Facultad de Ciencias UNAM
Lógica Computacional
Práctica 2: Tipos y lógica proposicional

Alejandro Hernández Mora *

Entrega: Jueves 8 de febrero de 2018

1. Objetivos y Anotaciones

Que el alumno aprenda a definir tipos en Haskell y lo aplique a la lógica proposicional.

Tipos en Haskell.
Una herramienta más que importante en un lenguaje de programación es la definición de nuevos tipos, estructuras y datos.

Como en muchos lenguajes de programación funcional, Haskell es un lenguaje fuertemente tipado, lo que significa que cuando una variable es de algún tipo, éste no podrá cambiar durante todo el proceso de interpretación. A su vez, Haskell es un lenguaje de tipado no explícito, por lo que no decimos de manera explícita el tipo de cada variable o parámetro en la función, Haskell va a inferir el tipo de cada una de estas variables.

Dada la explicación anterior, habrá varios factores a considerar cuando definimos un tipo.

1. Cuando necesitamos un tipo definido recursivamente, es importante el orden en el que lo vamos a definir, pues por la inferencia de tipos que tiene, Haskell tomará los argumentos recibidos y va a comparar desde la primera definición del tipo, hasta la última, en caso de no encontrar un patrón mandará un error (en tiempo de interpretación, pues al ejecutar el programa con el intérprete, no marcará error alguno).
   En otras palabras, debemos empezar la definición de nuestros tipos y funciones recursivas por el(los) caso(s) base (como vimos que ocurre con las listas).

*alejandrohmora@ciencias.unam.mx

2. El orden en el que definamos el tipo es que daremos la jerarquía a las operaciones más importantes, las primeras serán las operaciones más simples y las últimas las de mayor importancia, por lo que esto afecta el orden en el que Haskell hace la evaluación en las funciones. En la lógica proposicional, por ejemplo el operador de mayor jerarquía es la equivalencia lógica, motivo por el cuál debemos definirlo como último caso en los tipos, antes de él deberían ir la implicación, disyunción, conjunción y negación en ese orden.

Lógica 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 :|:
infixr 7 :=>:
infixl 8 :<=>:
```

La 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, y el orden es en el que lo definimos. 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 algúna letra mayúscula definida en el tipo *Var*.

A los 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 proposicional.

La precedencia la estamos definiendo con el número que aparece junto a la palabra ***infixl/infixr***. ***infixl*** es para decirle a Haskell que es un operador infijo (que va entre dos operandos) y que la precedencia es a la izquierda. ***infixr*** es para decirle a Haskell que se trata de un operador infijo, con asociatividad a la derecha. Todo esto ya lo sabemos de las clases de teoría que hemos tenido hasta ahora y sólo para remarcarlo, el único operador asociativo a la derecha es la implicación.

2. Ejercicios

1. Una función recursiva que recibe una fórmula y devuelve el conjunto (lista sin repeticiones) de variables que hay en la fórmula. HINT: Elimina las repeticiones utilizando una función de tu práctica anterior.

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

   Por ejemplo:

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

2. Una función que recibe una fórmula y devuelve su negación.

   ```
   negacion :: Formula -> Formula
   ```

   Por ejemplo:

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

3. Una función que recibe una fórmula y elimina implicaciones y equivalencias.

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

   Por ejemplo:

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

4. Una función recursiva que recibe una fórmula proposicional y una lista de parejas de variables proposicionales. La función debe sustituir todas las presencias de variables en la fórmula por la pareja ordenada que le corresponde en la lista.

   ```
   sustituye :: Formula -> [(Var,Var)] -> Formula
   ```

   Por ejemplo:

   ```
   sustituye (Prop P :=>: Prop Q :&: Neg (Prop R)) [(P,A),(Q,R),(O,U)] =
   Prop A :=>: Prop R :&: Neg (Prop R)
   ```

5. Una función recursiva que recibe una fórmula y una lista de parejas ordenadas de variables con estados (True y False) y evalua la fórmula asignando el estado que le corresponde a cada variable. Si existe alguna variable que no tenga estado asignado 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 estan definidas

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

6. Una función que recibe una fórmula y devuelve la fórmula en ***Forma normal negativa***. Decimos que una fórmula ψ está en forma normal negativa si y sólo si en ψ las negaciones quedan únicamente frente a fórmulas atómicas y no hay presencias de conectivo de implicación, ni equivalencia.

   ```
   fnn:: Formula -> Formula
   ```

   Por ejemplo:

   ```
   fnn(Neg (Prop Q :&: Prop R)) = Neg (Prop Q) :|: Neg (Prop R)
   fnn(Neg ((Prop R) :|: (Prop S)) :<=>: (Prop P)) = (Neg (Prop R) :&: Neg (Prop S)) :<=>: Prop P
   ```

7. En lógica de proposiciones, una ***literal*** es una fórmula atómica o la negación de una fórmula atómica, por ejemplo: P, $\neg Q$. Una ***cláusula*** es una fórmula que únicamente tiene disyunciones de literales o una literal, por ejemplo: $P \wedge Q$, R, $\neg P \wedge Q \wedge \neg R$.Una función que recibe una fórmula y devuelve la fórmula en ***Forma normal conjuntiva***. Decimos que una fórmula ψ está en forma normal conjuntiva, si ϕ es una conjunción de cláusulas. Es decir que todos los términos de ϕ son disyunciones de literales y el operador lógico entre ellos es la conjunción. **Nota:** Es importante tomar en cuenta que una literal, es el caso particular de un término que es disyunción.

   ```
   fnc:: Formula -> Formula
   ```

   Por ejemplo:

   ```
   fnc(Neg (Prop Q :&: Prop R)) = Neg (Prop Q) :|: Neg (Prop R)
   fnc(Neg ((Prop R) :|: (Prop S)) :<=>: (Prop P)) =
   ((Prop R):|:(Prop S):|:(Prop P))
   :&: ((Neg Prop R):|:Neg(Prop P))
   :&: (Neg(Prop R):|:Neg(Prop P))
   ```

3. Requerimientos

Deberás respetar la signatura de las funciones, es decir, está prohibido cambiar el tipo dado para las funciones; por supuesto, tampoco está permitido modificar el tipo de algún parámetro o del resultado.

Deberás enviar tu práctica al correo ***luismanuel@ciencias.unam.mx*** (y sólo a ese correo) antes de las 23:59 del día jueves 8 de marzo de 2018, tal y como lo indican los lineamientos de entrega, de lo contrario la práctica podría no ser calificada.

Deberás enviar por correo un archivo llamado ***Practica2.hs***. La entrega de esta práctica será en equipos de mínimo dos y máximo tres personas. El orden en el que definan las funciones en el archivo ***Practica2.hs*** debe ser el orden especificado en este PDF, de lo contrario se bajarán puntos. No olvides enviar el archivo **ReadMe.txt** como lo especifican los lineamientos de entrega.

¡Que tengas éxito en tu práctica!.