Final Febrero 2025 (segunda semana)

1. Demostrar isomorfismo de tipos

QvQ : a->(b,c) $\simeq$ (a->b, a->c)

Para esto basta con encontrar funciones f, g tipadas de la forma

```
f :: (a->(b,c)) -> (a->b, a->c)
g :: (a->b, a->c) -> (a->(b,c))
```

y ver que $f.g = id$ y $g.f = id$

Sean
$f(h) = (\lambda x.\pi_1(h(x)), \lambda x.\pi_2(h(x)))$
$g(p, q) = (\lambda x.(p(x), q(x)))$

Con su 'equivalente' en haskell

```
f h = (\x. fst(h x), \x. snd(h x))
g (p, q) = (\x. (p(x), q(x)))
```

Empiezo viendo $f.g = id$

```
-- Veo que la composición tipa:
f . g :: (a->b, a->c) -> (a->b, a->c)

-- Por extensionalidad de funciones, vemos para (p,q), p :: a->b, q :: a->c
f . g (p, q) = id (p, q) :: (a->b, a->c)
-- Aplico id
(f . g) (p, q) = (p, q)
-- Aplico .
= f (g (p, q)) = (p, q)
-- Aplico g
= f (\x. (p(x), q(x))) = (p, q)
-- Aplico f
= (\x. fst(\y. (p(y), q(y))), \x. snd(\z. (p(z), q(z)))) = (p, q)
-- Extensionalidad de funciones para x :: a
= (\x. fst(\y. (p(y), q(y))), \x. snd(\z. (p(z), q(z)))) x = (p, q) x
= (fst(\y. (p(y), q(y))) x, snd(\z. (p(z), q(z))) x) = (p, q) x
= (fst(p(x), q(x))), snd((p(x), q(x))) = (p, q) x
= (p(x), q(x)) = (p(x), q(x))
```

Ahora veo $g.f = id$

```
-- Veo que la composición tipa
g . f :: (a->(b,c)) -> (a->(b,c))

-- Por extensionalidad de funciones, vemos para h :: (a->(b,c))
(g . f) h = id h :: a->(b,c)
= g (f h) = id h
-- Aplico f
= g ((\x. fst(h x), \x. snd(h x))) = id h
-- Aplico g
= \x. ((\y. fst(h y)) x, (\z. snd(h z)) x)
-- Reduzco
= \x. ((fst(h x)), (snd(h x)))
-- Por extensionalidad de funciones, vemos para x :: a
= \x. ((fst(h x)), (snd(h x))) x = id h x :: (b, c)
-- Por extensionalidad de pares sea h x = (m, n) :: (b, c)
= (fst((m, n)), snd((m,n))) = id (m, n) :: (b, c)
= (m, n) = (m, n)
```

2. Resolución (a validar):

Dado un programa con los siguientes predicados unarios:

1. $c(x)$
2. $p(x)$
3. h
4. $p(h) \iff \exists X.(c(x) \wedge \neg p(X))$

Demostrar con resolución que $\sigma = c(h) \to p(h)$

> En cuaderno

3. Programación Lógica.

```
% Utilizo el esquema Generate & Test.

% listasGemelas(-L1, -L2):
% Vale cuando el concatenarlas, el resultado es capicúa
listasGemelas([], []).
listasGemelas(A, B) :-
    desde(0, N), % genero la longitud de las listas capicuas
    generarListaTam(N, A), % genero una lista de tamaño N
    generarListaTam(N, B), % genero otra lista de tamaño N
    sonEspejo(A, B).       % testeo la condicion

% generarListaTam(+N, -A)
generarListaTam(0, []).
generarListaTam(1, [X]) :- member(X, [0,1]).
generarListaTam(N, [X|XS]) :-
    N > 1,
    member(X, [0,1]),
    M is N-1,
    generarListaTam(M, XS).

% sonEspejo(+X, ?Y)
sonEspejo([], []).
sonEspejo([X], [X]).
sonEspejo([X|XS], [Y|YS]) :-
    tail(YS, X),
    tail(XS, Y),
    sacarCola([Y|YS], L),
    sonEspejo(XS, L).

% desde(+X, -Y)
desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).

% tail(+L, -X)
tail([X], X).
tail([X|XS], Y) :- tail(XS, Y).

% sacarCola(+L, -L)
sacarCola([X], []).
sacarCola([X|XS], [X|YS]) :- sacarCola(XS, YS).
```

4 Algoritmo de inferencia de tipos

Suponiendo que $\vdash M : x_1 \to x_2$ es un juicio tipado valido y el mas general de U ($erase(M) = U$).
Me pide hallar el juicio de tipado más general para UU y justificar por qué es el más general.

Vale que
$\vdash U_1 : x_1 \to x_2$
$\vdash U_2 : x_1 \to x_2$

Y la regla del algoritmo es que
$W(U_1U_2) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \vdash S(U_1U_2) : S(?_k)$

Con S el unificador más general,
$S = MGU\{x_1^{(1)} \to x_2^{(1)} = x_1^{(2)} \to x_2^{(2)} \to ?_k\}$

Luego esto no unifica por colisión, entonces no hay forma de tipar UU con tipos simples.

5. Programación funcional y esquemas de recursión.

```haskell
data Form = Prop String | And Form Form
type Ctx = [Form]
data Demo = Ax Ctx Form | AndI Demo Demo | AndE1 Demo | AndE2 Demo  -- modela las demos

foldDemo :: (Ctx -> Form -> a) -> (a -> a -> a) -> (a -> a) -> (a -> a) -> Demo -> a
foldDemo cAx cAndI cAndE1 cAndE2 d = case d of
    Ax c f -> cAx c f
    AndI d1 d2 -> cAndI (rec d1) (rec d2)
    AndE1 dd -> cAndE1 (rec dd)
    AndE2 dd -> cAndE2 (rec dd)
    where rec = foldDemo cAx cAndI cAndE1 cAndE2

eval :: Demo -> Maybe (Ctx, Form)
eval = foldDemo
    (\ctx form -> Just(ctx, form))          -- cAxioma
    (\r1 r2 ->                              -- cAndI
        case (r1, r2) of
            (Just(ctx1, form1), Just(ctx2, form2)) -> Just(ctx1 ++ ctx2, And form1 form2
            _ -> Nothing
    )
    (\r -> case r of                        --cAndE1
            Just(ctx, And p q) -> Just(ctx, p)
            _ -> Nothing
    )
    (\r -> case r of                        --cAndE2
            Just(ctx, And p q) -> Just(ctx, q)
            _ -> Nothing
    )
```