Procedimiento de decisión de SAT y Validez
Resolución en Lógica Proposicional

Laboratorio 3
Lógica para Computación
Otoño 2018

En el caso de la Lógica Proposicional (LP), la consecuencia lógica es *decidible*, es decir, existen algoritmos que pueden resolver el problema de determinar si, para un conjunto finito de premisas Γ y una conclusión α, se cumple o no la relación $\Gamma \vDash \alpha$. El objetivo de este laboratorio es implementar en Haskell un algoritmo basado en el *principio de resolución* para decidir instancias del problema de consecuencia lógica.

1. Introducción

Comenzamos introduciendo y repasando conceptos teóricos y notación necesarios para realizar este laboratorio.

1.1. Sintaxis y semántica

Definición 1. Sintaxis de fórmulas

El lenguaje *L* de *fórmulas bien formadas* para LP se define inductivamente por las siguientes reglas en notación BNF:

$$\alpha, \beta ::= p \mid \neg(\alpha) \mid (\alpha \wedge \beta) \mid (\alpha \vee \beta) \mid (\alpha \supset \beta)$$

donde p denota cualquier variable proposicional.

Definición 2. Literal

Un *literal* l es una variable proposicional o su negación; en el primer caso se dice que l es *positivo*, y en el segundo caso l es *negativo*.

Denotamos con $\overline{l}$ el complemento de l:

- Si $l = p$ entonces $\overline{l} = \neg p$
- Si $l = \neg p$ entonces $\overline{l} = p$

Definición 3. Forma Normal Conjuntiva (FNC)

Una fórmula $\alpha \in L$ se encuentra en *forma normal conjuntiva* ssi es una conjunción de disyunciones de literales:

$$\alpha = \bigwedge_{i \leq n} \bigvee_{j \leq m_i} l_{i,j} = \left(l_{1,1} \vee \ldots \vee l_{1,m_1}\right) \wedge \ldots \wedge \left(l_{n,1} \vee \ldots \vee l_{n,m_n}\right)$$

Ejemplos

La fórmula $(\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg r)$ es una FNC.

La fórmula $(\neg p \vee q \vee r) \wedge \neg(\neg q \vee r) \wedge (\neg r)$ no es una FNC, porque la segunda disyunción está negada.

La fórmula $(\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r)$ no es una FNC, porque $((p \wedge \neg q) \vee r)$ no es una disyunción de literales.

Teorema 1. Existencia de FNC
Para toda $\alpha \in L$ existe una *forma normal conjuntiva* $fnc(\alpha)$ tal que $\alpha \approx fnc(\alpha)$. [1]

Demostramos de manera constructiva la existencia de FNC equivalente para toda fórmula presentando un algoritmo basado en transformaciones. [2]

Algoritmo 1. FNC
Para convertir una fórmula $\alpha \in L$ arbitraria a $fnc(\alpha)$, realizar los siguientes pasos:

1. Eliminar todas las conectivas excepto $\wedge$, $\vee$ y $\neg$. Para esto se substituyen por equivalencias:
   - $\alpha \supset \beta \approx \neg\alpha \vee \beta$
2. Empujar las negaciones hacia adentro hasta que sólo queden sobre variables. Para esto se aplica De Morgan las veces que sea necesario:
   - $\neg(\alpha \wedge \beta) \approx (\neg\alpha \vee \neg\beta)$
   - $\neg(\alpha \vee \beta) \approx (\neg\alpha \wedge \neg\beta)$
3. Eliminar secuencias de negaciones aplicando la Doble Negación:
   - $\neg\neg\alpha \approx \alpha$
4. Eliminar las conjunciones que están dentro de las disyunciones. Para esto usamos la distributiva de $\vee$ sobre $\wedge$:
   - $\alpha \vee (\beta \wedge \gamma) \approx (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
   - $(\alpha \wedge \beta) \vee \gamma \approx (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Notar que todas las transformaciones se deben aplicar de izquierda a derecha.

Ejemplo
Sea $\alpha = (\neg p \supset \neg q) \supset (p \supset q)$. Entonces aplicando el algoritmo:

$\alpha \approx \neg(\neg\neg p \vee \neg q) \vee (\neg p \vee q)$, Paso 1: def. $\supset$ x3
$\approx (\neg\neg\neg p \wedge \neg\neg q) \vee (\neg p \vee q)$, Paso 2: De Morgan
$\approx (\neg p \wedge q) \vee (\neg p \vee q)$, Paso 3: doble neg. x2
$\approx (\neg p \vee \neg p \vee q) \wedge (q \vee \neg p \vee q)$, Paso 4: distr. $\vee$ sobre $\wedge$
$\approx fnc(\alpha)$

Además, vemos que hay oportunidades adicionales de simplificación:

$\approx (\neg p \vee \neg p \vee q) \wedge (q \vee q \neg p)$, Conmutatividad de $\vee$
$\approx (\neg p \vee q) \wedge (q \vee \neg p)$, Idempotencia de $\vee$
$\approx (\neg p \vee q) \wedge (\neg p \vee q)$, Conmutatividad de $\vee$
$\approx \neg p \vee q$, Idempotencia $\wedge$

[1] Usamos el símbolo $\approx$ para denotar la relación de equivalencia lógica entre formulas tal que $\approx \subseteq L \times L$.
[2] Los primeros tres pasos del algoritmo dejan α en FNN (forma normal negacional).

Teorema 2. Consecuencia lógica y condicional tautológico
Sean $\alpha \in L$ y $\Gamma \subseteq L$ con $\Gamma = \{\gamma_1, \dots, \gamma_n\}$:

$$\Gamma \vDash \alpha \quad \Leftrightarrow \quad \vDash \bigwedge_i \gamma_i \supset \alpha$$

Con este resultado podemos expresar el problema de determinar la validez de una consecuencia lógica como el problema de determinar si una fórmula es tautología.

Teorema 3. Dualidad Satisfacción-Tautología
Sea $\alpha \in L$:

$$\vDash \alpha \quad \Leftrightarrow \quad \neg\alpha \text{ unsat}$$

Una consecuencia útil de esta dualidad es que si tenemos un procedimiento de decisión para determinar si una fórmula es insatisfacible entonces también tenemos un procedimiento de decisión para determinar si una fórmula es tautología y viceversa.

1.2. Cláusulas y conjuntos de cláusulas

Definición 4. Forma Clausular (FC)
La *forma clausular* es una variante notacional de FNC, conveniente en ciertas manipulaciones sintácticas.

- Una *cláusula* c es un conjunto de literales que representa su disyunción implícita.
  $$c = \{\ell_1, \ell_2, \dots, \ell_n\} \quad \Leftrightarrow \quad \ell_1 \vee \ell_2 \vee \dots \vee \ell_n$$
  La cláusula vacía la denotamos con $\boxtimes$.
- Un conjunto de cláusulas $\mathcal{C}$ representa la conjunción implícita de sus cláusulas.
  $$\mathcal{C} = \{c_1, c_2, \dots, c_n\} \quad \Leftrightarrow \quad c_1 \wedge c_2 \wedge \dots \wedge c_n$$
  El conjunto de cláusulas vacío lo denotamos con $\emptyset$.

Notar que toda fórmula puede ser expresada en FC vía FNC, pero diferentes FNCs pueden tener la misma FC. La transformación de FNC a FC resulta en el colapso de múltiples ocurrencias de literales y cláusulas, pero esto no altera el valor de verdad. Más precisamente, FC es equivalente (lógicamente) a FNC *módulo* Asociatividad, Conmutatividad e Idempotencia de $\wedge$ y $\vee$.

Usaremos $fc(\alpha)$ para denotar la forma clausular de una fórmula α.

Ejemplo
Sea la FNC $\alpha = (p \vee r) \wedge \neg q \wedge (\neg p \vee p \vee q \vee p \vee \neg p) \wedge (r \vee p)$, entonces $fc(\alpha)$ es un conjunto $\mathcal{C}$ de tres clausulas:

$$\mathcal{C} = \{\ \{p, r\}, \{\neg q\}, \{\neg p, p, q\}\ \}$$

Definición 5. Cláusula Satisfacible

Una cláusula es satisfacible ssi existe al menos una interpretación bajo la cual hay un literal verdadero.

La cláusula vacía $\boxtimes$ es insatisfactible, porque no contiene ningún literal y entonces ningún literal es verdadero. Por lo tanto, $\boxtimes$ es la notación clausular equivalente a la constante lógica $\bot$ en el lenguaje de fórmulas: ambas tienen valor semántico $False$ bajo cualquier interpretación.

Definición 6. Conjunto de cláusulas válido

Un conjunto de cláusulas es válido ssi para toda cláusula en el conjunto la cláusula es verdadera bajo toda interpretación.

El conjunto de cláusulas vacío $\emptyset$ es válido, porque al no contener ninguna cláusula la definición se cumple vacuamente.

Definición 7. Clausula trivial

Una cláusula es *trivial* ssi contiene un par de literales complementarios.

Estas cláusulas se denominan triviales porque que no aportan al valor de verdad de un conjunto de cláusulas.

Ejemplo

$$\mathcal{C} = \{ \{p, \neg p, r\}, \{ p, r \}, \{ q \} \}$$

La clausula $\{p, \neg p, r\}$ es trivial y puede eliminarse de $\mathcal{C}$.

Esto es fácil de comprobar expresando $\mathcal{C}$ en el lenguaje de fórmulas:

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

Aplicando las equivalencias lógicas $(\alpha \vee \neg\alpha \approx \top)$, $(\alpha \vee \top \approx \top)$ y $(\alpha \wedge \top \approx \alpha)$ resulta:

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

que en forma clausular es simplemente $\mathcal{C} - \{p, \neg p, r\}$:

$$\{ \{ p, r \}, \{ q \} \}$$

1.3. Resolución

Definición 8. Regla de Resolución Proposicional (R)

Sean c_1, c_2 cláusulas tales que $l \in c_1$ y $\overline{l} \in c_2$. Se dice que las cláusulas c_1, c_2 son *conflictivas* y el conflicto es en el par de literales complementarios l y $\overline{l}$.
La *resolvente* es la cláusula que resulta de aplicar la siguiente regla sobre dos cláusulas conflictivas:

$$\mathrm{R}\frac{\{\, l \,\} \cup c_1' \quad \{\, \overline{l} \,\} \cup c_2'}{c_1' \cup c_2'}$$

Ejemplo
Las cláusulas $c_1 = \{p, q, \neg r\}$ y $c_2 = \{p, r, s\}$ tienen conflicto en el par de literales $\neg r, r$.
Al aplicar R sobre c_1 y c_2:

$$\mathrm{R}\frac{\{\, p,q,\neg r\} \quad \{p,r,s\}}{\{p,q,s\}}$$

obtenemos la resolvente $c = \{p, q, s\}$

Definición 9. Conjunto de cláusulas Saturado
Un conjunto de cláusulas $\mathcal{C}$ se encuentra *saturado* ssi para cada par de cláusulas de la forma $\{\, l \,\} \cup c_1$ y $\{\, \overline{l} \,\} \cup c_2$ entonces $c_1 \cup c_2$ ya se encuentra en $\mathcal{C}$.

En otras palabras, cuando un conjunto está saturado ya no se pueden agregar más resolventes porque ya se encuentran todas las resolventes posibles en el conjunto (esto puede incluir a ☒).

Ejemplo
El conjunto $\mathcal{C}$ está saturado:

$$\mathcal{C} = \{\, \{\neg p, \neg q, r\}, \{\, p \,\}, \{\, q \,\}, \{\, r \,\}, \{\, \neg q, r \,\}, \{\, \neg p, r \,\} \,\}$$

El conjunto $\mathcal{C}'$ no está saturado:

$$\mathcal{C}' = \{\, \{\neg q, \neg r, p\}, \{\neg q, r\}, \{q\}, \{\neg p, \neg q\}, \{\neg r, p\}, \{\neg r, \neg q\}, \{\neg q\} \,\}$$

Podemos, por ejemplo, resolver $R(\{\neg q, r\}, \{q\}) = \{r\}$ pero $\{r\} \notin \mathcal{C}'$.

Resolución como Sistema de demostración

La Resolución, al igual que Tableaux, es un sistema de refutación: para demostrar α se intenta refutar $\neg\alpha$ demostrando que $\neg\alpha$ es insatisfacible.

En el caso de Tableaux, las demostraciones son representadas como árboles, donde cada rama es la conjunción de las fórmulas que aparecen en ella, y el árbol es la disyunción de sus ramas. La Resolución trabaja con el concepto *dual*: una conjunción de disyunciones, pero la requiere desde el inicio del procedimiento. En la notación clausular de FNC, las disyunciones son expresadas como conjuntos de literales llamados *cláusulas*, y la conjunción es un conjunto de cláusulas.

El sistema de Resolución que consideraremos aquí sólo requiere de la regla *R* introducida en la definición 8. [3] Una demostración de α por Resolución comienza con $fc(\neg\alpha)$ como un conjunto inicial de cláusulas. A medida que la demostración progresa el conjunto se expande aplicando únicamente la regla *R* agregando las nuevas cláusulas (resolventes) al conjunto original. La regla *R* tiene la propiedad de preservar la satisfacibilidad del conjunto de cláusulas. Si luego de una cantidad suficiente de aplicaciones de la regla se deriva $\boxtimes$ (una contradicción), esto es suficiente para concluir que el conjunto original $fc(\neg\alpha)$ es insatisfacible (y entonces α es tautología). De lo contrario finalmente el conjunto quedará saturado y no se podrá expandir más, lo cual significa que no se puede demostrar α (y entonces α no es una tautología).

Definición. Deducción en R
Una *deducción* de la cláusula c a partir de un conjunto de cláusulas $\mathcal{C}$, denotado por $\mathcal{C} \vdash_R c$, es una secuencia finita de clausulas $c_1, \ldots, c_m$ tal que $c_m = c$ y cada c_i es o bien una cláusula de $\mathcal{C}$, o una resolvente de c_j, c_k con $j, k < i$.

Lema 1. Equivalencia
Sea $\mathcal{C}$ un conjunto de cláusulas tal que $c_1, c_2 \in \mathcal{C}$ son conflictivos y sea la resolvente $r = R(c_1, c_2)$. Entonces $\mathcal{C} \approx \mathcal{C} \cup \{r\}$.

Teorema 4. Solidez
Sea $\mathcal{C}$ un conjunto de clausulas, si $\mathcal{C} \vdash_R \boxtimes$ entonces $\mathcal{C}$ es unsat.

Teorema 5. Consistencia
Sea $\alpha \in L$, entonces se cumple $fc(\alpha) \vdash_R \boxtimes$ o $fc(\neg\alpha) \vdash_R \boxtimes$, pero no ambas.

[3] Existen variantes, por ejemplo si nuestras cláusulas fueran multiconjuntos (conjuntos que pueden tener repetidos) necesitaríamos de una regla de *factoreo* para operar con repetidos: si $c = \{l, l\} \cup c'$ es una clausula entonces $\{l\} \cup c$ es una nueva cláusula *factor* de c.

2. Algoritmo

La Resolución se presta con facilidad para un tratamiento mecánico [4], especialmente al tener sólo una regla. Presentamos un algoritmo de Resolución Proposicional que aplica la regla *R* hasta saturar el conjunto para finalmente tomar una decisión.

Algoritmo 2. Resolución
Entrada: Un conjunto de cláusulas $\mathcal{C}$
Salida: SAT o UNSAT
Descripción: Sea $\mathcal{C}_0 = \mathcal{C}$,

1) Mientras haya cláusulas conflictivas sin resolver, repetir los pasos a), b) y c) para obtener $\mathcal{C}_{i+1}$ a partir de $\mathcal{C}_i$:
   a) Elegir un par de cláusulas conflictivas $\{c_1, c_2\} \subseteq \mathcal{C}_i$ sin resolver.
   b) Computar $c = R(c_1, c_2)$
   c) $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{c\}$.
2) Si $\boxtimes \in \mathcal{C}_n$, retornar UNSAT
   De lo contrario, retornar SAT.

El procedimiento puede terminar de dos maneras, o bien el conjunto final $\mathcal{C}_n$ de cláusulas contiene la cláusula vacía $\boxtimes$ y retorna UNSAT, o no la contiene y retorna SAT. En ambos casos, $\mathcal{C}_n$ está saturado.

En la práctica, al derivar $\boxtimes$ en un paso del procedimiento, no es necesario continuar hasta saturar el conjunto. Esta observación puede usarse como un refinamiento más eficiente del algoritmo presentado.

[4] Resolución y Tableaux son métodos muy populares en la automatización de demostraciones. Ambos son clasificados como procedimientos *analíticos* que cumplen con el principio de subfórmula: en toda inferencia realizada la conclusión obtenida es subfórmula de sus premisas, de esta manera nada nuevo es introducido en la demostración aparte de lo que hay que demostrar. Esto contrasta con la Deducción Natural donde existe la posibilidad de introducir formulas arbitrarias.

2.1 Aplicación

Veremos dos ejemplos de resolución aplicada a la decisión de validez para entender mejor la mecánica del algoritmo y cómo se puede usar en la práctica. Esto a su vez será de ayuda para la etapa de implementación donde habrá que tomar decisiones de diseño.

Ejemplo 1

Queremos determinar si la siguiente relación de consecuencia lógica se cumple:

$$q \land r \supset p, \neg q \lor r, q \models ?\ p \land q$$

Por el teorema 2 queremos determinar si la siguiente formula, llamémosle α, es una tautología:

$$\alpha = (q \land r \supset p) \land (\neg q \lor r) \land q \supset (p \land q)$$

Trataremos de refutar su negación:

$$\neg\alpha \approx (q \land r \supset p) \land (\neg q \lor r) \land q \land \neg(p \land q)$$

Transformamos $\neg\alpha$ en FNC:

$$fnc(\neg\alpha) = (\neg q \lor \neg r \lor p) \land (\neg q \lor r) \land q \land (\neg p \lor \neg q)$$

Reescribimos $fnc(\neg\alpha)$ en FC, y éste será nuestro conjunto de cláusulas inicial:

$$\mathcal{C}_0 = \{\, \{\neg q, \neg r, p\}, \{\neg q, r\}, \{q\}, \{\neg p, \neg q\} \,\}$$

Ahora estamos en posición de aplicar el algoritmo de resolución. Etiquetamos las clausulas con números para llevar un registro de las resolventes generadas y las cláusulas usadas:

$$\mathcal{C}_0 = \{\, 1.\{\neg q, \neg r, p\}, \quad 2.\{\neg q, r\}, \quad 3.\{q\}, \quad 4.\{\neg p, \neg q\} \,\}$$

Ilustramos el procedimiento con un formato lineal numerado (izquierda) y como alternativa con un formato arborescente (derecha). La elección de uno u otro es cuestión de gusto personal.

5. $\{\, \neg r, p \,\}$ con 1,3
6. $\{\, \neg r, \neg q \,\}$ con 5,4
7. $\{\, \neg q \,\}$ con 6,2
8. $\boxtimes$ con 7,3

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\neg q, \neg r, p\} \quad \{q\}}{\{\, \neg r, p \,\} \quad \{\neg p, \neg q\}}}{\{\, \neg r, \neg q \,\} \quad \{\neg q, r\}}}{\{\, \neg q \,\} \quad \{q\}}}{\boxtimes}$$

Se ha realizado un paso de refutación al inferir $\boxtimes$. Terminamos el procedimiento habiendo realizado una demostración por refutación.

$$\mathcal{C}_3 = \{\, 1.\{\neg q, \neg r, p\}, 2.\{\neg q, r\}, 3.\{q\}, 4.\{\neg p, \neg q\}, 5.\{\neg r, p\}, 6.\{\neg r, \neg q\}, 7.\{\neg q\} \,\}$$

El resultado es que $\neg\alpha$ es UNSAT. Por el teorema 3 esto implica que α es tautología, por lo tanto se cumple la consecuencia lógica:

$$q \land r \supset p, \neg q \lor r, q \models p \land q$$

Ejemplo 2

Queremos determinar si la siguiente relación de consecuencia lógica se cumple:

$$p \wedge q \supset r, p, q \models ? \neg r$$

Por el teorema 2 queremos determinar si la siguiente fórmula, llamémosle α, es una tautología:

$$\alpha = (p \wedge q \supset r) \wedge p \wedge q \supset \neg r$$

Trataremos de refutar su negación:

$$\neg\alpha \approx (p \wedge q \supset r) \wedge p \wedge q \wedge r$$

Transformamos $\neg\alpha$ en FNC:

$$fnc(\neg\alpha) = (\neg p \vee \neg q \vee r) \wedge p \wedge q \wedge r$$

Reescribimos $fnc(\neg\alpha)$ en FC, y éste será nuestro conjunto de cláusulas inicial:

$$\mathcal{C}_0 = \{\, \{\neg p, \neg q, r\}, \{\, p \,\}, \{\, q \,\}, \{\, r \,\} \,\}$$

Ahora estamos en posición de aplicar el algoritmo de resolución. Etiquetamos las cláusulas con números para llevar un registro de las resolventes generadas y las cláusulas usadas:

$$\mathcal{C}_0 = \{\, 1.\{\neg p, \neg q, r\}, \quad 2.\{\, p \,\}, \quad 3.\{\, q \,\}, \quad 4.\{\, r \,\} \,\}$$

Ilustramos el procedimiento con un formato lineal numerado (izquierda) y como alternativa con un formato arborescente (derecha).

5. $\{\, \neg q, r \,\}$ con 1,2
6. $\{\, r \,\}$ con 5,3
7. $\{\, \neg p, r \,\}$ con 1,3
8. $\{\, r \,\}$ con 7,2

$$\dfrac{\dfrac{\{\neg p, \neg q, r\} \quad \{\, p \,\}}{\{\, \neg q, r \,\} \quad \{\, q \,\}}}{\{\, r \,\}} \qquad \dfrac{\dfrac{\{\neg p, \neg q, r\} \quad \{\, q \,\}}{\{\, \neg p, r \,\} \quad \{\, p \,\}}}{\{\, r \,\}}$$

El procedimiento termina luego de realizar cuatro pasos, porque ya no quedan cláusulas conflictivas sin resolver: $\mathcal{C}_4$ está saturado.

$$\mathcal{C}_4 = \{\, 1.\{\neg p, \neg q, r\}, 2.\{\, p \,\}, 3.\{\, q \,\}, 4.\{\, r \,\}, 5.\{\, \neg q, r \,\}, 7.\{\, \neg p, r \,\} \,\}$$

Aquí no agregamos las cláusulas 6 y 8 porque son repetidos de la cláusula 4.

El resultado es que $\neg\alpha$ es SAT. Por el teorema 3 esto implica que α no es tautología, por lo tanto no se cumple la consecuencia lógica:

$$p \wedge q \supset r, p, q \not\models \neg r$$

3. Implementación

La implementación consiste en la definición de tipos de datos adecuados para nuestro problema (sección 3.1) y luego varias funciones que agrupamos en tres categorías principales (secciones 3.2, 3.3, 3.4).

3.1. Tipos de datos

Definimos tipos de datos y alias de tipos convenientes para modelar los conceptos definidos en la primera parte.

Fórmulas lógicas

Una variable proposicional es un string, con lo cual tenemos una cantidad infinita de variables a nuestra disposición.

```
type V = String
```

Una formula se define de acuerdo a la notación BNF de la definición 1.

```
data F = Atom V | Neg F | Conj F F | Disy F F | Imp F F
```

Un razonamiento es un par compuesto por una lista (posiblemente vacía) de fórmulas consideradas premisas y una formula considerada la conclusión.

```
type Statement = ([F],F)
```

Literales, clausulas y conflictos

Un literal es una variable proposicional positiva o negativa.

```
data L = LP V | LN V
```

Una cláusula es una lista de literales.

```
type C = [L]
```

La cláusula vacía es simplemente `[]`.

Un conjunto de cláusulas es una lista de cláusulas, o sea una lista de listas.

```
type CSet = [C]
```

Ejemplo
Un conjunto de cláusulas que contiene la cláusula vacía tiene la forma `[c1,c2,…,[]]` (donde `c1` y `c2` son otras cláusulas cualesquiera).

Un conflicto entre dos cláusulas es una cuádrupla donde primer y tercer componente son las clausulas conflictivas y el segundo y cuarto componente son literales complementarios tales que el segundo pertenece a la primero y el cuarto pertenece al tercero.

```
type Clash = (C,L,C,L)
```

Modelar conjuntos utilizando listas significa que el orden en el que disponemos los elementos al construir listas no es importante (al menos desde el punto de vista de la corrección). Pero las operaciones que realizamos sobre estas listas deben asegurar un comportamiento consistente con el que esperamos de trabajar con conjuntos; en particular es necesario contemplar la presencia de repetidos y tomar medidas correctivas.[5]

[5] Podríamos considerar hacer uso del tipo `Set` de Haskell, pero para nuestros propósitos no es realmente necesario.

3.2. Interface

Funciones:

- `sat :: F -> Bool`
- `tau :: F -> Bool`
- `valid :: Statement -> Bool`

Estas son las funciones públicas exportadas por la implementación con las que interactúa el usuario. La más importante es `sat`, ya que las funciones `tau` y `valid` pueden definirse en términos de `sat` via teoremas 2 y 3.

La función `sat` debe convertir la formula parámetro a FC usando la función `f2CSet`, y luego aplicarle el algoritmo de resolución usando la función `resolveCSet`, con lo cual se obtiene una lista de cláusulas tal que si contiene `[]` (la cláusula vacía) el resultado debe ser `False`, de lo contrario debe ser `True`.

3.3. Fórmulas y cláusulas

Funciones:

- `f2CSet :: F -> CSet`
- `cnf2CSet :: F -> CSet`
- `f2cnf :: F -> F`

Éstas son funciones de soporte necesarias para poder aplicar el algoritmo de resolución sobre fórmulas. La función `f2cnf` convierte una fórmula a su forma normal conjuntiva, usando por ejemplo el Algoritmo 1. La función `cnf2CSet` convierte `una` forma normal conjuntiva a un conjunto de cláusulas. La función `f2CSet` convierte una formula a un conjunto de cláusulas (ésta puede ser definida en términos de `f2cnf`, `cnf2CSet` y quizá algo más).

3.4. Resolución

Funciones:

- `resolveCSet :: CSet -> CSet`
- `resolveClash :: Clash -> C`

Estas funciones son la esencia del algoritmo de resolución. La función `resolveCSet` recibe un conjunto de cláusulas y busca pares de cláusulas con conflicto para aplicar la regla *R* llamando a la función `resolveClash`. Esto se repite hasta saturar el conjunto de cláusulas. De la respuesta final, es decir si el resultado es SAT o UNSAT, se encarga la función pública `sat` mencionada anteriormente.

Mencionamos dos posibles estrategias que pueden ser consideradas para la implementación del algoritmo:

1. **Resolución por saturación**
   Se implementa la función auxiliar
   ```
   resolveClashes :: CSet -> CSet
   ```
   Esta función resuelve todos los conflictos que hay en un conjunto de cláusulas y retorna sus resolventes.
   Luego de aplicar suficientes veces la regla *R*, el conjunto de cláusulas finalmente quedará *saturado* (incluyendo o no la cláusula `[]`), llegado este momento aplicar la regla *R* no generará ninguna clausula que ya no esté en el conjunto.
   La función `resolveCSet` puede detectar cuándo el conjunto de cláusulas ha quedado *saturado* si se compara el largo del conjunto antes y después de invocar `resolveClashes` sobre el conjunto de cláusulas actual. Si estos son iguales entonces el conjunto ya está saturado y el procedimiento termina retornando el conjunto de cláusulas. De lo contrario `resolveCSet` debe invocarse a sí misma nuevamente con el nuevo conjunto de cláusulas para repetir lo anterior.

2. **Resolución paso a paso con retorno temprano**
   Se implementa la función auxiliar
   ```
   findClashes :: CSet -> [Clash]
   ```
   Esta función retorna todos los conflictos existentes en un conjunto de cláusulas.
   La función `resolveCSet` utiliza `findClashes` para obtener los conflictos en el conjunto de cláusulas actual, la idea es resolver uno a uno. Pero aquí se deben ignorar los conflictos ya resueltos, de lo contrario el procedimiento nunca terminaría, para esto `resolveCSet` puede ir acumulando los conflictos ya resueltos y restárselos al resultado de `findClashes` en cada invocación.
   Si luego de resolver un conflicto la resolvente es `[]`, se retorna el conjunto actual de cláusulas (posiblemente no saturado) con `[]` agregada.
   Cuando ya no hay conflicto por resolver se retorna el conjunto actual de cláusulas que estará saturado.

En cualquier caso, es importante ver que no alcanza con buscar únicamente los conflictos existentes en el conjunto de cláusulas inicial al comenzar el procedimiento, porque cada aplicación de la regla *R* genera una nueva cláusula resolvente que puede también tener conflictos con otras cláusulas previas.

3.5. Detalles de implementación

La descripción del algoritmo 2 es una descripción de muy *alto nivel*, es decir tiene un nivel de detalle suficiente para ser comprendido por un humano (esto es fácil de comprobar viendo los ejemplos de aplicación manual del algoritmo) pero no suficiente para ser comprendido por una máquina. La implementación que nos interesa es una descripción del algoritmo en un lenguaje de programación, particularmente Haskell. Pero no hay una sola implementación posible, si pensamos en la descripción del algoritmo 2 como una especificación de lo que el algoritmo debe hacer entonces hay detalles que requerimos pero no están presentes, por ejemplo podemos preguntarnos: si hay más de un par de cláusulas conflictivas, ¿cual debemos resolver primero?, o si hay más de un conflicto en una clausula, ¿sobre cual debemos aplicar la regla? No hay una sola respuesta a este tipo de preguntas,

diferentes decisiones resultaran en diferentes implementaciones, algunas más eficientes que otras.

A continuación se exploran algunas cuestiones de diseño que pueden ser útiles para la implementación.

Orden de resolución

La selección de un par conflictivo para resolver no necesita de un criterio especial desde el punto de vista de la corrección. Trabajando con listas se pueden resolver conflictos recorriendo naturalmente la lista de cláusulas de izquierda a derecha y concatenando las resolventes al final (o al inicio) de la lista actual de cláusulas.

Existen alternativas más elaboradas. Por ejemplo, se le puede dar prioridad a las cláusulas más chicas (con menos literales), con este criterio si existiera un paso de refutación (una derivación posible de [ ]) este tendría mayor prioridad y el procedimiento podría terminar con mayor antelación. Como desventaja, esto hace la implementación más complicada, requiriendo posiblemente una estructura de datos mejor equipada que una lista.

Múltiples conflictos en una clausula

Un par de cláusulas conflictivas puede tener más de un conflicto, por ejemplo las cláusulas $c_1 = \{p, \neg q, \neg r\}$ y $c_2 = \{p, r, q\}$ tienen conflicto en el par de literales $\neg r, r$ y en $\neg q, q$.

Por suerte, no es necesario hacer resolución sobre cláusulas que tengan más de un conflicto (aunque no es incorrecto). Si dos cláusulas conflictivas tienen más de un conflicto, la resolvente es *trivial*. Supongamos que tenemos un par de cláusulas:

$$c_1 = \{ l_1, l_2\} \cup c_1' \qquad c_2 = \{ \overline{l_1}, \overline{l_2} \} \cup c_2'$$

Aplicamos *R* sobre el conflicto $l_1, \overline{l_1}$ y obtenemos la resolvente:

$$c = \{ l_2, \overline{l_2} \} \cup c_1' \cup c_2'$$

Pero la resolvente *c* es trivial de acuerdo a la definición 7, y puede ser eliminada del conjunto de cláusulas sin afectar su valor de verdad. Por lo tanto, las cláusulas con más de un conflicto pueden ser ignoradas como candidatos a resolver.

Tratamiento de cláusulas

Las cláusulas son conjuntos, y las pretendemos modelar con listas, por lo tanto no queremos conservar repetidos. Por un lado, mantener los repetidos no aporta al valor de verdad de una cláusula (notar que $l \vee l \approx l$). Por otro lado, y más importante, puede provocar la no terminación de la implementación, como se puede apreciar en el siguiente ejemplo.

Consideremos la lista de cláusulas:

$$[\,[\neg p, q, r], [\neg p, r], [p, \neg q], [q, r]\,]$$

Podemos realizar una derivación sin fin:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\neg p, r],\quad [p, \neg q]}{[r, \neg q] \qquad [\neg p, q, r]}}{[r, \neg p, r] \qquad [p, \neg q]}}{[r, r, \neg q] \qquad [\neg p, q, r]}}{[r, r, \neg p, r] \qquad [p, \neg q]}}{[r, r, r, \neg q] \qquad [\neg p, q, r]}}{\vdots}$$

En el tercer paso se obtiene la resolvente $[r, \neg p, r]$, pero ésta no debería ser tratada como una nueva cláusula, de esta manera se siguen generando nuevas cláusulas con repetidos indefinidamente.

Una simplificación opcional que se puede realizar durante la conversión de fórmulas a cláusulas (antes de la resolución) es eliminar las cláusulas triviales. [6]

Tratamiento del conjunto de cláusulas

De manera similar a lo que sucede dentro de las cláusulas, tampoco queremos conservar cláusulas repetidas dentro del conjunto de cláusulas (notar que $\alpha \wedge \alpha \approx \alpha$).

Funciones de utilidad

Puede resultar de utilidad implementar las funciones:

- `sameElems :: (Eq a) => [a] -> [a] -> Bool`
  Comparación de listas sin importar el orden.
- `compl :: L -> L`
  Devuelve el complemento de un literal.

[6] Esta eliminación de redundancia también se puede realizar durante la resolución, pero formalmente esto significa tener una nueva regla de simplificación, y es preferible mantener el algoritmo lo más coherente posible con el sistema deductivo considerado.

4. Entrega

Para esta entrega se pide:

- Demostrar el lema 1 y los teoremas 4 y 5. Se pueden entregar en formato digital o papel.
- Implementar un procedimiento de decisión para SAT en fórmulas de la Lógica Proposicional basado en el algoritmo de resolución. Para esto, implementar las funciones definidas en el archivo template `Resolucion.hs`.
  Se sugiere seguir estos pasos:
  1) Implementar las funciones relacionadas con la manipulación de fórmulas: `f2CSet` y `cnf2CSet`.
  2) Implementar las funciones que se encargan del procedimiento de resolución: `resolveCSet`, `resolveClash`. En este punto, puede seguirse alguna de las recomendaciones dadas en el punto 3.4 u otro mecanismo que le parezca conveniente.
  3) Implementar las funciones `sat`, `tau` y `valid.`

En el archivo `Test.hs` se encuentra un juego de pruebas unitarias que podrá utilizar para testear la implementación. Tener en cuenta que las pruebas no son exhaustivas y por lo tanto, la correcta ejecución de todos los casos no garantiza la correctitud del programa. Se recomienda repasar los casos de prueba y generar propios.

Las pruebas dependen de HUnit (https://hackage.haskell.org/package/HUnit), un framework de pruebas unitarias basado en JUnit.
Para ejecutarlas:

1) Instalar HUnit.
2) Cargar el archivo `Test.hs`
3) Ejecutar comando: `runTestTT allTests`

Condiciones:

- La tarea tiene un puntaje máximo de 28 puntos.
- Puede realizarse en grupo de hasta 2 personas.
- La entrega del obligatorio debe hacerse por Aulas hasta el día 29/06/2018 a las 23:30hs.
- Todos los integrantes del grupo deberán cargar una copia de la entrega. Ésta deberá incluir en el encabezado del archivo `Resolucion.hs`, nombre y número de estudiante de ambos integrantes.
- Todos los archivos de código serán sujetos a verificación de plagio y copia. En los casos donde se detecte que efectivamente hubo plagio o copia de la tarea, todos los estudiantes involucrados serán responsables y quedarán sujetos a las acciones disciplinarias que la Universidad considere correspondientes.