Práctica Nº 1 - Programación Funcional

Para resolver esta práctica, recomendamos usar el intérprete "GHCI", de distribución gratuita, que puede bajarse de `https://www.haskell.org/ghc/`.

Para resolver los ejercicios **no** está permitido usar recursión explícita, a menos que se indique lo contrario.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Currificación y tipos en Haskell

Ejercicio 1 ★

Sean las siguientes definiciones de funciones:

```
- max2 (x, y)  | x >= y = x
               | otherwise = y
- normaVectorial (x, y) = sqrt (x^2 + y^2)
- subtract = flip (-)
- predecesor = subtract 1
- evaluarEnCero = \f -> f 0
- dosVeces = \f -> f.f
- flipAll = map flip
- flipRaro = flip flip
```

I. ¿Cuál es el tipo de cada función? (Asumir que todos los números son de tipo Float).

II. ¿Alguna de las funciones anteriores no está currificada? De ser así, escribir la versión currificada junto con su tipo para cada una de ellas.

Ejercicio 2 ★

I. Definir la función `curry`, que dada una función de dos argumentos, devuelve su equivalente currificada.

II. Definir la función `uncurry`, que dada una función currificada de dos argumentos, devuelve su versión no currificada equivalente. Es la inversa de la anterior.

III. ¿Se podría definir una función `curryN`, que tome una función de un número arbitrario de argumentos y devuelva su versión currificada?

Listas por comprensión

Ejercicio 3

¿Cuál es el valor de esta expresión?

```
[ x | x <- [1..3], y <- [x..3], (x + y) `mod` 3 == 0 ]
```

Ejercicio 4 ★

Una tripla pitagórica es una tripla (a, b, c) de enteros positivos tal que $a^2 + b^2 = c^2$.

La siguiente expresión intenta ser una definición de una lista (infinita) de triplas pitagóricas:

```
pitagóricas :: [(Integer, Integer, Integer)]
pitagóricas = [(a, b, c) | a <- [1..], b <-[1..], c <- [1..], a^2 + b^2 == c^2]
```

Explicar por qué esta definición no es útil. Dar una definición mejor.

Ejercicio 5 ★

Generar la lista de los primeros mil números primos. Observar cómo la evaluación *lazy* facilita la implementación de esta lista.

Ejercicio 6

Usando listas por comprensión, escribir la función `partir :: [a] -> [([a], [a])]` que, dada una lista xs, devuelve todas las maneras posibles de partirla en dos sublistas xs_1 y xs_2 tales que $xs_1 ++ xs_2 == xs$.

Ejemplo: `partir [1, 2, 3]` → `[([], [1, 2, 3]),([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]`

Ejercicio 7 ★

Escribir la función `listasQueSuman :: Int -> [[Int]]` que, dado un número natural n, devuelve todas las listas de enteros positivos (es decir, mayores o iguales que 1) cuya suma sea n. Para este ejercicio se permite usar recursión explícita.

Ejercicio 8

Definir en Haskell una lista que contenga todas las listas finitas de enteros positivos (esto es, con elementos mayores o iguales que 1).

Esquemas de recursión

Ejercicio 9

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b = (a -> Bool) -- determina si es o no el caso trivial
                       -> (a -> b)      -- resuelve el caso trivial
                       -> (a -> [a])    -- parte el problema en sub-problemas
                       -> ([b] -> b)    -- combina resultados
                       -> a             -- estructura de entrada
                       -> b             -- resultado
```

Definir las siguientes funciones

I. `dc :: DivideConquer a b` que implementa la técnica. Es decir, completar la siguiente definición:
`dc trivial solve split combine x = ...`
La forma en que funciona es, dado un input x, verifica si es o no un caso base utilizando la función *trivial*. En caso de serlo, utilizaremos *solve* para dar el resultado final. En caso de no ser un caso base, partimos el problema utilizando la función *split* y luego combinamos los resultados recursivos utilizando *combine*. Por ser este un esquema de recursión, puede utilizarse recursión explícita para definirlo.

II. Implementar la función `mergeSort :: Ord a =>[a] ->[a]` en términos de `dc`.
`mergeSort = dc ...` (se recomienda utilizar `break` y aplicación parcial para definir la función de *combine*).

III. Utilizar el esquema `dc` para reimplementar `map` y `filter`.
`map :: (a ->b) ->[a] ->[b]`
`filter :: (a ->Bool) ->[a] ->[a]`

Ejercicio 10 ★

I. Redefinir usando `foldr` las funciones `sum`, `elem`, `(++)`, `filter` y `map`.

II. Definir la función `mejorSegún :: (a -> a -> Bool) -> [a] -> a`, que devuelve el máximo elemento de la lista según una función de comparación, utilizando `foldr1`. Por ejemplo, `maximum = mejorSegún (>)`.

III. Definir la función `sumaAlt`, que realiza la suma alternada de los elementos de una lista. Es decir, da como resultado: el primer elemento, menos el segundo, más el tercero, menos el cuarto, etc. Usar `foldr`.

IV. Hacer lo mismo que en el punto anterior, pero en sentido inverso (el último elemento menos el anteúltimo, etc.). Pensar qué esquema de recursión conviene usar en este caso.

V. Definir la función `permutaciones :: [a] -> [[a]]`, que dada una lista devuelve todas sus permutaciones. Se recomienda utilizar `concatMap :: (a -> [b]) -> [a] -> [b]`, y también `take` y `drop`.

Ejercicio 11

I. Definir la función `partes`, que recibe una lista `L` y devuelve la lista de todas las listas formadas por los mismos elementos de `L`, en su mismo orden de aparición.

   Ejemplo: `partes [5, 1, 2]` → `[[], [5], [1], [2], [5, 1], [5, 2], [1, 2], [5, 1, 2]]` (en algún orden).

II. Definir la función `prefijos`, que dada una lista, devuelve todos sus prefijos.

   Ejemplo: `prefijos [5, 1, 2]` → `[[], [5], [5, 1], [5, 1, 2]]`

III. Definir la función `sublistas` que, dada una lista, devuelve todas sus sublistas (listas de elementos que aparecen consecutivos en la lista original).

   Ejemplo: `sublistas [5, 1, 2]` → `[[], [5], [1], [2], [5, 1], [1, 2], [5, 1, 2]]` (en algún orden).

Ejercicio 12 ★

El siguiente esquema captura la recursión primitiva sobre listas.

```
recr::(a->[a]->b->b)->b->[a]->b
recr \_ z [] = z
recr f z (x:xs) = f x xs (recr f z xs)
```

a. Definir la función `sacarUna :: Eq a => a -> [a] -> [a]`, que dados un elemento y una lista devuelve el resultado de eliminar de la lista la primera aparición del elemento (si está presente).

b. Explicar por qué el esquema `foldr` no es adecuado para implementar la función `sacarUna` del punto anterior.

c. La función `listasQueSuman` del ejercicio 7, ¿se ajusta al esquema de recursión `recr`? ¿Por qué o por qué no?

Ejercicio 13

I. Definir la función `genLista :: a -> (a -> a) -> Integer -> [a]`, que genera una lista de una cantidad dada de elementos, a partir de un elemento inicial y de una función de incremento entre los elementos de la lista. Dicha función de incremento, dado un elemento de la lista, devuelve el elemento siguiente.

II. Usando `genLista`, definir la función `desdeHasta`, que dado un par de números (el primero menor que el segundo), devuelve una lista de números consecutivos desde el primero hasta el segundo.

Ejercicio 14 ★

Definir las siguientes funciones para trabajar sobre listas, y dar su tipo. Todas ellas deben poder aplicarse a listas *finitas* e *infinitas*.

I. `mapPares`, una versión de `map` que toma una función currificada de dos argumentos y una lista de pares de valores, y devuelve la lista de aplicaciones de la función a cada par. **Pista:** recordar `curry` y `uncurry`.

II. `armarPares`, que dadas dos listas arma una lista de pares que contiene, en cada posición, el elemento correspondiente a esa posición en cada una de las listas. Si una de las listas es más larga que la otra, ignorar los elementos que sobran (el resultado tendrá la longitud de la lista más corta). Esta función en Haskell se llama `zip`. **Pista:** aprovechar la currificación y utilizar evaluación parcial.

III. `mapDoble`, una variante de `mapPares`, que toma una función currificada de dos argumentos y dos listas (de igual longitud), y devuelve una lista de aplicaciones de la función a cada elemento correspondiente de las dos listas. Esta función en Haskell se llama `zipWith`.

Ejercicio 15

I. Escribir la función `sumaMat`, que representa la suma de matrices, usando `zipWith`. Representaremos una matriz como la lista de sus filas. Esto quiere decir que cada matriz será una lista finita de listas finitas, todas de la misma longitud, con elementos enteros. Recordamos que la suma de matrices se define como la suma celda a celda. Asumir que las dos matrices a sumar están bien formadas y tienen las mismas dimensiones.
   `sumaMat :: [[Int]] -> [[Int]] -> [[Int]]`

II. Escribir la función `trasponer`, que, dada una matriz como las del ítem I, devuelva su traspuesta. Es decir, en la posición i, j del resultado está el contenido de la posición j, i de la matriz original. Notar que si la entrada es una lista de N listas, todas de longitud M, entonces el resultado debe tener M listas, todas de longitud N.
`trasponer :: [[Int]] -> [[Int]]`

Ejercicio 16 ★

Definimos la función `generate`, que genera listas en base a un predicado y una función, de la siguiente manera:

```
generate ::  ([a] -> Bool) -> ([a] -> a) -> [a]
generate stop next = generateFrom stop next []

generateFrom:: ([a] -> Bool) -> ([a] -> a) -> [a] -> [a]
generateFrom stop next xs | stop xs = init xs
                          | otherwise = generateFrom stop next (xs ++ [next xs])
```

I. Usando `generate`, definir `generateBase::([a] ->Bool) ->a ->(a ->a) ->[a]`, similar a `generate`, pero con un caso base para el elemento inicial, y una función que, en lugar de calcular el siguiente elemento en base a la lista completa, lo calcula solo a partir del último elemento. Por ejemplo: `generateBase (\l->not (null l) && (last l > 256)) 1 (*2)` es la lista las potencias de 2 menores o iguales que 256.

II. Usando `generate`, definir `factoriales::Int ->[Int]`, que dado un entero `n` genera la lista de los primeros `n` factoriales.

III. Usando `generateBase`, definir `iterateN :: Int -> (a -> a) -> a -> [a]` que, toma un entero `n`, una función `f` y un elemento inicial `x`, y devuelve la lista `[x, f x, f (f x), ..., f ( ...(f x) ...)]` de longitud `n`. **Nota:** `iterateN n f x` = `take n (iterate f x)`.

IV. Redefinir `generateFrom` usando `iterate` y `takeWhile`.

Otras estructuras de datos

En esta sección se permite (y se espera) el uso de recursión explícita *únicamente* para la definición de esquemas de recursión.

Ejercicio 17 ★

I. Definir y dar el tipo del esquema de recursión `foldNat` sobre los naturales. Utilizar el tipo `Integer` de Haskell (la función va a estar definida sólo para los enteros mayores o iguales que 0).

II. Utilizando `foldNat`, definir la función `potencia`.

Ejercicio 18

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Polinomio a  =  X
                  |  Cte a
                  |  Suma (Polinomio a) (Polinomio a)
                  |  Prod (Polinomio a) (Polinomio a)
```

Luego usar el esquema definido para escribir la función: `evaluar :: Num a => a -> Polinomio a- > a`

Ejercicio 19 ★

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si c es un conjunto y e un elemento, la expresión $c\ e$ devuelve `True` sii e pertenece a c.

I. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.

II. Escribir las funciones `intersección` y `unión` (ambas de tipo `Conj a -> Conj a-> Conj a`).

III. Definir un conjunto de funciones que contenga infinitos elementos, y dar su tipo.

IV. Definir la función `singleton :: Eq a => a -> Conj a`, que dado un valor genere un conjunto con ese valor como único elemento.

V. ¿Puede definirse un `map` para esta estructura? ¿De qué manera, o por qué no?

Ejercicio 20

En este ejercicio trabajaremos con matrices infinitas representadas como funciones:
`type MatrizInfinita a = Int->Int->a`
donde el primer argumento corresponde a la fila, el segundo a la columna y el resultado al valor contenido en la celda correspondiente.
Por ejemplo, las siguientes definiciones:

```
identidad = \i j->if i==j then 1 else 0
cantor = \x y->(x+y)*(x+y+1)`div`2+y
pares = \x y->(x,y)
```

corresponden a las matrices:

1	0	0	⋯
0	1	0	⋯
0	0	1	⋯
⋮	⋮	⋮	⋱

identidad

0	2	5	⋯
1	4	8	⋯
3	7	12	⋯
⋮	⋮	⋮	⋱

cantor

(0,0)	(0,1)	(0,2)	⋯
(1,0)	(1,1)	(1,2)	⋯
(2,0)	(2,1)	(2,2)	⋯
⋮	⋮	⋮	⋱

pares

Definir las siguientes funciones:

I. `fila::Int->MatrizInfinita a->[a]` y `columna::Int->MatrizInfinita a->[a]` que, dado un índice, devuelven respectivamente la fila o la columna correspondiente en la matriz (en forma de lista infinita). Por ejemplo, `fila 0 identidad` devuelve la lista con un 1 seguido de infinitos 0s.

II. `trasponer::MatrizInfinita a->MatrizInfinita a`, que dada una matriz devuelve su transpuesta.

III. `mapMatriz::(a->b)->MatrizInfinita a->MatrizInfinita b`,
`filterMatriz::(a->Bool)->MatrizInfinita a->[a]` y
`zipWithMatriz::(a->b->c)->MatrizInfinita a->MatrizInfinita b->MatrizInfinita c`, que se comportan como `map`, `filter` y `zipWith` respectivamente, pero aplicadas a matrices infinitas. En el caso de `filterMatriz` no importa el orden en el que se devuelvan los elementos, pero se debe pasar una y sólo una vez por cada posición de la matriz.

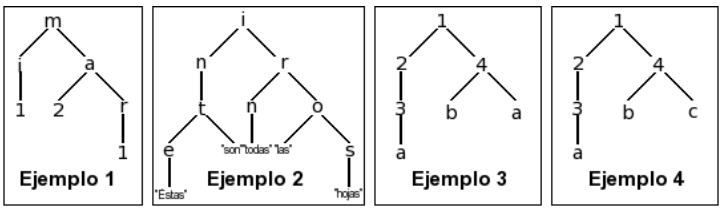
IV. `suma::Num a=>MatrizInfinita a->MatrizInfinita a->MatrizInfinita a`, y
`zipMatriz::MatrizInfinita a->MatrizInfinita b->MatrizInfinita (a,b)`. Definir ambas utilizando `zipWithMatriz`.

Ejercicio 21 ★

Consideremos el siguiente tipo de datos:

```
data AHD tInterno tHoja = Hoja tHoja
                        | Rama tInterno (AHD tInterno tHoja)
                        | Bin (AHD tInterno tHoja) tInterno (AHD tInterno tHoja)
```

que representa un árbol binario no vacío cuyos nodos internos pueden tener datos de un tipo diferente al de sus hojas. (AHD = árbol con hojas distinguidas).
Por ejemplo:
`Bin (Hoja ‘‘hola’’) ‘b’ (Rama ‘c’ (Hoja ‘‘chau’’))` tiene tipo `AHD Char String`
`Rama 1 (Bin(Hoja True)(-2)(Hoja False))` tiene tipo `AHD Int Bool`
A continuación mostramos algunos ejemplos de forma más gráfica:



I. Escribir el esquema de recursión estructural `foldAHD` para este tipo de datos, y dar su tipo.

II. Escribir, usando `foldAHD`, la función `mapAHD :: (a -> b) -> (c -> d) -> AHD a c -> AHD b d`, que actúa de manera análoga al `map` de listas, aplicando la primera función a los nodos internos y la segunda a las hojas. Por ejemplo:

```
mapAHD (+1) not (Bin(Rama 1 (Hoja False)) 2 (Bin(Hoja False) 3 (Rama 5 (Hoja True))))
```

devuelve `Bin (Rama 2 (Hoja True)) 3 (Bin (Hoja True) 4 (Rama 6 (Hoja False)))`.

Ejercicio 22

Sea el siguiente tipo, que representa a los árboles binarios:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

I. Definir el esquema de recursión estructural (*fold*) para estos árboles, y dar su tipo.

II. Definir las funciones `esNil`, `altura`, `ramas` (caminos desde la raíz hasta las hojas), `#nodos`, `#hojas` y `espejo` (para `esNil` puede utilizarse `case` en lugar de `fold`).

III. Definir la función `mismaEstructura :: AB a -> AB b -> Bool` que, dados dos árboles, indica si éstos tienen la misma forma, independientemente del contenido de sus nodos. **Pista:** usar evaluación parcial y recordar el ejercicio 15.

Ejercicio 23 ★

I. Definir el tipo de datos `RoseTree` de árboles no vacíos, donde cada nodo tiene una cantidad indeterminada de hijos.

II. Escribir el esquema de recursión estructural para `RoseTree`. **Importante** escribir primero su tipo.

III. Usando el esquema definido, escribir las siguientes funciones:

a) `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.

b) `distancias`, que dado un `RoseTree`, devuelva las distancias de su raíz a cada una de sus hojas.

c) `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Práctica N^{o} 2 - Introducción al cálculo lambda tipado

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

A menos que se especifiquen las extensiones a utilizar, trabajaremos con el cálculo λ con los tipos Bool, Nat y funciones.

Notación para este segmento de la materia:

- las letras $M, N, O, P, \ldots$ denotan términos.
- las letras $V, W, Y, \ldots$ denotan valores.
- las letras griegas $\sigma, \tau, \rho, \pi, \ldots$ denotan tipos.

Gramáticas a tener en cuenta:

- Términos
  $M ::= x \mid \lambda x\colon \sigma.\ M \mid M\ M \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid 0 \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{isZero}(M)$

  Donde la letra x representa un *nombre de variable* arbitrario. Tales nombres se toman de un conjunto infinito dado $\mathfrak{X} = \{w, w_1, w_2, \ldots, x, x_1, x_2, \ldots, y, y_1, y_2, \ldots, z, z_1, z_2, \ldots\}$
- Tipos
  $\sigma ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \sigma \to \sigma$

Sintaxis

Ejercicio 1 ★

Determinar qué expresiones son sintácticamente válidas (es decir, pueden ser generadas con las gramáticas presentadas) y determinar a qué categoría pertenecen (expresiones de términos o expresiones de tipos):

1. x
2. $x\ x$
3. M
4. $M\ M$
5. $\mathsf{true}\ \mathsf{false}$
6. $\mathsf{true}\ \mathsf{succ}(\mathsf{false}\ \mathsf{true})$
7. $\lambda x.\mathsf{isZero}(x)$
8. $\lambda x\colon \sigma.\ \mathsf{succ}(x)$
9. $\lambda x\colon \mathsf{Bool}.\ \mathsf{succ}(x)$
10. $\lambda x\colon \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{Bool}\ \mathsf{else}\ \mathsf{Nat}.\ x$
11. σ
12. Bool
13. $\mathsf{Bool} \to \mathsf{Bool}$
14. $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Nat}$
15. $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Nat}$
16. $\mathsf{succ}\ \mathsf{true}$
17. $\lambda x\colon \mathsf{Bool}.\ \mathsf{if}\ 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ 0\ \mathsf{succ}(\mathsf{true})$

Ejercicio 2

Mostrar un término que utilice al menos una vez **todas** las reglas de generación de la gramática y exhibir su *árbol sintáctico*.

Ejercicio 3 ★

1. Marcar las ocurrencias del término x como subtérmino en $\lambda x\colon \mathsf{Nat}.\ \mathsf{succ}((\lambda x\colon \mathsf{Nat}.\ x)\ x)$.
2. ¿Ocurre x_1 como subtérmino en $\lambda x_1\colon \mathsf{Nat}.\ \mathsf{succ}(x_2)$?
3. ¿Ocurre $x\ (y\ z)$ como subtérmino en $u\ x\ (y\ z)$?

Ejercicio 4 ★

Para los siguientes términos:

1. $u\ x\ (y\ z)\ (\lambda v\colon \mathsf{Bool}.\ v\ y)$
2. $(\lambda x\colon \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Bool}.\ \lambda y\colon \mathsf{Bool} \to \mathsf{Nat}.\ \lambda z\colon \mathsf{Bool}.\ x\ z\ (y\ z))\ u\ v\ w$
3. $w\ (\lambda x\colon \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Bool}.\ \lambda y\colon \mathsf{Bool} \to \mathsf{Nat}.\ \lambda z\colon \mathsf{Bool}.\ x\ z\ (y\ z))\ u\ v$

Se pide:

1. Insertar todos los paréntesis de acuerdo a la convención usual.
2. Dibujar el árbol sintáctico de cada una de las expresiones.
3. Indicar en el árbol cuáles ocurrencias de variables aparecen ligadas y cuáles libres.
4. ¿En cuál de los términos anteriores ocurre la siguiente expresión como subtérmino?
   $(\lambda x\colon \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Bool}.\ \lambda y\colon \mathsf{Bool} \to \mathsf{Nat}.\ \lambda z\colon \mathsf{Bool}.\ x\ z\ (y\ z))\ u$

Tipado

Ejercicio 5 *(Derivaciones ★)*

Demostrar o explicar por qué no puede demostrarse cada uno de los siguientes juicios de tipado.

1. $\emptyset \rhd \mathsf{if\ true\ then}\ 0\ \mathsf{else\ succ}(0)\ :\ \mathsf{Nat}$
2. $\{x : \mathsf{Nat}, y : \mathsf{Bool}\} \rhd \mathsf{if\ true\ then\ false\ else}\ (\lambda z\colon \mathsf{Bool}.\ z)\ \mathsf{true}\ :\ \mathsf{Bool}$
3. $\emptyset \rhd \mathsf{if}\ \lambda x\colon \mathsf{Bool}.\ x\ \mathsf{then}\ 0\ \mathsf{else\ succ}(0)\ :\ \mathsf{Nat}$
4. $\{x : \mathsf{Bool} \to \mathsf{Nat}, y : \mathsf{Bool}\} \rhd x\ y\ :\ \mathsf{Nat}$

Ejercicio 6

Determinar qué tipo representa σ en cada uno de los siguientes juicios de tipado.

1. $\emptyset \rhd \mathsf{succ}(0)\ :\ \sigma$
2. $\emptyset \rhd \mathsf{isZero}(\mathsf{succ}(0))\ :\ \sigma$
3. $\emptyset \rhd \mathsf{if}\ (\mathsf{if\ true\ then\ false\ else\ false})\ \mathsf{then}\ 0\ \mathsf{else\ succ}(0)\ :\ \sigma$

Ejercicio 7 ★

Determinar qué tipos representan σ y τ en cada uno de los siguientes juicios de tipado. Si hay más de una solución, o si no hay ninguna, indicarlo.

1. $\{x\colon \sigma\} \rhd \mathsf{isZero}(\mathsf{succ}(x))\ :\ \tau$
2. $\emptyset \rhd (\lambda x\colon \sigma.\ x)(\lambda y\colon \mathsf{Bool}.\ 0)\ :\ \sigma$
3. $\{y\colon \tau\} \rhd \mathsf{if}\ (\lambda x\colon \sigma.\ x)\ \mathsf{then}\ y\ \mathsf{else\ succ}(0)\ :\ \sigma$
4. $\{x\colon \sigma\} \rhd x\ y\ :\ \tau$
5. $\{x\colon \sigma, y\colon \tau\} \rhd x\ y\ :\ \tau$
6. $\{x\colon \sigma\} \rhd x\ \mathsf{true}\ :\ \tau$
7. $\{x\colon \sigma\} \rhd x\ \mathsf{true}\ :\ \sigma$
8. $\{x\colon \sigma\} \rhd x\ x\ :\ \tau$

Ejercicio 8

Mostrar un término que no sea tipable y que no tenga variables libres ni abstracciones.

Ejercicio 9

Mostrar un juicio de tipado que sea demostrable en el sistema actual pero que no lo sea al cambiar (T-ABS) por la siguiente regla. Mostrar la demostración del juicio original.

$$\frac{\Gamma \triangleright M \ : \ \tau}{\Gamma \triangleright \lambda x\colon \sigma.\ M \ : \ \sigma \to \tau}\ \text{T-ABS2}$$

Semántica

Ejercicio 10 ★

Sean $\sigma,\ \tau,\ \rho$ tipos. Según la definición de sustitución, calcular:

1. $(\lambda y\colon \sigma.\ x\ (\lambda x\colon \tau.\ x))\{x \leftarrow (\lambda y\colon \rho.\ x\ y)\}$
2. $(y\ (\lambda v\colon \sigma.\ x\ v))\{x \leftarrow (\lambda y\colon \tau.\ v\ y)\}$

Renombrar variables en ambos términos para no cambiar el significado del término.

Ejercicio 11 *(Valores)* **★**

Dado el conjunto de valores visto en clase:

$$V := \lambda x\colon \sigma.\ M \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid \mathsf{succ}(V)$$

Determinar si cada una de las siguientes expresiones es o no un valor:

1. $(\lambda x\colon \mathsf{Bool}.\ x)\ \mathsf{true}$
2. $\lambda x\colon \mathsf{Bool}.\ \underline{2}$
3. $\lambda x\colon \mathsf{Bool}.\ \mathsf{pred}(\underline{2})$
4. $\lambda y\colon \mathsf{Nat}.\ (\lambda x\colon \mathsf{Bool}.\ \mathsf{pred}(\underline{2}))\ \mathsf{true}$
5. x
6. $\mathsf{succ}(\mathsf{succ}(0))$

Ejercicio 12 *(Programa, Forma Normal)* **★**

Para el siguiente ejercicio, considerar el cálculo **sin** la regla $\mathsf{pred}(0) \to 0$

Un *programa* es un término que tipa en el contexto vacío (es decir, no puede contener variables libres).

Para cada una de las siguientes expresiones

(a) Determinar si puede ser considerada un **programa**.

(b) Si vale (a), ¿Cuál es el resultado de su evaluación? Determinar si se trata de una forma normal, y en caso de serlo, si es un **valor** o un **error**.

1. $(\lambda x\colon \mathsf{Bool}.\ x)\ \mathsf{true}$
2. $\lambda x\colon \mathsf{Nat}.\ \mathsf{pred}(\mathsf{succ}(x))$
3. $\lambda x\colon \mathsf{Nat}.\ \mathsf{pred}(\mathsf{succ}(y))$
4. $(\lambda x\colon \mathsf{Bool}.\ \mathsf{pred}(\mathsf{isZero}(x)))\ \mathsf{true}$
5. $(\lambda f\colon \mathsf{Nat} \to \mathsf{Bool}.\ f\ 0)\ (\lambda x\colon \mathsf{Nat}.\ \mathsf{isZero}(x))$
6. $(\lambda f\colon \mathsf{Nat} \to \mathsf{Bool}.\ x)\ (\lambda x\colon \mathsf{Nat}.\ \mathsf{isZero}(x))$
7. $(\lambda f\colon \mathsf{Nat} \to \mathsf{Bool}.\ f\ \mathsf{pred}(0))\ (\lambda x\colon \mathsf{Nat}.\ \mathsf{isZero}(x))$

8. $\mathsf{fix}\ (\lambda y\colon \mathsf{Nat}.\ \mathsf{succ}(y))$
9. $\mathsf{letrec}\ f = \lambda x\colon \mathsf{Nat}.\ \mathsf{succ}(f\ x)\ \mathsf{in}\ f\ 0$

Ejercicio 13 *(Determinismo)*

1. ¿Es cierto que la relación definida $\to$ es determinística (o una función parcial)?
   Más precisamente, ¿pasa que si $M \to N$ y $M \to N'$ entonces $N = N'$?
2. ¿Vale lo mismo con muchos pasos? Es decir, ¿es cierto que si $M \twoheadrightarrow M'$ y $M \twoheadrightarrow M''$ entonces $M' = M''$?
3. ¿Acaso es cierto que si $M \to M'$ y $M \twoheadrightarrow M''$ entonces $M' = M''$?

Ejercicio 14

1. ¿Da lo mismo evaluar $\mathsf{succ}(\mathsf{pred}(M))$ que $\mathsf{pred}(\mathsf{succ}(M))$? ¿Por qué?
2. ¿Es verdad que para todo M vale que $\mathsf{isZero}(\mathsf{succ}(M)) \twoheadrightarrow \mathsf{false}$? Si no lo es, ¿para qué términos vale?
3. ¿Para qué términos M vale que $\mathsf{isZero}(\mathsf{pred}(M)) \twoheadrightarrow \mathsf{true}$? (Hay infinitos).

Ejercicio 15

Al agregar la siguiente regla para las abstracciones:

$$\frac{M \to M'}{\lambda x\colon \tau.\ M \to \lambda x\colon \tau.\ M'}\ E - ABS$$

1. Repensar el conjunto de valores para respetar esta modificación, pensar por ejemplo si $(\lambda x\colon \mathsf{Bool}.\ Id_{bool}\ \mathsf{true})$ es o no un valor.
2. ¿Qué reglas deberían modificarse para no perder el determinismo?
3. Utilizando la nueva regla y los valores definidos, reducir la siguiente expresión
   $(\lambda x\colon \mathsf{Nat} \to \mathsf{Nat}.\ x\ \underline{23})\ (\lambda x\colon \mathsf{Nat}.\ 0)$
   ¿Qué se puede concluir entonces? ¿Es seguro o no agregar esta regla?

Ejercicio 16

La variante del Cálculo Lambda vista en clase utiliza el modo de reducción call-by-value: para reducir una aplicación a forma normal, se reduce primero el argumento y luego se ejecuta la aplicación. ¿Cómo cambiaría el cálculo si en lugar de esto se utilizara la estrategia call-by-name (es decir, reduciendo la aplicación antes que el argumento)[1]? Mencionar qué reglas se modifican, y reescribirlas para adaptarlas a esta estrategia.

Reducir el siguiente término a forma normal utilizando la estrategia:

$comp\ (\lambda x\colon \mathsf{Nat}.\ \mathsf{succ}(x))\ (\lambda x\colon \mathsf{Nat}.\ \mathsf{succ}(x))\ \underline{5}$

donde $comp \stackrel{\text{def}}{=} \lambda f\colon \mathsf{Nat} \to \mathsf{Nat}.\ \lambda g\colon \mathsf{Nat} \to \mathsf{Nat}.\ \lambda x\colon \mathsf{Nat}.\ f\ (g\ x)$

Extensiones

En esta sección puede asumirse, siempre que sea necesario, que el cálculo ha sido extendido con la suma de números naturales $(M + N)$, con las siguiente reglas de tipado y semántica:

$$\frac{\Gamma \rhd M\colon \mathsf{Nat} \quad \Gamma \rhd N\colon \mathsf{Nat}}{\Gamma \rhd M + N\colon \mathsf{Nat}}\ \text{T-+}$$

$$\frac{M \to M'}{M + N \to M' + N}\ \text{E-+}_1 \qquad \frac{N \to N'}{V + N \to V + N'}\ \text{E-+}_2 \qquad \frac{}{V + 0 \to V}\ \text{E-+0} \qquad \frac{}{V_1 + \mathsf{succ}(V_2) \to \mathsf{succ}(V_1) + V_2}\ \text{E-+Succ}$$

[1] Pista: la idea de la reducción call-by-name consiste en lo siguiente: en el caso de un término con la forma $(\lambda x\colon \sigma.M)\ N$, en lugar de reducir primero N (que es lo que haría la reducción call-by-value), se resuelve la aplicación directamente sobre el parámetro sin reducir; es decir, se reduce $(\lambda x\colon \sigma.M)\ N$ a $M\{x \leftarrow N\}$.

Ejercicio 17 ★

Este ejercicio extiende el Cálculo Lambda tipado con listas. Comenzamos ampliando el conjunto de tipos:

$$\sigma \quad ::= \quad \ldots \mid [\sigma]$$

donde $[\sigma]$ representa el tipo de las listas cuyas componentes son de tipo σ. El conjunto de términos ahora incluye:

$$M, N, O \quad ::= \quad \ldots \mid [\,]_\sigma \mid M :: N \mid \texttt{case}\, M\ \texttt{of}\ \{[\,] \rightsquigarrow N \mid h :: t \rightsquigarrow O\} \mid \texttt{foldr}\, M\, \text{base} \rightsquigarrow N; \text{rec}(h, r) \rightsquigarrow O$$

donde

- $[\,]_\sigma$ es la lista vacía cuyos elementos son de tipo σ;
- $M :: N$ agrega M a la lista N;
- $\texttt{case}\ M\ \texttt{of}\ \{[\,] \rightsquigarrow N \mid h :: t \rightsquigarrow O\}$ es el observador de listas. Por su parte, los nombres de variables que se indiquen luego del $\mid$ (h y t en este caso) son variables que pueden aparecer libres en O y deberán ligarse con la cabeza y cola de la lista respectivamente;
- $\texttt{foldr}\ M\ \ \text{base} \rightsquigarrow N;\ \text{rec}(h, r) \rightsquigarrow O$ es el operador de recursión estructural (no currificado). Los nombres de variables indicados entre parentesis (h y r en este caso) son variables que pueden aparecer libres en O y deberán ser ligadas con la cabeza y el resultado de la recursión respectivamente.

Por ejemplo,

- $\texttt{case}\ 0 :: \mathsf{succ}(0) :: [\,]_{\mathsf{Nat}}\ \texttt{of}\ \{[\,] \rightsquigarrow \mathsf{false} \mid x :: xs \rightsquigarrow \mathsf{isZero}(x)\} \twoheadrightarrow \mathsf{true}$
- $\texttt{foldr}\, \underline{1} :: \underline{2} :: \underline{3} :: (\lambda x\colon [\mathsf{Nat}].\ x)\ [\,]_{\mathsf{Nat}}\ \text{base} \rightsquigarrow 0;\ \text{rec}(head, rec) \rightsquigarrow head + rec \twoheadrightarrow \underline{6}$

1. Mostrar el árbol sintáctico para los dos ejemplos dados.
2. Agregar reglas de tipado para las nuevas expresiones.
3. Demostrar el siguiente juicio de tipado (recomendación: marcar variables libres y ligadas en el término antes de comenzar).

   $\{x : \mathsf{Bool}, y : [\mathsf{Bool}]\} \triangleright \texttt{foldr}\, x :: x :: y\ \ \text{base} \rightsquigarrow y;\ \text{rec}(y, x) \rightsquigarrow \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ [\,]_{\mathsf{Bool}}\ :\ [\mathsf{Bool}]$
4. Mostrar cómo se extiende el conjunto de valores. Estos deben reflejar la forma de las listas que un programa podría devolver.
5. Agregar los axiomas y reglas de reducción asociados a las nuevas expresiones.

Ejercicio 18 ★

A partir de la extensión del ejercicio 17, definir una nueva extensión que incorpore expresiones de la forma $\texttt{map}(M, N)$, donde N es una lista y M una función que se aplicará a cada uno de los elementos de N.

Importante: tener en cuenta las anotaciones de tipos al definir las reglas de tipado y semántica.

Ejercicio 19 ★

La aplicación parcial sobre funciones currificadas es una de las ventajas de los lenguajes funcionales, como el cálculo lambda tipado. Sin embargo, el mecanismo del cálculo lambda (que se repite en la mayoría de los lenguajes funcionales como Haskell) es limitado, ya que la aplicación parcial debe hacerse siempre en el orden de los argumentos. Por ejemplo, si tenemos la función potencia, podemos usarla con aplicación parcial para definir la función cuadrado, si su primer parámetro es el exponente, o la función dosALa si su primer parámetro es la base, pero no podemos hacer ambas cosas con la misma función potencia.

Para solucionar este problema introduciremos el cálculo μ, que es igual al cálculo lambda en todo, excepto en que el mecanismo para construir funciones ($\lambda x.M$) y el mecanismo para aplicarlas ($M\ N$) serán sustituidos por un nuevo mecanismo de construcción ($\mu x_1, ..., x_n.M$) y de aplicación ($M\ \#_i\ N$). Estos cambios también introducen un cambio en el sistema de tipos: en lugar de tener $\sigma \to \tau$ tendremos $\{\sigma_1, ..., \sigma_n\} \to \tau$. Notar que $\{\sigma_1, ..., \sigma_n\}$ no es un nuevo tipo, sino sólo una parte del nuevo tipo para funciones.

La sintaxis del cálculo μ y su conjunto de tipos, entonces, serán los siguientes:

$$M, N \ ::= \ ... \mid \mu x_1 \colon \sigma_1, ..., x_n \colon \sigma_n.M \mid M\ \#_i\ N \qquad \sigma_1 \ldots \sigma_n, \tau \ ::= \ ... \mid \{\sigma_1, ..., \sigma_n\} \to \tau$$

El término $\mu x_1 : \sigma_1, ..., x_n : \sigma_n.M$ sirve para construir una nueva función de n parámetros ordenados y el operador $\#_i$ sirve para aplicar el i-ésimo parámetro. Notar que si la cantidad de parámetros de una función es mayor a 1, al aplicarla se obtiene una nueva función con un parámetro menos, pero si la cantidad de parámetros es exactamente 1, al aplicarla se obtiene su valor de retorno. Notar además que que el orden de los tipos de los argumentos es importante: por ejemplo, $\{nat, nat, bool\} \to nat$ y $\{bool, nat, nat\} \to nat$ no son el mismo tipo.

1. Introducir las reglas de tipado para la extensión propuesta.
2. Dar formalmente la extensión de los valores e introducir las reglas de semántica para la extensión propuesta.
3. Escribir las construcciones básicas del cálculo lambda (λ y aplicación) como macros del cálculo μ para mostrar que este último puede emularlo.

Ejercicio 20

Definir una extensión que permita "unir" un registro $\{x_1 = M_1, \ldots, x_m = M_m\}$ con otro registro $\{y_1 = N_1, \ldots, y_n = N_n\}$, de manera tal que el registro resultante contenga todas las etiquetas de ambos, con los mismos valores y en el mismo orden.

Restricción: los registros a unir **no deben** tener etiquetas en común.

Ejercicio 21 *(Conectivos booleanos)*

Definir como macros (azúcar sintáctica) los términos **Not**, **And**, **Or**, **Xor**, que simulen desde la reducción los conectivos clásicos usuales, por ej. $And\ M\ N \twoheadrightarrow \mathsf{true} \Leftrightarrow M \twoheadrightarrow \mathsf{true}$ y $N \twoheadrightarrow \mathsf{true}$.

Notar que definir una macro no es lo mismo que hacer una extensión. Por ejemplo, definir el término $I_\sigma \stackrel{\text{def}}{=} \lambda x : \sigma.x$, que es la función identidad del tipo σ, es distinto de extender la sintaxis del lenguaje con términos de la forma $I(M)$, lo cual además requeriría agregar nuevas reglas de tipado y de evaluación.

Ejercicio 22 ★

Se desea extender el cálculo lambda tipado agregando *unión de funciones*. Para ello, extenderemos el conjunto de términos y el de tipos de la siguiente manera:

$$M_1 \ldots M_k ::= \cdots \mid [(M_1, \ldots, M_k)] \qquad \sigma ::= \cdots \mid \mathsf{Union}(\sigma_1, \ldots, \sigma_k)_\tau$$

Cada M_i dentro de "[()]" es una función con distinto dominio del resto pero con la misma imagen.

En el tipo $\mathsf{Union}(\sigma_1, \ldots, \sigma_k)_\tau$, cada σ_i representa el tipo del dominio de M_i (la función en la posición i), y τ el tipo de la imagen de todas las funciones.

Al aplicarse esta unión sobre un valor de tipo σ, el término reduce utilizando la función de esta unión cuyo tipo para el dominio sea σ. Es decir, aplicando la función que corresponda según el dominio.

Por ejemplo, sea

$$l \stackrel{def}{=} [(\lambda\, x : \mathsf{Nat}\,.\ x + 2,\ \lambda\, x : \mathsf{Bool}\,.\ \mathsf{if}\ x\ \mathsf{then}\ 4\ \mathsf{else}\ 3,\ \lambda\, f : \mathsf{Bool} \to \mathsf{Nat}\,.\,(f\ \mathsf{true})\ +\ 3)]$$

l tiene tipo $\mathsf{Union}(\mathsf{Nat}, \mathsf{Bool}, \mathsf{Bool} \to \mathsf{Nat})_{\mathsf{Nat}}$.

Luego, $l\ (\lambda\, b : Bool.\ \mathsf{if}\ b\ \mathsf{then}\ 3\ \mathsf{else}\ 4) \longrightarrow\!\!\!\!\rightarrow ((\lambda b : \mathsf{Bool}.\ \mathsf{if}\ b\ \mathsf{then}\ 3\ \mathsf{else}\ 4)\ \mathsf{true}) + 3) \longrightarrow\!\!\!\!\rightarrow 6$.

Se pide:

1. Extender las reglas de tipado acordemente.
2. Mostrar el árbol de derivación para el juicio: $\{y : \mathsf{Nat}\} \triangleright [(\lambda x : \mathsf{Bool}.\ y, \lambda x : \mathsf{Nat}.\ x)]\ y : \mathsf{Nat}$.
3. Indicar cómo se modifica el conjunto de valores. Justificar.
4. Modificar o extender las reglas de semántica operacional para la extensión propuesta.

Ejercicio 23

Definir las siguientes funciones en Cálculo Lambda con Listas (visto en el ejercicio 17). Pueden definirse como macros o como extensiones al cálculo.

Nota: en este ejercicio usamos la notación $M : \sigma$ para decir que la expresión M a definir debe tener tipo σ en cualquier contexto.

1. $head_\sigma : [\sigma] \to \sigma$ y $tail_\sigma : [\sigma] \to [\sigma]$ (asumir que $\bot_\sigma \stackrel{\text{def}}{=} fix\ \lambda x : \sigma.x$).

2. $iterate_\sigma$:$(\sigma \to \sigma) \to \sigma \to [\sigma]$ que dadas f y x genera la lista infinita $x :: f\ x :: f(f\ x) :: f(f(f\ x)) :: \ldots$ (y no termina).

3. $zip_{\rho,\sigma} : [\rho] \to [\sigma] \to [\rho \times \sigma]$ que se comporta como la función homónima de Haskell.

4. $take_\sigma : nat \to ([\sigma] \to [\sigma])$ que se comporta como la función homónima de Haskell.

Ejercicio 24 ★

Se desea extender el cálculo lambda tipado para tener un mayor control sobre el proceso de reducción. Para esto, se introducen expresiones capaces de detener la reducción de un término, o de continuar una reducción que estaba detenida.

El conjunto de tipos será: σ ::= ... | $\texttt{det}(\sigma)$ donde $\texttt{det}(\sigma)$ es el tipo de los términos que resultan de detener la reducción de términos de tipo σ.

El conjunto de términos será: M ::= ... | $\texttt{detener}(M)$ | $\texttt{continuar}(M)$

El comportamiento de estas expresiones es el siguiente: sea M un término tipable cualquiera, $\texttt{detener}(M)$ detiene la reducción de M. Es decir, no reduce por más que M pueda reducirse. Por otro lado, si N es un término detenido, $\texttt{continuar}(N)$ reanuda la reducción de N.

Por ejemplo, $\texttt{continuar}((\lambda x\colon \texttt{det}(\mathsf{Nat}).x)\ \texttt{detener}(\mathsf{pred}(\mathsf{succ}(0)))) \to \texttt{continuar}(\texttt{detener}(\mathsf{pred}(\mathsf{succ}(0)))) \to \mathsf{pred}(\mathsf{succ}(0)) \to 0$.

Además, las funciones que esperan argumentos detenidos, pueden recibir argumentos del tipo correspondiente sin detener. En ese caso, en lugar de reducir el argumento hasta obtener un valor, lo detienen. Esto permite definir funciones que toman parámetros por nombre (call-by-name). Por ejemplo:

$(\lambda x\colon \texttt{det}(\mathsf{Nat}).\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \texttt{continuar}(x)\ \mathsf{else}\ 0)\ \mathsf{succ}(\mathsf{pred}(\mathsf{succ}(0))) \to$
$(\lambda x\colon \texttt{det}(\mathsf{Nat}).\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \texttt{continuar}(x)\ \mathsf{else}\ 0)\ \texttt{detener}(\mathsf{succ}(\mathsf{pred}(\mathsf{succ}(0)))) \to$
$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \texttt{continuar}(\texttt{detener}(\mathsf{succ}(\mathsf{pred}(\mathsf{succ}(0))))))\ \mathsf{else}\ 0 \to$
$\texttt{continuar}(\texttt{detener}(\mathsf{succ}(\mathsf{pred}(\mathsf{succ}(0))))) \to \mathsf{succ}(\mathsf{pred}(\mathsf{succ}(0))) \to \mathsf{succ}(0)$.

1. Introducir las reglas de tipado para la extensión propuesta.

2. Exhibir la derivación de tipado para el siguiente juicio:
$\{y\colon \mathsf{Bool}\} \triangleright (\lambda x\colon \texttt{det}(\mathsf{Bool}).\mathsf{if}\ y\ \mathsf{then}\ \texttt{continuar}(x)\ \mathsf{else}\ \mathsf{false})\ \mathsf{isZero}(0)\ :\ \mathsf{Bool}$.

3. Indicar formalmente cómo se modifica el conjunto de valores, y dar la semántica operacional de a un paso para la extensión propuesta. Notar que puede ser necesario modificar alguna de las reglas preexistentes.

Práctica N° 3 - Inferencia de Tipos

Aclaraciones:

- Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.
- Usaremos las expresiones de tipos y términos vistas en clase, con los tipos Bool, Nat y funciones ya definidos.
- Para esta práctica será necesario utilizar los axiomas y reglas de tipado e inferencia vistos en clase (tanto en las teóricas como en las prácticas).
- Siempre que se pide definir extensiones, se asume que el algoritmo de unificación (`MGU`) y el de borrado (`Erase`) ya se encuentran correctamente extendidos, de manera que sólo es necesario extender el algoritmo $\mathbb{W}$ (también conocido como `Principal Typing`).

Gramáticas a tener en cuenta:

- Términos **anotados**
  $M ::= x \mid \lambda x\colon \sigma.M \mid M\ M \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid 0 \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{isZero}(M)$ Donde la letra x representa un *nombre de variable* arbitrario. Tales nombres se toman de un conjunto infinito dado $\mathfrak{X} = \{w, w_1, w_2, \ldots, x, x_1, x_2, \ldots, y, y_1, y_2, \ldots, f, f_1, f_2, \ldots\}$
- Términos **sin anotaciones**
  $M' ::= x \mid \lambda x.M' \mid M'\ M' \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{if}\ M'\ \mathsf{then}\ M'\ \mathsf{else}\ M' \mid 0 \mid \mathsf{succ}(M') \mid \mathsf{pred}(M') \mid \mathsf{isZero}(M')$
- Tipos
  $\sigma ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \sigma \to \sigma \mid \mathbf{s}$
  Donde la letra $\mathbf{s}$ representa una *variable de tipos* arbitraria. Tales nombres se toman de un conjunto infinito dado $\mathfrak{T} = \{\mathbf{s}, \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{t}, \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots\}$

Ejercicio 1

Determinar qué expresiones son sintácticamente válidas y, para las que sean, indicar a qué gramática pertenecen.

I. $\lambda x\colon \mathsf{Bool}.\mathsf{succ}(x)$
II. $\lambda x.\mathsf{isZero}(x)$
III. $\mathbf{s} \to \sigma$
IV. $\mathit{Erase}(f\ y)$
V. $\mathbf{s}$
VI. $\mathbf{s} \to (\mathsf{Bool} \to \mathbf{t})$
VII. $\lambda x\colon \mathbf{s}_1 \to \mathbf{s}_2.\mathsf{if}\ 0\ \mathsf{then}\ \mathsf{True}\ \mathsf{else}\ 0\ \mathsf{succ}(\mathsf{True})$
VIII. $\mathit{Erase}(\lambda f\colon \mathsf{Bool} \to \mathbf{s}.\lambda y\colon \mathsf{Bool}.f\ y)$

Ejercicio 2

Determinar el resultado de aplicar la sustitución `S` a las siguientes expresiones

I. $S = \{\mathbf{t} \leftarrow \mathsf{Nat}\} \qquad S(\{x : \mathbf{t} \to \mathsf{Bool}\})$
II. $S = \{\mathbf{t}_1 \leftarrow \mathbf{t}_2 \to \mathbf{t}_3,\ \mathbf{t} \leftarrow \mathsf{Bool}\} \quad S(\{x : \mathbf{t} \to \mathsf{Bool}\}) \triangleright S(\lambda x\colon \mathbf{t}_1 \to \mathsf{Bool}.x)\colon S(\mathsf{Nat} \to \mathbf{t}_2)$

Ejercicio 3 ★

Determinar el resultado de aplicar el `MGU` ("most general unifier") sobre las ecuaciones planteadas a continuación. En caso de tener éxito, mostrar la sustitución resultante.

I. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathbf{t}_2 \doteq \mathsf{Nat} \to \mathsf{Bool}\}$

II. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathbf{t}_2 \doteq \mathbf{t}_3\}$

III. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathbf{t}_2 \doteq \mathbf{t}_2\}$

IV. $\texttt{MGU}\ \{(\mathbf{t}_2 \to \mathbf{t}_1) \to \mathsf{Bool} \doteq \mathbf{t}_2 \to \mathbf{t}_3\}$

V. $\texttt{MGU}\ \{\mathbf{t}_2 \to \mathbf{t}_1 \to \mathsf{Bool} \doteq \mathbf{t}_2 \to \mathbf{t}_3\}$

VI. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathsf{Bool} \doteq \mathsf{Nat} \to \mathsf{Bool}, \mathbf{t}_1 \doteq \mathbf{t}_2 \to \mathbf{t}_3\}$

VII. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathsf{Bool} \doteq \mathsf{Nat} \to \mathsf{Bool}, \mathbf{t}_2 \doteq \mathbf{t}_1 \to \mathbf{t}_1\}$

VIII. $\texttt{MGU}\ \{\mathbf{t}_1 \to \mathbf{t}_2 \doteq \mathbf{t}_3 \to \mathbf{t}_4, \mathbf{t}_3 \doteq \mathbf{t}_2 \to \mathbf{t}_1\}$

Ejercicio 4

Unir con flechas los tipos que unifican entre sí (entre una fila y la otra). Para cada par unificable, exhibir el *mgu* ("most general unifier").

$$\mathbf{t} \to \mathbf{u} \qquad \mathsf{Nat} \qquad \mathbf{u} \to \mathsf{Bool} \qquad \mathbf{a} \to \mathbf{b} \to \mathbf{c}$$

$$\mathbf{t} \qquad \mathsf{Nat} \to \mathsf{Bool} \qquad (\mathsf{Nat} \to \mathbf{u}) \to \mathsf{Bool} \qquad \mathsf{Nat} \to \mathbf{u} \to \mathsf{Bool}$$

Ejercicio 5

Decidir, utilizando el método del árbol, cuáles de las siguientes expresiones son tipables. Mostrar qué reglas y sustituciones se aplican en cada paso y justificar por qué no son tipables aquéllas que fallan.

I. $\lambda z.\ \text{if } z \text{ then } 0 \text{ else } \mathsf{succ}(0)$

II. $\lambda y.\ \text{succ}((\lambda x.x)\ y)$

III. $\lambda x.\ \text{if isZero}(x) \text{ then } x \text{ else (if } x \text{ then } x \text{ else } x)$

IV. $\lambda x.\lambda y.\ \text{if } x \text{ then } y \text{ else succ}(0)$

V. $\text{if } \mathsf{True} \text{ then } (\lambda x.\ 0)\ 0 \text{ else } (\lambda x.\ 0)\ \mathsf{False}$

VI. $(\lambda f.\ \text{if } \mathsf{True} \text{ then } f\ 0 \text{ else } f\ \mathsf{False})\ (\lambda x.\ 0)$

VII. $\lambda x.\lambda y.\lambda z.\ \text{if } z \text{ then } y \text{ else } \mathsf{succ}(x)$

VIII. $\text{fix}\ (\lambda x.\ \text{pred}(x))$

Para el punto VIII, asumir extentido el algoritmo de inferencia con $\mathbb{W}(\text{fix}) = \emptyset \triangleright \text{fix}_{\mathbf{a}} : (\mathbf{a} \to \mathbf{a}) \to \mathbf{a}$ donde $\mathbf{a}$ es una variable fresca.

Ejercicio 6 ★

Utilizando el árbol de inferencia, inferir el tipo de las siguientes expresiones o demostrar que no son tipables. En cada paso donde se realice una unificación, mostrar el conjunto de ecuaciones a unificar y la sustitución obtenida como resultado de la misma.

- $\lambda x.\ \lambda y.\ \lambda z.\ (z\ x)\ y\ z$
- $\lambda x.\ x\ (w\ (\lambda y.w\ y))$
- $\lambda x.\lambda y.\ xy$
- $\lambda x.\lambda y.\ yx$
- $\lambda x.(\lambda x.\ x)$
- $\lambda x.(\lambda y.\ y)x$
- $(\lambda z.\lambda x.\ x\ (z\ (\lambda y.\ z)))\ \mathsf{True}$

Ejercicio 7 *(Numerales de Church)*

Indicar tipos σ y τ apropiados de modo que los términos de la forma $\lambda y : \sigma.\lambda x : \tau.y^n(x)$ resulten tipables para todo n natural. El par (σ, τ) debe ser el mismo para todos los términos. Observar si tienen todos el mismo tipo. Notación: $M^0(N) = N, M^{n+1}(N) = M(M^n(N))$. *Sugerencia:* empezar haciendo inferencia para $n = 2$ – es decir, calcular $\mathbb{W}(\lambda y.\lambda x.y(yx))$ – y generalizar el resultado.

Ejercicio 8

I. Utilizar el algoritmo de inferencia sobre la siguiente expresión: $\lambda y.(x\ y)\ (\lambda z.x_2)$

II. Una vez calculado, demostrar (utilizando chequeo de tipos) que el juicio encontrado es correcto.

III. ¿Qué ocurriría si x_2 fuera x?

Ejercicio 9 ★

Tener en cuenta un nuevo tipo par definido como: $\sigma ::= \ldots \mid \sigma \times \sigma$

Con expresiones nuevas definidas como: $M ::= \ldots \mid \langle M, M\rangle \mid \pi_1(M) \mid \pi_2(M)$

Y las siguientes reglas de tipado:

$$\frac{\Gamma \rhd M : \sigma \quad \Gamma \rhd N : \tau}{\Gamma \rhd \langle M, N\rangle : \sigma \times \tau} \qquad \frac{\Gamma \rhd M : \sigma \times \tau}{\Gamma \rhd \pi_1(M) : \sigma} \qquad \frac{\Gamma \rhd M : \sigma \times \tau}{\Gamma \rhd \pi_2(M) : \tau}$$

I. Adaptar el algoritmo de inferencia para que funcione sobre esta versión extendida.

II. Tipar la expresión $(\lambda$f.$\langle$f,$\underline{2}\rangle)$ $(\lambda$x.x $\underline{1})$ utilizando la versión extendida del algoritmo.

III. Intentar tipar la siguiente expresión utilizando la versión extendida del algoritmo.
$(\lambda f.\langle f\ \underline{2},\ f\ \mathsf{True}\rangle)\ (\lambda x.x)$
Mostrar en qué punto del mismo falla y por qué motivo.

Ejercicio 10

a) Extender el sistema de tipado y el algoritmo de inferencia con las reglas necesarias para introducir los tipos `Either` σ σ y
`Maybe` σ, cuyos términos son análogos a los de Haskell.

b) Utilizando estas reglas y el método del árbol, tipar la expresión:
λx`.if` x `then Just (Left 0) else Nothing`

Ejercicio 11 ★

a) Extender el algoritmo de inferencia para soportar la inferencia de tipos de árboles binarios. En esta extensión del algoritmo sólo se considerarán los *constructores* del árbol.

La sintaxis de esta extensión es la siguiente:

$\sigma ::= ... \mid AB_\sigma \qquad M ::= ... \mid Nil_\sigma \mid Bin(M, N, O)$

Y sus reglas de tipado, las siguientes:

$$\frac{}{\Gamma \rhd Nil_\sigma : AB_\sigma} \qquad \frac{\Gamma \rhd M : AB_\sigma \quad \Gamma \rhd O : AB_\sigma \quad \Gamma \rhd N : \sigma}{\Gamma \rhd Bin(M, N, O) : AB_\sigma}$$

Nota: la función *Erase*, que elimina la información de tipos que el inferidor se encargará de inferir, se extiende de manera acorde para la sintaxis nueva:

$$\begin{aligned} Erase(Nil_\sigma) &= Nil \\ Erase(Bin(M, N, O)) &= Bin(Erase(M), Erase(N), Erase(O)) \end{aligned}$$

Recordar que una entrada válida para el algoritmo es un pseudo término con la información de tipos eliminada. Por ejemplo:

$$(\lambda x.Bin(Nil, 5, Bin(Nil, x, Nil)))\ 5$$

b) Escribir la regla de tipado para el case de árboles binarios, y la regla análoga en el algoritmo de inferencia.

Ejercicio 12 ★

Extender el algoritmo de inferencia $\mathbb{W}$ para que soporte el tipado del *switch* de números naturales, similar al de C o C++. La extensión de la sintaxis es la siguiente:

$$M = \ldots \mid \texttt{switch}\ M\ \{\texttt{case}\ \underline{n_1} :\ M_1\ \ \ldots\ \texttt{case}\ \underline{n_k} :\ M_k\ \texttt{default} :\ M_{k+1}\}$$

donde cada $\underline{n_i}$ es un numeral (un *valor* de tipo Nat, como 0, $\mathsf{succ}(0)$, $\mathsf{succ}(\mathsf{succ}(0))$, etc.). Esto forma parte de la sintaxis y no hace falta verificarlo en el algoritmo.
La regla de tipado es la siguiente:

$$\frac{\Gamma \rhd M : \mathsf{Nat} \quad \forall i, j (1 \leq i, j \leq k \wedge i \neq j \Rightarrow n_i \neq n_j) \quad \Gamma \rhd N_1 : \sigma \ \ldots\ \Gamma \rhd N_k : \sigma \quad \Gamma \rhd N : \sigma}{\Gamma \rhd \texttt{switch}\ M\ \{\texttt{case}\ \underline{n_1} :\ N_1\ \ldots\ \texttt{case}\ \underline{n_k} :\ N_k\ \texttt{default} : N\} : \sigma}$$

Por ejemplo, una expresión como:

$$\lambda x.\ \texttt{switch}\ (x)\ \{\texttt{case}\ 0 :\ \mathsf{True}\ \texttt{default} :\ \mathsf{False}\}$$

debería tipar a $\mathsf{Nat} \to \mathsf{Bool}$. En cambio, la expresión:

$$\texttt{switch}\ \underline{3}\ \{\texttt{case}\ \underline{1} :\ \underline{1}\ \texttt{case}\ \underline{2} :\ 0\ \texttt{default} :\ \mathsf{False}\}$$

no tiene tipo, pues entre los casos hay números y booleanos. Y finalmente, la expresión:

$$\texttt{switch}\ \underline{3}\ \{\texttt{case}\ \underline{1} :\ \underline{1}\ \texttt{case}\ \underline{2} :\ \underline{2}\ \texttt{case}\ \underline{1} :\ \underline{3}\ \texttt{default} :\ 0\}$$

tampoco tiene tipo, ya que el número 1 se repite entre los casos.

Ejercicio 13

En este ejercicio extenderemos el algoritmo de inferencia para soportar operadores binarios. Dichos operadores se comportan de manera similar a las funciones, excepto que siempre tienen 2 parámetros y su aplicación se nota de manera infija. Para esto extenderemos la sintaxis y el sistema de tipos del cálculo lambda tipado de la siguiente manera:

$$M\ ::=\ \ldots\ \mid\ \varphi x : \sigma\ y : \tau.M\ \mid\ \langle M\ N\ O\rangle \qquad \sigma\ ::=\ \ldots\ \mid\ \mathrm{Op}(\sigma, \tau \to \upsilon)$$

Aquí φ es el constructor de operadores que liga las variables x (parámetro anterior al operador) e y (parámetro posterior) y $\langle M\ N\ O\rangle$ es la aplicación del operador N a los parámetros M y O (lo ponemos entre $\langle$ y $\rangle$ para evitar problemas de ambigüedad con la aplicación estándar). $\mathrm{Op}(\sigma, \tau \to \upsilon)$, por otro lado, representa el tipo de los operadores cuyo parámetro anterior es de tipo σ, el posterior de tipo τ y dan como resultado un tipo υ.

Las reglas de tipado que se incorporan son las siguientes:

$$\frac{\Gamma \cup \{x : \sigma,\ y : \tau\} \rhd M : \upsilon}{\Gamma \rhd \varphi x : \sigma\ y : \tau.M : \mathrm{Op}(\sigma, \tau \to \upsilon)}$$

$$\frac{\Gamma \rhd M : \sigma \qquad \Gamma \rhd N : \mathrm{Op}(\sigma, \tau \to \upsilon) \qquad \Gamma \rhd O : \tau}{\Gamma \rhd \langle M\ N\ O \rangle : \upsilon}$$

I. Dar la extensión al algoritmo necesaria para soportar el tipado de las nuevas expresiones. Recordar que el parámetro de entrada es un término **sin anotaciones de tipos**.

II. Aplicar el algoritmo extendido con el método del árbol para tipar: $\langle (\lambda x.\mathsf{succ}(x))\ (\varphi xy.xy)\ 0 \rangle$

Ejercicio 14

Considerar el algoritmo de inferencia extendido para soportar listas:

$\mathbb{W}([\,]) \stackrel{def}{=} \emptyset \rhd [\,]_{\mathbf{t}} : [\mathbf{t}]$, con $\mathbf{t}$ variable fresca.

$\mathbb{W}(M : N) \stackrel{def}{=} S\Gamma_1 \cup S\Gamma_2 \rhd S(U : V) : [S\sigma]$, con:

$\mathbb{W}(M) = \Gamma_1 \rhd U : \sigma$

$\mathbb{W}(N) = \Gamma_2 \rhd V : \tau$

$S = \mathtt{MGU}(\{\tau \doteq [\sigma]\} \cup \{\alpha \doteq \beta / x : \alpha \in \Gamma_1 \wedge x : \beta \in \Gamma_2\})$

I. Extender el algoritmo de inferencia para soportar expresiones de la forma "$\exists x$ in M/N".

$$\frac{\Gamma \cup \{x : \sigma\} \rhd N : \mathsf{Bool} \qquad \Gamma \rhd M : [\sigma]}{\Gamma \rhd \exists x \text{ in } M/N : \mathsf{Bool}}$$

II. Aplicar el algoritmo extendido con el método del árbol para tipar las siguientes expresiones. Si alguna de ellas no tipa, indicar el motivo.

i) $(\lambda x.\exists y \text{ in } x/y)(0 : [\,])$
ii) $(\lambda x.\exists y \text{ in } x/y)(\mathrm{iszero}(z) : [\,])$
iii) $\exists x \text{ in } [\,]/\mathsf{True}$
iv) $\exists x \text{ in } [\,]/(\lambda y.\mathsf{True})$
v) $\exists x \text{ in } (0 : [\,])/\mathrm{iszero}(x)$

Ejercicio 15

Se desea diseñar un algoritmo de inferencia de tipos para el cálculo λ extendido con fórmulas proposicionales de la siguiente manera:

$M ::= \cdots \mid \neg M \mid M \supset M \mid \mathsf{esTautología}(M)$

$\sigma := \cdots \mid \mathsf{Prop}$

Las reglas de tipado son:

$$\frac{\Gamma \rhd M : \mathsf{Prop}}{\Gamma \rhd \neg M : \mathsf{Prop}}\text{TNEG} \qquad \frac{\Gamma \rhd M : \mathsf{Prop} \qquad \Gamma \rhd N : \mathsf{Prop}}{\Gamma \rhd M \supset N : \mathsf{Prop}}\text{TIMP}$$

$$\frac{\Gamma, x_1 : \mathsf{Prop}, \ldots, x_n : \mathsf{Prop} \rhd M : \mathsf{Prop} \qquad \mathsf{fv}(M) = \{x_1, \ldots, x_n\}}{\Gamma \rhd \mathsf{esTautología}(M) : \mathsf{Bool}}\text{TTAUT}$$

Notar que $\mathsf{esTautología}(M)$ liga **todas** las variables libres de M. Por ejemplo, $\mathsf{esTautología}(p \supset (q \supset p))$ es un término cerrado y bien tipado (de tipo Bool).

I. Extender el algoritmo de inferencia para admitir las expresiones incorporadas al lenguaje, de tal manera que implemente las reglas de tipado tNeg, tImp y tTaut.

II. Aplicar el algoritmo extendido con el método del árbol para tipar las siguientes expresiones (exhibiendo siempre las sustituciones utilizadas). Si alguna de ellas no tipa, indicar el motivo.

- $\lambda y.\neg((\lambda x.\neg x)(y \supset y))$
- $(\lambda x.\mathsf{esTautología}(\mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z))\mathsf{True}$

Ejercicio 16 ★

En este ejercicio modificaremos el algoritmo de inferencia para incorporar la posibilidad de utilizar letrec en nuestro cálculo.

- $M ::= \ldots \mid \mathsf{letrec}\ f = M\ \mathsf{in}\ N$
- letrec permite por ejemplo representar el factorial de 10 de la siguiente manera:

$$\mathsf{letrec}\ f = \ (\lambda\, x : \mathsf{Nat}\,.\,\mathsf{if}\ \mathsf{isZero}(x)\ \mathsf{then}\ 1\ \mathsf{else}\ x \times f\ (\mathsf{pred}(x)))\ \mathsf{in}\ f\ 10$$

Para ello se agrega la siguiente regla de tipado:

$$\frac{\Gamma \cup \{f : \pi \to \tau\} \rhd M : \pi \to \tau \qquad \Gamma \cup \{f : \pi \to \tau\} \rhd N : \sigma}{\Gamma \rhd \mathsf{letrec}\ f = M\ \mathsf{in}\ N : \sigma}$$

Suponiendo que se propone el siguiente pseudocódigo:

$\mathbb{W}(\mathsf{letrec}\ f = M\ \mathsf{in}\ N) \stackrel{def}{=} \Gamma \rhd S\,(\mathsf{letrec}\ f = M'\ \mathsf{in}\ N') : S\,\sigma$
donde

- $\mathbb{W}(M) = \Gamma_1 \rhd M' : \pi \to \tau$
- $\mathbb{W}(N) = \Gamma_2 \rhd N' : \sigma$
- $\tau_1 = \rho / f : \rho \in \Gamma_1$
- $\tau_2 = \delta / f : \delta \in \Gamma_2$
- $\mathrm{S} = \texttt{MGU}\ \{\tau_1 \doteq \tau_2,\ \text{COMPLETAR}\}$
- $\Gamma = S\,\Gamma_1 \cup S\,\Gamma_2$

I. Explicar cuál es el error en los llamados recursivos. Dar un ejemplo que debería tipar y no lo hace debido a este error.

II. Explicar cuál es el error en el pseudocódigo con respecto la definición de τ_1 y τ_2. Dar un ejemplo que debería tipar y no lo hace debido a este error.

III. El contexto Γ ¿puede contener a f? ¿Es un comportamiento deseable? Mostrar un ejemplo donde esto trae conflictos (ayuda: usar letrec dentro de un término más grande).

IV. Reescribir el pseudocódigo para que funcione correctamente (corregir los errores y completar la definición de S).