Facultad de Ciencias - UNAM
Lógica Computacional 2023-1
Práctica 1 - Implementación FNN y FNC

Favio E Miranda Perea
Javier Enríquez Mendoza
José Manuel Madrigal Ramírez

2/septiembre/2022
Fecha de entrega: 9/septiembre/2022 a las 23:59 hrs

Preámbulo

En nuestras clases de laboratorio implementamos el tipo de dato **Prop**. Vale la pena recordar los nombres de los elementos sintácticos de Haskell ocupados en esta construcción:

```
data Prop = VarP Int
          | T
          | F
          | Neg Prop
          | And Prop Prop
          | Or Prop Prop
          | Imp Prop Prop
          | Equiv Prop Prop
    deriving (Show, Eq)
```

Prop es el tipo de dato para nuestras fórmulas. Lo utilizaremos en la firma de tipo en la mayor parte de las funciones que implementemos. Por otro lado *VarP* es el constructor de las variables proposicionales que recibe un entero como argumento. Cuando lo utilicemos en las cláusulas de nuestras funciones tiene que estar entre paréntesis con una variable representando su argumento, por ejemplo: `(VarP x)`.
T es un constructor sin parámetros que usaremos para representar el valor de verdad True. *And* también es un constructor y recibe dos parámetros que son, a su vez, del tipo *Prop*. Al utilizarlo en las cláusulas, lo haremos en forma de patrón, escribiéndolo entre paréntesis y colocando variables que representen sus argumentos: `(And x y)`

Parte 1

1. Implementa la función `subconj` que dado un subconjunto representado en una lista devuelve su conjunto potencia representado en listas de listas.

   Ejemplo:

   ```
   *Main> subconj [1, 2, 3]
   [[1, 2, 3], [1, 2], [2, 3], [1, 3], [2], [1], [3], []]

   *Main> subconj [1]
   [1, []]
   ```

   Pistas:

   - El orden de los subconjuntos en el resultado de la función puede ser distinto al ejemplo.
   - Podemos utilizar comprensión de listas.
     Ejemplo:

     ```
     *Main> [2*x | x <- [2, 3]]
     [4, 6]
     ```

     En general la sintáxis de la comprensión de listas es:

     ```
     [ EXP | QUAL, ..., QUAL ]
     ```

     Donde `QUAL` es una expresión booleana o un generador.

     En el siguiente enlace hay más ejemplos: `http://www.zvon.org/other/haskell/Outputsyntax/listQcomprehension_reference.html`

2. Implementar la función `vars` que dada una proposición regrese una lista con sus variables proposicionales sin repetición.

   Pista: Podemos utilizar funciones auxiliares y después hacerlas converger en la función que cumpla con nuestro propósito final

3. **extra**: Definir nuestra versión de `Show` para *Prop*. Cuando el interprete reciba una expresión de nuestro lenguaje Prop quisiéramos que en vez de que se imprimiera:

   ```
   Imp (Neg (VarP 1)) (Or (VarP 2) (VarP 3))
   ```

   se imprimiera algo como:

   ```
   (¬v1 -> (v2  v3))
   ```

   Para ello utilizamos la siguiente sintaxis:

```
instance Show Prop where
    show (Constructor x1 .. xn) = <cadena>
      .
      .
```

Por ejemplo, para las variables proposicionales podríamos tener algo como:

```
instance Show Prop where
   show (VarP i) = "v" ++ (show i)
   show (Neg p) = ???
```

Observa que `Show` después de `instance` se escribe con mayúsculas y en las cláusulas va con minúsculas. No olvides quitar `Show` de la definición de `Prop`.

Parte 2

1. Sabemos que el estado de una fórmula es una función I que va de las variables proposicionales a los valores de verdad. $I : VarP \rightarrow Bool$.

   Implementaremos los estados como listas de variables proposicionales. Consideraremos que las variables en la lista están asociadas al valor de verdad T mientras que las que no se encuentren en ella estarán asociadas a F.

   Ejemplos:
   Si tenemos la fórmula `(Imp (VarP 1) (VarP 2))` escrita con los constructores de `Prop`. Para implementar el estado

   $$I_1(Var_1) = 0$$
   $$I_1(Var_2) = 1$$

   utilizaríamos la lista en donde sólo figura Var_2

   ```
   i1 = [(VarP 2)]
   ```

   Por otro lado, un estado I_2 en el que fuesen verdaderas Var_1 y Var_2 se implementaría colocando ambas variables en la lista:

   ```
   i1 = [(VarP 1), (VarP 2)]
   ```

   Así, para definir a los estados como listas de variables comenzaremos escribiendo:

   ```
   type Estado = [Prop]
   ```

   (type nos permite establecer un "alias" con tipos primitivos de Haskell. Más información al respecto aquí: `http://aprendehaskell.es/content/ClasesDeTipos.html`)

Una vez hecho esto vamos a implementar la función que interpreta una fórmula con un estado dado.

```
interp ::  Estado > Prop -> Bool
```

Algunos ejemplos de su ejecución:

```
*Main>interp [VarP 1] (Imp (VarP 1) (VarP 2))
False

*Main>interp [(VarP 1), (VarP 2)] T
True

*Main>interp [] (Imp (VarP 1) (VarP 2))
True
```

2. Ahora vamos a implementar la función `estados ::  Prop -> [[Prop]]` Que devuelve una lista con todos los posibles estados de una fórmula.

   Para ello utilizaremos la función `subconj` y la función `vars` que implementamos en el laboratorio.

   Aquí hay algunos ejemplos de su ejecución (suponiendo que implementamos la función *Show* al menos para las variables):

```
*Main> estados (Imp (VarP 1) (VarP 2))
[[v1,v2],[v1],[v2],[]]

*Main> estados (VarP 1)
[[v1],[]]
```

Parte 3: Forma normal negativa

1. Comenzaremos por implementar el par de funciones que aparece en las Notas de clase 2, en la sección 4.2 las cuales eliminan equivalencias e implicaciones respectivamente:

```
elimEquiv ::  Prop -> Prop

elimImp ::  Prop -> Prop
```

2. En la **sección 1.1** de las notas de clase 3, tenemos detalles sobre la implementación de la forma *normal negativa (fnn)*.

   Al convertir una fórmula en su equivalente *fnn* necesitamos que todas las negaciones aparezcan únicamente frente a variables proposicionales. Para ello requerimos de una función que elimine las dobles negaciones y que interiorice las negaciones si es que están frente a una conjunción o una disyunción.

   Para ello podemos servirnos de las *leyes De Morgan* y las equivalencias vistas en clase:

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg\neg A \equiv A.$$

Implementemos la función `iNeg :: Prop -> Prop` que por entrada espera una fórmula sin implicaciones ni equivalencias y cuya salida es una fórmula sin doble negación y dónde las únicas fórmulas negadas son atómicas.

3. Utilizando las funciones implementadas en las secciones anteriores podemos implementar la función `fnn::Prop -> Prop` que devuelve la fórmula de entrada en su *forma normal negativa*.

Parte 4: Forma normal conjuntiva

1. En la definición de la forma normal conjuntiva utilizamos literales, que son fórmulas atómicas o sus negaciones. Implementemos una función que nos permita distinguir si una fórmula dada es literal o no.

   `literal :: Prop -> Bool`

   Veamos unos ejemplos de su ejecución:

   ```
   *Main> literal (And (VarP 1) (VarP 2))
   False
   *Main> literal (VarP 1)
   True
   *Main> literal (Neg (VarP 1))
   True
   ```

2. Antes de implementar la función de la forma normal conjuntiva, vamos a implementar una función auxiliar que nos permitan aplicar las reglas de la distribución cuya firma de tipo sería: `distr :: Prop -> Prop -> Prop`

   En la sección 3.1 de las Notas de clase 3, aparecen algunas observaciones que nos ayudan a implementar esta función. Aquí hay también algunos ejemplos de su ejecución:

   ```
   *Main> distr (And (Neg (VarP 1)) (Neg (VarP 2))) (VarP 3)
   ((v3 \/ ¬v1) /\ (v3 \/ ¬v2))

   *Main> distr (Neg (VarP 1)) (VarP 2)
   (¬v1 \/ v2)
   ```

3. Finalmente podemos unir la función que devuelve la forma normal negativa de una fórmula junto a las funciones auxiliares que recién implementamos para construir la forma normal conjuntiva:

   `fnc :: Prop -> Prop`

Parte 5: Regla de resolución binaria (Extra)

(a) Implementemos una cláusula con una lista de literales en una fórmula como sigue: `type Clausula = [Literal]`

(b) Ahora vamos a implementar una función que dada una literal ℓ nos regrese su literal contraria ℓ^c:

```
*Main> compLit (Neg (VarP 1))
v1
*Main> compLit (VarP 1)
¬v1
```

(c) Con lo anterior podemos implementar la regla de resolución binaria tal como está descrita en la sección 4 de las Notas de clase 3:

```
res ::  Clausula -> Clausula -> Clausula
```

Pistas:

- Haskell tiene una función predefinida llamada `elem` que recibe un elemento y una lista. Nos regresa `True` si tal elemento se encontraba en la lista que pasamos como segundo argumento. Regresa `False` en caso contrario. Aquí algunos ejemplos:

```
*Main> elem 3 [1,2,3]
True
*Main> elem 1 []
False
*Main> elem 'a' "hola mundo"
True
```

- Tal vez también pueda ser de utilidad una función predefinida llamada `filter` cuya sintaxis en general es la siguiente:

```
filter ::  (a -> Bool) -> [a] -> [a]
```

Y puede utilizarse de la siguiente manera:

```
*Main> filter (not.(== 3)) [1,2,3]
[1,2]
*Main> filter (not.(== 0)) [1,2,3]
[1,2,3]
```

Entrega

La entrega se realizará por **parejas** y consistirá en un archivo comprimido que debe contener:

- El archivo `practica1.hs` con el código necesario para que sus funciones se ejecuten adecuadamente.
- Un archivo readMe.txt que incluya sus nombres y una bitácora sobre el desarrollo de la práctica donde indiquen los problemas que se presentaron y cómo los resolvieron. Asimismo incluyan comentarios de cada uno al respecto.

Envíen su archivo comprimido a mi correo: *jose.manuel.madrigal.ramirez@gmail.com* con el siguiente formato:

Practica1-Apellido1Nombre1Apellido2Nombre2.zip

Por ejemplo, si su equipo esta conformado por Alan Turing y Ada Lovelace su archivo debería llamarse:

Practica1-TuringAlanLovelaceAda.zip

Cualquier duda que tengan no duden en enviarme un correo o comunicarse por Telegram. 😄