Lógica Computacional

Práctica 02: Sintaxis y Semántica de la Lógica Proposicional

Dicter Tadeo García Rosas Erik Rangel Limón Manuel Soto Romero

Semestre 2024-2
Facultad de Ciencias UNAM

Objetivos

1. Construir un lenguaje para representar la lógica proposicional usando los tipos de dato algebraicos de *Haskell*.
2. Definir la semántica del lenguaje que construimos para representar la lógica proposicional.

Instrucciones

1. Para esta práctica necesitan tener instalado *GHC* y *Cabal*.

   Pueden usar una de las siguientes dos alternativas.

   - **ghcup:**
     Esta es una opción recomendada para instalar las dos herramientas usando un mismo instalador.
     Las instrucciones de cómo instalarlo para sistemas basados en *Unix* y *Windows* vienen en la página `https://www.haskell.org/ghcup/`.
     Una vez instalado, pueden verificar la instalación de `ghc` y de `cabal` utilizando el comando `ghcup tui`, el cual abrirá una interfaz de terminal (utilicen las versiones recomendadas de `ghc` y de `cabal`).
   - **Administrador de paquetes:**
     Por lo general los paquetes que necesitan llevan los nombres `ghc` y `cabal-install`, investiguen si esto difiere según su distribución.
     Debian:

     ```
     sudo apt install ghc cabal-install
     ```

     Fedora:

     ```
     sudo dnf install ghc cabal-install
     ```

2. Necesitan también la biblioteca *QuickCheck*.

   Una vez instalados *GHC* y `Cabal`, lo pueden instalar con la siguientes líneas de comandos:

   ```
   cabal update
   cabal install --lib QuickCheck
   ```

   Pueden ver si la herramienta se instaló bien si pueden interpretar con `ghci` el archivo `src/Test.hs` y no salen errores.

   En la carpeta de la práctica:

```
    ghci Test.hs
```

Pueden salir de `ghci` con `:q` o con la combinación de teclas `Ctrl+D`.

3. Resolver todas las funciones que se encuentran en el archivo `src/Prop.hs`.
4. Pueden verificar sus funciones y recibir una calificación **tentativa** con el archivo `src/Test.hs`.
   En la carpeta de la práctica:

```
runhaskell Test.hs
```

```
Pruebas show:
+++ OK, passed 1000 tests.
Pruebas conjPotencia:
+++ OK, passed 1000 tests.
Pruebas vars:
+++ OK, passed 1000 tests.
Pruebas interpretacion:
+++ OK, passed 1000 tests.
Pruebas modelos:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas tautologia:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas contradiccion:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas esSatisfacible:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas deMorgan:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas elimImplicacion:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas elimEquivalencias:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas exitosas: 18/18
Calificación tentativa: 10.0
```

Ejercicios

Recuerda la definición inductiva de las fórmulas de la lógica proposicional:

1. Una variable ($p, q, r, s, \ldots$) es una expresión atómica y es una fórmula de la lógica proposicional.
2. Los valores de verdad $\top$ (verdadero) y $\bot$ (falso), son expresiones atómicas y fórmulas de la lógica proposicional.
3. Sean φ y ψ fórmulas de la lógica proposicional; las expresiones $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$ y $\varphi \Leftrightarrow \psi$ son fórmulas de la lógica proposicional

Su definición en Haskell sería la siguiente:

```
data Prop = T | F | Var String
          | Neg Prop
          | Conj Prop Prop | Disy Prop Prop
          | Impl Prop Prop | Equiv Prop Prop deriving Eq
```

Por otra parte, una interpretación de una fórmula de la lógica proposicional, es el resultado de la asignación de valores de verdad a las variables que aparecen en una fórmula dado un estado de las variables. En *Haskell* un estado lo definiremos como una lista de variables proposicionales, en donde las variables que figuren en dicha la lista serán consideradas como verdaderas, mientras que el resto serán falsas.

```
type Estado = [String]
```

Realiza las siguientes funciones:

1. *Show*:

   Definir un ejemplar de la clase `Show` para el tipo de dato `Prop` que muestre una cadena que represente las fórmulas proposicionales en notacion infija.

   ```
   instance Show Prop where
       show :: Prop -> String
   ```

   Los símbolos de los conectivos lógicos deben ser los siguientes:

   ```
   -- Negación: ¬
   -- Disyunción: \/ (dentro de una cadena se debe escribir como: "\\/")
   -- Conjunción: /\ (dentro de una cadena se debe escribir como: "/\\")
   -- Implicación: ->
   -- Equivalencia: <->
   ```

   Los conectivos lógicos binarios deben de estar entre paréntesis.

   Ejemplos:

   ```
   ghci> Equiv (Var "p") (Conj (Neg (Var "q")) (T))
   ( p <-> ( ¬q /\ T ) )
   ghci> Disy (Neg (Var "x")) (Impl (Var "y") (Neg (Equiv (Var "w") (Neg (Var "z")))))
   ( ¬x \/ ( y -> ¬( w <-> ¬z ) ) )
   ```

Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_show n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_show 1000
+++ OK, passed 1000 tests.
```

2. *Conjunto potencia*:

   Dada una lista x regresa la lista que contiene todos los subconjuntos de x.

```
conjPotencia :: [a] -> [[a]]
```

   No importa el orden de los subconjuntos

   Ejemplos:

```
ghci> conjPotencia [1,2,3]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
ghci> conjPotencia "abcde"
["","e","d","de","c","ce","cd","cde","b","be","bd","bde","bc","bce","bcd","bcde"
,"a","ae","ad","ade","ac","ace","acd","acde","ab","abe","abd","abde","abc","abce"
,"abcd","abcde"]
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_conjPotencia n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_conjPotencia 1000
+++ OK, passed 1000 tests.
```

3. *Variables en una fórmula:*

   Dada una fórmula de la lógica proposicional, devuelve la lista de las variables que aparecen en ella:

```
vars :: Prop -> [String]
```

   Ejemplos:

```
ghci> vars $ Equiv (Var "p") (Conj (Neg (Var "q")) (T))
["p","q"]
ghci> vars $ Disy (Neg (Var "x")) (Impl (Var "y") (Neg (Equiv (Var "w")
(Neg (Var "z")))))
["x","y","w","z"]
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_show n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_vars 1000
+++ OK, passed 1000 tests.
```

4. *Interpretación*:

   Dada una fórmula de la lógica propocional y un estado, regresa la interpretación obtenida de la fórmula en dicho estado.

```
interpretacion :: Prop -> Estado -> Bool
```

```
ghci> interpretacion (Equiv (Var "p") (Conj (Neg (Var "q")) (T))) []
False
ghci> interpretacion (Equiv (Var "p") (Conj (Neg (Var "q")) (T))) ["p"]
True
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_interpretacion n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_interpretacion 1000
+++ OK, passed 1000 tests.
```

5. *Modelos*:

   Dada una fórmula de la lógica proposicional, devuelve la lista de estados que satisfacen la fórmula.

```
modelos :: Prop -> [Estado]
```

   Ejemplos:

```
ghci> modelos $ Equiv (Var "p") (Conj (Neg (Var "q")) (T))
[["q"],["p"]]
ghci> modelos $ Disy (Neg (Var "x")) (Impl (Var "y") (Neg (Equiv (Var "w")
(Neg (Var "z")))))
[[],["z"],["w"],["w","z"],["y"],["y","z"],["y","w"],["y","w","z"],["x"],["x","z"]
,["x","w"],["x","w","z"],["x","y"],["x","y","w","z"]]
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_modelos n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_modelos 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

6. *Tautología*:

   Dada una fórmula de la lógica proposicional, indica si es o no una tautología.

```
tautologia :: Prop -> Bool
```

   Ejemplos:

```
ghci> tautologia (Disy (Var "p") (Neg (Var "p")))
True
ghci> tautologia (Disy (Var "p") (Var "p"))
False
ghci> tautologia (Impl (Conj (Impl (Var "p") (Var "q")) (Neg (Var "q")))
(Neg (Var "p")))
True
```

Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_tautologia n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_tautologia 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

7. *Contradicción*:

   Dada una lógica de la lógica proposicional, indica si es o no una contradicción

```
contradiccion :: Prop -> Bool
```

   Ejemplos:

```
ghci> contradiccion (Conj (Var "p") (Neg (Var "p")))
True
ghci> contradiccion (Disy (Var "p") (Var "q"))
False
ghci> contradiccion (Neg (Impl (Conj (Impl (Var "p") (Var "q")) (Neg (Var "q")))
(Neg (Var "p"))))
True
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_contradiccion n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_contradiccion 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

8. *Satisfacibilidad*:

   Dada una fórmula de la lógica proposicional, indica si es o no satisfacible.

```
esSatisfacible :: Prop -> Bool
```

   Ejemplos:

```
ghci> esSatisfacible (Conj (Var "p") (Var "q"))
True
ghci> esSatisfacible (Impl (Var "p") (Var "q"))
True
ghci> esSatisfacible (Neg ((Impl (Conj (Impl (Var "p") (Var "q")) (Neg (Var "q")))
(Neg (Var "p")))))
False
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_esSatisfacible n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_esSatisfacible 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

9. *Leyes de De Morgan*:

   Dada una fórmula de la lógica proposicional, regresa una fórmula equivalente que elimina dobles negaciones y aplica las leyes de *De Morgan* de ser necesario.

```
deMorgan :: Prop -> Prop
```

   Ejemplos:

```
ghci> deMorgan (Neg (Neg (Neg (Var "x"))))
¬x
ghci> deMorgan (Disy (Neg (Neg (Var "s"))) (Conj (Neg (Var "l"))
(Neg (Neg (Var "r")))))
( s \/ ( ¬l /\ r ) )
ghci> deMorgan $ fst $ head $ prs prop "¬( w /\\ ( m \\/ r ) )"
( ¬w \/ ( ¬m /\ ¬r ) )
```

   Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_deMorgan n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_deMorgan 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

10. *Quitando implicaciones*:

    Dada una fórmula de la lógica proposicional, regresa una fórmula equivalente que elimina las implicaciones de la fórmula.

```
elimImplicacion :: Prop -> Prop
```

    Ejemplos:

```
ghci> elimImplicacion (Disy (Var "p") (Var "q"))
( p \/ q )
ghci> elimEquivalencias  $ fst $ head $ prs prop
"¬( ( g -> v ) -> ( ( e <-> r ) -> ( u -> v ) ) )"
¬( ( g -> v ) -> ( ( ( e -> r ) /\ ( r -> e ) ) -> ( u -> v ) ) )
ghci> elimImplicacion $ fst $ head $ prs prop "(a <-> (b <-> ((c <-> d) <-> e)))"
( ¬a \/ ( ¬b \/ ( ¬( ¬c \/ d ) \/ e ) ) )
```

    Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_elimImplicacion n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_elimImplicacion 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

11. *Quitando equivalencias*:

    Dada una fórmula de la lógica proposicional, regresa una fórmula equivalente que elimina las implicaciones de la fórmula.

```
elimEquivalencias :: Prop -> Prop
```

Ejemplos:

```
ghci> elimEquivalencias (Disy (Var "p") (Var "q"))
( p \/ q )
ghci> elimEquivalencias  $ fst $ head $ prs prop
"¬( ( g -> v ) -> ( ( e <-> r ) -> ( u -> v ) ) )"
¬( ( g -> v ) -> ( ( ( e -> r ) /\ ( r -> e ) ) -> ( u -> v ) ) )
ghci> elimEquivalencias $ fst $ head $ prs prop "(a <-> (b <-> (c <-> d)))"
( ( a -> ( ( b -> ( ( c -> d ) /\ ( d -> c ) ) ) /\ ( ( ( c -> d ) /\
( d -> c ) ) -> b ) ) ) /\ ( ( ( b -> ( ( c -> d ) /\ ( d -> c ) ) ) /\
( ( ( c -> d ) /\ ( d -> c ) ) -> b ) ) -> a ) )
```

Para ejecutar las pruebas de esta función puedes evaluar la expresión `check_elimEquivalencias n` siendo `n` el número de pruebas que se le van a hacer a la función.

```
ghci> check_elimEquivalencias 1000
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

Entrega

1. La tarea se entrega en *Github Classroom*.
2. La tarea se entrega en equipos de hasta 4 personas.
3. Además de resolver los ejercicios, deben entregar un `README` (`.md` o `.org`) en la raiz de la carpeta de la práctica que contenga los datos de todos los integrantes, y por cada una de sus funciones dejar una breve explicación de su solución.

Consideraciones

1. Todas las prácticas con copias totales o parciales tanto en el código como en el `README` serán evaluadas con cero.
2. Las únicas funciones con soluciones iguales admisibles son todas aquellas que sean iguales a las resueltas por el grupo en el laboratorio, sin embargo la explicación de su solución en el `README` debe ser única para cada equipo.
3. No entregar el `README` o tenerlo incompleto se penalizará con hasta dos puntos menos sobre su calificación en la práctica.
4. Cada día de retraso se penalizará con un punto sobre la calificación de la práctica.
5. Pueden usar las funciones `map, filter, reverse, sum, elem, all, any, head, tail`. Pueden utilizar otras funciones siempre y cuando no resuelvan directamente el ejercicio, expliquen en el `README` qué es lo que hace, y pertenezca a alguna biblioteca estándar de *Haskell* (es decir, no es válido utilizar paqueterías que requieran una instalación manual).
6. Los tests de ésta práctica tardan entre uno y tres minutos. Al final del archivo `src/Prop.hs` hay una variable `pruebas`, ésta indica el número de pruebas que se realizarán a cada una de las funciones. Por defecto está configurada para hacer 1000 pruebas, pero pueden ponerle un valor menor si consideran que las pruebas tardan mucho en completarse, siempre y cuando éste valor no sea menor a 100. Si las pruebas tardan más de los tres minutos aún con 100 pruebas, consideren otra solución.
7. Si a la variable `pruebas` le pusieron un valor menor a 1000, asegúrense que pasen las pruebas de manera consistente, es decir, que puedan ejecutar varias veces las pruebas sin dar errores.
8. Pueden hacer tantas funciones auxiliares como quieran, pero no deben modificar la firma de las funciones ni de las variables, ni la definición de tipos de dato que se les dió.
9. No se recibirán prácticas que no compilen (no debe arrojar errores la orden `ghci Test.hs`). Si no resuelven alguna de las funciones déjenlas como `undefined`, pero no eliminen la función, ya que ésto lanzará errores.
10. No deben modificar el archivo `src/Test.hs`. Si encuentran errores o tienen dudas sobre las pruebas, manden un correo al ayudante de laboratorio (si encuentran errores tendrán una participación).
11. Las participaciones en el laboratorio se aplicarán de manera individual sobre la calificación que tuvieron en la práctica.