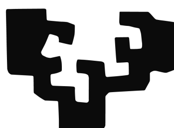
eman ta zabal zazu



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática

Computación

Proyecto de Fin de Grado

Restricciones de igualdad sobre el dominio de los naturales

Autor

Jon Ander García

Directores

Hubert Chen y Montserrat Hermo

informatika fakultatea

facultad de informática

2015

Agradecimientos

En primer lugar quisiera dar las gracias a mis directores, Hubert Chen y Montserrat Hermo por ofrecerme este proyecto y ayudarme a llevarlo adelante.

Gracias a mis padres, por su apoyo incondicional a lo largo de la carrera y durante toda mi vida. Ellos me han permitido estudiar lo que ansiaba desde pequeño.

Agradezco también el apoyo de mi pareja, Sarai, a la que tanto he aburrido explicándole cosas del proyecto.

Gracias a todos aquellos que hayan colaborado o aportado su granito de arena en el proyecto.

Resumen

El objetivo de este proyecto consiste en implementar una aplicación que busca todos los modelos de una fórmula recibida como entrada. Esta fórmula de entrada pertenece a la lógica NatEq, la cual es un subconjunto de la lógica de primer orden. Esta lógica consta de un sólo predicado, que es la igualdad. Además, no contiene símbolos de funciones, incluye los cuantificadores universal y existencial, y se interpreta sobre el dominio de los números naturales $\mathbb{N}$.

Índice general

Resumen I

Índice general III

Índice de figuras V

Indice de tablas VII

1. Introducción 1

2. Restricciones de igualdad sobre el dominio de los naturales 3

2.1. Sintaxis . . . 3

2.2. Semántica . . . 5

2.3. Forma Normal Negativa . . . 7

2.4. Conjunto de modelos normalizados . . . 8

3. Implementación 19

3.1. Módulo sintáctico . . . 19

3.1.1. Analizador léxico . . . 19

3.1.2. Gramática . . . 21

3.1.3. Abstracciones funcionales . . . 25

3.1.4. Atributos . . . 26

3.1.5. Esquema de Traducción Dirigido por la Sintaxis 26

3.2. Módulo semántico 30

3.2.1. Estructura de datos de las fórmulas 30

3.2.2. Estructura de datos auxiliar 38

4. Pruebas y resultados **45**

4.1. Pruebas unitarias 45

4.1.1. Igualdades 45

4.1.2. Operadores 46

4.1.3. Negación 48

4.1.4. Cuantificadores 49

4.2. Pruebas globales 50

5. Conclusiones **53**

6. Bibliografía **55**

Anexos

Índice de figuras

3.1. AFD del analizador léxico. 21

3.2. Diagrama de clases UML de Formula. 31

3.3. Diagrama de clases UML de la estructura auxiliar. 39

Indice de tablas

3.1. Los diferentes tokens 20
3.3. Atributos de la gramática. 26
3.5. Atributos de la clase Formula. 32
3.6. Métodos de la clase Formula. 32
3.7. Atributos de la clase Negation. 33
3.8. Métodos de la clase Negation. 33
3.9. Atributos de la clase EqualityOperator. 33
3.10. Métodos de la clase EqualityOperator. 33
3.11. Métodos de la clase Equals. 34
3.12. Métodos de la clase Notequals. 34
3.13. Atributos de la clase LogicalOperator. 35
3.14. Métodos de la clase LogicalOperator. 35
3.15. Métodos de la clase And. 35
3.16. Métodos de la clase Or. 36
3.17. Métodos de la clase Implication. 36
3.18. Métodos de la clase Biconditional. 37
3.19. Atributos de la clase Quantifier. 37
3.20. Métodos de la clase Quantifier. 37

3.21. Métodos de la clase Exists. 38
3.22. Métodos de la clase Forall. 38
3.23. Atributos de la clase Relation. 39
3.24. Métodos de la clase Relation. 42
3.25. Atributos de la clase Tupla. 42
3.26. Métodos de la clase Tupla. 43
3.27. Métodos de la clase VarList. 44

4.1. Resultado de las pruebas globales. 51

1. CAPÍTULO

Introducción

El objetivo de este proyecto consiste en implementar una aplicación que busca todos los modelos de una fórmula recibida como entrada. Esta fórmula de entrada pertenece a la lógica NatEq, la cual es un subconjunto de la lógica de primer orden. Esta lógica consta de un sólo predicado, que es la igualdad. Además, no contiene símbolos de funciones, incluye los cuantificadores universal y existencial, y se interpreta sobre el dominio de los números naturales $\mathbb{N}$.

La aplicación consta de dos partes, la sintáctica y la semántica. La sintáctica se encarga de reconocer la fórmula de entrada y construir una estructura virtual que la parte semántica resolverá por partes e irá hallando todos los modelos de la fórmula de entrada.

La estructura de la memoria se divide en tres partes: La sección de teoría, donde se explica la lógica *NatEq*; la sección donde se explica la implementación de la aplicación; la sección de pruebas. Finalmente, se incluye un anexo con un pequeño tutorial sobre cómo ejecutar y usar la aplicación.

2. CAPÍTULO

Restricciones de igualdad sobre el dominio de los naturales

La Lógica Ecuacional que se presenta y que llamaremos *NatEq*, consta de un único predicado: la igualdad. Incluye los cuantificadores universal y existencial, y su interpretación será sobre el dominio de los números naturales $\mathbb{N}$. Esta lógica es un subconjunto de la lógica de primer orden, que nos permitirá describir afirmaciones tales como que algunos "objetos" son distintos o iguales, o que todos ellos son el mismo o lo contario. Un ejemplo más concreto puede ser el siguiente:

$$\text{Para todo } x \neq z \text{ existe algún } y\text{, tal que } y = x.$$

De aquí en adelante se formalizará el marco sintáctico y semántico donde analizaremos las fórmulas de *NatEq*.

2.1. Sintaxis

El alfabeto de *NatEq* consta de los siguientes símbolos:

- Variables: $V = \{x_1, x_2, x_3, \ldots\}$
- Predicados: $\{=\}$
- Conectivos lógicos: $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$

- $\neg$: Negación
- $\wedge$: Conjunción
- $\vee$: Disyunción
- $\Rightarrow$: Implicación
- $\Leftrightarrow$: Doble implicación

- Cuantificadores: $\{\forall, \exists\}$
  - $\forall$: Cuantificador universal
  - $\exists$: Cuantificador existencial

- Símbolos impropios: Paréntesis

La noción de fórmula se define inductivamente con las siguientes reglas:

1. Siendo x_1 y x_2 variables, entonces $(x_1 = x_2)$ es una fórmula.
2. Si F es una fórmula, entonces $\neg F$ también lo es[1].
3. Si F y G son fórmulas, entonces $(F \wedge G)$ y $(F \vee G)$, también lo son.
4. Si F es una fórmula y x una variable, entonces $(\forall x F)$ y $(\exists x F)$ son fórmulas.

Las fórmulas atómicas son aquellas que se han construido de acuerdo a la regla 1. Si F es una fórmula y F ocurre como parte de la fórmula G, entonces F es una subfórmula de G.

El alcance de un cuantificador de una fórmula es la subfórmula a la cual afecta dicho cuantificador.

Todas las ocurrencias de una variable en una fórmula se distinguen por ser ocurrencias libres o ligadas. Una ocurrencia de la variable x en la fórmula F es ligada si x ocurre dentro de una subfórmula de F con la forma de $\forall x G$ o $\exists x G$. Una ocurrencia de una variable es libre si no está ligada. Una variable está libre en una fórmula si tiene al menos una ocurrencia libre en ella. Una fórmula se denomina cerrada o sentencia si no tiene ocurrencias de una variable libre.

Para interpretar las fórmulas de *NatEq* (i.e. para darles una semántica, esto es, un "significado"), es necesario interpretar las variables en un dominio fijo. Nuestro dominio es

[1]A partir de ahora la fórmula $\neg(x_1 = x_2)$ la escribiremos $(x_1 \neq x_2)$

el conjunto de los números naturales, $\mathbb{N}$, el cual es infinito. Cuando interpretamos todas las variables libres de un fórmula sobre $\mathbb{N}$, entonces podemos establecer un siginificado, es decir, un valor de verdad para la fórmula. Esta explicación intuitiva se hará formal a continuación.

2.2. Semántica

En la lógica *NatEq*, una interpretación $\mathcal{I}$ es una aplicación parcial de las variables de V en $\mathbb{N}$.

En otras palabras, $\mathcal{I}$ es una aplicación cuyo dominio es un subconjunto de V y cuyo rango es un subconjunto de $\mathbb{N}$. En lo que sigue, se abreviará la notación de forma que escribiremos $x^{\mathcal{I}}$ en vez de $\mathcal{I}(x)$.

El único predicado que tiene *NatEq*, la $=$, se interpretará como la igualdad clásica entre naturales.

Fijado el conjunto de variables $V = \{x_1, x_2, \ldots, x_n\}$, para expresar las variables libres de una fórmula, a veces será conveniente indicar las posiciones que ocupan estas variables dentro del conjunto V. Por ello, dada cualquier fórmula F, el conjunto $var(F) = \{i_1, i_2, \ldots i_k\}$ indicará, de forma ordenada, las posiciones dentro de V de las variables libres de F. Dicho de otra forma, si $var(F) = \{i_1, i_2, \ldots i_k\}$, el conjunto de variables libres de F es $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ y además $\forall j(1 \leq j < k \rightarrow i_j < i_{j+1})$.

Sea F una fórmula y sea $\mathcal{I}$ una interpretación. Se dice que $\mathcal{I}$ es apropiada para F si $\mathcal{I}$ está definida para todas las variables libres de F. Además, como veremos a continuación, el valor de verdad de F según $\mathcal{I}$, dependerá sólo del valor de $\mathcal{I}$ sobre cada variable libre de F.

Ejemplo:
Sea $F \equiv (\forall x_1((x_1 = x_2) \vee (x_1 \neq x_2)))$ una fórmula. $var(F) = \{2\}$ porque la variable x_2 es la única libre en F. Un ejemplo de interpretación $\mathcal{I}$ apropiada para F es la siguiente:

$$\mathcal{I}(x_2) = x_2^{\mathcal{I}} = 0.$$

Sea F una fórmula e $\mathcal{I}$ una interpretación apropiada para F. Para cada subfórmula atómica f que ocurra en F, se denota su valor de verdad bajo la interpretación $\mathcal{I}$ como $\mathcal{I}(f)$, y se define de la siguiente manera:

Si f es de la forma $(x = y)$, donde x e y son variables

$$\mathcal{I}(f) = \begin{cases} cierto & \text{si } x^{\mathcal{I}} = y^{\mathcal{I}} \\ falso & \text{si } x^{\mathcal{I}} \neq y^{\mathcal{I}} \end{cases}$$

Análogamente, se define el valor de una fórmula no atómica F, denotada $\mathcal{I}(F)$, bajo la interpretación $\mathcal{I}$ con una definición inductiva:

1. Si F es de la forma $\neg G$, donde G es una fórmula, entonces

$$\mathcal{I}(F) = \begin{cases} falso & \text{si } \mathcal{I}(G) = cierto \\ cierto & \text{si } \mathcal{I}(G) = falso \end{cases}$$

2. Si F es de la forma $(G \wedge H)$, donde G y H son fórmulas, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si } \mathcal{I}(G) = cierto \text{ y } \mathcal{I}(H) = cierto \\ falso & \text{en otro caso.} \end{cases}$$

3. Si F es de la forma $(G \vee H)$, donde G y H son fórmulas, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si } \mathcal{I}(G) = cierto \text{ ó } \mathcal{I}(H) = cierto \\ falso & \text{en otro caso.} \end{cases}$$

4. Si F es de la forma $(G \Rightarrow H)$, donde G y H son fórmulas, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si } \mathcal{I}(G) = falso \text{ ó } \mathcal{I}(H) = cierto \\ falso & \text{en otro caso.} \end{cases}$$

5. Si F es de la forma $(G \Leftrightarrow H)$, donde G y H son fórmulas, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si } \mathcal{I}(G) = cierto \text{ y } \mathcal{I}(H) = cierto \\ cierto & \text{si } \mathcal{I}(G) = falso \text{ y } \mathcal{I}(H) = falso \\ falso & \text{en otro caso.} \end{cases}$$

6. Si F es de la forma $(\forall x G)$, donde G es una fórmula, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si para todo } n \in \mathbb{N}, \mathcal{I}[n/x](G) = cierto \\ falso & \text{en otro caso.} \end{cases}$$

Aquí, $\mathcal{I}[n/x]$ es una interpretación $\mathcal{I}'$, la cual es idéntica a $\mathcal{I}$ con la excepción de la definición de $x^{\mathcal{I}'}$: No importa si $\mathcal{I}$ está definida en x o no, se cumple que $x^{\mathcal{I}'} = n$.

7. Si F tiene la forma $(\exists x G)$, donde G es una fórmula, entonces

$$\mathcal{I}(F) = \begin{cases} cierto & \text{si existe al menos un } n \in \mathbb{N}, \text{ tal que } \mathcal{I}[n/x](G) = cierto \\ falso & \text{en otro caso.} \end{cases}$$

Aquí, $\mathcal{I}[n/x]$ es como se ha definido en el punto anterior.

Si para una fórmula F y una interpretación apropiada $\mathcal{I}$ tenemos $\mathcal{I}(F) = cierto$, entonces se denota como $\mathcal{I} \models F$ (se dice que, F es verdad en $\mathcal{I}$, o $\mathcal{I}$ es modelo de F). Si toda interpretación apropiada para F es modelo de F, entonces se denota como $\models F$ (F es válida), en otro caso $\not\models F$. Si al menos hay un modelo para la fórmula F, entonces se dice que F es satisfactible, y en otro caso, insatisfactible o contradictoria.

A partir de ahora, y siempre que no haya confusión, cuando hablemos de una interpretación para una fórmula, supondremos que es una interpretación apropiada para ella.

2.3. Forma Normal Negativa

A continuación se presentan unas reglas de transformación estándar en la lógica de primer orden. Según la semántica definida en el apartado anterior es fácil comprobar que estas reglas son válidas.

Reglas válidas de *NatEq*:

Dadas dos fórmulas F y G, las siguientes fórmulas son válidas en *NatEq*:

- $\neg\neg F \Leftrightarrow F$
- $\neg(F \wedge G) \Leftrightarrow (\neg F \vee \neg G)$
- $\neg(F \vee G) \Leftrightarrow (\neg F \wedge \neg G)$
- $\neg(\forall x F) \Leftrightarrow (\exists x \neg F)$
- $\neg(\exists x F) \Leftrightarrow (\forall x \neg F)$

Dadas dos fórmulas F y G, diremos que F y G son equivalentes cuando $F \Leftrightarrow G$ es una fórmula válida.

Aplicando las reglas anteriores y un razonamiento inductivo es fácil demostrar que cualquier fórmula es equivalente a otra donde las negaciones están únicamente delante de las subfórmulas atómicas.

Definición: Se dice que F está en Forma Normal Negativa si F no contiene dobles negaciones y todas las negaciones que aparecen en F están delante de subfórmulas atómicas.

Ejemplo:
La fórmula $\neg(\forall x((x = y) \vee (\exists z(x \neq z))))$ no está en Forma Normal Negativa, pero es equivalente a otra fórmula que sí lo está:

$$(\exists x((x \neq y) \wedge (\forall z(x = z)))).$$

Teorema de la Forma Normal Negativa
Toda fórmula de *NatEq* es equivalente a otra fórmula de *NatEq* que está en Forma Normal Negativa.

Demostración del teorema:
Se base en las reglas válidas descritas anteriormente. ■

2.4. Conjunto de modelos normalizados

El propósito de este trabajo es encontar todas las interpretaciones que son modelo de una fórmula dada, las cuales pueden ser una cantidad infinita. Para ello, tendremos que representar una cantidad infinita de interpretaciones de manera finita.

Dada una fórmula F, con

$$var(F) = \{i_1, i_2, \ldots, i_k\}$$

vamos a representar cada interpretación $\mathcal{I}$ para F con la tupla:

$$(x^{\mathcal{I}}_{i_1}, x^{\mathcal{I}}_{i_2}, \ldots, x^{\mathcal{I}}_{i_k})$$

Ejemplo:
Sea F la fórmula $F \equiv ((x_1 = x_2) \wedge (x_1 \neq x_3))$, con $var(F) = \{1,2,3\}$. La interpretación $(2,2,3)$ es modelo de F. Esto es, $(2,2,3) \models F$. También $(4,4,0) \models F$, pero $(6,6,6) \not\models F$.

La fórmula del ejemplo es satisfactible y hay una cantidad infinita de modelos para ella. Sin embargo, podemos representar a todos ellos de una única forma.

Modelo normalizado

Definición de tupla normalizada
Diremos que una tupla $(a_1, a_2, \ldots, a_k)$ de naturales está normalizada cuando se verifique:

1. $a_1 = 0$.
2. Dado cualquier j con $1 < j \leq k$, se verifica que si $a_j \notin \{a_1, a_2, \ldots, a_{j-1}\}$, entonces $a_j = 1 + max\{a_1, a_2, \ldots, a_{j-1}\}$.

Definición de modelo normalizado
Dada una fórmula F con $var(F) = \{i_1, i_2, \ldots, i_k\}$, diremos que la tupla $(a_1, a_2, \ldots, a_k)$ de naturales es un modelo normalizado para F cuando se verifique:

1. $(a_1, a_2, \ldots, a_k)$ es una tupla normalizada.
2. $(a_1, a_2, \ldots, a_k) \models F$.

En el caso del ejemplo, el único modelo normalizado para F sería $(0, 0, 1)$.

Se puede demostrar que si una fórmula tiene un modelo, entonces tiene un modelo normalizado.

Teorema del modelo normalizado
Dada una fórmula F, con $var(F) = \{i_1, i_2, \ldots, i_k\}$, se puede demostrar que existe un modelo para F si y sólo si existe un modelo normalizado para F.

Demostración del teorema
Obviamente, si hay un modelo normalizado para una fórmula, entonces existe un modelo para ella. Falta demostrar que si existe un modelo para F, entonces también hay un modelo normalizado. La demostración es constructiva. Supongamos que $\mathcal{I}$ es el modelo para F, representado por la tupla $(a_1, a_2, \ldots, a_k)$. El modelo normalizado $(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k)$ (lo denotaremos $\hat{\mathcal{I}}$) se construye de la siguiente forma:

Función Normalizar $((a_1, a_2, \ldots, a_k))$
Construcción del modelo normalizado a partir del modelo $\mathcal{I}$.

$\hat{\mathcal{I}} := (?, ?, \ldots, ?)$;
$max := 0$;
for i in $1..k$ **loop**
  if $\forall j(1 \leq j < i \rightarrow a_j \neq a_i)$ **then**
    Cambiar ? por max en todas las posiciones p de $\hat{\mathcal{I}}$ tales que $\hat{a_p} = \hat{a_i}$;
    $max := max + 1$;
  end if;
end loop;
devolver $\hat{\mathcal{I}}$;

Tenemos que justificar que la tupla $\hat{\mathcal{I}}$ constuida es un modelo normalizado para F.

1. El bucle mantiene como invariante que para cualquier i y j se verifica que

$$a_i = a_j \Longleftrightarrow \hat{a}_i = \hat{a}_j$$

   Por lo tanto $\hat{\mathcal{I}} \models F$.

2. $a_1 = 0$ por construcción.

3. El bucle mantiene como invariante que al terminar la i-esima vuelta, con $i \geq 2$

$$max = max\{\hat{a_1}, \hat{a_2}, \ldots, \hat{a_{i-1}}\} + 1.$$

   En el caso $i = 1$, $max = 1$. Por lo tanto, $\hat{a_i}$ es algún valor previo que es menor o igual a $max - 1$ o bien, en el cuerpo del bucle, se le asigna la cantidad max. ■

Ejemplo

$\mathcal{I} = \{3, 2, 1, 2, 0, 1\}$
$\hat{\mathcal{I}} = \{?, ?, ?, ?, ?, ?\}$
$\hat{\mathcal{I}} = \{0, ?, ?, ?, ?, ?\}$ $max = 0; i = 1$
$\hat{\mathcal{I}} = \{0, 1, ?, 1, ?, ?\}$ $max = 1; i = 2$
$\hat{\mathcal{I}} = \{0, 1, 2, 1, ?, 2\}$ $max = 2; i = 3$
$\hat{\mathcal{I}} = \{0, 1, 2, 1, 3, 2\}$ $max = 3; i = 4$
$\hat{\mathcal{I}} = \{0, 1, 2, 1, 3, 2\}$ $max = 3; i = 5$

$\hat{\mathcal{I}} = \{0,1,2,1,3,2\}$ $max = 3; i = 6$

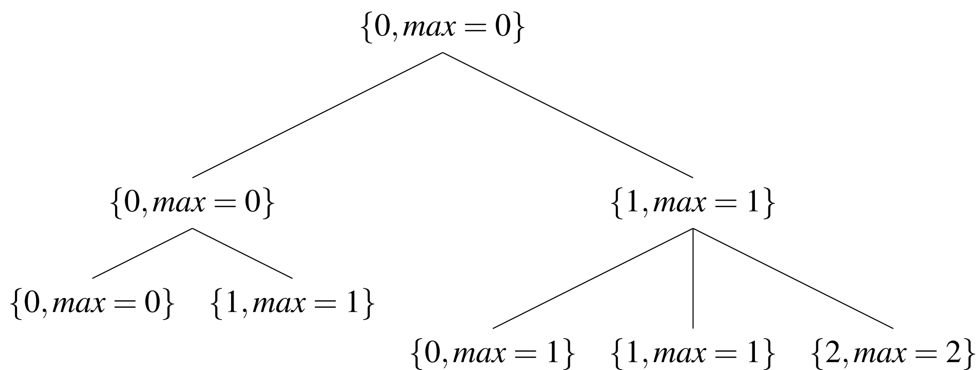
Conjunto de modelos normalizados para una fórmula válida

Supongamos que F es la fórmula $((x_1 = x_2) \vee (x_1 \neq x_2)) \wedge ((x_1 = x_3) \vee (x_1 \neq x_3))$ con $var(F) = \{1,2,3\}$. F es válida y por tanto todas las interpretaciones para F son modelos de la misma. Sin embargo, por el teorema anterior sabemos que podemos representar al conjunto de todos los modelos con el conjunto de todos los modelos normalizados. En este caso el conjunto:

$$\mathcal{T}_3 = \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(0,1,2)\}$$

No hay más modelos normalizados con tres variables, porque a x_1 necesariamente hay que asignarle un 0 (punto (1) de la definición de tupla normalizada). Fijado el valor de x_1, a la variable x_2 sólo se le puede asignar o bien 0, o bien 1 para cumplir el punto (2) de la definición. Si a x_2 se le asigna el valor 0, el máximo de valores asignados hasta ahora es 0, y para cumplir el punto (2) de la definición, a x_3 sólo se le puede asignar o bien el 0 o bien el 1. Sin embargo, si a x_2 se le asigna el valor 1, entonces x_3 puede moverse entre los valores $\{0,1,2\}$.

Todos los modelos normalizados para tres variables se pueden construir fácilmente por medio de un árbol $\mathcal{A}_3$.

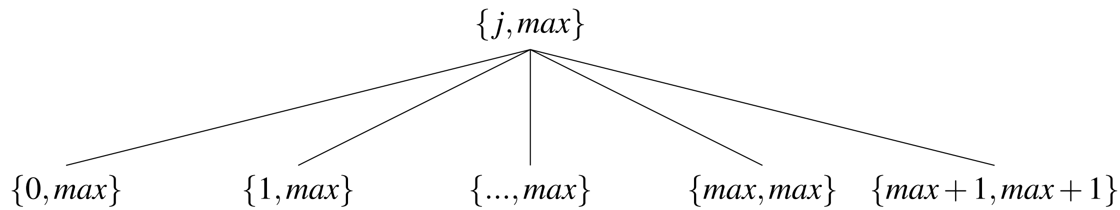


Estas ideas se pueden generalizar a cualquier número de variables. Por ejemplo el árbol $\mathcal{A}_4$ tendría un nivel más donde cada hoja tendría los hijos correspondientes. Por ejemplo la hoja $\{2, max = 2\}$ pasaría a tener cuatro hijos: $\{0, max = 2\}, \{1, max = 2\}, \{2, max = 2\}, \{3, max = 3\}$.

El conjunto de todos los modelos normalizados para 4 variables sería el siguiente:

$$\mathcal{T}_4 = \left\{ \begin{array}{lllll} (0,0,0,0), & (0,0,0,1), & (0,0,1,0), & (0,0,1,1), & (0,0,1,2), \\ (0,1,0,0), & (0,1,0,1), & (0,1,0,2), & (0,1,1,0), & (0,1,1,1), \\ (0,1,1,2), & (0,1,2,0), & (0,1,2,1), & (0,1,2,2), & (0,1,2,3) \end{array} \right\}$$

En general la construcción de $\mathcal{T}_n$ se hace a partir del árbol $\mathcal{A}_n$. Mediante $\mathcal{A}_n$ somos capaces de construir todos los modelos de $\mathcal{T}_n$. Cada camino del árbol $\mathcal{A}_n$ genera uno de los modelos de $\mathcal{T}_n$, y todos los modelos de $\mathcal{T}_n$ se pueden obtener a partir de los diferentes caminos del árbol $\mathcal{A}_n$. Siguiendo los diferentes caminos del árbol, el nodo de profundidad i indica el valor que va a tener en la posición i el modelo generado a partir de ese camino, donde $1 \leq i \leq n$. Un nodo a profundidad $i-1$, con valor j, y máximo valor max en el camino que va de la raíz a j, genera $max+2$ hijos a profundidad i, de los cuales $max+1$ nodos tendrán un valor del intervalo $[0, max]$ sin que ese valor se repita entre sus hermanos y un máximo valor hasta el momento de max. El otro nodo, tendrá un valor de $max+1$ y un máximo valor del camino de $max+1$.



La primera posición de cada nodo indica el valor que tiene, y la segunda el máximo valor del camino hasta el momento.

Sabiendo cómo se construye el árbol $\mathcal{A}_n$, podemos calcular la cantidad de modelos que forman el conjunto $\mathcal{T}_n$, pues éstos coinciden con el número de hojas de $\mathcal{A}_n$.

Número de modelos posibles para n variables

Definimos el valor $f(m,k)$ como el número de modelos posibles a los que les falta por asignar las $m+1$ últimas variables $\{x_{i_1}, \ldots, x_{i_{m+1}}\}$, y que hasta ahora han asignado k como valor máximo. Por ejemplo el cardinal del conjunto $\mathcal{T}_3$ es exactamente $f(2,0)$ porque faltan por asignar 3 variables y el máximo valor asignado hasta ahora es 0. El cardinal de $\mathcal{T}_4$ es exactamente $f(3,0)$ y así sucesivamente.

Vamos a encontrar la regla de recurrencia que debe cumplir la función f. En general para calcular $f(m,k)$ podemos asignar a la variable x_{i_1}

(1) o bien los valores del intervalo $[0,k]$, quedando por asignar las variables $\{x_{i_2},\ldots,x_{i_{m+1}}\}$ y habiendo usado como valor máximo k.

(2) o bien el valor $k+1$, quedando por asignar las variables $\{x_{i_2},\ldots,x_{i_{m+1}}\}$ y habiendo usado como valor máximo $k+1$.

El cálculo de (1) se corresponde con $(k+1)f(m-1,k)$ mientras que el cálculo de (2) se corresponde con $f(m-1,k+1)$. Por tanto la regla sería:

$$f(m,k) = (k+1)f(m-1,k) + f(m-1,k+1)$$

Para $m = 1$, $f(1,k) = k+2$. La recurrencia quedaría:

$$f(0,k) = 1$$

$$f(m,k) = (k+1)f(m-1,k) + f(m-1,k+1)$$

Efectivamente, $|\mathcal{T}_2| = f(1,0) = 2$; $|\mathcal{T}_3| = f(2,0) = f(1,0) + f(1,1) = 2+3 = 5$. Aplicando la recurrencia $|\mathcal{T}_4| = f(3,0) = f(2,0)+f(2,1) = 5+f(2,1) = 5+2f(1,1)+f(1,2) = 5+6+4 = 15$. Para 5 variables, $|\mathcal{T}_5| = f(4,0) = f(3,0)+f(3,1) = 15+f(3,1) = 15+2f(2,1)+f(2,2) = 15+20+f(2,2) = 35+f(2,2) = 35+3f(1,2)+f(1,3) = 35+12+5 = 52$.

Para terminar esta sección vamos a dar una idea del orden de crecimiento de la función $f(m,k)$. Usando una prueba por inducción sobre m, se pude demostrar la siguiente propiedad.

Propiedad: $\forall m \geq 0\ \forall k \geq 0 : (k+1)^m \leq f(m,k) \leq (k+m+1)^m$

Conjunto de modelos normalizados para cualquier fórmula

En esta sección vamos a explicar cómo se construye el conjunto de todos los modelos normalizados de una fórmula de *NatEq*.

Previamente, vamos a fijar alguna notación relativa a operaciones sobre tuplas.

Definiciones de operaciones sobre tuplas
Sean $\bar{a} = (a_1,a_2,\ldots,a_m)$ y $\bar{b} = (b_1,b_2,\ldots,b_r)$ dos tuplas de naturales.

- La concatenación de dos tuplas: $\bar{a} \bullet \bar{b} = (a_1,a_2,\ldots,a_m,b_1,b_2,\ldots,b_r)$.

- La permutación de una tupla: $\mathcal{P}_{\{i_1,i_2,\ldots,i_m\}}(\bar{a}) = (a_{i_1}, a_{i_2}, \ldots, a_{i_m})$. La permutación $\mathcal{P}_{\{1,2,\ldots,m\}}$, que se corresponde con el orden preestablecido entre las m variables, se identificará con $\mathbb{P}$.
- La permutación de un conjunto de tuplas: $\mathcal{P}_{\{i_1,i_2,\ldots,i_m\}}(\mathrm{E}) = \{\mathcal{P}_{\{i_1,i_2,\ldots,i_m\}}(\bar{a}) : \bar{a} \in E\}$. $\mathbb{P}$(E) es la reordenación de todas las tuplas de E según el orden preestablecido entre las variables.
- La proyección de una tupla: $\mathcal{J}_i(\bar{a}) = \mathbf{Normalizar}((a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_m))$
- La proyección de un conjunto de tuplas: $\mathcal{J}_i(E) = \{\mathcal{J}_i(\bar{a}) : \bar{a} \in E\}$
- La expansión n-ésima de un conjunto de tuplas: $\mathcal{E}_n(A)$ se define inductivamente.
  - $\mathcal{E}_0(A) = \{\bar{a} : \bar{a} \in A\}$.
  - $\mathcal{E}_{n+1}(A) = \{(b_1, \ldots, b_r) \bullet (z) : (b_1, \ldots, b_r) \in \mathcal{E}_n(A) \wedge 0 \leq z \leq 1 + max\{b_1, \ldots, b_r\ \}\ \}$.

Ejemplo

Si $A = \{(0,0,1)\}$, entonces $\mathcal{E}_0(A) = \{(0,0,1)\}$; $\mathcal{E}_1(A) = \{(0,0,1,0),(0,0,1,1),(0,0,1,2)\}$;

$$\mathcal{E}_2(A) = \left\{ \begin{array}{llll} (0,0,1,0,0), & (0,0,1,0,1), & (0,0,1,0,2), & \\ (0,0,1,1,0), & (0,0,1,1,1), & (0,0,1,1,2), & \\ (0,0,1,2,0), & (0,0,1,2,1), & (0,0,1,2,2), & (0,0,1,2,3) \end{array} \right\}$$

y así sucesivamente.

Definición de la unión ordenada de variables libres

Dadas dos fórmulas G y H, con $var(G) = \{i_1, i_2, \ldots i_m\}$ y $var(H) = \{j_1, j_2, \ldots j_r\}$. El conjunto $var(G) + var(H)$ corresponde a la unión conjuntista de los índices de $var(G)$ y $var(H)$ donde el orden establecido es primero los índices de G, y después los índices de H que no estén en G.

Ejemplo

Supongamos $V = \{x_1, x_2, x_3, x_4, x_5\}$, $G \equiv ((x_1 = x_2) \wedge (x_1 \neq x_5))$ y $H \equiv (x_3 = x_1)$. Entonces $var(G) = \{1,2,5\}$, $var(H) = \{1,3\}$ y $var(G) + var(H) = \{1,2,5,3\}$. La permutación $\mathbb{P}$, que se refiere al orden preestablecido por V es exactamente $\mathcal{P}_{\{1,2,3,5\}}$.

La definición de unión ordenada permire asegurar la siguiente propiedad.

Propiedad de la unión ordenada de variables libres

Dadas dos fórmulas G y H, los conjuntos $var(G) + var(H)$ y $var(H) + var(G)$ contienen las mismas variables, aunque puede ser que en diferente orden.

A continuación vamos a definir, por inducción en la estructura de las fórmulas de *NatEq*, el conjunto de todos los modelos de una fórmula F. Lo denotaremos por $\mathcal{M}_F$.

Por motivos de eficiencia, la construcción se hace a partir de la fórmula equivalente a F que está en Forma Normal Negativa. Por tanto, la negación sólo aparecerá en las subfórmulas atómicas. La construcción es como sigue:

Modelos normalizados de una fórmula de *NatEq*

- Si F es de la forma $(x = x)$, entonces $\mathcal{M}_F = \{(0)\}$.
- Si F es de la forma $(x \neq x)$, entonces $\mathcal{M}_F = \varnothing$.
- Si F es de la forma $(x = z)$, entonces $\mathcal{M}_F = \{(0,0)\}$.
- Si F es de la forma $(x \neq z)$, entonces $\mathcal{M}_F = \{(0,1)\}$.
- Sean G y H dos fórmulas, con $|var(G) - var(H)| = p$, $|var(H) - var(G)| = q$, donde $\mathcal{E}_q(\mathcal{M}_G)$ contiene la expansión de los modelos de G a las variables $var(G) + var(H)$ y $\mathcal{E}_p(\mathcal{M}_H)$ contiene la expansion de los modelos de H a las variables $var(H) + var(G)$.
  - Si F es de la forma $(G \wedge H)$, entonces
  $$\mathcal{M}_{G \wedge H} = \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G))) \cap \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)))$$
  - Si F es de la forma $(G \vee H)$, entonces
  $$\mathcal{M}_{G \vee H} = \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G))) \cup \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)))$$
- Si F es de la forma $F = (\forall x_j\ G)$, con $var(G) = \{g_1, g_2, \ldots, g_i, \ldots, g_k\}$ y $j = g_i$, entonces $\mathcal{M}_F = \{\bar{g} : \bar{g} \in \mathcal{J}_i(\mathcal{M}_G) \wedge \mathcal{E}_1(\bar{g}) \in \mathcal{P}_{\{g_1, g_2, \ldots, g_{i-1}, g_{i+1}, \ldots, g_k, g_i\}}(\mathcal{M}_G)\}$
- Si F es de la forma $F = (\exists x_j\ G)$, con $var(G) = \{g_1, g_2, \ldots, g_i, \ldots, g_k\}$ y $j = g_i$, entonces $\mathcal{M}_F = \{\bar{g} : \bar{g} \in \mathcal{J}_i(\mathcal{M}_G)\}$

Ejemplo
Sean G y H las formulas del ejemplo anterior. El conjunto de variables libres es $V = \{x_1, x_2, x_3, x_4, x_5\}$, $G \equiv ((x_1 = x_2) \wedge (x_1 \neq x_5))$ y $H \equiv (x_3 = x_1)$. Entonces,

- $var(G) = \{1, 2, 5\}$

- $var(H) = \{1,3\}$
- $|var(G) - var(H)| = |\{2,5\}| = 2$
- $var(H) + var(G) = \{1,3,2,5\}$, Este es el orden de la expansión $\mathcal{E}_2(\mathcal{M}_H)$.
- $|var(H) - var(G)| = |\{3\}| = 1$
- $var(G) + var(H) = \{1,2,5,3\}$. Este es el orden de la expansión $\mathcal{E}_1(\mathcal{M}_G)$.
- $\mathcal{M}_G = \{(0,0,1)\}$
- $\mathcal{M}_H = \{(0,0)\}$
- $\mathbb{P}(\mathcal{E}_1(\mathcal{M}_G)) = \mathbb{P}(\mathcal{E}_1(\{(0,0,1)\}) = \mathbb{P}(\{(0,0,1,0),(0,0,1,1),(0,0,1,2)\})$
  $= \{(0,0,0,1),(0,0,1,1),(0,0,2,1)\}$. Esta última igualdad es debida a que se permuta el orden de las variables $\{1,2,5,3\}$ por $\{1,2,3,5\}$. Una vez normalizado el conjunto obtenemos $\{(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{1}),(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{2})\}$.
- $\mathbb{P}(\mathcal{E}_2(\mathcal{M}_H)) = \mathbb{P}(\mathcal{E}_2(\{(0,0)\}) = \mathbb{P}(\mathcal{E}_1(\{(0,0,0),(0,0,1)\})$
  $= \mathbb{P}\{(0,0,0,0),(0,0,0,1),(0,0,1,0),(0,0,1,1),(0,0,1,2)\})$
  $= \{(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{2})\}$, que ya se encuentra normalizado. Esta última igualdad es debida a que se permuta el orden de las variables $\{1,3,2,5\}$ por $\{1,2,3,5\}$.
- $\mathcal{M}_{G\wedge H} = \{(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1})\}$
- $\mathcal{M}_{G\vee H} = \{(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{1}),(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{2}),(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{2})\}$

Ejemplo

Sean las fórmulas F, G y H. El conjunto de variables libres es $V = \{x_1,x_2,x_3\}$, $F \equiv \forall x_2 H$, $G \equiv \exists x_2 H$ y $H \equiv ((x_1 = x_2) \vee (x_1 \neq x_3))$. Entonces,

- $var(H) = \{1,2,3\}$
- $\mathcal{M}_H = \{(0,0,0),(0,0,1),(0,1,1),(0,1,2)\}$
- $\mathcal{P}_{\{1,3,2\}}(\mathcal{M}_H) = \{(0,0,0),(0,1,0),(0,1,1),(0,2,1)\}$
- **Normalizar**$(\mathcal{P}_{\{1,3,2\}}(\mathcal{M}_H)) = \{(0,0,0),(0,1,0),(0,1,1),(0,1,2)\}$
- $\mathcal{J}_2(\mathcal{M}_H) = \{(0,0),(0,1)\}$

- $\mathcal{E}_1((0,0)) = \{(0,0,0),(0,0,1)\}$
- $\mathcal{E}_1((0,1)) = \{(0,1,0),(0,1,1),(0,1,2)\}$
- $\mathcal{M}_F = \{(0,0)\}$. Esto es debido a que no todos los modelos de la expansión $(0,1)$ se encuentran en **Normalizar**$(\mathcal{P}_{\{1,3,2\}}(\mathcal{M}_H))$.
- $\mathcal{M}_G = \mathcal{J}_2(\mathcal{M}_H) = \{(0,0),(0,1)\}$

Vamos a justificar la corrección de la construcción de los modelos normalizados de una fórmula de *NatEq*.

La construcción de los modelos normalizados de una fórmula es correcta

- Si F es de la forma $(x \neq x)$, es obvio que no hay ningún modelo normalizado F. Por lo tanto, $\mathcal{M}_F = \varnothing$.
- Si F es de la forma $(x = x)$, el único modelo normalizado de F es $\mathcal{M}_F = \{(0)\}$.
- Si F es de la forma $(x = z)$ ó $(x \neq z)$, es obvio que el único modelo normalizado de F es $\mathcal{M}_F = \{(0,0)\}$ ó $\mathcal{M}_F = \{(0,1)\}$ respectivamente.
- Si F es de la forma $(G \wedge H)$ ó $(G \vee H)$, vamos a justificar que cada paso que damos en la construcción de $\mathcal{M}_{G\wedge H}$ ó $\mathcal{M}_{G\vee H}$ es correcto. Los conjuntos $\mathcal{E}_q(\mathcal{M}_G)$ y $\mathcal{E}_p(\mathcal{M}_H)$ contienen la expansión de los modelos de G a las variables $var(G) + var(H)$ y la expansión de los modelos de H a las variables $var(H) + var(G)$ respectivamente. Según la propiedad de la unión ordenada de variables libres, ambos conjuntos contienen las mismas variables aunque en diferente orden. Si aplicamos la reordenación $\mathbb{P}$ a ambos conjuntos, conseguimos el mismo orden en los dos. Por tanto, $\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G))$ y $\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)$ contienen todos los modelos de G expandidos a las variables de H y todos los modelos de H expandidos a las variables de G respectivamente, y además en el mismo orden. Por lo tanto, **Normalizar**$(\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G)))$ contiene todos los modelos normalizados de G expandidos a las variables de H y **Normalizar**$(\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)))$ contiene todos los modelos normalizados de H expandidos a las variables de G. Como el orden de las variables en las tuplas de estos cojuntos es el mismo, si intersecamos ambos obtenemos todos los modelos normalizados de $G \wedge H$. De la misma forma, si unimos ambos conjuntos obtenemos todos los modelos normalizados de $G \vee H$. Concluimos pues que

$$\mathcal{M}_{G\wedge H} = \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G))) \cap \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)))$$

$$\mathcal{M}_{G\vee H} = \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_q(\mathcal{M}_G))) \cup \textbf{Normalizar}(\mathbb{P}(\mathcal{E}_p(\mathcal{M}_H)))$$

son definiciones correctas.

- Si F es de la forma $\forall x_j G$, vamos a justificar que cada paso que damos en la construcción de $\mathcal{M}_F$ es correcto. Como $j = g_i$, el conjunto $\mathcal{J}_i(\mathcal{M}_G)$ contiene la proyección de los modelos de G en la variable x_j, y el conjunto **Normalizar**($\mathcal{P}_{\{g_1,g_2,\ldots,g_{i-1},g_{i+1},\ldots,g_k,g_i\}}(\mathcal{M}_G))$ contiene los modelos de G de tal manera que la variable x_j está representada en la última posición. Así pues, $\mathcal{M}_F$ estará compuesto por aquellos modelos de $\mathcal{J}_i(\mathcal{M}_G)$ cuya expansión estén totalmente contenidas en **Normalizar**($\mathcal{P}_{\{g_1,g_2,\ldots,g_{i-1},g_{i+1},\ldots,g_k,g_i\}}(\mathcal{M}_G))$.
- Si F es de la forma $\exists x_j G$, vamos a justificar que cada paso que damos en la construcción de $\mathcal{M}_G$ es correcto. Como $j = g_i$, el conjunto $\mathcal{J}_i(\mathcal{M}_G)$ contiene la proyección de los modelos de G en la variable x_j. Por lo tanto, $\mathcal{M}_F = \mathcal{J}_i(\mathcal{M}_G)$.

3. CAPÍTULO

Implementación

En esta sección se presenta la implementación, que está estructurada en dos paquetes o módulos principales. El módulo sintáctico, el parser, que analiza las fórmulas introducidas por la entrada estándar y se encarga de reconocerlas, y el módulo semántico, se encarga de buscar todos los modelos de las fórmulas reconocidas.

3.1. Módulo sintáctico

En el módulo sintáctico, se va a definir una sola clase, que es la principal del programa. En ella se implementa un Autómata Finito Determinista, que será el analizador léxico, para determinar los diferentes tipos de tokens que se encuentran en la entrada, y una gramática concreta, para generar la estructura de la fórmula al terminar cada regla de derivación. Este módulo sintáctico nos permite escribir las fórmulas sin tener que escribirla entre paréntesis a pesar de que la lógica no esté definida de esa manera. Esto se ha hecho para una mayor comodidad a la hora de escribir las fórmulas.

3.1.1. Analizador léxico

Se definen los diferentes tokens admitidos y se reconocen mediante el Autómata Finito Determinista.

La siguiente tabla muestra el nombre de los diferentes tokens y la cadena de caracteres que le corresponde a cada uno.

Nombre	**Cadena de caracteres**
Id	[A-Za-z][A-Za-z0-9]*
Implicación	->
DobleImplicación	<->
ParaTodo	@
Existe	%
ParentAbrir	(
ParentCerrar	)
Igual	=
Diferente	!
Conjunción	&
Disyunción	|
Negación	¬

Tabla 3.1: Los diferentes tokens

En la siguiente imagen, se observa los estados y transiciones del AFD.

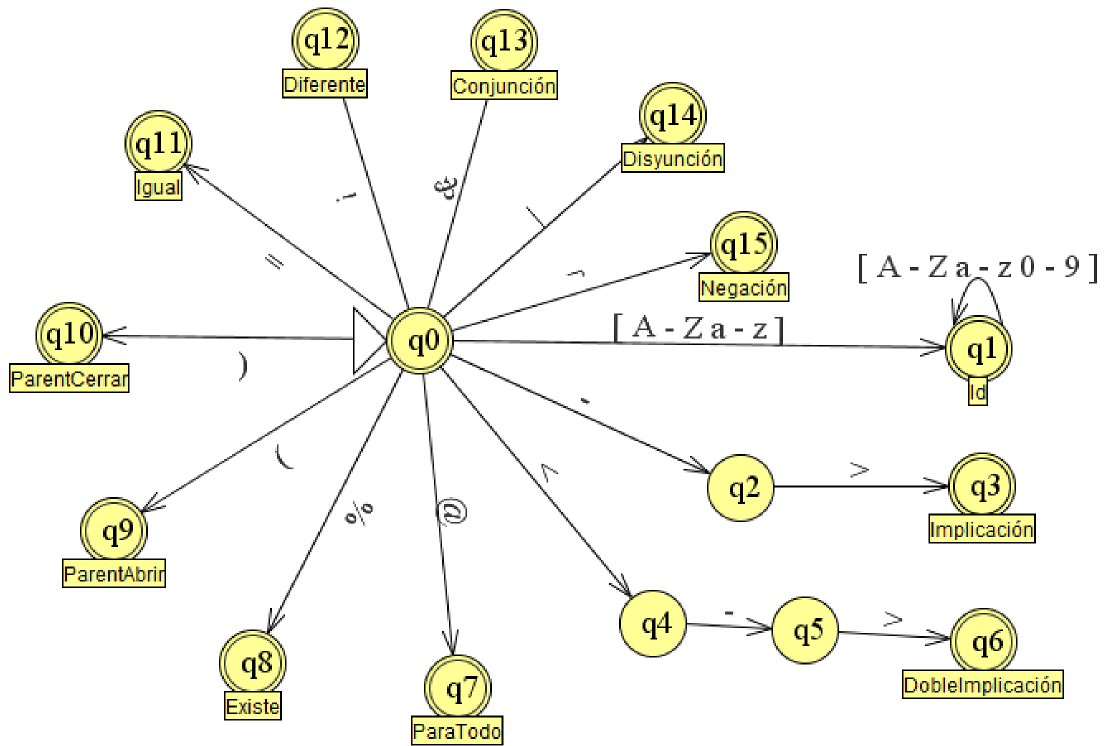


Figura 3.1: AFD del analizador léxico.

3.1.2. Gramática

A continuación se presenta la gramática no ambigua, en la notación de Backus-Naur, que se ha implementado para parsear la entrada. Esta gramática hace un análisis sintáctico descendente predictivo.

<F> ::= <G><F2>

<F2> ::= “<->” <F>

|

<G> ::= <H><G2>

<G2> ::= “->” <G>

|

<H> ::= <I><H2>

<H2> ::= “|” <H>

|

<I> ::= <J><I2>

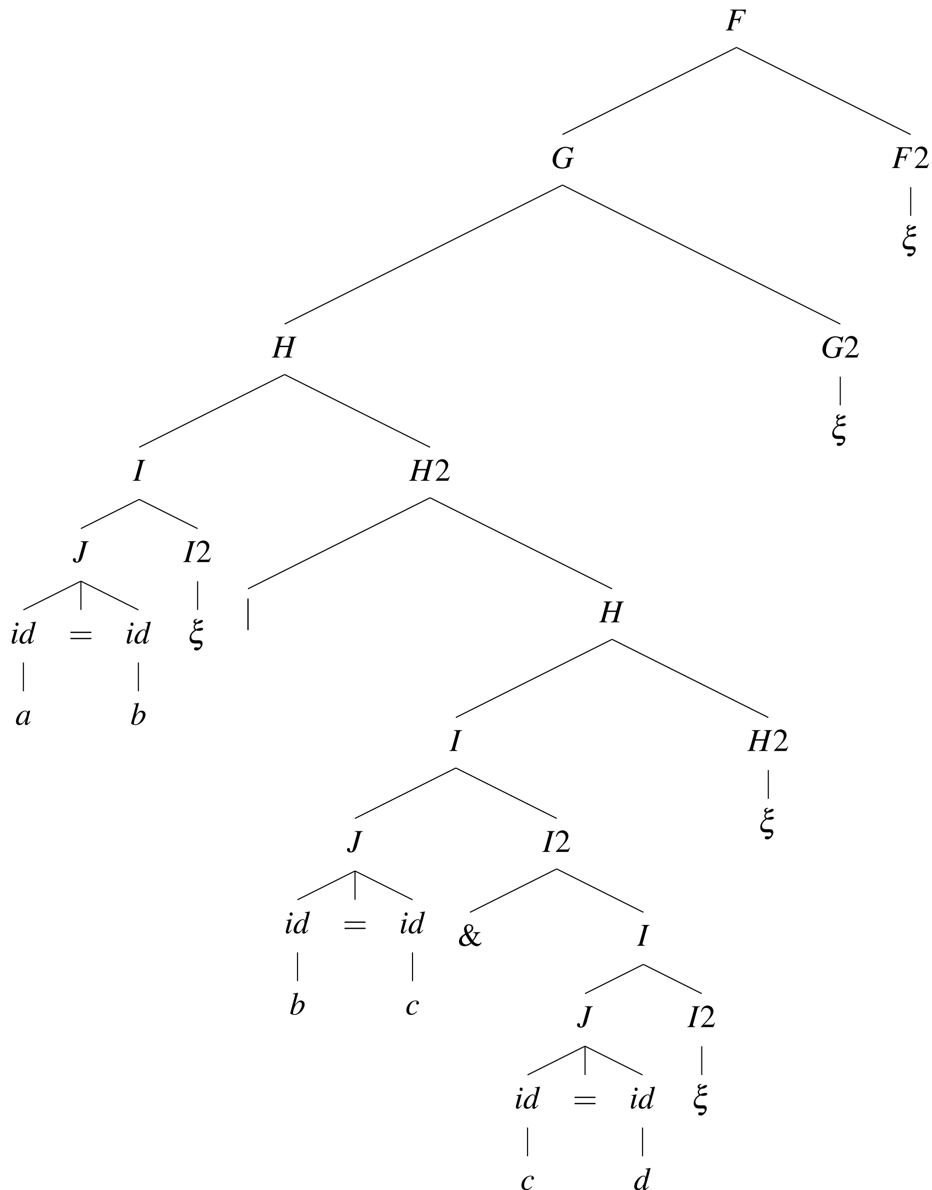
<I2> ::= “&” <I>
|
<J> ::= “¬” <J>
| “@” <id>“(” <F>“)”
| “%” <id>“(” <F>“)”
| “(” <F>“)”
| <id>“!” <id>
| <id>“=” <id>
<id> ::= <letra><idAux>
<idAux> ::= <letra><idAux>
| <digito><idAux>
|
<letra> ::= “a”
| “b”
| “c”
| “d”
| “e”
| “f”
| “g”
| “h”
| “i”
| “j”
| “k”
| “l”
| “m”
| “n”
| “o”
| “p”
| “q”
| “r”
| “s”
| “t”
| “u”
| “v”
| “w”
| “x”

| “y”
| “z”
| “A”
| “B”
| “C”
| “D”
| “E”
| “F”
| “G”
| “H”
| “I”
| “J”
| “K”
| “L”
| “M”
| “N”
| “O”
| “P”
| “Q”
| “R”
| “S”
| “T”
| “U”
| “V”
| “W”
| “X”
| “Y”
| “Z”
<digito> ::= “0”
| “1”
| “2”
| “3”
| “4”
| “5”
| “6”
| “7”

| “8”
| “9”

Las reglas de derivación <id>, <idAux>, <letra>y <digito>sólo están puestas para ilustrar cómo se derivan. En la implementación de la gramática no existen dichas reglas, ya que es el analizador léxico quien se encarga de ello.

Ejemplo derivación
a=b | b=c & c=d



3.1.3. Abstracciones funcionales

A medida que la entrada se va analizando carácter por carácter y se van identificando los tokens, se va generando la estructura de la fórmula. Para ello, es necesario ejecutar unas instrucciones después de la derivación de cada regla. Dichas instrucciones utilizarán la estructura de datos definida en el módulo semántico. Las diferentes instrucciones a ejecutar son:

- Biconditional(Formula f1, Formula f2): genera un objeto Biconditional con las fórmulas f1 y f2 como atributos.
- Implication(Formula f1, Formula f2): genera un objeto Implication con las fórmulas f1 y f2 como atributos.
- Or(Formula f1, Formula f2): genera un objeto Or con las fórmulas f1 y f2 como atributos.
- And(Formula f1, Formula f2): genera un objeto And con las fórmulas f1 y f2 como atributos.
- Negation(Formula f1): genera un objeto Negation con la fórmulas f1 como atributo.
- Forall(String varName, Formula f): genera un objeto Forall con el nombre de variable dado por varName y con la fórmula f como atributos.
- Exists(String varName, Formula f): genera un objeto Exists con el nombre de variable dado por varName y con la fórmula f como atributos.
- Equals(String varName1, String varName2): genera un objeto Equals con los nombres de variables dados por varName1 y varName2 como atributos.
- Notequals(String varName1, String varName2): genera un objeto Notequals con los nombres de variables dados por varName1 y varName2 como atributos.

Todas estas instrucciones devuelven un objeto de su propia clase. Estas clases heredan de la clase Formula, que más adelante se explicará su estructura de datos.

3.1.4. Atributos

Para poder guardar en memoria los objetos generados con las instrucciones intermedias, se definen una serie de atributos, que son los siguientes:

Terminal	**Atributo**	**Tipo**	**Contenido**
F	form	Sintetizado	Objeto Formula que se genera.
F2	form	Sintetizado	Objeto Formula que se genera.
G	form	Sintetizado	Objeto Formula que se genera.
G2	form	Sintetizado	Objeto Formula que se genera.
H	form	Sintetizado	Objeto Formula que se genera.
H2	form	Sintetizado	Objeto Formula que se genera.
I	form	Sintetizado	Objeto Formula que se genera.
I2	form	Sintetizado	Objeto Formula que se genera.
J	form	Sintetizado	Objeto Formula que se genera.
id	varName	Léxico	El nombre de la variable del identificador.

Tabla 3.3: Atributos de la gramática.

Una vez parseada la entrada, tendremos un objeto Formula que representará, en forma de árbol, a la fórmula introducida.

3.1.5. Esquema de Traducción Dirigido por la Sintaxis

Este esquema de traducción, especifica el código intermedio que se ejecutará tras realizar una derivación.

<F> ::= <G><F2> {*if(F2.form!=null)*
F.form = new Biconditional(G.form, F2.form);
else F.form = G.form;}

<F2> ::= “<->” <F> {*F2.form = F.form;*}

| {*F2.form = null;*}

<G> ::= <H><G2> {*if(G2.form!=null)*
G.form = new Implication(H.form, G2.form);
else G.form = H.form;}

<G2> ::= “->” <G> {*G2.form = G.form;*}

| {*G2.form = null;*}

<table>
<tr><td><H></td><td>::=</td><td><I><H2></td><td>{if(H2.form!=null)
H.form = new Implication(I.form, H2.form);
else H.form = I.form;}</td></tr>
<tr><td><H2></td><td>::=</td><td>“|” <H></td><td>{H2.form = H.form;}</td></tr>
<tr><td></td><td>|</td><td></td><td>{H2.form = null;}</td></tr>
<tr><td><I></td><td>::=</td><td><J><I2></td><td>{if(I2.form!=null)
I.form = new Implication(J.form, I2.form);
else I.form = J.form;}</td></tr>
<tr><td><I2></td><td>::=</td><td>“&” <I></td><td>{I2.form = I.form;}</td></tr>
<tr><td></td><td>|</td><td></td><td>{I2.form = null;}</td></tr>
<tr><td><J></td><td>::=</td><td>“¬” <J></td><td>{J.form = new Negation(J1.form);}</td></tr>
<tr><td></td><td>|</td><td>“@” <id>“(” <F>“)”</td><td>{J.form = new Forall(id.varName, F.form);}</td></tr>
<tr><td></td><td>|</td><td>“%” <id>“(” <F>“)”</td><td>{J.form = new Exists(id.varName, F.form);}</td></tr>
<tr><td></td><td>|</td><td>“(” <F>“)”</td><td>{J.form = F.form;}</td></tr>
<tr><td></td><td>|</td><td><id>“!” <id></td><td>{J.form = new Notequals(
id_1.varName, id_2.varName);}</td></tr>
<tr><td></td><td>|</td><td><id>“=” <id></td><td>{J.form = new Equals(
id_1.varName, id_2.varName);}</td></tr>
<tr><td><id></td><td>::=</td><td><letra><idAux></td><td>{id.varName = letra.nombre + idAux.nombre;}</td></tr>
<tr><td><idAux></td><td>::=</td><td><letra><idAux></td><td>{idAux.nombre = letra.nombre + idAux.nombre;}</td></tr>
<tr><td></td><td>|</td><td><digito><idAux></td><td>{idAux.nombre = digito.nombre + idAux.nombre;}</td></tr>
<tr><td></td><td>|</td><td></td><td>{idAux.nombre = “”;}</td></tr>
<tr><td><letra></td><td>::=</td><td>“a”</td><td>{letra.nombre = “a”;}</td></tr>
<tr><td></td><td>|</td><td>“b”</td><td>{letra.nombre = “b”;}</td></tr>
<tr><td></td><td>|</td><td>“c”</td><td>{letra.nombre = “c”;}</td></tr>
<tr><td></td><td>|</td><td>“d”</td><td>{letra.nombre = “d”;}</td></tr>
<tr><td></td><td>|</td><td>“e”</td><td>{letra.nombre = “e”;}</td></tr>
<tr><td></td><td>|</td><td>“f”</td><td>{letra.nombre = “f”;}</td></tr>
<tr><td></td><td>|</td><td>“g”</td><td>{letra.nombre = “g”;}</td></tr>
<tr><td></td><td>|</td><td>“h”</td><td>{letra.nombre = “h”;}</td></tr>
<tr><td></td><td>|</td><td>“i”</td><td>{letra.nombre = “i”;}</td></tr>
<tr><td></td><td>|</td><td>“j”</td><td>{letra.nombre = “j”;}</td></tr>
<tr><td></td><td>|</td><td>“k”</td><td>{letra.nombre = “k”;}</td></tr>
<tr><td></td><td>|</td><td>“l”</td><td>{letra.nombre = “l”;}</td></tr>
<tr><td></td><td>|</td><td>“m”</td><td>{letra.nombre = “m”;}</td></tr>
</table>

| “n” {*letra.nombre = “n”;*}
| “o” {*letra.nombre = “o”;*}
| “p” {*letra.nombre = “p”;*}
| “q” {*letra.nombre = “q”;*}
| “r” {*letra.nombre = “r”;*}
| “s” {*letra.nombre = “s”;*}
| “t” {*letra.nombre = “t”;*}
| “u” {*letra.nombre = “u”;*}
| “v” {*letra.nombre = “v”;*}
| “w” {*letra.nombre = “w”;*}
| “x” {*letra.nombre = “x”;*}
| “y” {*letra.nombre = “y”;*}
| “z” {*letra.nombre = “z”;*}
| “A” {*letra.nombre = “A”;*}
| “B” {*letra.nombre = “B”;*}
| “C” {*letra.nombre = “C”;*}
| “D” {*letra.nombre = “D”;*}
| “E” {*letra.nombre = “E”;*}
| “F” {*letra.nombre = “F”;*}
| “G” {*letra.nombre = “G”;*}
| “H” {*letra.nombre = “H”;*}
| “I” {*letra.nombre = “I”;*}
| “J” {*letra.nombre = “J”;*}
| “K” {*letra.nombre = “K”;*}
| “L” {*letra.nombre = “L”;*}
| “M” {*letra.nombre = “M”;*}
| “N” {*letra.nombre = “N”;*}
| “O” {*letra.nombre = “O”;*}
| “P” {*letra.nombre = “P”;*}
| “Q” {*letra.nombre = “Q”;*}
| “R” {*letra.nombre = “R”;*}
| “S” {*letra.nombre = “S”;*}
| “T” {*letra.nombre = “T”;*}
| “U” {*letra.nombre = “U”;*}
| “V” {*letra.nombre = “V”;*}
| “W” {*letra.nombre = “W”;*}

	|	“X”	{*letra.nombre = “X”;*}
	|	“Y”	{*letra.nombre = “Y”;*}
	|	“Z”	{*letra.nombre = “Z”;*}
<digito>	::=	“0”	{*digito.nombre = “0”;*}
	|	“1”	{*digito.nombre = “1”;*}
	|	“2”	{*digito.nombre = “2”;*}
	|	“3”	{*digito.nombre = “3”;*}
	|	“4”	{*digito.nombre = “4”;*}
	|	“5”	{*digito.nombre = “5”;*}
	|	“6”	{*digito.nombre = “6”;*}
	|	“7”	{*digito.nombre = “7”;*}
	|	“8”	{*digito.nombre = “8”;*}
	|	“9”	{*digito.nombre = “9”;*}

De manera análoga que en la gramática, las reglas de derivación <id>, <idAux>, <letra>y <digito>están puestas para ilustrar el código intermedio que generarían, aunque es el analizador léxico quien se encarga de darle un valor al atributo de <id>.

3.2. Módulo semántico

Este módulo será el que se encargue de resolver el problema principal: encontrar todos los modelos para una fórmula dada. En este módulo se implementa la estructura de datos de las fórmulas y la estructura de datos auxiliar.

3.2.1. Estructura de datos de las fórmulas

Gracias a la orientación a objetos y polimorfismo de Java, se ha podido implementar fácilmente la estructura de una fórmula. Se han detectado e identificado los diferentes tipos de fórmulas que hay en la lógica, con lo que se ha podido definir la siguiente jerarquía:

- Formula
    - Negation
    - EqualityOperator
        - Equals
        - Notequals
    - LogicalOperator
        - And
        - Or
        - Implication
        - Biconditional
    - Quantifier
        - Forall
        - Exists

La siguiente imagen muestra el diagrama de clases UML que representa la estructura de datos de la fórmula.

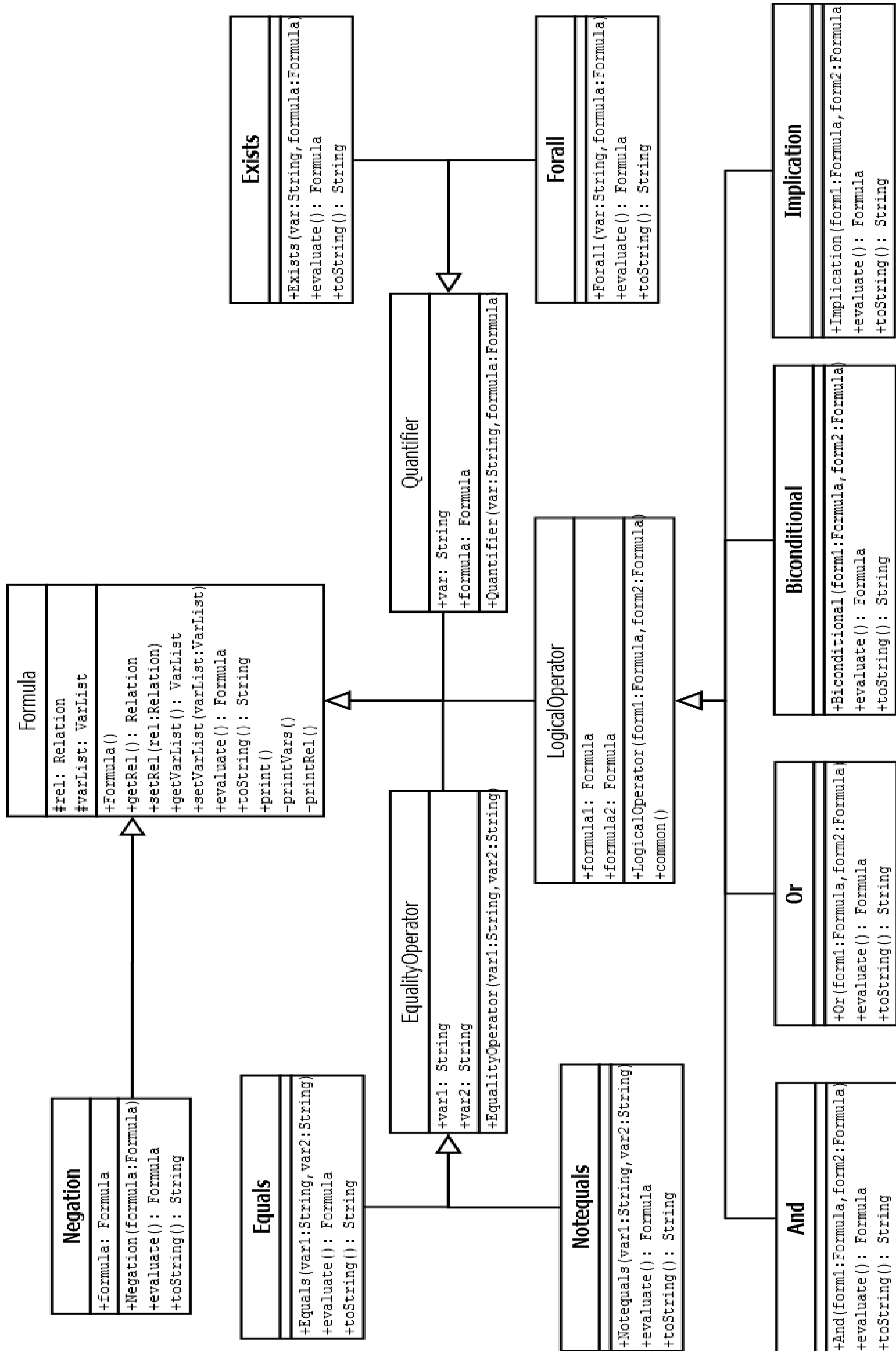


Figura 3.2: Diagrama de clases UML de Formula.

A continuación se explica el funcionamiento de las clases.

Formula

Es la clase padre. Después de que el analizador sintáctico haya parseado la fórmula de la entrada, se obtiene un objeto de la clase Formula con la estructura correspondiente. En este punto, se han de calcular los modelos para ese objeto Formula, llamando al método evaluate().

Modificador	Atributo	Descripción
Protected	Relation rel	Atributo donde se almacenarán los modelos de la fórmula.
Protected	VarList varList	Atributo donde se almacenará la lista de variables que aparecen en la fórmula.

Tabla 3.5: Atributos de la clase Formula.

Modificador	Método	Descripción
Public	Formula()	Constructora.
Public	Relation getRel()	Devuelve los modelos de la fórmula.
Public	setRel(Relation rel)	Asigna los modelos que se le pasan por parámetro.
Public	VarList getVarList()	Devuelve la lista de variables que aparecen en la fórmula.
Public	setVarList(VarList varList)	Asigna la lista de variables que se le pasa por parámetro.
Public abstract	evaluate()	Calcula todos los modelos de la fórmula.
Public abstract	toString()	Representación de la fórmula en formato de texto.
Public	print()	Imprime, por la entrada estándar, la lista de variables y todos los modelos de la fórmula.
Private	printVars()	Imprime, por la entrada estándar, la lista de variables.
Private	printRel()	Imprime, por la entrada estándar, todos los modelos de la fórmula.

Tabla 3.6: Métodos de la clase Formula.

Negation

Clase que extiende directamente la clase Formula. Se le añade un nuevo atributo e imple-

menta los métodos abstractos.

Modificador	**Atributo**	**Descripción**
Protected	Formula formula	Fórmula a negar.

Tabla 3.7: Atributos de la clase Negation.

Modificador	**Método**	**Descripción**
Public	Negation()	Constructora. Crea una nueva instancia de la clase e inicializa los atributos.
Public	evaluate()	Calcula los modelos de la fórmula en forma normal negativa. A la hora de evaluar una negación, primero se calcula la forma normal negativa, y después se calculan los modelos.
Public	toString()	Representación de la fórmula en formato de texto. Ejemplo: ¬(a=b)

Tabla 3.8: Métodos de la clase Negation.

EqualityOperator

Es una clase abstracta. Extiende directamente la clase Formula. Se le añaden dos nuevos atributos, que serán las variables con las que se va a operar.

Modificador	**Atributo**	**Descripción**
Public	String var1	Primera variable. Representa a la variable de la izquierda.
Public	String var2	Segunda variable. Representa a la variable de la derecha.

Tabla 3.9: Atributos de la clase EqualityOperator.

Modificador	**Método**	**Descripción**
Public	EqualityOperator(String var1, String var2)	Constructora.

Tabla 3.10: Métodos de la clase EqualityOperator.

Equals
Clase que extiende EqualityOperator. Es el predicado de la igualdad entre dos variables.

Modificador	Método	Descripción
Public	Equals(String var1, String var2)	Constructora.
Public	evaluate()	Calcula el modelo de la fórmula.
Public	toString()	Representación de la fórmula en formato de texto. Ejemplo: a=b

Tabla 3.11: Métodos de la clase Equals.

A la hora de evaluar una igualdad, se comprueba si las variables son iguales. En caso de serlo, el modelo generado es 0,0, si no, 0.

Notequals
Clase que extiende EqualityOperator. Es el predicado de la igualdad entre dos variables.

Modificador	Método	Descripción
Public	Equals(String var1, String var2)	Constructora.
Public	evaluate()	Calcula el modelo de la fórmula.
Public	toString()	Representación de la fórmula en formato de texto. Ejemplo: a!b

Tabla 3.12: Métodos de la clase Notequals.

A la hora de evaluar una desigualdad, se comprueba si las variables son iguales. En caso de serlo, no se genera ningún modelo, ya que no existe. Por lo contrario, si son diferentes, el modelo generado es 0,1

LogicalOperator
Es una clase abstracta. Extiende directamente de la clase Formula. Esta fórmula necesita de dos fórmulas como atributos.

Modificador	Atributo	Descripción
Public	Formula form1	Primera fórmula. Representa a la fórmula de la izquierda.
Public	Formula form2	Segunda fórmula. Representa a la fórmula de la derecha.

Tabla 3.13: Atributos de la clase LogicalOperator.

Modificador	Método	Descripción
Public	LogicalOperator(Formula form1, Formula form2)	Constructora.
Public	common()	Cálculo parcial de los modelos de la fórmula. El método common es un procedimiento común a la hora de evaluar las fórmulas tipo And y Or. Se evalúan las dos subfórmulas que componen esta fórmula. Después, según las variables que haya en cada fórmula, se hacen las expansiones en los conjuntos de las subfórmulas, y se hacen las permutaciones correspondientes para que ambos conjuntos tengan representadas las variables en el mismo orden.

Tabla 3.14: Métodos de la clase LogicalOperator.

And

Clase que extiende la clase LogicalOperator. Esta fórmula representa a la conjunción de dos fórmulas.

Modificador	Método	Descripción
Public	And(Formula form1, Formula form2)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. Para calcular los modelos de una fórmula And, es necesario ejecutar previamente el procedimiento común. Después, se hace la intersección entre el conjunto de modelos de la subfórmula de la izquierda y el de la derecha.
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: (a=b) & (a!c)

Tabla 3.15: Métodos de la clase And.

Or

Clase que extiende la clase LogicalOperator. Esta fórmula representa a la disyunción de dos fórmulas.

Modificador	Método	Descripción
Public	Or(Formula form1, Formula form2)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. Para calcular los modelos de una fórmula Or, es necesario ejecutar previamente el procedimiento común. Después, se hace la unión entre el conjunto de modelos de la subfórmula de la izquierda y el de la derecha.
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: (a=b) | (a!c)

Tabla 3.16: Métodos de la clase Or.

Implication

Clase que extiende la clase LogicalOperator. Esta fórmula representa a la implicación de dos fórmulas.

Modificador	Método	Descripción
Public	Implication(Formula form1, Formula form2)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. El cálculo de los modelos se hace mediante el cálculo de los modelos de una fórmula disyuntiva equivalente. Ejemplo: G ->H es equivalente a ¬G | H
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: (a=b) | (a!c)

Tabla 3.17: Métodos de la clase Implication.

Biconditional

Clase que extiende la clase LogicalOperator. Esta fórmula representa a la doble implicación de dos fórmulas.

Modificador	**Método**	**Descripción**
Public	Biconditional(Formula form1, Formula form2)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. El cálculo de los modelos se hace mediante el cálculo de los modelos de una fórmula conjuntiva de la implicación entre ambas subfórmulas. Ejemplo: G<->H es equivalente a (G->H) & (H->G)
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: (a=b) | (a!c)

Tabla 3.18: Métodos de la clase Biconditional.

Quantifier

Es una clase abstracta. Extiende directamente la clase Formula. Se le añaden dos nuevos atributos: una variable y una fórmula.

Modificador	**Atributo**	**Descripción**
Public	String var	Variable que se va a cuantificar.
Public	Formula formula	La fórmula a la que se le aplica la cuantificación.

Tabla 3.19: Atributos de la clase Quantifier.

Modificador	**Método**	**Descripción**
Public	Quantifier(String var, Formula formula)	Constructora.

Tabla 3.20: Métodos de la clase Quantifier.

Exists

Clase que extiende la clase Quantifier. Esta fórmula representa al cuantificador existencial.

Modificador	Método	Descripción
Public	Exists(String var, Formula formula)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. Para calcular el conjunto de modelos, primero se calcula el conjunto de modelos de la fórmula donde se aplica la cuantificación y después se hace una proyección sobre la variable.
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: exists c ((a=b) | (b!c))

Tabla 3.21: Métodos de la clase Exists.

Forall

Clase que extiende la clase Quantifier. Esta fórmula representa al cuantificador existencial.

Modificador	Método	Descripción
Public	Forall(String var, Formula formula)	Constructora.
Public	evaluate()	Calcula los modelos de la formula. Para calcular el conjunto de modelos, primero se calcula el conjunto de modelos de la fórmula donde se aplica la cuantificación y después se hace una proyección sobre la variable. De este último conjunto, sólo estarán en el conjunto de modelos del resultado aquellos modelos que expandiéndolos con la variable, estén todas sus expansiones en el conjunto original.
Public	String toString()	Representación de la fórmula en formato de texto. Ejemplo: forall c ((a=b) | (b!c))

Tabla 3.22: Métodos de la clase Forall.

3.2.2. Estructura de datos auxiliar

Está compuesta por tres clases:

- Relation: para representar los modelos de una fórmula.
- Tupla: para representar un modelo.

- VarList: para representar la lista de variables que hay en una fórmula.

La siguiente imagen muestra el diagrama de clases UML que representa la estructura de datos auxiliar.

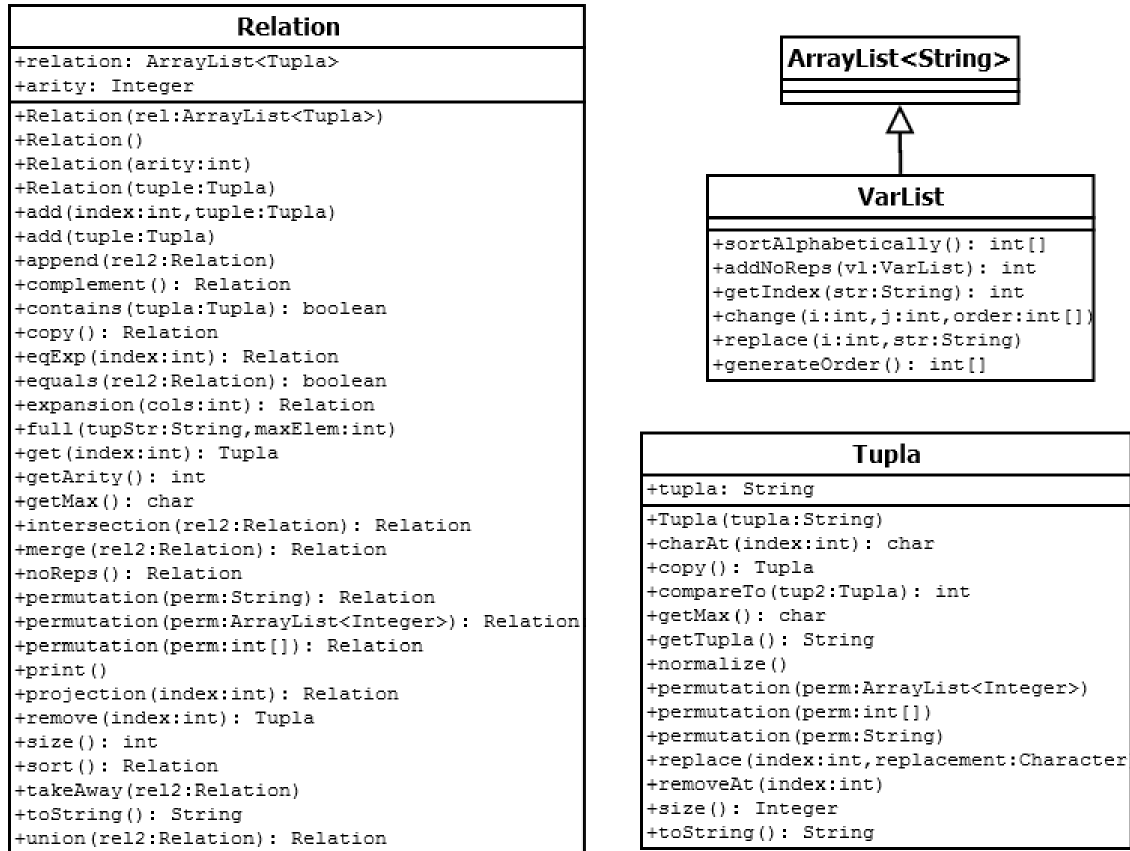


Figura 3.3: Diagrama de clases UML de la estructura auxiliar.

Relation

Sirve para representar los modelos de una fórmula. Esta clase contiene las diferentes operaciones definidas para un conjunto de modelos, a saber: la expansión, intersección, permutación, proyección y unión.

Modificador	Atributo	Descripción
Public	ArrayList<Tupla> relation	Representa un conjunto de modelos mediante una lista de tuplas.
Public	Integer arity	Representa la aridad.

Tabla 3.23: Atributos de la clase Relation.

Modificador	Método	Descripción
Public	Relation(ArrayList<Tupla> rel)	Constructora. Crea una nueva relación a partir de una lista de tuplas.
Public	Relation()	Constructora. Crea una nueva relación vacía de aridad 0.
Public	Relation(int arity)	Constructora. Crea una nueva relación vacía con una aridad definida por arity.
Public	Relation(Tupla tuple)	Constructora. Crea una nueva relación con el modelo pasado por parámetro.
Public	add(int index, Tupla tuple)	Introduce la tupla en la posición especificada.
Public	add(Tupla tuple)	Introduce la tupla al final de la lista.
Public	append(Relation rel2)	Introduce todas las tuplas de rel2.
Public	Relation complement()	Genera una relación complementaria a la actual. Para ello se calculan todas las diferentes tuplas de la misma aridad y se eliminan aquellas que están en la relación.
Public	boolean contains(Tupla tuple)	Comprueba si la tupla especificada se encuentra actualmente en la relación.
Public	Relation copy()	Crea una copia exacta de la relación.
Public	Relation eqExp(int index)	Devuelve la expansión de la relación. Para hacer la expansión se copia, en una nueva columna, la posición indicada por el parámetro.
Public	boolean equals(Relation rel2)	Comprueba si la relación actual es igual a la pasada por parámetro.
Public	Relation expansion(int cols)	Expande la relación tantas veces como lo indique el parámetro.
private	full(String tupStr, int maxElem)	Genera tantas expansiones como resulte la diferencia entre la aridad y la cantidad de caracteres de tupStr, siendo maxElem el valor más alto hasta ahora.

Public	Tupla get(int index)	Devuelve la tupla en la posición indicada.
Public	int getArity()	Devuelve la aridad de la relación.
Public	char getMax()	Devuelve el elemento más alto entre todas las tuplas.
Public	Relation intersection(Relation rel2)	Devuelve la intersección entre la relación actual y rel2.
Public	Relation merge(Relation rel2)	Combina la relación actual con rel2.
Public	Relation noReps()	Devuelve la relación sin tuplas repetidas.
Public	Relation permutation(String perm)	Permuta todas las tuplas de la relación.
Public	Relation permutation(ArrayList<Integer> perm)	Permuta todas las tuplas de la relación.
Public	Relation permutation(int[] perm)	Permuta todas las tuplas de la relación.
Public	print()	Imprime por la salida estándar la relación.
Public	Relation projection(int index)	Elimina la columna y normaliza todas las tuplas.
Public	Tupla remove(int index)	Remueve la tupla en la posición indicada.
Public	int Size()	Devuelve la cantidad de tuplas que hay en la relación.
Public	Relation sort()	Ordena de menor a mayor todas las tuplas. El algoritmo utilizado es el mergesort.
Public	takeAway(Relation rel2)	Devuelve la resta entre la relación actual y rel2.
Public	String toString()	Representación de la relación en formato texto.

Public	Relation union(Relation rel2)	Devuelve la unión entre la relación actual y rel2.

Tabla 3.24: Métodos de la clase Relation.

Tupla

Sirve para representar un modelo. Esta clase contiene las diferentes operaciones definidas para un modelo, a saber: la expansión, permutación y proyección.

Modificador	Atributo	Descripción
Public	String tupla	Representa un modelo mediante un String.

Tabla 3.25: Atributos de la clase Tupla.

Modificador	Método	Descripción
Public	Tupla(String tupla)	Constructora. Crea un nuevo modelo a partir de un String.
Public	char charAt(int index)	Devuelve el carácter en la posición indicada.
Public	Tupla copy()	Crea una copia exacta de la tupla.
Public	int compareTo(Tupla tup2)	Compara las tuplas y devuelve un entero indicando cual es mayor alfabéticamente. El entero será negativo si la tupla actual es menor que tup2, 0 si son iguales y positivo si es mayor.
Public	eqExp(int index)	Agrega al final de la tupla el carácter de la posición indicada.
Public	boolean equals(Tupla tupla2)	Comprueba si las tuplas son iguales.
Public	char getMax()	Devuelve el mayor carácter alfabético que se encuentra en la tupla.
Public	String getTupla()	Representación de la tupla en formato texto.
Public	Normalize()	Normaliza la tupla.
Public	Tupla permutation(String perm)	Permuta la tupla.

Public	Tupla permutation(ArrayList<Integer> perm)	Permuta la tupla.
Public	Tupla permutation(int[] perm)	Permuta la tupla.
Public	Replace(int index, char replacement)	Reemplaza el carácter en la posición indicada por el nuevo carácter.
Public	removeAt(int index)	Elimina el carácter en la posición indicada.
Public	Integer size()	Devuelve el tamaño de la tupla.
Public	String toString()	Representación de la tupla en formato texto. Lo mismo que getTupla.

Tabla 3.26: Métodos de la clase Tupla.

VarList

Sirve para representar la lista de variables que contiene una fórmula. Esta clase ha sido implementada para poder extender la clase ArrayList.

Modificador	**Método**	**Descripción**
Public	VarList()	Constructora. Crea una lista de variables vacía.
Public	int [] sortAlphabetically()	Reordena la lista alfabéticamente y devuelve un array que contiene la permutación realizada.
Public	int addNoReps(VarList vl)	Añade a la lista las variables de vl que aún no estén en la lista actual.
Public	int getIndex(String str)	Devuelve la posición donde se encuentra la variable indicada. Si dicha variable no se encuentra en la lista, devuelve -1.
Public	int[] change(int i, int j, int[] order)	Intercambia las variables de las posiciones i y j y devuelve dicha permutación en un array.

Public	replace(int i, String str)	Reemplaza la variable en la posición indicada por la que se pasa por parámetro.
Public	int[] generateOrder()	Tomando como original el orden actual, devuelve un array con la permutación del orden, esto es, [0, 1, 2,. . . , n-1]

Tabla 3.27: Métodos de la clase VarList.

4. CAPÍTULO

Pruebas y resultados

En este apartado se exponen las diferentes pruebas que se han realizado para confirmar la correcta implementación de la aplicación.

4.1. Pruebas unitarias

Se han probado cada una de las diferentes fórmulas individualmente, y después algunas en conjunto. El texto en verde indica la fórmula introducida, y la siguiente línea la fórmula reconocida por la aplicación. La tabla nos muestra todos los modelos para la fórmula introducida. La primera línea indica las variables que hay, mientras que las siguientes indican los modelos.

4.1.1. Igualdades

- A=A

  (A = A)

A
0

- A!A

  (A ! A)

A

- `A=B`

  `(A = B)`

A	B
0	0

- `A!B`

  `(A = B)`

A	B
0	1

4.1.2. Operadores

- `A=B & B=C`

  `((A = B) & (B = C))`

A	B	C
0	0	0

- `A=B & C=D`

  `((A = B) & (C = D))`

A	B	C	D
0	0	0	0
0	0	1	1

- `A=B | B=C`

  `((A = B) | (B = C))`

A	B	C
0	0	0
0	0	1
0	1	1

- `A=B | C=D`

  `((A = B) | (C = D))`

A	B	C	D
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	0	1	2
0	1	0	0
0	1	1	1
0	1	2	2

- `A=B ->B=C`

  `((A = B) ->(B = C))`

A	B	C
0	0	0
0	1	0
0	1	1
0	1	2

- `A=B ->C=D`

  `((A = B) ->(C = D))`

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	0	2
0	1	1	0
0	1	1	1
0	1	1	2
0	1	2	0
0	1	2	1
0	1	2	2
0	1	2	3

- `A=B <->B=C`

  `((A = B) <->(B = C))`

A	B	C
0	0	0
0	1	0
0	1	2

- `A=B <->C=D`

  `((A = B) <->(C = D))`

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	0	2
0	1	1	0
0	1	1	2
0	1	2	0
0	1	2	1
0	1	2	3

4.1.3. Negación

- ¬(A=B & B=C)

  ¬(((A = B) & (B = C)))

A	B	C
0	0	1
0	1	0
0	1	1
0	1	2

- ¬(A=B & C=D)

  ¬(((A = B) & (C = D)))

A	B	C	D
0	0	0	1
0	0	1	0
0	0	1	2
0	1	0	0
0	1	0	1
0	1	0	2
0	1	1	0
0	1	1	1
0	1	1	2
0	1	2	0
0	1	2	1
0	1	2	2
0	1	2	3

- ¬(A=B | B=C)

  ¬(((A = B) | (B = C)))

A	B	C
0	1	0
0	1	2

- ¬(A=B | C=D)

  ¬(((A = B) | (C = D)))

A	B	C	D
0	1	0	1
0	1	0	2
0	1	1	0
0	1	1	2
0	1	2	0
0	1	2	1
0	1	2	3

- `¬(A=B ->B=C)`

  `¬(((A = B) ->(B = C)))`

A	B	C
0	0	1

- `¬(A=B ->C=D)`

  `¬(((A = B) ->(C = D)))`

A	B	C	D
0	0	0	1
0	0	1	0
0	0	1	2

- `¬(A=B <->B=C)`

  `¬(((A = B) <->(B = C)))`

A	B	C
0	0	1
0	1	1

- `¬(A=B <->C=D)`

  `¬(((A = B) <->(C = D)))`

A	B	C	D
0	0	0	1
0	0	1	0
0	0	1	2
0	1	0	0
0	1	1	1
0	1	2	2

4.1.4. Cuantificadores

- `@A(A=B & B=C)`

  `forAll A (((A = B) & (B = C)))`

B	C

- `@A(A=B & C=D)`

  `forAll A (((A = B) & (C = D)))`

B	C	D

- `@A(A=B | B=C)`

  `forAll A (((A = B) | (B = C)))`

B	C
0	0

- `@A(A=B | C=D)`

  `forAll A (((A = B) | (C = D)))`

B	C	D
0	0	0
0	1	1

- `%A(A=B & B=C)`

  `exists A (((A = B) & (B = C)))`

B	C
0	0

- `%A(A=B & C=D)`

  `exists A (((A = B) & (C = D)))`

B	C	D
0	0	0
0	1	1

- `%A(A=B | B=C)`

  `exists A (((A = B) | (B = C)))`

B	C
0	0
0	1

- `%A(A=B | C=D)`

  `exists A (((A = B) | (C = D)))`

B	C	D
0	0	0
0	0	1
0	1	0
0	1	1
0	1	2

4.2. Pruebas globales

Se han probado tautologías de diferente aridad para medir el tiempo que tarda la aplicación en resolverlas. Estas pruebas se han hecho con un ordenador con el procesador i7-4710HQ, de 4 núcleos y 8 hilos de ejecución, a 3.33GHz. Esta vez no se muestran los modelos de las fórmulas, ya que la cantidad de modelos que tiene una tautología es elevada. El tiempo indicado es la media de 10 ejecuciones de la misma fórmula.

<table>
<tr><th>Fórmula</th><th>Aridad</th><th>Modelos</th><th>Tiempo(ms)</th></tr>
<tr><td>A=A</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>A=B | A!B</td><td>2</td><td>2</td><td>0</td></tr>
<tr><td>A=B | A!B | B=C</td><td>3</td><td>5</td><td>0</td></tr>
<tr><td>A=B | A!B | B=C | C=D</td><td>4</td><td>15</td><td>0</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E</td><td>5</td><td>52</td><td>0</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E | E=F</td><td>6</td><td>203</td><td>2</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E | E=F | F=G</td><td>7</td><td>877</td><td>18</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E | E=F | F=G | G=H</td><td>8</td><td>4140</td><td>302</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E | E=F | F=G | G=H | H=I</td><td>9</td><td>21147</td><td>7176</td></tr>
<tr><td>A=B | A!B | B=C | C=D | D=E | E=F | F=G | G=H | H=I | I=J</td><td>10</td><td>115975</td><td>195176</td></tr>
</table>

Tabla 4.1: Resultado de las pruebas globales.

5. CAPÍTULO

Conclusiones

Este proyecto presenta la implementación de una aplicación que busca todos los modelos de una fórmula de la lógica NatEq recibida como entrada. Para realizar el proyecto han sido necesarias varias fases. El primer paso consistió en repasar la lógica de predicados, ya que la lógica que se presenta es una restricción de la primera.

Por otro lado, fue necesario volver a estudiar algunos conceptos sobre compiladores, ya que el proyecto integra un parser.

A la hora de escoger un lenguaje de programación y un entorno de desarrollo, se optó por utilizar JAVA y eclipse. JAVA, además de haber sido uno de los lenguajes más utilizados a lo largo de la carrera, es muy intuitivo y está orientado a objetos. Por otro lado, eclipse incluye muchos plugins que facilitan enormemente la tarea del programador.

En un principio, ya que eclipse permitía el uso de plugins, se optó por utilizar uno para generar el parser a partir de una gramática y tokens. Esta idea, aunque buena, fracasó y hubo que programar el parser entero desde el principio. Este proceso fue algo complicado de llevar a cabo, pero implicó que se consolidaran los conocimientos adquiridos en la carrera sobre compiladores.

Gracias a la orientación a objetos de JAVA, se pudo diseñar una estructura de datos para las fórmulas fácil de entender. Lo más complicado de esta parte fue el diseño de los algoritmos correctos capaces de obtener todos los modelos para cada tipo de fórmula.

Integrar el parser con la parte semántica, esto es, con la estructura de datos, fue muy fácil, ya que se diseñó para que se pudiera ampliar y poder añadir mejoras en un futuro. Una

vez integrado, se pasó a la fase de testeo, donde se realizaron diferentes pruebas unitarias y globales. Esto permitió encontrar pequeños fallos en la implementación del algoritmo para obtener los modelos de las fórmulas, que fueron rápidamente corregidos. Tras los resultados de las pruebas globales, se concluyó que el coste en tiempo era muy elevado para fórmulas con más de 10 variables, ya que tenían una cantidad ingente de modelos.

Posibles ampliaciones

Desde el punto de vista de la presentación, una posible ampliación sería integrar una interfaz gráfica para el usuario. También se podría generar una imagen con el árbol resultante para una mejor visualización de la fórmula parseada. Sería interesante crear una aplicación Android o una página web, quedando así accesible para fines académicos o de investigación.

6. CAPÍTULO

Bibliografía

1. Uwe Schöning. *Logic for Computer Scientists.* Birkhäuser, reprint of the 1989 edition, 2008.

2. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compiladores. Principios, técnicas y herramientas.* Segunda edición. Pearson Educación, 2008.

3. First-order logic. [En línea] https://en.wikipedia.org/wiki/First-order_logic

4. Java^{TM} platform Standard edition 7 API Specification [En línea] http://docs.oracle.com/javase/7/docs/api/.

Anexos

Manual del usuario

Para ejecutar la aplicación Será necesario tener instalado en el equipo la versión JRE 1.6 o superior. Una vez lo tengamos instalado, abrimos una línea de comando donde esté situada la aplicación y escribimos “java –jar nateqparser.jar” sin las comillas.

Cómo escribir bien una fórmula Si x1 y x2 son variables, entonces escribiremos “X1 = X2” o “x1 ! x2” para obtener las fórmulas más simples. Una vez tengamos una fórmula escrita, podemos relacionarlas con otra añadiendo conectivos lógicos y cuantificadores. La siguiente tabla muestra cómo escribir correctamente los conectivos lógicos y cuantificadores entre fórmulas en esta aplicación:

<table>
<tr><th>Nombre</th><th>Símbolo en la lógica</th><th>Símbolo(s) en la aplicación</th></tr>
<tr><td>Negación</td><td>¬</td><td>¬</td></tr>
<tr><td>Conjunción</td><td>∧</td><td>&</td></tr>
<tr><td>Disyunción</td><td>∨</td><td>|</td></tr>
<tr><td>Implicación</td><td>⇒</td><td>-></td></tr>
<tr><td>Doble implicación</td><td>⇔</td><td><-></td></tr>
<tr><td>Cuantificador universal</td><td>∀</td><td>@</td></tr>
<tr><td>Cuantificador existencial</td><td>∃</td><td>%</td></tr>
</table>

He aquí unos ejemplos:

Fórmula en la lógica	**Fórmula en la aplicación**
$(x_1 = x_2)$	X1=X2
$(x_1 \neq x_2)$	X1!X2
$\neg(x_1 = x_2)$	¬ X1=X2
$((x_1 = x_2) \wedge (x_2 = x_3))$	X1=X2 & X2=X3
$((x_1 = x_2) \vee (x_2 = x_3))$	X1=X2 | X2=X3
$\neg((x_1 = x_2) \vee (x_2 = x_3))$	¬(X1=X2 | X2=X3)
$(\forall x_1 (x_1 = x_2))$	@X1 (X1=X2)
$(\exists x_1 (x_1 = x_2))$	%X1 (X1=X2)