Facultad de Ciencias - UNAM
Lógica Computacional 2023-1
Práctica 2 - Implementación del algoritmo DPLL

Favio E Miranda Perea
Javier Enríquez Mendoza
José Manuel Madrigal Ramírez

27/septiembre/2022
Fecha de entrega: 7/octubre/2022 a las 23:59 hrs

Preámbulo

En la cuarta entrega de las notas de clase quedó descrito el algoritmo *Davis-Putnam-Logemann-Loveland* como un procedimiento de búsqueda hacia atrás para decidir la satisfacibilidad de fórmulas proposicionales en su forma normal conjuntiva. En esta práctica vamos a definir una implementación funcional del mismo.

En el archivo *Practica2.hs* encontrarán la firma de tipo de las funciones que nos serán de utilidad. Guárdenlo en la misma carpeta donde estaba su archivo de la práctica anterior y las actividades de laboratorio, de modo que podamos importar su contenido en la práctica actual.

En ese archivo también encontrarán los siguientes tipos de dato (en principio los primeros dos ya estaban definidos en entregas anteriores, si no es el caso, añádanlos a esta práctica):

```
type Literal = Prop
type Clausula = [Literal]

type Formula = [Clausula]
type Modelo = [Literal]
type Configuracion = (Modelo, Formula)
```

El tipo de dato `Configuración` representará las expresiones del tipo $\mathcal{M} \models_? F$, presentes en las transiciones de la sección `3.1` de la cuarta entrega de las notas.

Cuando hayamos terminado esta práctica tendremos una función `dpll` cuya firma de tipo será:

`dpll::Configuracion -> Configuracion`

y que recibirá una configuración con un modelo vacío y una fórmula F en su forma normal conjuntiva. Por ejemplo, si la fórmula es:

$$(\neg p \vee r \vee \neg t) \wedge (\neg q \vee \neg r)$$

Podríamos definir la configuración inicial siguiente:

```
    p = VarP 1
    q = VarP 2
    r = VarP 3
    s = VarP 4
    phi = [[(Neg p), r, (Neg s)],[(Neg q),(Neg r)]]
    conf = ([], phi)
```

y al aplicar nuestra función `dpll` en consola obtendríamos:

```
*Practica2> dpll conf
*Practica2>([¬v2,¬v1],[])
```

Lo cual querría decir que un modelo para F es aquel en el que las literales $\neg p$ y $\neg q$ se evalúan a verdadero.

Parte 1 - ¿Cuándo se aplica cada regla?

En la definición de DPLL como búsqueda de modelos hacia atrás, tenemos seis reglas que definen la manera en que realizamos transiciones de una configuración a otra:

- Regla de la cláusula unitaria
- Regla de eliminación
- Regla de reducción
- Regla de separación
- Regla de conflicto
- Regla de éxito

Podemos utilizar guardias dentro de la definición de nuestra función `dpll` para indicar en qué estado de las configuraciones haremos uso de cada regla de transición:

```
dpll :: Configuracion -> Configuracion
dpll (m, f)
    | (usarElim (m, f)) ) =
    | (usarRed (m,f)) =
    | (exito (m, f)) =
    | (conflicto (m, f)) =
    | (usarUnit (m, f)) =
    | otherwise =
```

Para ello requerimos funciones que sirvan como condición de activación de cada guardia. Estas funciones deben tomar una configuración y devolver un booleano según se pueda o no ejecutar cada una de las reglas de transición.

1. Implementa la función

   ```
   usarUnit::Configuracion ->Bool
   ```

   que nos permita saber si en una configuración dada podemos utilizar la **regla de la cláusula unitaria** :

   ```
   *Practica2>usarUnit ([p], [[(Neg p),r],[(Neg q),(Neg r)]])
   *Practica2>False

   *Practica2>usarUnit ([(Neg q)], [[q], [(Neg p),r,s]])
   *Practica2>False


   *Practica2>usarUnit ([(Neg q)], [[t], [p,(Neg r),s]])
   *Practica2>True
   ```

2. Implementa la función

   ```
   usarElim::Configuracion ->Bool
   ```

   que nos permita saber si en una configuración dada podemos utilizar la **regla de eliminación**:

   ```
   *Practica2>usarElim ([p], [[(Neg p),r],[(Neg q),(Neg r)]])
   *Practica2>False

   *Practica2>usarElim ([(Neg q)], [[t], [p,(Neg q),s]])
   *Practica2>True
   ```

3. Implementa la función

   ```
   usarRed::Configuracion ->Bool
   ```

   que nos permita saber si dada una configuración podemos utilizar la **regla de reducción**:

   ```
   *Practica2>usarRed ([p], [[(Neg p),r],[(Neg q),(Neg r)]])
   *Practica2>True

   *Practica2>usarRed ([(Neg q)], [[t], [p,(Neg q),s]])
   *Practica2>False
   ```

4. Implementa la función

   ```
   exito ::  Configuracion -> Bool
   ```

   que nos permite saber, si hemos conseguido un modelo para nuestra fórmula de la configuración inicial:

```
*Practica2>exito ([(Neg q), t, p], []
*Practica2>True

*Practica2>exito ([(Neg q), t, p], [[], [(Neg p), q]]
*Practica2>False
```

5. Implementa la función

   ```
   conflicto ::  Configuracion -> Bool
   ```

   que nos permite saber, si es imposible tener un modelo para la fórmula en una configuración dada:

   ```
   *Practica2>conflicto ([(Neg q), t, p], []
   *Practica2>False

   *Practica2>exito ([(Neg q), t, p], [[], [(Neg p), q]]
   *Practica2>True
   ```

   La regla `split` se ejecuta en el caso de que ninguna de las reglas anteriores haya podido ejecutarse, es decir, se ejecuta en el caso *otherwise* de las guardias.

Parte 2 - Definiendo las reglas de eliminación y reducción

Una vez implementadas las funciones que nos brindan condiciones para las guardias, podemos escribir lo que sucederá cuando se activen, aplicando la función `dpll` a la configuración ligeramente modificada. Estas modificaciones dependerán de la regla de transición que utilicemos. Por ejemplo, en la regla de *cláusula unitaria* modificamos el modelo y la fórmula, mientras que en la regla de *eliminación* sólo modificamos la fórmula.

En esta parte de la práctica definiremos las reglas de transición que sólo modifican a la fórmula. Cabe aclarar que en las notas de clase, las reglas de transición operan con una sóla cláusula a la vez. Sin embargo, para optimizar el desempeño de nuestro programa aprovechando los mecanismos de Haskell, implementaremos estas reglas de modo que simplifiquen todo el conjunto de cláusulas cada vez que se apliquen:

1. Define una función que implemente la regla de eliminación:

   ```
   elim ::  Literal -> Formula -> Formula
   ```

   Pista: Puede quedar definida en una línea usando listas de comprensión y la condición adecuada. A continuación algunos ejemplos de su uso:

   ```
   *Practica2>elim q [[q, p],[(Neg q), r, s],[q,s]]
   *Practica2>[[(Neg q), r, s]]
   ```

```
*Practica2>elim s [[q, p],[(Neg q), r, s],[q,s]]
*Practica2>[[q, p]
```

2. Define una función que implemente la regla de reducción:

```
red ::  Literal -> Formula -> Formula
```

Algunos ejemplos de su uso serían:

```
*Practica2>red q [[q, p],[(Neg q), r, s],[q,s]]
*Practica2>[[q, p],[r, s],[q,s]]

*Practica2>red (Neg s) [[q, p],[(Neg q) r, s],[q,s]]
*Practica2>[[q, p],[(Neg q) r],[q]]
```

Pista: `delete` es una función predefinida que elimina un elemento x de una lista.

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

Pista 2: Podemos utilizar funciones en la primera parte de la comprensión de listas.

```
*Main>[delete 1 c|c <- [[1,2], [1,3], [1,5]]]
*Main>[[2],[3],[5]]
```

Parte 3 - Definiendo la regla de cláusula unitaria y la regla de separación

1. Si la función `usarUnit` que definimos previamente se evalúa a verdadero, significa que existe al menos una cláusula unitaria cuya literal debemos eliminar de la fórmula y añadir a nuestro modelo:

$$\mathcal{M} \models_? F, \ell \triangleright \mathcal{M}, \ell \models_? F$$

Para esto nos conviene implementar dos funciones auxiliares:

- `unitaria ::  Formula -> Literal`
  Que regresa la literal de una cláusula unitaria en la fórmula.
- `quitaUnitaria ::  Formula -> Formula`
  Que elimina la primera cláusula unitaria de una fórmula dada.

2. Por otro lado, para implementar la regla de *separación*:

$$\mathcal{M} \models_? F \triangleright \mathcal{M}, \ell \models_? F;\ \mathcal{M}, \ell^c \models_? F$$

debemos considerar la forma en que *rescataremos* el modelo conseguido en alguna de las configuraciones, si es que lo hubo. Una sugerencia es utilizar una función auxiliar:

```
unirConfig::Configuracion -> Configuracion -> Configuracion
```

que una los resultados de aplicar dpll en dos configuraciones distintas.

También necesitamos una función auxiliar que nos devuelva una literal cualquiera de una fórmula dada para poder añadirla al modelo:

```
sigLit ::  Formula -> Literal
```

Poniendo estas dos funciones adecuadamente en la última guardia de `dpll` podremos replicar el efecto de utilizar la regla de *separación*.

Entrega

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

- El archivo `Practica2.hs` con el código necesario para que sus funciones se ejecuten adecuadamente.
- Los archivos que hayan importado dentro de su práctica de modo que esta pueda compilar. En principio éstos son la práctica anterior y la actividad 3.
- 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:

Practica2-Apellido1Nombre1Apellido2Nombre2.zip

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

Practica2-TuringAlanLovelaceAda.zip

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