin embargo, utilizando el tipo `PNat` esto se reduce a recursión primitiva.
   Define la función `mist`, es decir, su firma deberá ser:

   ```
   mist : PNat → PNat → PNat
   ```

3. Como se mencionó el gran inconveniente de $PNat$ es que el conjunto de expresiones representantes del 0 es infinita, esto indica que la gramática no está normalizada. Tú trabajo **no** será normalizar la gramática, pero si normalizar las representaciones del cero, es decir, define una función que reciba un $PNat$ y devuelva su representación normalizada. La firma de la función es:

   ```
   norm : PNat → PNat
   ```

   Entonces `norm (D (D Cero)) = Cero` o bien `norm (I (D (D Cero))) = I Cero`.
   Sugerencia: Hacer análisis de casos en el caso recursivo del resultado de `norm (D x)`.

2. Programación lógica

2.1. Relaciones sobre $\mathbb{N}$

Cuando le hablan de relaciones sobre números naturales una de las primeras elecciones es la relación menor igual, analicemos esta relación:

En `Haskell` definir la función menor o igual se hace de la siguiente forma:

```
leq :: Int → Int → Bool
leq 0 _ = True
leq _ 0 = False
leq (S n) (S m) = leq n m
```

Sin embargo, al momento de querer hacer demostraciones utilizando esta función, la hipótesis de inducción resulta demasiado débil, es decir, no nos da suficientes hipótesis para poder llegar al resultado.
Es por esto que se utiliza la siguiente definición recursiva:

Definición 1. *La relación menor o igual se define recursivamente de la siguiente forma:*

- *Para todo $n \in \mathbb{N}$, se tiene que $n \leq n$.*
- *Para cualesquiera $n, m \in \mathbb{N}$, se tiene que si $n \leq m$, entonces $n \leq S(m)$.*

Con esto por ejemplo demostrar el siguiente hecho es realmente sencillo:

Teorema 1. *La relación $\leq$ es reflexiva.*

Demostración. Por definición el caso base de la relación $\leq$, indica que para cualquier $n \in \mathbb{N}$ se tiene que $n \leq n$. □

Observamos que la demostración se volvió trivial a diferencia si utilizamos la definición de `leq`, que tendría que ser por inducción sobre n.

Se puede demostrar que `leq` y la definición recursiva son equivalentes, es decir:

Teorema 2. *Para cualesquiera* $n, m \in \mathbb{N}$ *se cumple que* `leq` $n\ m =$ `True` *syss* $n \leq m$.

Demostración.
La ida se demuestra con inducción sobre la estructura de números naturales.
El regreso se demuestra con inducción sobre la estructura de la relación $\leq$. □

El archivo `IntroCoq.v` contiene la relación previamente descrita. Haciendo un análisis similar define dos relaciones:

1. Que determine si un número es par, nombra a tal relación `par`.

2. Que determine si un número es impar, nombra a tal relación `impar`.

En tu `README` brinda el análisis que hiciste para llegar a las definiciones que brindaste.
Para probar tu solución hay algunos resultados dados en el archivo `IntroCoq.v` que utilizan las definiciones anteriores.

2.2. Programación funcional vs lógica

1. Brinda la versión funcional de las relaciones anteriores, es decir:

   - `npar n` es válida syss `nparF n = true`.
   - `nimpar n` es válida syss `nimparF n = true`.

   Donde `nparF` es la función que determina si un número es par y `nimparF` es la función que determina si un número es impar.
   Observación 1: En `Coq` la función módulo se manda a llamar con la siguiente expresión `Nat.modulo`.
   Por ejemplo `Nat.modulo 4 2` devuelve 0.
   Observación 2: En `Coq` la igualdad que devuelve valores booleanos se manda a llamar con el siguiente operador `=?`.
   Por ejemplo `4 =? 2` devuelve false.

2. Tanto en `Haskell` como en `Prolog` se definió un algoritmo de búsqueda en listas (en `Haskell` fue `elem` y en `Prolog` fue `member`). Brinda ambas definiciones en `Coq` para el caso de listas de números naturales, es decir, define una función cuya firma es
   `elemF : nat ->lista nat ->bool` y define una relación cuyo nombre sea `elem` la cual sea cierta syss un número natural pertenece a una lista de números naturales.

3. Finalmente en tu `README` indica de manera informal y con tus propias palabras los pros y contras de cada implementación (funcional vs lógica).

3. Punto extra

Para obtener un punto extra en la práctica demuestra el teorema 2 descrito en la subsección de relaciones sobre números naturales, tal demostración deberá ser dada en un archivo `pdf` generado por LaTeX. En caso de que tu `README` sea generado por LaTeXpodrás anexar la demostración a este mismo.

4. Reglas

- No se podrán importar mas bibliotecas de las dadas en el archivo.
- A excepción de los ejercicios de la sección de programación lógica, toda función deberá tener al menos tres ejemplos utilizando la expresión `Eval compute in`.

El software es como las catedrales, primero lo construimos y luego rezamos. - Anónimo.