Lógica Computacional
Práctica 03: Solucionadores SAT

Dicter Tadeo García Rosas Erik Rangel Limón Manuel Soto Romero

Semestre 2024-2
Facultad de Ciencias UNAM

Objetivos

1. Conocer el funcionamiento práctico de un solucionador del problema *SAT*, a partir de un programa que resuelve Sudokus.

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 *MiniSat*.

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

   ```
   cabal update
   cabal install --lib minisat-solver
   ```

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

   En la carpeta de la práctica:

```
    ghci Main.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/Sudoku.hs`.

4. Pueden verificar su solución si el programa en el archivo `src/Main.hs` es capaz de dar una solución correcta para un Sudoku.

   En la carpeta de la práctica:

```
runhaskell Main
```

```
==== Sudoku Solver 1.0.0. ====
1. Resolver Sudoku
2. Salir
Elige una opción:
1
Ingresa los números de la fila 1 (los espacios en blanco se denotan con un 0):
001620057
Ingresa los números de la fila 2 (los espacios en blanco se denotan con un 0):
000000009
Ingresa los números de la fila 3 (los espacios en blanco se denotan con un 0):
040100000
Ingresa los números de la fila 4 (los espacios en blanco se denotan con un 0):
000060400
Ingresa los números de la fila 5 (los espacios en blanco se denotan con un 0):
030007065
Ingresa los números de la fila 6 (los espacios en blanco se denotan con un 0):
005000900
Ingresa los números de la fila 7 (los espacios en blanco se denotan con un 0):
000000600
Ingresa los números de la fila 8 (los espacios en blanco se denotan con un 0):
800003000
Ingresa los números de la fila 9 (los espacios en blanco se denotan con un 0):
004070021
La solución es la siguiente:
  | 1 2 3 4 5 6 7 8 9
---------------------
1 | 9 8 1 6 2 4 3 5 7
2 | 6 5 2 7 3 8 1 4 9
3 | 7 4 3 1 5 9 2 8 6
4 | 2 7 8 9 6 5 4 1 3
5 | 1 3 9 2 4 7 8 6 5
6 | 4 6 5 3 8 1 9 7 2
7 | 5 1 7 4 9 2 6 3 8
8 | 8 2 6 5 1 3 7 9 4
9 | 3 9 4 8 7 6 5 2 1

==== Sudoku Solver 1.0.0. ====
1. Resolver Sudoku
2. Salir
Elige una opción:
```

Ejercicios

La biblioteca `SAT.MiniSat` tiene su propia definición de tipo de dato para representar a las fórmulas de la lógica proposicional.

```
data Formula v = Var v -- Variables
               | Yes -- Constante de verdad
               | No -- Constante de falsedad
               | Not (Formula v) -- Negación
               | (Formula v) :&&: (Formula v) -- Conjunción
               | (Formula v) :||: (Formula v) -- Disyunción
               | (Formula v) :++: (Formula v) -- Disyunción exclusiva
               | (Formula v) :->: (Formula v) -- Implicación
               | (Formula v) :<->: (Formula v) -- Equivalencia
               | All [Formula v] -- Todos son verdad
               | Some [Formula v] -- Alguno es verdad
               | ExactlyOne [Formula v] -- Exactamente uno es verdad
               | AtMostOne [Formula v] -- A lo más uno es verdad
               | Let (Formula v) (Formula v -> Formula v)
               -- Asignación local; ejemplo:
               -- Let (Var "v") (\v -> v :||: Not v) = Var "v" :||: Not (Var "v")
```

Una diferencia sustancial con nuestra definición de la práctica 2 es que para los operadores binarios se utilizan constructores infijos.

En *Haskell* para poder hacer constructores infijos es necesario que éstos comiencen con el símbolo `(:)`.

Gracias a ésto podemos escribir expresiones de la lógica proposicional más legibles:

```
Var "p" :->: Var "q" :->: Var "r" :<->: Var "p" :&&: Var "q" :->: Var "r"
```

El único problema que surge a raíz de usar tanto operadores como constructores infijos es la forma de asociar; considerando la expresión anterior, ¿cómo debería asociar?

```
(Var "p" :->: (Var "q" :->: Var "r")) :<->: ((Var "p" :&&: Var "q") :->: Var "r")
((Var "p" :->: Var "q") :->: Var "r") :<->: (Var "p" :&&: (Var "q" :->: Var "r"))
(Var "p" :->: Var "q") :->: (Var "r" :<->: ((Var "p" :&&: Var "q") :->: Var "r"))
```

Haskell ofrece las palabras reservadas `infixl`, `infixr` e `infix`, con el que se denota si un operador es asociativo hacia la izquierda, hacia la derecha, o si no es asociativo respectivamente. Con esta expresión también se determina la jerarquía del operador.

En el caso particular de la biblioteca `SAT.MiniSat` se definió la asociatividad de sus constructores de la siguiente forma:

```
infixr 6 :&&:
infixr 5 :||:
infixr 4 :++:
infixr 2 :->:
infix 1 :<->:
```

Después del uso de la palabra reservada, se denota la jerarquía del operador, la cual debe ser un número del 0 al 9, y por último el nombre del operador.

De esta forma la correcta asociación de la expresión anterior es la siguiente:

```
(Var "p" :->: (Var "q" :->: Var "r")) :<->: ((Var "p" :&&: Var "q") :->: Var "r")
```

Además de introducir constructores infijos, tenemos también nuevas expresiones que denotan expresiones para un conjunto de fórmulas; como `All` que denota si todas las fórmulas en el conjunto son verdaderas, `Some` para denotar si alguna fórmula en el conjunto es verdadera, `ExactlyOne` para denotar que hay exactamente una fórmula en el conjunto que es verdadera y `AtMostOne` para denotar que a lo más una fórmula en el conjunto es verdad.

Para esta práctica daremos un sinónimo de tipo para un Sudoku:

```
type Sudoku = [(Int,Int,Int)]
```

Va a ser una lista de triplas, donde cada elemento (i, j, n) denota que la celda (i, j) tiene el número n (si una celda no tiene número, no se coloca en la lista).

Debes considerar que la matriz del Sudoku es 1-indexada.

Implementa la siguiente función:

1. *Fórmula para resolver un Sudoku*:

   Dado un Sudoku inicial (la lista que tiene únicamente la información de las celdas con valores iniciales), devuelve la fórmula de la lógica proposicional en la que, de ser satisfacible, cada uno de sus modelos es una solución correcta para el Sudoku dado.

   ```
   sudokuFormula :: Sudoku -> Formula (Int,Int,Int)
   ```

   La fórmula resultante debe de asegurar que se cumplan los siguientes puntos:

   - Cada elemento del sudoku inicial debe ocurrir.
     Es decir, si la entrada fue `[(1,1,9),(3,2,4),(5,4,2),(6,7,2)]`, tu fórmula debe incluir que cada una de éstas casillas ocurren, por ejemplo:

     ```
     sudokuFormula [(1,1,9),(3,2,4),(5,4,2),(6,7,2)]
       -> Var (1,1,9) :&&: Var (3,2,4) :&&: Var (5,4,2) :&&: Var (6,7,2) :&&: ...
     ```

     o bien:

     ```
     sudokuFormula [(1,1,9),(3,2,4),(5,4,2),(6,7,2)]
       -> All [Var (1,1,9), Var (3,2,4), Var (5,4,2), Var (6,7,2)] :&&: ...
     ```

   - Cada una de las celdas guarda exactamente un número del 1 al 9.
     Por ejemplo, consideremos la casilla $(5,5)$, en alguna parte de la fórmula resultante debe aparecer una expresión similar o equivalente a la siguiente:

     ```
     sudokuFormula ...
       -> ... :&&: ExactlyOne [Var (5,5,1), Var (5,5,2), ..., Var (5,5,9)] :&&: ...
     ```

     Ésto debe suceder para cada casilla en el sudoku.

- En cada fila aparece exactamente una vez cada uno de los números del 1 al 9.
  Por ejemplo, consideremos la fila 3 del sudoku, en alguna parte de la fórmula resultante debe aparecer algo similar o equivalente a las siguientes expresiones.

```
sudokuFormula ...
  -> ... :&&: ExactlyOne [Var (3,1,1), Var (3,2,1), ..., Var (3,9,1)] :&&:
              ExactlyOne [Var (3,1,2), Var (3,2,2), ..., Var (3,9,2)] :&&:
              ... :&&:
              ExactlyOne [Var (3,1,9), Var (3,2,9), ..., Var (3,9,9)] :&&: ...
```

  Esto debe susceder para cada fila en el sudoku.

- En cada columna aparece exactamente una vez cada uno de los números del 1 al 9.
  Por ejemplo, consideremos la columna 3 del sudoku, en alguna parte de la fórmula resultante debe aparecer algo similar o equivalente a las siguientes expresiones.

```
sudokuFormula ...
  -> ... :&&: ExactlyOne [Var (1,3,1), Var (2,3,1), ..., Var (9,3,1)] :&&:
              ExactlyOne [Var (1,3,2), Var (2,3,2), ..., Var (9,3,2)] :&&:
              ... :&&:
              ExactlyOne [Var (1,3,9), Var (2,3,9), ..., Var (9,3,9)] :&&: ...
```

  Esto debe susceder para cada columna en el sudoku.

- En cada bloque 3×3 aparece exactamente una vez cada uno de los números del 1 al 9.
  Por ejemplo, consideremos el bloque compuesto por las filas 4 a 6 y las columnas 1 a 3, en alguna parte de la fórmula resultante debe aparecer algo similar o equivalente a las siguientes expresiones.

```
sudokuFormula ...
  -> ... :&&: ExactlyOne [ Var (4,1,1), Var (4,2,1), Var (4,3,1)
                         , Var (5,1,1), Var (5,2,1), Var (5,3,1)
                         , Var (6,1,1), Var (6,2,1), Var (6,3,1) ] :&&:
              ExactlyOne [Var (4,1,2), Var (4,2,2), ..., Var (6,3,2)] :&&:
              ... :&&:
              ExactlyOne [Var (4,1,9), Var (4,2,9), ..., Var (6,3,9)] :&&: ...
```

  Esto debe suceder para cada bloque del sudoku.

Procuren no escribir toda la fórmula de manera extensiva, utilicen listas por comprensión.

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. Asegúrense de que su programa pueda resolver correctamente varios Sudokus, su evaluación será determinada por el número de sudokus que resuelva su programa al momento de calificar.
7. 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.
8. No se recibirán prácticas que no compilen (no debe arrojar errores la orden `runhaskell Main`).
9. No deben modificar el archivo `src/Main.hs`. Si encuentran errores o tienen dudas sobre el archivo, manden un correo al ayudante de laboratorio (si encuentran errores tendrán una participación).
10. Las participaciones en el laboratorio se aplicarán de manera individual sobre la calificación que tuvieron en la práctica.