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

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

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

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.

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.

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

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

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.

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?

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.

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

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.

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.

5) definiendo fix::(a→a)→a como fix f=f(fix f),
escribir foldL sin recursión explicita, usando fix

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.

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.

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

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

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)))

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.

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

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.

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.

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)

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