Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 2^{do} cuat. 2019

Fecha de entrega: 17 de septiempre

1. Introducción

En este trabajo práctico trabajaremos con proposiciones lógicas definidas en Haskell de la siguiente manera:

```
data Proposition = Var String
                 | Not Proposition
                 | And Proposition Proposition
                 | Or Proposition Proposition
                 | Impl Proposition Proposition
```

donde:

- `Var x` representa a una variable proposicional con nombre `x`.
- `Not q` representa la negación de una proposición `q`.
- `And p q` representa la conjunción de dos proposiciones `p` y `q`.
- `Or p q` representa la disyunción de dos proposiciones `p` y `q`.
- `Impl p q` representa la implicación entre las funciones `p` y `q`.

Por otro lado, dado que la valuación de una proposición es una función que asigna a cada una de sus variables un valor de verdad, la modelaremos como:

```
type Assignment = String → Bool
```

2. Resolver

Ejercicio 1

Definir y dar el tipo de las siguientes funciones. Por tratarse de esquemas de recursión, para definir estas funciones se permite utilizar **recursión explícita**.

- `recProp`, que representa el esquema de recursión primitiva *recr* sobre `Proposition`.
- `foldProp`, que representa el esquema de recursión estructural *foldr* sobre `Proposition`. Notar que puede definirse en términos de `recProp`.

Ejercicio 2

Definir la función `show` para `Proposition` de manera tal que `Proposition` pase a ser una instancia de la clase de tipos `Show` y las proposiciones se muestren por pantalla de la siguiente manera:

```
> Var "p"
p
> Not (Var "p")
¬p
> And (Var "p") (Var "q")
(p ∧ q)
> Or (Var "p") (Var "q")
(p ∨ q)
> Impl (Var "p") (Var "q")
(p ⊃ q)
> Not (And (Impl (Var "p") (Var "q")) (Not (Var "p")))
¬((p ⊃ q) ∧ ¬p)
```

Recordar que la pertenencia a la clase `Show` para un tipo `A` se establece de la siguiente manera:

```
instance Show A where
  show ...
```

Ejercicio 3

Definir la función `assignTrue` que dada una lista de variables proposicionales devuelve una asignación tal que la variable `p` es verdadera si y solo si es una elemento de la lista pasada por parámetro.

Ejercicio 4

Definir la función `eval`, que dadas una valuación y una proposición devuelve el valor de verdad de la proposición para esa valuación.

Por ejemplo:

```
> eval ("p" ==) (Not (Var "p"))
False
> eval (assignTrue ["p","q"]) (Impl (And (Var "p") (Var "r")) (And (Not (Var "q")) (Var "q")))
True
```

Ejercicio 5

Definir las siguientes funciones:

- `elimImpl` que dada una proposición devuelve otra equivalente pero sin ninguna implicación.
- `negateProp` que dada una proposición devuelve una proposición equivalente a su negación, con la restricción de que no se debe introducir el operador `Not` a menos que lo que se esté negando sea una variable proposicional. Para esto, se deben usar las leyes de De Morgan. Por ejemplo:

```
> negateProp (Not $ Var "q")
q

> negateProp (And (Var "p") (Not (Var "q")))
(¬p ∨ q)

> negateProp (Impl (Var "p") (Var "q"))
(p ∧ ¬q)
```

- `nnf` (Negated Normal Form) que dada una proposición devuelve otra equivalente pero en forma normal negada. El conjunto de las fórmulas en forma normal negada (FNN) se define inductivamente como:

  - Las variables proposicionales están en FNN.
  - Si $\varphi, \psi \in$ FNN, entonces $(\varphi \vee \psi), (\varphi \wedge \psi) \in$ FNN. Por ejemplo, $(p \vee \neg p)$ está en FNN, pero $\neg(p \wedge \neg p)$ no.

  En otras palabras, se trata de que las negaciones estén lo más adentro posible. (Notar también que no se admiten implicaciones).

  Algunos ejemplos de pasaje a forma normal negada:

$$\begin{array}{ll} \neg\neg p & \rightsquigarrow p \\ \neg(q \wedge r) & \rightsquigarrow (\neg q \vee \neg r) \\ \neg(p \supset q) & \rightsquigarrow (p \wedge \neg q) \end{array}$$

Ejercicio 6

Definir las siguientes funciones:

- `vars` que dada una proposición devuelve una lista (sin elementos repetidos) con los nombres de las variables que aparecen en ella.
- `parts` que dada una lista (que puede suponerse sin elementos repetidos) devuelve su conjunto de partes. Por ejemplo:

```
> parts [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

- `sat` que dada una proposición devuelve una lista de listas de (nombres de) variables, para cada una de las cuales la valuación que asigna verdadero a sus variables (y solo a ellas) hace verdadera a la proposición. Dicho de otra manera, tiene que valer:

$$ls \in sat\ prop \iff eval\ (assignTrue\ ls)\ prop$$

- `satisfiable` que dada una proposición indica si existe alguna valuación que la hace verdadera.
- `tautology` que dada una proposición indica si la proposición es verdadera para cualquier valuación.
- `equivalent` que dadas dos proposiciones `p` y `q` devuelve verdadero si vale $p \iff q$

Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[1] como HSpec[2] 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 Main             ( tp1.hs, interpreted )
Ok, modules loaded: Main.
> main
```

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

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell 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 (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 se 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.

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

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.