Examen Parcial, MD2, Dic. 2001 [1]

1. Deduce el tipo de la expresión:

```
[x:Prop; H:(~(x\/~x))]
 (False_ind False
   (H
     (or_intror x x->False
       [H7:x](False_ind False (H (or_introl x x->False H7))))))
```

sabiendo que:

```
False_ind : (P:Prop)False->P

Inductive or [A:Prop; B:Prop] : Prop :=
      or_introl : A->(or A B) | or_intror : B->(or A B)
```

Resultado *El resultado es:* `(x:Prop)~(x\/~x)->False`

2. Definimos el condicional en coq:

```
Definition si := [X:Set][b:bool][x,y:X] Cases b of true => x | _ => y end.
```

Teniendo en cuenta esta definición, explica el siguiente comportamiento de Coq en comparación con el de Caml:

```
Coq < Fixpoint fact [n:nat]:nat:=
Coq < (si nat (egal_nat n O) (S O) (mult n (fact (pred n)))).
Error:
Recursive call applied to an illegal term
The recursive definition fact :=
[n:nat](si nat (egal_nat n O) (S O) (mult n (fact (pred n))))
is not well-formed
```

en caml

```
#let si b x y = if b then x else y;;
si : bool -> 'a -> 'a -> 'a = <fun>
#let rec bad_fac n = si (n=0) 1 (n*(bad_fac (n-1)));;
bad_fac : int -> int = <fun>
#bad_fac 0;;
Uncaught exception: Out_of_memory
#bad_fac 1;;
Uncaught exception: Out_of_memory
```

en cambio

[1] Calificación: 1:(2pt);2:(2.5pt);3:(3.5pt);4:(2pt)

```
#let rec fact n = if (n=0) then 1 else (n*(fact (n-1)));;
fact : int -> int = <fun>
#fact 0;;
- : int = 1
#fact 23;;
- : int = 862453760
```

Respuesta (breve indicación)

La evaluación de la función `fact` *que intentamos definir en Coq mediante una función recursiva sería por valor (eager). Afortunadamente el mecanismo de Coq no admite una función recursiva cuya terminación no está garantizada mediante el análisis del código. Este análisis trata de comprobar que a las llamadas recursivas de la función que se trata de definir se le pasan valores estructuralmente menores que el argumento principal.*

La diferencia con Caml (cuya evaluación es también por valor) es que éste admite definiciones como `bad_fact` *que fallan en ejecución porque hemos utilizado la función* `si` *definida por nosotros y, como tal función, será evaluada por valor, es decir, primero todos los argumentos. Ello lleva a evaluar la llamada recursiva lo cual produce un bucle. Ello a pesar de que el compilador le asigna un tipo y la admite, cosa que no sucede en Coq.*

Si, por el contrario,si en caml utilizamos el `if..then..else` *que claramente no está implementada como una función y tiene por tanto una evaluación diferente, entonces funciona bien.*

3. Consideremos el tipo:

```
Coq < Inductive List : Set:=
Coq < Nil : List
Coq < |Cons:(A:Set)A->List->List.
```

Describe los elementos de List.

Sabiendo que:

```
List_rec : (P:(List->Set))
        (P Nil)
        ->((A:Set; a:A; l:List)(P l)->(P (Cons A a l)))
        ->(l:List)(P l)
```

definir, utilizando `List_rec`, una función que satisfaga la siguiente especificación:

```
Inductive spec:List->List->Prop:=
   spec_intro1:(spec Nil Nil)
  |spec_intro2:(A:Set;a:A;l:List)(spec (Cons A a l) l).
```

Respuesta

Los elementos del tipo `List` *serán listas heterogeneas, como por ejemplo:*

```
(Cons nat O (Cons bool true Nil))
```

que representaría la lista formada por un natural y un booleano.

Nótese que se trata de un tipo recursivo no paramétrico. Es, por lo tanto, un tipo distinto del de las listas homogéneas que tiene un parámetro `A:Set` *que obliga a que todos los elementos de la lista tengan ese mismo tipo.*

Para que una función satisfaga esta especificación está claro que en `Nil` *el resultado ha de ser* `Nil` *y el resultado en* `(Cons A a l)` *ha de ser* `l`*. Denotaremos la función que buscamos como* `cdr` *por lo que su tipo ha de ser el que nos provee* `List_rec` *es decir* `(l:List)(P l)`*.*

En primer lugar, al tratarse `cdr` *de una función de* `List -> List` *es claro que el parámetro* `P:List -> Set` *será* `[_:List]List`*, y, por la especificación, dado que* `(P Nil)=List` *debemos tomar aquí* `Nil`*.*

Además, el tercer parámetro, deberá ser del tipo:

```
(A:Set; a:A; l:List)List->List
```

o lo que es lo mismo:

```
(A:Set)A->List->List->List
```

de modo que, el tercer parámetro ha de ser:

```
[A:Set][a:A][l:List][l':List]l''
```

de forma que si `l'` *es el resultado de* `(cdr l)` *(pues este es el elemento que tomamos en* `(P l)`*), entonces* `l''` *debe ser el resultado de* `(cdr (Cons A a l))` *es decir* `l`*. De modo que* `l''` *debe coincidir, en este caso, con* `l`*.*

Por todo ello la función puede programarse como:

```
Definition cdr:=(List_rec [_:List]List Nil [A:Set;_:A;l:List;_:List]l).
```

Veamos ahora que, efectivamente, `cdr` *cumple la especificación. Esto podemos hacerlo de dos formas. Una viendo directamente que se cumplen:*

```
(cdr Nil)=Nil

(cdr (Cons X x li))=li.
```

para lo cual no hay más que tener en cuenta como funciona `List_rec` *en general:*

```
(List_rec P p F Nil)=p.

(List_rec P p F (Cons X x li))=(F X x li (List_rec P p F li))
```

siendo, claro está, `P:List->Set`*,* `p:(P Nil)` *y* `F:(X:Set;x:A; li:List)(P li)->(P (Cons X x li))`*. Por lo tanto,*

```
(cdr Nil)=Nil
(cdr (Cons X x li))=([A:Set;_:A;l:List;_:List]l X x li
                       (List_rec [_:List]List Nil
                            [A:Set;_:A;l:List;_:List]l li))
                  =([_:List]li (List_rec [_:List]List Nil
                                  [A:Set;_:A;l:List;_:List]l li))
                  = li
```

Otra forma sería emular la prueba en Coq:

```
Theorem test:  (l:List)(spec l (cdr l)).

Destruct l.
test < Destruct l.
2 subgoals

  l : List
  ============================
   (spec Nil (cdr Nil))

subgoal 2 is:
 (A:Set; a:A; l0:List)(spec (Cons A a l0) (cdr (Cons A a l0)))

test < Simpl.
2 subgoals

  l : List
  ============================
   (spec Nil Nil)

subgoal 2 is:
 (A:Set; a:A; l0:List)(spec (Cons A a l0) (cdr (Cons A a l0)))
test < Constructor 1.
1 subgoal

  l : List
  ============================
   (A:Set; a:A; l0:List)(spec (Cons A a l0) (cdr (Cons A a l0)))
test < Simpl.                                  <---------------------
1 subgoal                        Notese que para justificar este /
                                 paso hay que hacer el cálculo  /
  l : List                       anterior también              /
  ============================                                /
   (A:Set; a:A; l0:List)(spec (Cons A a l0) l0)              /
                                              <--------------/
Constructor 2.
Subtree proved!
```

La máxima calificación se alcanzará sólamente si se hace de la primera forma, o bien de la segunda pero en ese caso también hay que incluir la primera para justificar el último `simpl`.

4. Explica el significado de las siguientes reglas de tipado y cómo, a partir de ellas se pueden obtener tácticas de prueba:

$$\frac{E[\Gamma] \vdash (x:T)U : s \quad E[\Gamma :: (x:T)] \vdash f : U}{E[\Gamma] \vdash [x:T]f : (x:T)U}\, Lam$$

$$\frac{E[\Gamma] \vdash f : (x:T)U \quad E[\Gamma] \vdash t : T}{E[\Gamma] \vdash (f\ t) : U\{x/t\}}\, App$$

Respuesta