Estructuras Discretas
Práctica 3

Fecha de Entrega: 18 de octubre de 2019

1 Tipos en Haskell

Una de las herramientas más importantes en un lenguaje de programación es la definición de nuevos tipos, estructuras y datos.

Como ya se menciono en clase, Haskell es un lenguaje fuertemente tipado, lo que significa que el tipo con el que definimos alguna variable, no va a poder cambiar duarante el proceso de interpretación de nuestro programa. Por otro lado, Haskell es un lenguaje de tipado no explícito, por lo tanto, no necesitamos especificar el tipo de la variable cada vez que la usemos.

Hay que considerar los siguientes puntos a la hora de definir un tipo:

- Cuando definimos un tipo recursivo, hay que tener cuidado con el orden de la definición. Haskell tomará los argumentos recibidos y va a comparar desde la primera definición de tipo, hasta la última, en caso de no encontrar un patrón, mandará un error al momento de interpretar el código.
- Hay que definir el tipo de acuerdo a la jerarquiía de operaciones, es decir, primero van las definiciones mas simples.

2 Logica proposicional

Considera el siguiente tipo recursivo en Haskell:

```
data Var     = A|B|C|D|E|F|G|H|I|J|K|L|M
              |N|O|P|Q|R|S|T|U|V|W|X|Y|Z deriving (Show, Eq, Ord)
data Formula = Prop Var
              |Neg Formula
              |Formula :&: Formula
              |Formula :|: Formula
              |Formula :=>: Formula
              |Formula :<=>: Formula deriving (Show, Eq, Ord)
infixl 9 :&:
infixl 9 :|:
infixl 7 :=>:
infixl 8 :<=>:
```

Las indicación al final de los datos **deriving Show, Eq, Ord)** hacen referencia a diferentes características que queremos que tenga el tipo de dato que estamos creando.

- Show: Es para decirle a Haskell que el tipo que estamos definiendo se puede representar con una cadena.
- Eq: Es porque necesitamos que los valores que puede tomar algo del tipo a definir son comparables. Es decir, nos permite usar los operadores == y ! = para comparar valores de este tipo.
- Ord: Nos dice que es un tipo ordenado, el orden está dado por nuestra definición. Por ejemplo, en el tipo Var, estamos diciendo que el orden que tendrán nuestras proposiciones es el orden léxicográfico. En el caso de las fórmulas, comenzamos por la proposición atómica Prop Var donde Var es alguna letra mayúscula definida en el tipo Var.

A lo operadores lógicos también les estamos dando una especificación extra, como lo son la precedencia y la asociatividad, de manera que ambas cosas coincidan con lo utilizado en la lógica propocicional.

La precedencia la estamos definiendo con el número que aparece junto a la palabra ***infixl ó infixr***. ***infixl*** le indica a Haskell que es un operador infijo y que la asociatividad es a la izquierda, ***infixr*** le indica a Haskell que la asocitividad del operador es a la derecha.

3 Ejercicios

Para esta práctica, programarás algunas funciones en Haskell:

- **Función Variables de una Fórmula**. Una función recursiva que recibe una fórmula y devuelve el conjunto (lista sin repeticiones) de variables que hay en la fórmula.

  ```
  varList :: Formula -> [Var]
  ```

  Por ejemplo:

  ```
  varList (Prop P :=>: Neg (Prop Q :<=> Prop W :&: Neg (Prop P))) = [Q,W,P]
  ```

- **Función Negación**. Una función recibe una fórmula y devuelve su negación.

  ```
  Negar :: Formula -> Formula
  ```

  Por ejemplo:

  ```
  negar (Prop Q :=>: Prop W :&: Neg (Prop P)) =
  (Prop Q :&: (Neg (Prop W) :|: Prop)) :|:
  ((Prop W :&: Neg (Prop P)) :&: Neg (Prop Q))
  ```

- **Función Equivalencia**. Una función que recibe una fórmula y devuelve su equivalencia lógica, sin implicaciones y bicondicionales, además la negación se encuentra únicamente frente a variables proposicionales.

  ```
  equivalencia :: Formula -> Formula
  ```

  Por ejemplo:

  ```
  equivalencia (Prop Q :=>: Neg (Prop P)) =
  (Neg (Prop Q) :|: Neg (Prop p)) :&: (Prop P :|: Prop Q)
  ```

- **Función Interpretación**. Una función recursiva que recibe una fórmula y una lista de parejas ordenadas, las parejas están formadas por variables y sus respectivos estados (True o False), la función devuelve la formula evaluada con la lista de estados de las variables. Si existe alguna variable dentro de la fórmula que no esté en la lista de estados y sea necesaria para calcular el valor de la proposición, muestra el error "No todas las variables están definidas".

  ```
  interp :: Formula -> [(Var,Bool)] -> Bool
  ```

  Por ejemplo:

  ```
  interp (Prop P :=>: Prop Q :&: Neg (Prop R)) [(Q,True),(P,False)] = True

  interp (Prop P :=>: Prop Q :&: Neg (Prop R)) [(Q,True),(P,True)] =
  Program error: No todas las variables están definidas.
  ```

- **Función Combinaciones**. Una función que recibe una fórmula y devuelve una lista con los posibles combinaciones de las variables de la fórmula, con sus posibles valores de verdad. Es una lista con 2^n posibles valores de verdad, donde n es el número de variables en la fórmula.

  ```
  combinaciones :: Formula -> [[(Var,Bool)]]
  ```

  Por ejemplo:

  ```
  combinaciones ((Prop P) :=>: (Prop Q)) =
  [[(P,True),(Q,True)],[(P,True),(Q,False)],[(P,False),(Q,True)],[(P,False),(Q,Fals
  ```

- **Función Tabla de Verdad**. Una función que recibe una fórmula y el resultado es una lista de 2^n pares ordenados (a, b), donde el a es una lista de estados para cada variable de la fórmula y b es el resultado de la interpretación de la fórmula con esa lista de estados.

  ```
  combinaciones :: Formula -> [[(Var,Bool)]]
  ```

  Por ejemplo:

  ```
  combinaciones ((Prop P) :&: (Prop Q)) =
  [([(P,True),(Q,True)],True),([(P,True),(Q,False)],False),
  ([(P,False),(Q,True)],False),([(P,False),(Q,False)],False)]
  ```

4 Entrega

Deberás enviar un archivo con las funciones que programes al correo **luismanuel@ciencias.unam.mx** a más tardar el **18 de octubre de 2019** antes de las **11:59:59**. El nombre del archivo debe ser **Practica3.hs**, además de las funciones programadas, el archivo debe llevar tu nombre completo empezando por apellidos. El asunto del correo debe ser **ED20201[Práctica 3]**.