Tarea 1: Haskell (ツ)

Carlo Miguel Herrera Di Giacinto (18-10451)

Parte I. Implementación

1. ***(3 ptos)*** Considere la estructura de datos `Conjunto`, que representa conjuntos potencialmente infinitos. Para ser capaces de conocer la pertenencia de elementos en dichos conjuntos (aún siendo infinitos) los mismos no deben representarse como enumeraciones explícitas de sus elementos, si no como la función que sabe distinguir los elementos que pertenecen al mismo (función característica del conjunto). Por ejemplo, el conjunto de los números enteros pares puede representarse como la función: `(\num -> even num)`.

A continuación se presenta la definición del tipo de datos `Conjunto` en Haskell:

```
type Conjunto a = a -> Bool
```

Tomando en cuenta la definición anterior, debe implementar entonces cada una de las siguientes funciones (válidas independientemente del tipo concreto que tome `a` y sin cambiar sus firmas).

```
a) miembro :: Conjunto a -> a -> Bool
miembro f = f

b) vacio :: Conjunto a
vacio _ = False

c) singleton :: (Eq a) => a -> Conjunto a
singleton x = (\y -> y == x)

d) desdeLista :: (Eq a) => [a] -> Conjunto a
desdeLista x = (\y -> y `elem` x)

e) complemento :: Conjunto a -> Conjunto a
complemento f = (\x -> miembro f x == False)

f) union:: Conjunto a -> Conjunto a -> Conjunto a
union f g = (\x -> miembro f x || miembro g x)

g) interseccion:: Conjunto a -> Conjunto a -> Conjunto a
interseccion f g = (\x -> miembro f x && miembro g x)

h) diferencia :: Conjunto a -> Conjunto a -> Conjunto a
diferencia f g = (\x -> miembro f x && miembro g x == False)

i) transformar :: (b->a) -> Conjunto a -> Conjunto b
transformar t f = (\x -> f (t x))
```

Parte II. Investigación

a) ¿Cual es la forma normalizada para la expresión:

> *(λx. λy. x y y) (λz. z O) L*

Aplicamos la sustitucion de λx sobre (λz.z O)

> *(λy. (λz. z O) y y) L*

Hacemos la sustitucion de λz que recibe un y

> *(λy. [(λz. z O) y] y) L =* ***(λy.(y O) y) L***

Realizamos la sustitucion de y

> *(λy.y O) L*

Finalmente sustituimos L en y, obteniendo la forma normalizada de la expresion.

> ***L O***

c) Considere una evaluación para una λ–expresión de la forma ((λx . E) F). ¿Qué cambios haría a la semántica formal de la función eval para este caso, si se permitiesen identificadores repetidos? *[Considere, como ejemplo, lo que ocurre al evaluar la siguiente expresión: (λx . (λy . x y)) y]*

> Un problema en la evaluación de expresiones lambda de la forma ((λx . E) F) es una posible colisión de nombres. Un cambio que se puede hacer sobre la funcion `eval` es incluir una revisión sobre los indentificadores de las expresiones E y F de tal manera que encaso de haber coincidencias se incluya un renombramiento de identificadores de tal manera que no se alteren inesperadamente los valores de los identificadores.

2. ***(3 ptos)*** Considere las siguientes funciones:

```
id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x

subs:: (a -> b -> c) -> (a -> b) -> a -> c
subs x y z = x z (y z)
```

a) Evalúe la expresión: `subs (id const) const id` . No evalúe la expresión en Haskell, pues si el resultado es una función no podrá imprimirlo. Evalúe la expresión a mano y exponga el resultado en término de las funciones antes propuestas (utilice evaluación normal: primero la función, luego los argumentos).

> Al evaluar la expresión obtenemos

```
ex : subs (id const) const id = (id const) id (const id)
```

> Esto nos deja
>
> - Una funcion f = id const
> - Una funcion g = id
> - Una funcion h = const id

> f resulta de evaluar const en id, lo que nos devuelve const.
> El resultado sera entonces f que recibe dos paramateros: la funcion g y la funcion h h es el resultado de evaluar const con id, que resulta en una funcion auxiliar que espera recibir el segundo parametro de const.

> Por lo tanto, la expresion resulta en una funcion que espera dos parametros: uno que espera h y otro el que evalua la funcion g, que es la funcion que retorna f.

b) Proponga una expresión (únicamente compuesta por las funciones definidas anteriormente), que no esté en forma normal, cuya evaluación resulte en la misma expresión y por lo tanto nunca termine.

```
subs id id (subs id id)
```

Esta evaluación nos devuelve a si misma en tres pasos:

> P0: `subs id id (subs id id)` P1: `id (subs id id) (id (subs id id))` P2: `subs id id (id (subs id id))` P3: `subs id id (subs id id)`

Evaluaciones de esta forma son un ejemplo de la indecibilidad del cálculo combinatorio.

c) Reimplemente la función `id` en términos de `const` y `subs` . ***(Pista:*** *puede utilizar el tipo* `unitario ()` *para representar un argumento del cual no importa su valor, pero que igual debe ser pasado como parámetro a una función.)*

```
id x = (subs const const) x
```

d) Discuta la relación entre las funciones propuestas y el cálculo de combinadores SKI

> Las funciones propuestas en el enunciado coinciden con la estructura de los combinadores SKI. La funcion `id` coincide con el funcionamiento del combinador I, la funcion `const` matchea con el combinador K y la funcion `subs` con el combinador S.