Inferencia/Tipado

1) Inferencia/MGU: sutil diferencia con uno tomado en otro final pero que cambia mucho :
Sea S un MGU de (C U { sigma1 =.= sigma2})
Verdadero o falso :
a) S es un unificador de C
b) S es un MGU de C

Sea C' = (C U { sigma1 =.= sigma2})
a) Verdadero. Dado que S es MGU de C', es por lo tanto unificador de C'. Como C es un subconjunto de cláusulas de C', la sustitución S unifica sus ecuaciones, dado que ya de por sí las unificaba junto a otras.

b) Falso. Dado C = {Bool -> t = r}, si tomamos sigma1=t, sigma2=Nat, el MGU de C' es S={Nat\t, Bool->Nat\r} y sería menos general que el unificador S'={Bool->t\r}. Por lo tanto, aunque S unifica a C, no es el unificador más general, dado que S' también unifica a C y S es instancia de él.

2) Verdadero o Falso. Justificar.
A) S es MGU de G1, y G1 contenido en G2, luego S es MGU de G2
B)S es MGU de G1, y G2 contenido en G1, luego S es MGU de G2
C)S es MGU de G1 y tambien de G2, luego S es MGU de G1 interseccion G2
D) S es MGU de G1 y tambien de G2, luego S es MGU de G1 union G2

a) Falso. Dado que G1 contenido en G2, puede haber ecuaciones en G2 que no estén en G1. Mismo podría pasar que haya clausulas más restrictivas. Por ej,
G1 = {r → s = t → Nat}
G2 = {r → s = t → Nat, r = Bool}
En este caso, el conjunto de goals G1 esta contenido en G2. En ese caso S siendo MGU de G1 no sería instancia de tipado de G2. S = {t\r, Nat\s}, que es MGU de G1, pero que no es solución de G2, dado que no unifica con la ecuación r = Bool.

b) Falso. Dado un S que es MGU de G1, como G2 esta contenido en G1, puede tener ecuaciones de menos. Esto significa que el MGU que se le da a G1 puede ser más específico que el MGU de G1. Por ej
G2 = {r → s = t → Nat}
G1 = {r → s = t → Nat, r = Bool}
En este caso S sería {Bool/r, Bool/t, Nat/s} y si bien sería unificador de G2, no sería el unificador más general posible. El MGU de G1 es instancia del MGU de G2 (es más chico).

c) Falso. Dado

S es {r/t, s/Nat, q/Bool} pero la intersección es {q = Bool}, con lo cual se puede encontrar un unificador mas general que S, donde S sea instancia de ella.

d) Verdadero. Pendiente demostrar.

3) Agarra la regla de occur check y la cambia para que en vez de fallar, no hace nada y elimina la ecuación
a. Encontrar un problema de unificación donde dependiendo del orden de elección, falle o no falle
b. Idem pero dependiendo del orden de sustituciones distintas
c. Encontrar un conjunto de cláusulas satisfactible desde el cual se pueda llegar a la cláusula vacía con esta modificación del algoritmo de MGU

a) {r -> t =.= Bool -> t, Bool =.= t, r = r -> s}
1 -> {r =.= Bool, t =.= t, Bool =.= t, r =.= r->s}
2 -> { r =.= Bool, Bool =.= t, r =.= r->s}
4 -> {Bool =.= t, Bool =.= Bool -> s}
{Bool/r}
5 -> FALLA (porque elegí primero unificar Bool/r que usar occur check)

{r -> t =.= Bool -> t, Bool =.= t, r = r -> s}
1 -> {r =.= Bool, t =.= t, Bool =.= t, r =.= r->s}
2 -> { r =.= Bool, Bool =.= t, r =.= r->s}
6 -> { r =.= Bool, Bool =.= t }
4 -> {Bool =.= t}
{Bool/r}
3 -> {t =.= Bool}
4 -> {}
{Bool/t}

b) {t -> Nat =.= s, t =.= r, r =.= s, s =.= Bool}
4 -> {r -> Nat =.= s, r =.= s, s =.= Bool}
{t/r}
4 -> {s -> Nat =.= s, s =.= Bool}
{r/s}
6 -> {s =.= Bool}
4 -> {}
{Bool/s}

{t -> Nat =.= s, t =.= r, r =.= s, s =.= Bool}
4 -> {t -> Nat =.= Bool, t =.= r, r =.= Bool}
{Bool/s}
4 -> {r -> Nat =.= Bool, r =.= Bool}
{r/t}
4 -> {Bool -> Nat =.= Bool}
5 -> FALLA

c) C1 = {P(x1), P(S(S(x1)))}

C2 = {!P(x2)}
C1 con c1 = {x1 = x2}
C3 = {P(S(S(x3)))}
C2 con c2 = {x3 = S(S(x3)} (funciona dado que martelli montanari me dice que puedo sustituirlo)
C4 = {}

4) bool <: bool -> bool y viceversa
a. Dar el juicio de tipado de true true. Explicar qué propiedad se perdió
b. tipar (\x. x x) (\x. x x) y dar un juicio de tipado. Explicar qué propiedad se perdió.

a) Se perdió la propiedad del determinismo semántico: ya la semántica no alcanza para decidir de qué tipo es una expresión, dado que existen infinitos juicios de tipado que llevan a distintos resultados

b) Dadas las extensiones que se usan en este contexto, podíamos usar la propiedad de que todo término bien tipado y cerrado terminaba. Se perdió esa propiedad, dado que este término tipa pero al reducirlo vemos que se queda ciclando infinitamente sobre el mismo término

5) sean S1 y S2 sustituciones de un goal de unificación G, definimos que S1≤S2 significa que S1 es más general que S2.
Decidir y justificar falso o verdadero:
a) o bien S1≤S2 o bien S2≤S1
b) si S1≤S2 y S2≤S1 entonces S1=S2
c) siempre existe T tal que T≤S1 y T≤S2, sin importar el G.

a) Si no aplica la igualdad… Falso. Dada
s instancias de S1 y S2 con variables renombradas, no sería cierto que S1≤S2 ni tampoco que S2≤S1. Ej. S1 = {Nat\r, r\s}, S2={Nat\s, s\r}, G = {Nat =.=s, r=.=s}.
Si aplica la igualdad… Falso. G = {r =.=s}. S1 = {Nat\r, Nat\s}. S2={Bool\r, Bool\s}. Ambas son sustituciones pero ninguna es más general que la otra.

b) Si no aplica la igualdad.. Falso porque no tiene sentido que pase a la vez
Si aplica la igualdad… Verdadero. Si S1 es más general que S2, es porque no es más restrictiva con su definición de tipos que S2. Y viceversa. Entonces, dado que son igualmente generales, la única diferencia que pueden tener es renombre de variables. Por lo tanto, S1=S2, donde = significa “son igualmente generales. Si la igualdad no considera renombre de variables, entonces es falso (porque ambas son más generales que la otra, pero no son literalmente iguales)

c) Verdadero. Esto es así porque ambas son sustituciones de G, y o bien S1 o S2 son el MGU de G (con lo cual T debería ser S1 o S2) o bien existe una sustitución T distinta a S1 y S2 tal que es la más general posible sobre G, siendo su MGU.

6) También, ejercicio similar ya tomado

Cambian la regla E-PredSucc por una qué hace:
pred(succ(N)) -> N
Decir si se pierden alguna, ni una o ambas propiedades:
I) Preservabilidad
II) Determinismo

I) La preservabilidad se mantiene, dado que los tipos no se ven afectados ni las reglas de tipado relacionadas a ellos. Es decir: será necesario que la expresión tipe correctamente para aplicar la regla, y la regla aplicada no afectará los tipos, sino que los preservará

II) Se pierde el determinismo semántico. No es necesario que N esté en forma normal, como si se pedía en la regla de evaluación anteriormente definida. Entonces, dada la expresión pred(succ(pred(succ(0)) y la regla de evaluación E-Pred y E-PredSucc nueva, podrían aplicarse los pasos de evaluación de distintas maneras. Así, la evaluación no está más dirigida por sintaxis.

7) Dado let f: Ref (Nat -> { l: Nat }) = ref (\x:Nat. {}) in 0
determinar si typechequea asumiendo que la regla de Ref es contravariante respecto del tipo

Lo hice en una hoja aparte pero typechequea.

8) Supongamos que, en cálculo lambda extendido con naturales, tenemos un algoritmo de inferencia de tipos que consiste en:

1. Reducir el término hasta una forma normal.
2. Si el resultado es un número cualquiera (0, 1, 2, ... representado con succ/pred), entonces el tipo es Nat. Si no, no.

¿Funcionaría bien este algoritmo? ¿Por qué o por qué no? Dar ejemplos. Asumir que no hay subtipado.

El algoritmo no funcionaría bien en general. Sí funcionaría para términos cerrados y bien tipados, porque en cálculo lambda todo término cerrado y bien tipado reduce a un valor en una cantidad finita de pasos (por ejemplo **(\x.succ(x)) 0**). Sin embargo, para términos bien tipados pero con variables libres, el algoritmo no va a funcionar. Tampoco lo va a hacer para términos que estén cerrados pero no bien tipados.

Ejemplos:

- **if true then x else 2** tipa a Nat bajo el contexto {x:Nat}, pero el algoritmo lo reducirá a **x** y no podrá determinar su tipo porque no es un número.
- **if true then 2 else false** debería hacer fallar al algoritmo, pero sí va a tipar porque simplemente reducirá a 2, en vez de causar un error por tener un término de tipo Bool en una rama del if y un término de tipo Nat en la otra.

9) Supongamos que extendemos el cálculo lambda con un operador de igualdad == que toma dos parámetros y se escribe de forma prefija, e.g. == 1 2.
a) Dar el tipo de este operador y extender el algoritmo de inferencia para tipar términos que lo usen.
b) Supongamos que queremos restringir el operador para que sólo pueda tomar elementos que no sean de tipo función. ¿Cómo debería modificarse el algoritmo de inferencia y/o el de Martelli-Montanari?

Punto (a): el tipo del operador es σ -> σ -> Bool.

Extendemos primero el conjunto de términos como M ::= … | ==. Observemos que no hace falta incluir los operandos, ya que == simplemente se comporta como una función lambda de dos parámetros.

Si el único término nuevo que tenemos es ==, sólo hace falta extender el algoritmo de inferencia para este caso:

$$\mathbb{W}(==) \triangleq \varnothing \square == : t_1 \to t_1 \to \text{Bool}$$

con t_1 variable fresca

La razón por la que no hace falta unificar los operandos es que el mismo algoritmo de inferencia va a tratar == M N como dos aplicaciones de una función currificada y se va a encargar de unificar los tipos de M y N a través de t_1.

Punto (b): la opción más ingenua es que, al encontrar un ==, el algoritmo de inferencia intente unificar la variable nueva con cada posible tipo no compuesto hasta encontrar uno que no cause una falla. Por ejemplo, primero intentar unificar el tipo de == con Bool -> Bool -> Bool, después con Nat -> Nat -> Bool, después con Int -> Int -> Bool, etc. Esta solución sería correcta, pero tendría la desventaja de que, si hay más instancias de == en las subexpresiones, tendría que hacer backtracking para cada una, y su complejidad sería exponencial.

Una opción un poco más complicada de especificar, pero que no tendría este problema de rendimiento, es introducir una nueva variante del algoritmo de Martelli-Montanari que *sólo* se ejecute cuando el algoritmo de inferencia se encuentra con un ==. Es decir, si está infiriendo el tipo de cualquier término de los que ya existían, usa la versión ya conocida de Martelli-Montanari; pero, si está infiriendo el tipo de un ==, usa esta variante en la que se introduce una regla particular para la nueva variable t_1. La regla es similar a la de colisión (o clash) y consiste simplemente en fallar si hay algún par a unificar $t_1 \stackrel{?}{=} \sigma \to \tau$ o $\sigma \to \tau \stackrel{?}{=} t_1$.

Haskell

1) La estructura del 3 era algo así si mal no recuerdo
type Var = String
type Lab = String
data Expr = EVar Var | EObj [(Lab, Var, Expr)] | ESel Expr Lab | EUpd Expr Lab Var Expr
a. Definir fold
b. El 3 b creo que era justificar por qué no se puede implementar una función eval en haskell que tome esa estructura y devuelva otra del tipo objeto, asumiendo que eval funciona con bigstep como cálculo sigma

```
type Var = String
type Lab = String
data Expr = EVar Var | EObj [(Lab, Var, Expr)] | ESel Expr Lab | EUpd Expr Lab Var Expr

foldExpr :: (Var -> b) -> (Lab -> Var -> b -> b) -> (Lab -> b -> b) -> (Lab -> Var -> b -> b -> b) -> Expr -> b

foldExpr fvar _ _ _ (EVar vvar) = fvar vvar

foldExpr fvar fobj fsel fupd (EObj [(vlab, vvar, vexpr)]) = fobj vlab vvar (foldExpr fvar fobj fsel fupd vexpr)

foldExpr fvar fobj fsel fupd (ESel vexpr vlab) = fsel vlab (foldExpr fvar fobj fsel fupd vexpr)

foldExpr fvar fobj fsel fupd (EUpd vexpr1 vlab vvar vexpr2) = fupd vlab vvar (foldExpr fvar fobj fsel fupd vexpr1) ...
```

2) Dar la lista de todos los predicados sobre naturales donde la preimagen de True sea finita

```
listasQueSuman :: Int -> [[Int]]
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <- listasQueSuman (n-x)]

listasFinitas :: [[Int]]
listasFinitas = [xs | n <- [1..], xs <- listasQueSuman n]

preds :: [[Int -> Bool]]
preds = [(\x -> elem x l) | l <- listasFinitas]
```

3) Haskell: Ejercicio idéntico a otro final.
Te dice que hay un tipo de datos recursivo: D a
Tal que foldD::(a - >b - >) -> D a -> b
También te dice que tenes un elemento unD:: D Int
Y te da: foldD (+) unD
Decir si es verdadero o falso que: la expresión tipa pero independientemente de como se defina unD, nunca termina.

Verdadero. Dado que es una estructura recursiva, y como la función (+) necesita si o si de resolver ambos lados de la ecuación, seguirá infinitamente sumando. Distinto sería el caso donde la función pudiera resolverse en algún paso recursivo sin necesidad de continuar resolviendo el siguiente paso del fold.

4) En haskell no podemos operar modificando variables como en imperativo. Podemos simularlo tomando un entero adicional que representa el valor inicial de la variable y devolver adicionalmente un entero que representa el valor resultante de la variable luego de la ejecución.
Definimos en haskell el siguiente sinónimo type FunX a b = (a -> Int -> (b, Int))
a- Definir idX :: FunX a a Que representa el programa que no modifica la variable. Definir incX :: FunX a a Que incrementa el valor de la variable en 1.
b- Definir componer :: (FunX b c) -> (FunX a b) -> (FunX a c) Que compone 2 programas. EJ: CompX IncX IncX "hola" que si le pasa 5 como argumento, devuelve ("hola",7)
c- Usando árboles binarios, data AB a = Nil | Bin (AB a) a (AB a). Definir mapX :: (FunX a b) -> (FunX (AB a) (AB b)) Que funciona como el map en árboles binarios.

a)
```
-- Sinónimo de tipo
type FunX a b = a -> Int -> (b, Int)

-- `idX` no modifica la variable, simplemente devuelve el mismo valor de entrada y deja la variable sin cambios
idX :: FunX a a
idX x n = (x, n)

-- `incX` incrementa el valor de la variable en 1, pero no modifica el valor de entrada
incX :: FunX a a
incX x n = (x, n + 1)
```

b)
```
componer :: FunX b c -> FunX a b -> FunX a c
componer f g x n = f y n'
  where (y, n') = g x n
```

c)

```
-- `mapX` aplica una función `FunX` a todos los elementos de un árbol binario
mapX :: FunX a b -> FunX (AB a) (AB b)
mapX _ Nil n = (Nil, n)
mapX f (Bin izq x der) n = (Bin izq' x' der', n''')
  where
    (x', n') = f x n              -- Aplicar `f` al valor del nodo actual
    (izq', n'') = mapX f izq n'    -- Aplicar `mapX` al subárbol izquierdo
    (der', n''') = mapX f der n''      -- Aplicar `mapX` al subárbol derecho
```

5) definiendo fix::(a→a)→a como fix f=f(fix f),

escribir foldL sin recursión explicita, usando fix

```
fix :: (a -> a) -> a
fix f = f (fix f)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f = fix (\fixfunc z l -> if null l then z else (fixfunc (f z (head l)) (tail l)))

foldl2 :: (b -> a -> b) -> b -> [a] -> b
foldl2 = fix (\fl f z l -> if null l then z else fl f (f z (head l)) (tail l))
```

6) Define el tipo de dato Nat como Zero | Succ(Nat)
Dada una función F
F :: [Nat] -> Nat .
Que te aseguran que es total y biyectiva.
Si es posible, implementar en Haskell F^-1. Si no es posible justificar porque.

Dado que F es total, su dominio son todas las listas de naturales.
Una función es biyectiva si es al mismo tiempo inyectiva y sobreyectiva.
Es decir, si todos los elementos del conjunto de salida tienen una imagen distinta en el conjunto de llegada, y a cada elemento del conjunto de llegada le corresponde un elemento del conjunto de salida.

Entonces, la función F^-1 es basicamente dado un natural X, devolver la lista L tal que F(L) = X
Entonces, si nos dan un valor X, como tenemos F y sabemos que es total y biyectiva, podemos ir probando con todas las listas posibles de naturales. Dado que es biyectiva, existe un elemento en el dominio que dará como resultado ese X, y es único. Y dado que F es total, para todas las listas existe un resultado válido.
Por lo tanto, podemos ir probando con todas las listas hasta llegar a una que nos de el resultado

Sea g = F^-1

```
listasQueSuman :: Int -> [[Int]]
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <- listasQueSuman (n-x)]

listasFinitas :: [[Int]]
listasFinitas = [xs | n <- [1..], xs <- listasQueSuman n]

g :: Int -> [Int]
g x = foldr (\l rec -> if (f l) == x then l else rec) False listasFinitas
```

7) a) Dar el tipo de una función en Haskell análoga a foldr, pero que tome dos listas y una función que procese las dos. Puede asumirse que las listas tienen la misma longitud.

b) Implementar la función. Puede usarse recursión explícita.
c) Dar el tipo de zipWith. Usar la función para implementar zipWith sin recursión explícita.

Punto (a): primero observemos que la función a aplicar recursivamente, en vez de tomar un único parámetro de una lista como lo hace foldr, toma un parámetro de cada lista por cada paso de la recursión. Además, toma el resultado del paso recursivo previo. Entonces, si una lista tiene elementos de tipo a y la otra tiene elementos de tipo b, y el resultado de la recursión es de tipo c, el tipo de la función debería ser (a -> b -> c -> c). El caso base es sencillo, porque podemos asumir que, cuando una de las listas está vacía, la otra también lo está (si no asumiéramos esto, tendríamos que usar aplicación parcial y cambiar el comportamiento al llegar al final de la primera lista en función de si la segunda es o no vacía). Por lo tanto, alcanza con un elemento de tipo c. Finalmente, vienen las dos listas y el resultado.

```
foldrDoble :: (a -> b -> c -> c) -> c -> [a] -> [b] -> c
```

Punto (b): muy similar al foldr que ya conocemos, pero con un parámetro más.
Recordemos que foldr aplica la función sobre la cabeza y el resultado de la recursión, en vez de aplicar recursión sobre la llamada como foldl.

```
foldrDoble _ z [] [] = z
foldrDoble f z (x:xs) (y:ys) = f x y : (foldrDoble f z xs ys)
```

Punto (c):

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f = foldrDoble (\x y rec -> (f x y):rec) []
```

Prolog/Resolución

1. Árboles binarios como los de la práctica y otros finales. Tengo:
arbol(nil).
arbol(bin(I, D)):-arbol(I), arbol(D)
esTotalmenteInterior(bin(bin(A,B), bin(C,D))).
a. explicar que devuelve la consulta ?- arbol(X), esTotalmenteInterior(X)
b. mostrar una resolución SLD que pruebe la existencia de algún X que cumple

a) Prolog va a resolver primero la cláusula arbol(X), y luego con esa unificación va a seguir resolviendo esTotalmenteInterior(X). Sin embargo, aunque existen árboles que unifican X en la consulta, por como funciona el motor de Prolog esta consulta nunca va a terminar.
Primero, arbol(X) entrara en la C1, arbol(nil). {X = nil}
Luego, esTotalmenteInterior(nil) falla. Entonces el motor vuelve hasta arbol(X) y unifica con la siguiente cláusula. arbol(bin(I , D)) :- arbol(I), arbol(D).
Resuelve ambas clausulas con nil porque es la 1ra cláusula que satisface y llega a esTotalmenteInterior(bin(nil, nil)), que falla.
Nuevamente el motor vuelve para atras, hasta el ultimo paso pre fallo: arbol(D). Y lo abre con la 2da clausula, usando arbol(bin(I, D)). Pero el arbol izquierdo inicial que teniamos ya nos quedó nil, y a medida que el motor siga fallando al alcanzar esTotInt siempre va a optar por este camino, abrir D, y nunca cambiar la rama izquierda. Es decir, nuestro arbol siempre se vera
arbol (bin(nil, bin(nil, bin(...bin(nil, nil)...))).

b) C1 = {arbol(nil)}
C2 = {!arbol(I), !arbol(D), arbol(bin(I, D))}
C3 = {esTotInt(bin(bin(A,B), bin(C,D)))}
C4 = {!arbol(X), !esTotInt(X)}

C2 — C2' {I = bin(I', D')}
C6 = {!arbol(D), !arbol(I'), !arbol(D'), arbol(bin(bin(I', D'), D))}

C6 — C2'' {D = bin(I'', D'')}
C7 = {!arbol(I'), !arbol(D'), !arbol(I''), !arbol(D''), arbol(bin(bin(I', D'), bin(I'', D'')))}

C7 — C1 {I' = nil}
C8 = {!arbol(D'), !arbol(I''), !arbol(D''), arbol(bin(bin(nil, D'), bin(I'', D'')))}

C8 — C1 {D' = nil}
C9 = {!arbol(I''), !arbol(D''), arbol(bin(bin(nil, nil), bin(I'', D'')))}

C9 — C1 {I'' = nil}
C10 = {!arbol(D''), arbol(bin(bin(nil, nil), bin(nil, D'')))}

C10 — C1 {D'' = nil}
C11 = {arbol(bin(bin(nil, nil), bin(nil, nil)))}

C11 — C4 {X = bin(bin(nil, nil), bin(nil, nil))}
C12 = {!esTotInt(bin(bin(nil, nil), bin(nil, nil)))}

C12 — C3 {A=nil, B=nil, C=nil, D=nil}
C13 = {}

2) Definimos una cadena K-equilibrada como una lista de 0s y 1s que contenga todas las listas posibles de k elementos (0s y 1s) entre sus elementos contiguos una sola vez cada una (una k-equilibrada de 2 es 11001).
escribir el programa equilibrada(+K,-L) en prolog que devuelva todas las cadenas K-equilibradas de K.
No hay por qué demostrar que funciona, pero hay que explicar los contratos e invariables de las auxiliares (?).
aca daba el ejemplo de dos respuestas de equilibrada(K,L) con K=3

```
% listas01(+N, -L)
listas01(0, []).
listas01(N, [1 | Xs ]) :- N > 0 , N1 is N-1, listas01(N1, Xs).
listas01(N, [0 | Xs ]) :- N > 0 , N1 is N-1, listas01(N1, Xs).

% occur(+L, +Y, -O)
occur([], _, 0).
occur([X | Xs], Y, N) :- number(N), prefijo([X|Xs], Y), N1 is N-1, occur(Xs, Y, N1).
occur([X | Xs], Y, N) :- number(N), not(prefijo([X|Xs], Y)), occur(Xs, Y, N).

% prefijo(+L, ?P)
prefijo(L, P) :- append(P, _, L).

% noKEquilibrado(+K, +L)
noKEquilibrado(K, L) :- listas01(K, L2), not(occur(L, L2, 1)).

% equilibrada(+K, -L)
equilibrada(K, L) :- desde(K, inf, N), listas01(N, L), not(noKEquilibrado(K, L)).
```

3- Sea P un predicado unario, Z una constante que representa el 0 y S la función unaria que representa el sucesor (números naturales). S^n(Z) representa S(S(…S(Z)) n veces la S.
Dada la siguiente fórmula:
\forall x. P(S(S(x))) => P(x)
Demostrar usando resolución que para cualquier n en los naturales:
P(S^n(Z)) => (P(Z) \lor P(S(Z)))
Ayuda: usar inducción, separando los casos 0 y 1.

4) Te daba dos fórmulas de primer orden y tenias que probar si eran válidas o no.
A) \forAll x \Exist y : x + y = x
B) (\Exist x (P(x))) v (\Exist x (P(~x)))

a) Si tomo M = ({1, 2}, {suma(a, b) = 2})
Para x = 1, no existe y tal que suma(x, y) = 1. Entonces no es válida, dado que existe una estructura en la cual no lo es.

b) (∃x (P(x))) v (∃x (~P(x)))
Demuestro que es válida porque veo que su negación es insatisfactible
Negación
~(∃x (P(x))) and ~(∃x (~P(x)))
(∀x (~P(x))) and (∀x (P(x)))
(∀x (~P(x))) and (∀y (P(y))) renombro

forma normal prenexa

∀x ∀y (~P(x) and P(y))

C1 = {~P(x)}
C2 = {P(y)}

C1 — C2.
{}.

5) Define la descomposición en 1 paso de una lista en Prolog.
Ejemplo: [2,*3*,1] -> [2, * 2, 1,* 1]
O sea, cambia un elemento por dos elementos que sumen el elemento original. Tiene que cumplirse que ambos sean mayores o iguales a uno.
Define la descomposición en N pasos que es lo que uno espera. Ejemplo para N =2. [2,3,1] -> [2, 1,1,1,1].
El ejercicio pedía hacer una función que dada una lista te dé todas las descomposiciones.

```
% descomponer1(+L, -R)
descomponer1([X | Xs], R) :- X > 1, descomponerVal(X, Y), append(Y, Xs, R).
descomponer1([X | Xs], [X | Rs] :- descomponer1(Xs, Rs).

% descomponerVal(+V, -Y)
descomponerVal(V, Y) :- LIM is V-1, desde(1, LIM, I), D is V - I, append([I], [D], Y).

% descomponerK(+L, +K, -R)
descomponerK(L, 1, R) :- descomponer1(L, R)
descomponerK(L, K, R) :- K > 1, descomponer1(L, LL), K1 is K-1, descomponerK(LL, K1, R).
```

6)
p(a);
p(b);

q(c);
q(d);
r(X) :- not(not(X)), not(not(X))
Describir si falla o no, cual es el resultado y qué variables se instancian en caso de que no falle

Veamos primero como es el not en Prolog.

not(G) :- call(G), !, fail.
not(G).

Si la consulta es r(X), va a fallar. Dado que not(X) hace una llamada a call(X), es necesario que, al ser un operador extra-lógico, X esté instanciado. De otra forma, call(X) no puede instanciarse en un valor que resuelva y no puede continuar el not.
En cambio, si se hace una llamada a r(q(X)), o r(p(X)), call(p(X)) o call(q(X)) si encuentra una instanciación que satisface la consulta. Supongamos que toma el valor X=b.
La consulta de not(G) encuentra un G que matchea, entonces hace el cut y fail. Es decir, devuelve un false.

Ahora bien, ese not esta metido adentro de otro not. Entonces, lo que va a ocurrir es que primero se va a hacer un call(not(p(X)). Dado que, como vimos, el not(p(X)) resuelve a false por el cut, not(not(X)) va a resolver a true.

Luego, se repite la misma cláusula not(not(X)). Sin embargo, el not no instancia las variables libres de G. Por lo tanto, se repite el procesamiento que explicamos arriba desde cero, volviendo a resolver a true finalmente. Por lo tanto r(p(X)) o r(q(X)) resolverán a true.

7) sea una función f:(X,X)→X sobre un conjunto X, decimos que "e" es un elemento neutro de f ↔ ∀x:X. f(e,x)=f(x,e)=x
Considerar el predicado "si f tiene un elemento neutro, es único"
a) Escribir el predicado como una mer de LPO
b) Decidir si la fórmula es válida. De serlo hay que probarlo con resolución. Y sino, mostrar una interpretación que la invalide.

8) Dado un programa de Prolog, del cual podemos asumir que usamos solo los operadores extra lógicos, cut y not, pero podes usar todos los que quieras si querés. Decir verdadero o falso:
a) si p(a) tiene éxito => a es solución de p(X)
b) si a es solución de p(X) => p(a) tiene éxito.
Podes armar el programa que quieras, agregar otras clausulas etc.

a) Verdadero.

- Si `p(a)` tiene éxito, significa que hay una cláusula en el programa que permite probar `p(a)` como verdadera.
- Al preguntar por `p(X)`, Prolog intenta unificar `X` con cualquier valor que satisfaga el predicado `p`.

- Como `p(a)` tiene éxito, al ejecutar `p(X)`, el intérprete unificará `X = a` como una solución válida.

codigo:

```
p(a).
p(b).
```

- **Si preguntamos `p(a).`, Prolog responde true.**
- **Si preguntamos `p(X).`, Prolog nos da como soluciones: `X = a` y `X = b`.**
- **Por lo tanto, `a` es efectivamente una solución de `p(X)`.**

b) falso

```
p(b) :- !.
p(a)
```

9) Sea P un conjunto de cláusulas de definición y G un goal, y A el árbol SLD de Prolog. Decir verdadero o falso.
a) A encuentra (o algo así) todas las refutacion es lineales de P U G
b) A encuentra todas las refutaciones SLD de P U G.

10) a) Definimos el tamaño de una fórmula de lógica de primer orden como la cantidad de instancias de símbolos de predicado que tiene. Al pasar una fórmula a forma clausal, ¿de qué orden es el crecimiento de su tamaño en función de la cantidad de operadores (&&, ||, ->)? ¿Lineal, cuadrático o exponencial?
b) ¿Cuál es la diferencia entre SLD y Prolog? Mostrar un programa y una consulta de Prolog para los que haya al menos una solución, pero Prolog no pueda hallar ninguna.

Punto (a): el crecimento sería exponencial. Si tenemos ((A ∧ B) ∨ C), cada subfórmula en forma clausal, y distribuimos el C, se va a duplicar la cantidad de literales por cada ∨ que haya dentro del C, porque estamos distribuyendo lo que está a la izquierda de éste con A y con B. Como era oral, no lo tuve que demostrar muy formalmente, sino explicar la idea.

Punto (b): SLD es un método de resolución lógica que consiste en empezar por una negación de lo que se quiere probar (que debería resultar en una cláusula objetivo que consiste sólo de literales negados), usar sólo cláusulas de Horn (cláusulas con a lo sumo un literal positivo), resolver eliminando, en cada paso, una única cláusula negativa (resolución binaria) y usar en cada paso la cláusula resolvente obtenida en el paso anterior (resolución lineal). SLD no especifica en qué orden se tienen que eliminar los literales ni en qué orden se tienen que elegir las cláusulas a comparar con ellos. SLD es completo: si una cláusula objetivo tiene una refutación a partir de las demás cláusulas, la va a hallar.

Prolog es un lenguaje de programación cuyo motor implementa una instancia de SLD con dos reglas de búsqueda: eliminar primero el literal de más a la izquierda y recorrer las cláusulas de arriba hacia abajo. Debido a estas restricciones, hay consultas de Prolog que tienen una refutación en base a un programa dado, pero el motor no la encuentra porque se queda trabado reemplazando una cláusula infinitas veces por una equivalente. Por ejemplo:

```
arbol(bin(Izq, _, Der)) :- arbol(Izq), arbol(Der).
arbol(nil).

?- arbol(X).
```

Esta consulta debería recorrer todas las estructuras posibles de árbol binario, pero en cambio, va a sustituir X por bin(Izq, _, Der), después arbol(bin(Izq, _, Der)) por arbol(Izq), arbol(Der), y hacer lo mismo con arbol(Izq), sin llegar nunca a instanciar una variable. Con SLD podemos ir directamente a la segunda cláusula y determinar que X = nil refuta la negación de arbol(X), o reemplazar X <- bin(Izq, _, Der), Izq <- nil, Der <- nil.

Cálculo Lambda

1. En calculo lambda con booleanos y naturales, se agrega un nuevo constructor de terminos fixdoble x y.<M1, M2>
M::= ...| fixdoble x y.<M1, M2> | Pi1(M) | Pi2(M)
que permite recursion entre los elemntos del par. Su tipo es sigma x tau. Ejemplo: fixdoble esPar esImpar.<\n:Nat.if (n==0) then true else esImpar(n-1),
\n:Nat.if (n==0) then true else esPar(n-1)>
es de tipo (Nat->Bool)x(Nat->Bool).
Pi1 evalua al primer elemento de fixdoble, y Pi2 al segundo. Pi1(el ejemplo de arriba) x es lo mismo que esPar x
a) dar reglas de tipado para fixdoble
b)V::= ...|fixdoble x y.<M1, M2>. Dar reglas de semantica para fixdoble

2- Cálculo lambda solo con booleanos, M es un término cerrado de tipo Bool -> Bool verdadero o falso: (->> significa evalúa en muchos pasos)
a- M true ->> M' entonces o bien M' = true o bien M' = false
b- M true ->> true o bien M true ->> false
c- M es un término en el cálculo lambda de booleanos, referencias y enteros. M true | mu ->> V | mu' con mu' potencialmente distinto a mu (es decir, puede ser igual a mu o no)

a) Falso. Podría evaluar en muchos pasos a M' pero aún asi que M' no esté en forma normal aún, y que aún queden pasos de la evaluación para hacer hasta llegar a un valor
b) Verdadero. Si M es un término cerrado y tipa en Bool -> Bool, puede ser evaluado y va a terminar reduciendo a un valor Bool: true o false.
c) Ni idea

3) En calculo lambda, definir las reglas de tipado y de reducción para el Rec de listas.