Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Septiembre–Diciembre 2012

Tarea I: Haskell – *Solución*

(5 pts)

Implementación:

(2.5 pts) – 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. 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) (0.4 pts) – `miembro :: Conjunto a -> a -> Bool`

Debe devolver la pertenencia en el conjunto proporcionado, de un elemento dado.

Solución:

`miembro = id`

Nótese que la solución mas sencilla para esta función sería `miembro c x = c x` pero, como hemos visto en clase, se puede eliminar ese último argumento de ambos lados resultando en `miembro c = c`. Ahora, recordemos que `id x = x`, por lo que podemos escribir `miembro c = id c` y eliminar nuevamente el último argumento de ambos lados.

b) (0.3 pts) – `vacio :: Conjunto a`

Debe devolver un conjunto vacío.

Solución:

`vacio = const False`

Queremos que la función evalúe a `False` para cualquier posible argumento, dado que nada pertenece al conjunto vacío.

c) (0.3 pts) – `singleton :: (Eq a) => a -> Conjunto a`

Debe devolver un conjunto que contenga únicamente al elemento proporcionado.

Solución:

`singleton = (==)`

Dado un elemento (que será el singleton), solo pertenecerá algún otro si el mismo es igual al primero.

d) (0.3 pts) – `desdeLista :: (Eq a) => [a] -> Conjunto a`

Debe devolver un conjunto que contenga a todos los elementos de la lista proporcionada.

Solución:

```
desdeLista = flip elem
```

Dado una lista y el elemento que se quiere saber pertenece al conjunto, se debe verificar la contencion en la misma. La función `elem` del preludio ya hace este trabajo, pero toma sus argumentos en orden inverso (esto se arregla con la función de orden superior `flip`).

e) (0.3 pts) – `complemento :: Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga únicamente todos los elementos que no estén en el conjunto proporcionado (pero que sean del mismo tipo).

Solución:

```
complemento = (not .)
```

Si una función sabe distinguir los elementos de un conjunto, no hay sino que componerla luego con la negación para obtener su complemento.

f) (0.3 pts) – `union :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga todos los elementos de cada conjunto proporcionado.

Solución:

```
union s t x = s x || t x
```

Dados dos conjuntos, un elemento pertenece a la unión si pertenece a alguno de los dos.

g) (0.3 pts) – `interseccion :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga solo los elementos que estén en los dos conjuntos proporcionados.

Solución:

```
interseccion s t x = s x && t x
```

Dados dos conjuntos, un elemento pertenece a la intersección si pertenece a ambos.

h) (0.3 pts) – `diferencia :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga los elementos del primer conjunto proporcionado, que no estén en el segundo.

Solución:

```
diferencia s t = interseccion s (complemento t)
```

Dados dos conjuntos, un elemento pertenece a la diferencia si pertenece a la intersección del primero con el complemento del segundo.

Investigación:

(2.5 pts) – La programación funcional está basada en el Lambda Cálculo, propuesto por Alonzo Church. Dicho cálculo está basado en llamadas λ–*expresiones*. Estas expresiones toman una de tres posibles formas:

x	–	donde x es un identificador.
$\lambda x\ .\ E$	–	donde x es un identificador y E es una λ–*expresión*, es llamada λ–*abstracción*.
$E\ F$	–	donde E y F son λ–*expresiones*, es llamada *aplicación funcional*.

Se dice que una λ–*expresión* está normalizada si para cada sub–expresión que contiene, correspondiente a una aplicación funcional, la expresión del lado izquierdo no evalúa a una λ–*abstracción*. De lo contrario, dicha expresión aún puede evaluarse. A continuación se presenta una semántica formal para la evaluación de λ–*expresiones*, suponiendo que todos los identificadores presentes en las mismas son diferentes:

$$
\begin{array}{lcl}
eval(x) & = & x. \\
eval(\lambda x\ .\ E) & = & \lambda x.eval(E) \\
eval(x\ F) & = & x\ eval(F) \\
eval((\lambda x\ .\ E)\ F) & = & eval(E)\ [x := eval(F)] \\
eval((E\ F)\ G) & = & eval(eval(E\ F)\ eval(G))
\end{array}
$$

Tomando esta definición en cuenta, conteste las siguientes preguntas:

a) (0.5 pts) – ¿Cual es la forma normalizada para la expresión: $(\lambda x\ .\ \lambda y\ .\ x\ y\ y)\ (\lambda z\ .\ z\ O)\ L$?

Solución:

`L O L`

b) (1 pt) – Considere una aplicación funcional, de la forma $E\ F$. ¿Existen posibles expresiones E y F, tal que el orden en el que se evalúen los mismos sea relevante (arroje resultados diferentes)? De ser así, proponga tales expresiones E y F. En cualquier caso, justifique su respuesta.

Solución:

Nótese que en la expresión propuesta deben realizarse tres evaluaciones, una para E, una para F y otra para la aplicación funcional $E\ F$. Para demostrar que el orden es relevante, propondremos E como una función constante (que siempre evalúa en c) y F como una función que nunca termine. De lograr hacer esto, resultaría que:

- Evaluar E y la aplicación funcional antes, resultaría en que F no necesitara evaluarse y por ende el programa terminaría evaluando directamente a la constante c.
- Evaluar F antes, resultaría en una expresión que nunca termine de evaluarse y por ende la aplicación funcional no podría culminarse.

Consideremos como E y F entonces, las siguientes λ–*expresiones*:

- $E = (\lambda x\ .\ c)$
- $F = ((\lambda x\ .\ x\ x)\ (\lambda x\ .\ x\ x))$

Nótese que al evaluar F solo se obtiene una copia de la expresión original, por lo que la evaluación no puede terminar. Este comportamiento está muy relacionado a aquel del combinador Y, con el cual es posible hacer funciones recursivas en Lambda Cálculo.

c) (1 pt) – Considere una evaluación para una λ–*expresión* de la forma $((\lambda x\ .\ E)\ F)$. ¿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: $(\lambda x\ .\ (\lambda y\ .\ x\ y))\ y$]

Solución:

Se debe incorporar un mecanismo que sustituya los nombres que pudieran entrar en conflicto. Aunque existen maneras más compactas, una posible solución es la de replantear la expresión completa, sustituyendo cada variable que forme parte una una λ–*abstracción* por una variable nueva (que no aparezca en ningún otro lugar de la expresión). Esto debe hacerse con cuidado, pues una expresión de la forma $(\lambda x\ .\ (\lambda y\ .\ (\lambda x\ .\ x\ y)\ z\ x))$ puede resultar problemática. Las dos ocurrencias de la variable x corresponden a dos identificadores distintos.

A este proceso de sustituir los nombres de variables asociados a λ–*expresiones* se le conoce como el proceso de α–*sustitución*. A la evaluación de una aplicación funcional se le conoce como el proceso de β–*reducción*. Estos dos procesos (en ocasiones combinados con la η–*conversión*, que permite transformar funciones en otras equivalentes) conforman la base para el poder expresivo del Lambda Cálculo.