Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 1^{er} cuat. 2021

Fecha de entrega: 22 de abril de 2021

1. Introducción

Este trabajo consiste en implementar en Haskell y luego testear varias funciones que modelan el comportamiento de fábricas.

Las fábricas van a ser representadas de la siguiente manera:

```
type Fabrica a b = [a] → [b]
```

donde `Fabrica a b` representa una fábrica que toma varias **entradas** de tipo `a` y las transforma en productos de tipo `b`. Los tamaños de las listas no necesariamente se mantienen. Por ejemplo una `Fabrica Carta Mazo` produce un mazo por cada cuarenta cartas. Es importante el orden en que llegan las **entradas** en la lista de entrada. Para que una `Fabrica Ingrediente PostreVigilante` funcione correctamente, la lista de ingredientes debe contener quesos y dulces intercalados, o terminará produciendo un postre con dos capas del mismo ingrediente. Por último, **las fábricas deben poder aceptar listas infinitas como entrada**.

2. Resolver

Ejercicio 1

a) Definir y **dar el tipo** de la función `crearFabricaSimple` que, dada una función `a → b`, devuelve una fábrica que aplique esta función a cado uno de los elementos en la lista de entrada. Notar que para este tipo de fábricas (que llamaremos "fábricas simples"), la cantidad de productos se mantiene respecto a la cantidad de **entradas**.

b) Usando la función `crearFabricaSimple`, definir las siguientes fábricas:

- `neg :: Fabrica Bool Bool`, que dada una lista de booleanos devuelve una lista con la negación de cada uno de ellos.
- `esPar :: Fabrica Int Bool`, que dada una lista de enteros devuelve una lista que indica si cada uno es par.
- `sumarTuplas :: Num a ⇒ Fabrica (a, a) a`, que dada una lista de tuplas devuelve una lista con las sumas de las dos componentes de cada tupla.

Ejercicio 2

Vamos a representar determinados **materiales**[1] que las fábricas pueden procesar de la siguiente forma:

```
data Material a =
  MateriaPrima a |
  Mezclar (Material a) Float (Material a)
```

Dado un tipo cualquiera `a`, `MateriaPrima a` representa un **material** puro, mientras que el constructor `Mezclar` representa la mezcla de dos **materiales**. El `Float` es un número entre 0.0 y 100 que representa el porcentaje del primer material en la mezcla. Llamaremos "elemento base" de un material a los argumentos de tipo `a` dentro de un término de tipo `Material a`.

Por ejemplo, `MateriaPrima True` es un `Material Bool` con elemento base `True` y
`Mezclar (MateriaPrima 5) 50 (MateriaPrima 7)` es un `Material Int` con elementos base `5` y `7`.

Definir y **dar el tipo** de la función `foldMaterial`, que implemente un esquema de recursión estructural (fold) para el tipo de datos `Material a`. Por tratarse de un esquema de recursión, **se puede utilizar recursión explícita** para definir esta función.

Ejercicio 3

Definir la función `crearFabricaDeMaterial :: (a → b) → Fabrica (Material a) (Material b)`, que dada una función devuelve una fábrica cuyo procesamiento consiste en aplicar esta función a los **elementos base** de los **materiales** en la lista de entrada.

Ejercicio 4

a) Definir y **dar el tipo** de la función `secuenciar`, que dadas dos fábricas simples, devuelve otra fábrica simple que consiste en conectar la salida de la primera con la entrada de la segunda (las fábricas pueden ser de cualquier tipo, no necesariamente de **materiales**).

   Por ejemplo, `secuenciar esPar neg [1, 2, 3, 5] = [True, False, True, True]`

b) Cuando dos fábricas simples producen cosas del mismo tipo, se puede paralelizar su producción. De esta forma, dos fábricas simples se convierten en una sola que toma una lista de pares (cada par contiene una **entrada** para cada una de las fábricas originales) y devuelve una única lista que intercala los productos de ambas fábricas.

   Definir y **dar el tipo** de la función `paralelizar` que, dadas dos fábricas simples paralelizables (es decir, que coinciden en el tipo de lo que producen), devuelve una fábrica simple paralela.

   Por ejemplo,
   `paralelizar neg esPar [(True,1),(False,3),(False,1)] = [False, False, True, False, True, False]`

   Sugerencia: definir una función auxiliar `entrelazar` que dada una lista de tuplas de elementos del mismo tipo devuelva una lista con los elementos de las tuplas en el mismo orden. Por ejemplo, `entrelazar [(1,2),(3,4)] = [1,2,3,4]` (recordar que debe funcionar con **listas infinitas**)

[1] Ojo: a partir de ahora cuando hablemos de **materiales** nos referiremos a elementos del tipo que estamos definiendo en este ejercicio, y llamaremos **entradas** a los elementos – de cualquier tipo – que puedan ser ingresados en una fábrica.

Ejercicio 5

a) Definir la función pureza :: (Eq a) ⇒ a → Material a → Float, que dado un elemento base y un **material** calcula el porcentaje del elemento en el material.

Por ejemplo,
verdad = MateriaPrima True
mentira = MateriaPrima False
pureza True (Mezclar (Mezclar verdad 50.0 mentira) 50.0 mentira) = 25.0

b) Definir la función filtrarPorPureza :: (Eq a) ⇒ [Material a] → [(a, Float)] → [Material a] que dada una lista de **materiales** y una lista de tuplas de elementos base y purezas (no necesariamente exhaustiva), devuelva una lista formada por los materiales de la primer lista cuyos elementos cumplan todas las restricciones de pureza en la segunda lista (es decir, que sus elementos base tengan una pureza igual o superior a la especificada por cada tupla en la segunda lista). Asumimos que en la segunda lista no hay dos tuplas con la misma primer coordenada.

Por ejemplo, filtrarPorPureza
[Mezclar verdad 44.5 mentira, Mezclar verdad 80.0 mentira, Mezclar mentira 99.0 verdad]
[(True, 50.0), (False, 1.0)] =
[Mezclar verdad 80.0 mentira]

Ejercicio 6

a) Un emparejador es una fábrica que, en lugar de producir algo, agrupa las **entradas** de a pares.

Definir y **dar el tipo** de emparejador.

Por ejemplo, emparejador [1, 2, 3, 4] = [(1, 2), (3, 4)]

Pista: pueden usar la función ya implementada paresEImpares :: [a] → ([a],[a]), que dada una lista la separa en las listas correspondientes a los elementos en posiciones pares e impares respectivamente.

b) Las fábricas complejas requieren dos unidades de **entradas** para producir una unidad del producto.

Definir la función crearFabricaCompleja :: (a → a → b) → Fabrica a b, que dada una función devuelve la fábrica compleja correspondiente a aplicarla a los pares sucesivos de **entradas**.

Por ejemplo, crearFabricaCompleja (+) [2, 3, 12, −3, 11, 0] = [5, 9, 11]

Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[2] como HSpec[3] permiten hacerlo con facilidad.

En el archivo de esqueleto que proveemos se encuentran tests básicos utilizando *HUnit*. Para correrlos, ejecutar dentro de *ghci*:

```
> :l TP1.hs
[1 of 1] Compiling TP1              ( TP1.hs, interpreted )
Ok, one module loaded.
> tests
```

[2] https://hackage.haskell.org/package/HUnit
[3] https://hackage.haskell.org/package/hspec

Para instalar HUnit usar: `> cabal install hunit` o bien `apt install libghc-hunit-dev`.

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Pautas de Entrega

Todo el código producido por ustedes debe estar en el archivo `TP1.hs` y es el **único** archivo que deben entregar. Cada función (incluso las auxiliares) asociada a los ejercicios debe contar con un conjunto de casos de test que muestren que exhibe la funcionalidad esperada. Para esto se deben agregar casos de test en las funciones *testEjercicio* en el archivo fuente *taller.hs*, agregando todos los tests que consideren necesarios (se incluyen algunos tests de ejemplo). Se debe enviar un e-mail a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del **nombre del grupo**.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto con nombre `TP1.hs` (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar lo ya definido. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en: http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.