Demostración Automática de Teoremas
(utilizando el principio de resolución de Robinson)

Ait Ismael y Postan Ezequiel
Introducción a la Inteligencia Artificial
Trabajo Práctico Final
LCC - UNR

23 de diciembre de 2013

Resumen

En este trabajo se estudia el problema de la demostración automática de teoremas de la lógica de primer orden. En particular nos enfocamos en el algoritmo de resolución de Robinson para demostrar la insatisfacibilidad de un conjunto de cláusulas.

Se implementaron los algoritmos necesarios para llevar una fórmula de lógica de predicados general a sus formas normales de prenex y de Skolem, para luego aplicar el algoritmo de resolución mencionado. Construimos un programa que decide la validez de una implicación en lógica de primer orden utilizando estos algoritmos.

1. Motivación

Para que un sistema informático demuestre un comportamiento inteligente en la resolución de problemas, debe poseer una gran cantidad de conocimiento y un potente mecanismo de razonamiento. Para la representación del conocimiento se utilizan diferentes tipos de lenguajes.

Los lenguajes de programación más comúnmente usados son de tipo imperativo, orientados a objetos o funcionales. Los programas generados con estos lenguajes representan, en sentido directo, procesos computacionales. Las estructuras de datos dentro de los programas son representaciones *ad hoc* de los hechos. Cada "actualización" de estos hechos es realizada por un procedimiento específico para el dominio de aplicación. Estos lenguajes de programación, conocidos como **lenguajes procedurales**, carecen de mecanismos generales para derivar nuevos hechos a partir de los anteriores.

Los **lenguajes naturales** (como el inglés y el español) tienen un mayor poder de expresividad. Además de ser mecanismos puros de representación, son un medio de comunicación. En este sentido, el significado de una sentencia en lenguaje natural depende tanto de la propia sentencia como del **contexto** al que hace referencia. Estos lenguajes

poseen grandes inconvenientes de **ambigüedad** que pueden causar obstáculos en la comprensión, lo cual hace extremadamente compleja o imposible la inferencia automatizada de nuevos hechos.

El enfoque de los **lenguajes procedurales** contrasta con la naturaleza declarativa de los **lenguajes lógicos**, en los cuales el mecanismo de inferencia y el conocimiento se encuentran bien separados, y la inferencia es totalmente independiente del dominio de aplicación. Su semántica está basada en la relación de verdad entre las sentencias y las posibles interpretaciones del mundo. En particular, el lenguaje de la **lógica de primer orden**[1] tiene el poder de expresividad suficiente para tratar con una gran cantidad de problemas, pero a su vez elimina la **ambigüedad** y la dependencia del **contexto** de los lenguajes naturales. Además, los lenguajes lógicos proporcionan métodos bien conocidos de inferencia automatizada de nuevos hechos.

La **programación lógica** es una tecnología que está bastante relacionada con alcanzar el ideal declarativo: construir sistemas expresando el conocimiento en lenguaje formal y resolver los problemas ejecutando procedimientos de inferencia automática sobre dicho conocimiento. Existen lenguajes declarativos bien conocidos que implementan la programación lógica como Prolog o Jess. Sin embargo, estos sistemas no nacen como un lenguaje para la demostración de teoremas y, en su búsqueda por resolver problemas de programación, han sacrificado la naturalidad para expresar fórmulas lógicas generales por eficiencia computacional.

Esto nos motivó a implementar el método de resolución más general, permitiendo toda la expresividad de la lógica de predicados. La idea principal es la de permitir el ingreso de cualquier tipo de sentencia en lógica de primer orden, sin la necesidad de ningún tratamiento previo. Como consecuencia, el programa será poco eficiente comparado con los sistemas ya nombrados. Sin embargo, la finalidad del trabajo es la de crear una herramienta educativa y no para el desarrollo de sistemas KBS complejos. Esperamos que el trabajo pueda extenderse a futuro para asistir en la enseñanza de estos temas, puesto que el enfoque es ofrecer una buena visualización de todos los pasos de la inferencia.

2. Objetivos

Como primer objetivo, nos propusimos que el software resuelva satisfactoriamente y en un tiempo razonable todos los problemas de programación lógica vistos en la práctica de la materia. Estos tipos de problemas podrán ser ingresados con fórmulas de lógica de primer orden sin ninguna restricción y de manera mucho más simple y natural que en Prolog (el software realizará las conversiones a las formas normales necesarias automáticamente). Además, queremos que la respuesta del programa sea útil para entender el método de resolución y justificar así el resultado.

De manera general, queremos que el programa sea capaz de, teniendo el tiempo necesario, responder si una implicación cualquiera en lógica de primer orden representa un

[1] También llamada **lógica de predicados** o **cálculo de predicados** (de primer orden). Los lenguajes de primer orden son lenguajes formales con cuantificadores que alcanzan sólo a variables de individuo.

teorema. En este sentido, deseamos un sistema de inferencia que sea **sólido** y **completo** para este tipo de preguntas.

Los objetivos principales de desarrollo comprenden la creación de software para:

- Parsear fórmulas generales de lógica de primer orden;
- Convertir de manera automática a forma normal de Prenex;
- Convertir de manera automática a forma normal de Skolem;
- Convertir de manera automática a forma normal conjuntiva;
- Generar una prueba mediante el algoritmo de Robinson;
- Mostrar el árbol de resolución generado, en el caso exitoso.

Como un objetivo secundario, nos propusimos crear una interfaz amigable del tipo TELL/ASK mediante la cual se pueda ingresar sucesivas sentencias a la base de conocimiento y realizar preguntas al respecto. Además, esta interfaz podría ofrecer comandos para cargar bases de conocimientos desde archivos con distintos formatos, realizar conversiones a las distintas formas normales, mostrar los árboles de refutación y volcar los resultados en otros archivos. En principio, la interfaz es en modo de texto pero pensamos desarrollar más adelante una interfaz gráfica.

3. Métodos

3.1. Agentes lógicos

La finalidad de nuestro trabajo será la de estudiar y construir un sistema de inferencia automatizado para un lenguaje de programación lógica, el cual permita el desarrollo, entre otras cosas, de agentes lógicos.

El componente principal de un agente lógico es su **base de conocimiento**, KB. Informalmente, una base de conocimiento es un conjunto de sentencias. Cada sentencia se representa en un lenguaje denominado **lenguaje de representación del conocimiento** y representa alguna aserción acerca del mundo. El objetivo del agente es realizar **inferencias**, es decir, derivar nuevas sentencias de las antiguas.

El diseño del lenguaje de representación que permita, de forma más fácil, expresar el conocimiento mediante sentencias simplifica enormemente el problema de la construcción del agente. Este enfoque en la construcción de sistemas se denomina **enfoque declarativo**. Por el contrario el enfoque procedural codifica los comportamientos que se desea obtener directamente en código de programación.

3.2. Elección de la lógica

Una lógica puede ser caracterizada principalmente por sus compromisos ontológico y epistemológico. Algunos ejemplos se muestran en la Figura 1. El compromiso ontológico se refiere a qué se asume de la realidad natural, qué existe en el mundo. El compromiso epistemológico se refiere a los posibles estados de conocimiento, qué puede creer el agente acerca de los hechos. Además de la lógica de predicados, existen otras lógicas con diferentes compromisos como las lógicas de mayor orden, la lógica temporal, la teoría de probabilidad o la lógica difusa (fuzzy logic), entre otras.

Lenguaje	Compromiso ontológico (lo que sucede en el mundo)	Compromiso epistemológico (lo que el agente cree de los hechos)
Lógica proposicional	Hechos	Verdadero/falso/desconocido
Lógica de primer orden	Hechos, objetos, relaciones	Verdadero/falso/desconocido
Lógica temporal	Hechos, objetos, relaciones, tiempos	Verdadero/falso/desconocido
Teoría de las probabilidades	Hechos	Grado de creencia $\in [0, 1]$
Lógica difusa	Hechos con un grado de verdad $\in [0, 1]$	Valor del intervalo conocido

Figura 1: Lenguajes formales y sus compromisos ontológicos y epistemológicos

Elegimos trabajar con la lógica de primer orden puesto que, a pesar de su simpleza, posee un poder expresivo suficiente para representar una gran cantidad de problemas. Mientras que la lógica proposicional se compromete sólo con la existencia de hechos, la lógica de primer orden se compromete con la existencia de objetos y sus relaciones, y por ello gana poder expresivo. Asimismo, existen métodos de inferencia para la lógica de primer orden, derivados de la lógica proposicional, que preservan la solidez (*soundness*) y completitud, como el método de resolución de Robinson que se implementa.

El único inconveniente que introduce la lógica de primer orden frente a la lógica proposicional es que la inferencia se vuelve **semidecidible**. Esto significa que el método de inferencia utilizado responderá de manera positiva, llegando a la demostración, para cualquier implicación válida (teóricamente, si se dispone del tiempo suficiente), pero podría quedar ejecutándose de manera indefinida para una implicación no válida. Sin embargo para ciertos problemas (particularmente para problemas que no poseen funciones) el método se torna decidible. Por lo tanto, en ocasiones sí podremos contestar que la implicación no es válida.

Este es un precio que decidimos pagar, a favor de ofrecer un mayor poder de expresividad en el lenguaje y tornar el problema mucho más interesante que usando sólo la lógica proposicional.

3.3. Sintaxis y semántica de la lógica de primer orden

Las bases de conocimiento están compuestas de sentencias. Estas sentencias se construyen de acuerdo a la **sintaxis** del lenguaje de representación, que especifica todas las sentencias que están bien formadas. El razonamiento implica generar y manipular estas sentencias. La sintaxis de la lógica de primer orden se muestra en la Figura 2.

Una lógica también debe definir la **semántica** del lenguaje. La semántica trata el significado de las sentencias. La semántica de un lenguaje de lógica define el valor de verdad

de cada sentencia respecto de cada **modelo** o mundo posible. Los modelos son abstracciones matemáticas que simplemente nos permiten representar los entornos reales en los que el agente puede o no estar.

Un mundo posible, o modelo, se define para la lógica de primer orden como un conjunto de **objetos**, las **relaciones** entre ellos y las **funciones** que se les puede aplicar. Los símbolos de constantes identifican a los objetos, los símbolos de predicados identifican a las relaciones y los símbolos de función identifican las funciones. Una **interpretación** especifica una aplicación de los símbolos al modelo. Dado un modelo y una interpretación, se puede determinar el valor de verdad de la sentencia. La introducción de los cuantificadores permite expresar proposiciones generales sobre algunos o todos los objetos del **dominio** del modelo. El dominio del modelo es el conjunto de objetos que éste posee.

```
Sentencia    → Literal
             | Sentencia Conectivo Sentencia
             | [ Cuatificador Variable : Sentecia ]

Literal      → Atomo
             | ¬ Atomo

Atomo        → Predicado
             | Predicado ( Termios )

Predicado    → <cadena de letras que comience con mayuscula>

Terminos     → Termino
             | Termino , Terminos

Termino      → Variable
             | Constante
             | Funcion

Variable     → <cadena de letras minusculas>

Constante    → <cadena de letras que comience con mayuscula>

Funcion      → <cadena de letras minusculas> ( Termios )

Conectivo    → ∧ | ∨ | ⇒ | ⇔

Cuantificador → ∃ | ∀
```

Figura 2: Sintáxis de la lógica de primer orden expresada en *BNF*.

El árbol de sintaxis abstracto se define con tipos `data` en el archivo `AST.hs`. Un fragmento del código se muestra en a continuación.

```haskell
{- Estructura de datos para las Formulas Logicas -}
data Form   = Pred String [Term]
            | Not Form
            | And Form Form
            | Or  Form Form
            | Imp Form Form
            | Eq  Form Form
            | Forall String Form
            | Exists String Form
```

```haskell

{- Estructura de datos para los Terminos -}
data Term   = Var String
            | Const String
            | Func String [Term]
```

El archivo `Parser.hs` contiene el parser para la lógica de primer orden. Este módulo utiliza una librería de parsing funcional del capítulo 8 de [5].

3.4. Inferencia

El valor de verdad de cualquier sentencia se determina por un modelo y por una interpretación de los símbolos de la sentencia. Por lo tanto, la implicación, la validez, etc., se determinan en base a todos los modelos posibles y todas las interpretaciones posibles. El número de elementos del dominio en cada modelo puede ser infinito. Por eso, el número de modelos posibles es infinito, igual que el número de interpretaciones. La comprobación de la implicación mediante la enumeración de todos los modelos posibles, que funciona para la lógica proposicional, no es una opción para la lógica de primer orden. Incluso aunque el número de objetos esté restringido, el número de combinaciones puede ser enorme.

La notación matemática para la relación de **implicación lógica** entre sentencias es el siguiente:

$$\alpha \models \beta$$

lo cual significa que la sentencia α implica la sentencia β. La definición formal de implicación es esta: $\alpha \models \beta$ si y sólo si en cada modelo en que α es verdadera, β también lo es. Un agente lógico tendrá como premisas a la base de conocimiento y estaremos interesados en saber si una sentencias α es implicada (consecuencia semántica) por la base de conocimiento (i.e. cuando $\mathrm{KB} \models \alpha$).

Para lograr este objetivo, existen diferentes algoritmos de **inferencia**. Los algoritmos de inferencia derivan una sentencia a partir de otra mediante aplicación de reglas sintácticas. Su notación formal es la siguiente: si el algoritmo de inferencia i puede derivar α de la KB, entonces escribimos

$$\mathrm{KB} \vdash \alpha$$

que se lee como "α deriva de la base de conocimiento mediante i".

Se dice que un algoritmo que deriva sólo sentencias implicadas es **sólido** (*sound*). El *soundness* es una propiedad muy deseable pues un procedimiento de inferencia no sólido podría inventar cualquier conclusión. También es muy deseable la propiedad de **completitud**: un algoritmo es completo si es capaz de derivar cualquier sentencia implicada. El algoritmo de resolución que usaremos es sólido y completo para implicación en lógica de primer orden. Cada algoritmo de inferencia que se conoce en lógica proposicional tiene un peor caso, cuya complejidad es exponencial respecto de la entrada. No esperamos mejorarlo, ya que demostrar la implicación en lógica proposicional es un problema NP-completo.

Existen tres principales familias de algoritmos de inferencia en lógica: **forward chaining**, **backward chaining** y basados en **resolución**. La completitud de la resolución hace que éste sea un método de inferencia muy importante. Sin embargo, en muchos casos prácticos no se necesita todo el poder de la resolución. Ciertas bases de conocimiento contienen sólo cláusulas de un tipo restringido, denominadas **cláusulas de Horn**[2]. La inferencia con cláusulas de Horn se puede realizar mediante los algoritmos de encadenamiento hacia delante y de encadenamiento hacia atrás, los cuales suelen ser más eficientes que el método de resolución.

3.4.1. Encadenamiento hacia delante (*forward chaining*)

El algoritmo de encadenamiento hacia delante comienza a partir de los hechos conocidos (literales positivos) de la base de conocimiento. Si todas las premisas de una implicación se conocen, entonces la conclusión se añade al conjunto de hechos conocidos. Este proceso continúa hasta que la petición q es añadida o hasta que no se pueden realizar más inferencias.

Es fácil descubrir que el encadenamiento hacia delante es un proceso **sólido**: cada inferencia es esencialmente una aplicación de la regla *modus ponendo ponens*. El encadenamiento hacia delante también es **completo** para cláusulas de Horn si se emplea un algoritmo de búsqueda completo.

3.4.2. Encadenamiento hacia atrás (*backward chaining*)

El algoritmo de encadenamiento hacia atrás, tal como lo sugiere su nombre, trabaja hacia atrás a partir de la petición. Si se sabe que la petición q es verdadera, entonces no se requiere realizar ningún trabajo. En el otro caso, el algoritmo encuentra aquellas implicaciones de la base de conocimiento de las que se concluye q. Si se puede probar que todas las premisas de una de esas implicaciones son verdaderas (mediante encadenamiento hacia atrás), entonces q es verdadera.

A menudo, el coste del encadenamiento hacia atrás es mucho menor que el del encadenamiento hacia delante, porque el proceso sólo trabaja con los hechos relevantes. En la programación lógica, el encadenamiento hacia atrás, es la forma más ampliamente usada de razonamiento automático. El lenguaje de programación lógica más extensamente utilizado, Prolog, utiliza este tipo de algoritmo.

3.4.3. Resolución

El algoritmo de resolución es el que implementamos en este trabajo. Tiene la ventaja con respecto a los otros de que puede trabajar con todo el poder expresivo de la lógica de primer orden, sin limitarse a cláusulas de Horn. Su desventaja es que suele realizar una inferencia más lenta. Sin embargo, si la base de conocimiento está compuesta solamente por cláusulas de Horn, este algoritmo se puede ejecutar en la misma complejidad computacional que el encadenamiento hacia atrás.

El resto del informe explica detalladamente este método y su implementación.

[2]Una cláusula de Horn es una disyunción de literales de los cuales a lo sumo uno es positivo.

4. Demostración automática mediante resolución

4.1. Un poco de historia

Una sentencia es **válida**[3] si es verdadera para todas sus interpretaciones definidas sobre cualquier dominio. De la definición de implicación podemos derivar el **teorema de la deducción** que indica lo siguiente: para dos sentencias α y β cualesquiera,

$$\alpha \models \beta \text{ si y sólo si } \textit{la sentencia } (\alpha \Rightarrow \beta) \textit{ es válida}$$

Sabemos que si una fórmula es válida, su negación es insatisfacible. Por lo tanto, podemos concluir que

$$\alpha \models \beta \text{ si y sólo si } \textit{la sentencia } (\alpha \wedge \neg\beta) \textit{ es insatisfacible}$$

La demostración de β a partir de α averiguando la insatisfacibilidad de $(\alpha \wedge \neg\beta)$ se denomina **refutación**[4] o demostración por contradicción. Asumimos que la sentencia β es falsa y observamos si se llega a una contradicción con las premisas de α.

El proceso de demostrar la insatisfacibilidad de una sentencia en lógica de primer orden mediante la enumeración de modelos es, en general, imposible. Cuando la base de conocimiento contiene algún símbolo de función, el conjunto de sustituciones de los términos base se vuelve infinito.

El paso más importante en la demostración automática de teoremas fue dado por Herbrand en 1930. El método de Herbrand es un procedimiento de refutación que intenta probar que una fórmula es insatisfacible utilizando sólo las interpretaciones definidas sobre un dominio especial, llamado universo de Herbrand, y limitándose a un tipo especial de interpretaciones. Sin embargo, este método resultó ser muy ineficiente.

El siguiente paso más importante fue dado por Robinson, en 1965, quien introdujo el llamado *principio de resolución*. Este procedimiento de demostración es mucho más eficiente que cualquier otro procedimiento anteriormente desarrollado, porque en lugar de trabajar con instancias de cláusulas, como hacían los anteriores, lo hace directamente sobre las cláusulas que se obtienen a partir de la fórmula cuya insatisfacibilidad se pretende probar.

4.2. Forma prenex

El primer paso para determinar la insatisfacibilidad de una fórmula mediante el método de resolución de Robinson consiste en llevarla su forma prenex. Diremos que una fórmula en la lógica de predicados está en su forma prenex si y sólo si está expresada de la siguiente manera

$$Q_1 x_1 \cdots Q_n x_n M$$

donde cada $Q_i x_i$ para $i = 1, \cdots, n$ es o bien $\exists x_i$ o bien $\forall x_i$, y M es una fórmula que sólo contiene literales posiblemente conectados con los operadores de disyunción y conjunción, es decir, no contiene cuantificadores ni operadores de implicación o equivalencia.

[3] Las sentencias válidas también se conocen como **tautologías**.

[4] La prueba por refutación se corresponde exactamente con la técnica de demostración en matemática conocida como "reducción al absurdo".

La fórmula obtenida debe ser lógicamente equivalente a la original, para que la insatisfacibilidad de mantenga.

El archivo `Prenex.hs` implementa el procedimiento que lleva una fórmula lógica a su forma prenex. Lo primero que debemos hacer es eliminar los operadores de implicación y equivalencia. El segundo paso es desplazar las negaciones al interior de la sentencia (por sobre las conjunciones, disyunciones y cuantificadores). El siguiente paso es normalizar las variables para luego poder sacar los cuantificadores lo más afuera posible. Finalmente, se desplazan los cuantificadores hacia la izquierda para que afecten a toda la fórmula y no sólo a fórmulas parciales.

Todos los pasos son implementados en cuatro funciones que se componen para definir el procedimiento `prenex` que buscamos, como se puede apreciar en el código siguiente.

```
1 {- Lleva una formula a su forma prenex -}
2 prenex  :: Form -> Form
3 prenex   = moveQuan . norm . moveNot . elimImp
```

4.3. Forma normal de Skolem

Si a partir de una fórmula en forma prenex se eliminan los cuantificadores existenciales y se sustituyen las variables afectadas por funciones de Skolem, se obtiene una nueva forma, llamada de Skolem, que conserva la insatisfacibilidad.

Sea $A = Q_1 x_1 \cdots Q_n x_n M$ una fórmula en forma prenex, su forma de Skolem se obtiene de la siguiente manera: se suprimen todos los cuantificadores existenciales que haya en el prefijo $Q_1 x_1 \cdots Q_n x_n$ y además:

1. Si no hay cuantificadores universales a la izquierda del cuantificador $\exists x_r$ que se elimina, se elige una nueva constante C y se reemplaza todas las apariciones de x_r en M por C.

2. Si $Q_{s_1}, \cdots, Q_{s_m}$ son los cuantificadores universales que hay a la izquierda del cuantificador que se elimina $\exists x_r$, con $1 \leq s_1 < s_2 < \cdots < s_m < r$, se elige una nueva función f de aridad m y se reemplaza todas las apariciones de x_r en M por $f(x_{s_1}, x_{s_2}, \cdots, x_{s_m})$.

Las constantes y funciones usadas para reemplazar las variables de los cuantificadores existenciales se llaman funciones de Skolem.

La forma normal de Skolem se obtiene llevando M a su forma normal conjuntiva (i.e. como una conjunción de disyunciones de literales).

El archivo `Skolem.hs` implementa un procedimiento llamado `skolem` que lleva una fórmula en forma prenex a su forma normal de Skolem. Primero, se eliminan los cuantificadores existenciales sustituyendo las variables afectadas por constantes y funciones de Skolem y luego se eliminan los cuantificadores universales. Seguidamente se convierte la sentencia a su forma normal conjuntiva. Todo esto se puede apreciar en la siguiente porción de código.

```haskell
{- Lleva una formula en forma prenex a forma normal de skolem -}
skolem  :: Form -> State Int ClauseSet
skolem f = do f' <- elimQuan f []
              return (fromList . map fromList . distribute $ f')
```

En este punto cambiamos la estructura de datos usada para almacenar las sentencias a un conjunto de conjunto de literales. Esta estructura está representada por el tipo `ClauseSet`, definido en el archivo `Clauses.hs`. Un conjunto de literales representa a una cláusula, y un conjunto de cláusulas representa a toda una sentencia cualquiera.

La función `skolem` toma como entrada una fórmula en forma prenex de tipo `Form` y devuelve un conjunto de cláusulas de tipo `ClauseSet` embebido en la mónada `State`. Usamos una mónada `State` para llevar en el estado la cuenta de las funciones de Skolem ya utilizadas. De esta manera, podemos aplicar muchas veces la función `skolem` sin peligro de repetir nombres de funciones de Skolem ya usadas. El estado que elegimos es un simple entero, por lo que los nombres de las constantes y funciones de Skolem generados serán del tipo C_1, C_2, f_3, f_4, etc.

La función que primero se aplica, `elimQuan`, elimina los cuantificadores. Seguidamente se aplica la función `distribute`, que distribuye la conjunción sobre la disyunción para llevar la fórmula a su forma normal conjuntiva. Por último, se aplican las funciones de la librería `Data.Set` para pasar a conjunto de conjuntos.

4.4. Conjunto de cláusulas

Un literal es un átomo (predicado) o la negación de un átomo. Una cláusula es una disyunción de literales. Podemos representar una cláusula por un conjunto, donde cada elemento es un disyuntor. Para que una cláusula sea satisfacible, lo tiene que ser al menos uno de sus elementos. Por ende la cláusula vacía, que la representamos con el símbolo $\bot$, es insatisfacible.

Un conjunto de cláusulas se interpreta como la conjunción de las mismas. No debe confundirse la cláusula vacía con el conjunto vacío de cláusulas. Mientras que la cláusula vacía es insatisfacible, el conjunto vacío de cláusulas es satisfacible, de hecho es una tautología que se suele representar por $\top$.

Una forma normal de Skolem está compuesta por un conjunto de cláusulas, donde cada variable en cada una de las cláusulas está afectada por un cuantificador universal. Podemos representar una sentencia en su forma normal de Skolem simplemente con un conjunto de cláusulas o un conjunto de conjuntos de literales, al cual denotaremos con la letra S. Un resultado primordial es que la fórmula original es insatisfacible si y sólo si el conjunto de cláusulas S es también insatisfacible.

Como se explicó más arriba, el archivo `Clause.hs` define las estructuras de datos que representan al conjunto S. A continuación mostramos algunas de éstas.

```haskell
{- Estructura de datos para representar literales -}
data Literal = Atom String [Term]
             | Neg  String [Term]
```

```haskell

{- Sinonimo de tipo que representa una clausula -}
type Clause  = Set Literal

{- Sinonimo de tipo que representa un conjunto de clausulas -}
type ClauseSet = Set Clause
```

Para representar conjuntos utilizamos la librería de Haskell `Data.Set`. Esta librería proporciona una implementación eficiente de conjuntos basada en árboles binarios de tamaño balanceado. Ofrece funciones de inserción, búsqueda, eliminación, unión, etc.

4.5. El principio de resolución

Resolución en lógica proposicional

El método de resolución es un procedimiento de inferencia mediante refutación para la lógica proposicional. Este método utiliza una sola regla de inferencia sencilla, la **resolución**, que nos llevará a un algoritmo de inferencia completo si se empareja con un algoritmo de búsqueda completo.

La regla de resolución, para lógica proposicional, es la siguiente: sean α y β dos cláusulas cualesquiera, y l un literal cualquiera,

$$\frac{\alpha \vee l, \;\; \beta \vee \neg l}{\alpha \vee \beta}$$

La idea básica es aplicar la regla de resolución sucesivamente a las distintas cláusulas. Cada par que contiene literales complementarios se resuelve para generar una nueva cláusula, que se añade al conjunto ya presente. El proceso continúa hasta que se deriva en la cláusula vacía, o hasta que no haya nuevas cláusulas que se puedan añadir.

Denominamos cierre de la resolución de S, $\mathrm{CR}(S)$, al conjunto de todas las cláusulas derivables mediante la aplicación de la regla de resolución a partir del conjunto S. Es fácil ver que el conjunto $\mathrm{CR}(S)$ debe ser finito en la lógica proposicional, pues sólo hay un conjunto finito de cláusulas que se pueden generar a partir de los símbolos que aparecen en S. Por eso, este procedimiento de resolución siempre termina y es completo para la lógica proposicional.

Extensión a la lógica de primer orden

Este procedimiento de inferencia completo puede ampliarse o elevarse a la lógica de primer orden. La regla de resolución para la lógica de primer orden es simplemente una versión generalizada de la regla de resolución proposicional. Dos cláusulas, que se asume están con las variables estandarizadas y así no comparten ninguna variable, se pueden resolver si contienen literales complementarios. Dos literales en lógica de primer orden son complementarios si uno se unifica con la negación del otro. De este modo tenemos,

$$\frac{\alpha \vee l_1, \;\; \beta \vee \neg l_2}{\textsc{Subst}(\sigma, \; \alpha \vee \beta)}$$

donde α y β son dos cláusulas cualesquiera y $\text{UNIFY}(l_1, l_2) = \sigma$, es decir, el literal l_1 unifica con el literal l_2 y su unificador más general es σ. Más adelante explicamos cuándo dos predicados unifican. Básicamente, se refiere a cuando se pueden igualar mediante sustituciones de sus variables.

Esta regla se conoce como regla de **resolución binaria**, porque resuelve exactamente dos literales. La regla de resolución binaria por sí misma no nos da un procedimiento de inferencia completo. Si usamos la **factorización** (la eliminación de los literales redundantes) para el caso de la lógica de primer orden, y la combinamos con la resolución binaria, obtenemos un procedimiento de inferencia completo. La factorización proposicional reduce dos literales a uno si son idénticos; la factorización de primer orden reduce dos literales a uno si éstos son unificables.

En el Algoritmo 1 se muestra el procedimiento básico de resolución implementado. La función RESOLVE toma dos cláusulas y devuelve el conjunto de todas las resolventes binarias y todos sus factores, para obtener un algoritmo completo. Se realiza una búsqueda a lo ancho en el espacio de cláusulas derivadas, obteniendo así un procedimiento de inferencia completo. Sin embargo, se divide el conjunto de cláusulas en dos subconjuntos disjuntos new y old para asegurarnos de no generar cláusulas repetidas.

Algorithm 1: $\text{PROVE}(old, new)$ algoritmo de resolución

Input: new, conjunto de cláusulas generadas en el paso anterior
old, conjunto del resto de las cláusulas
Output: el árbol de resolución cuando $new \cup old$ sea insatisfacible o fallo

$old' \leftarrow new \cup old$
$resolvents \leftarrow \{\}$
while $new \neq \{\}$ **do**
  $x \in new$
  $new \leftarrow new \backslash \{x\}$
  foreach $y \in (new \cup old)$ **do**
    $resolvents \leftarrow resolvents \cup \text{RESOLVE}(x, y)$
  end
end
$new' \leftarrow resolvents \backslash old'$
if $new' = \{\}$ **then return** $fallo$
else
  return $\text{PROVE}(new', old')$
end

La implementación del algoritmo se encuentra en el archivo `Resolution.hs`, el cual exporta la siguiente función.

```
{- Toma dos conjuntos de clausulas, premisas y conclusion negada
 Convierte a conjuntos de arboles y le aplica resolucion -}
resolution :: ClauseSet -> ClauseSet -> Result
```

La función `resolution` toma dos conjuntos de cláusulas. Utilizamos esta función de dos maneras distintas, como explicaremos más adelante. En una de las maneras, los dos conjuntos de cláusulas representan la conclusión negada y las premisas respectivamente; de la otra manera, el primer conjunto tendrá todo el teorema a probar (premisas más conclusión negada) y el segundo conjunto estará vacío.

Lo primero que hace `resolution` es poner cada cláusula como raíz de un árbol con ese único nodo, y trabaja con conjuntos de árboles. Cuando dos raíces de dos árboles unifican, se genera una nueva cláusula que se coloca en la raíz de un nuevo árbol y como hijos se colocan las dos cláusulas de la cual deriva. De esta manera, podemos llevar la cuenta de la forma en que se deriva cada nueva cláusula y al derivar la cláusula vacía tenemos el árbol de refutación ya a disposición.

Debido a la evaluación *lazy* de Haskell y a las características que ofrece el compilador `ghc`, pudimos implementar grandes conjuntos de árboles que comparten muchos nodos entre sí sin preocuparnos del uso excesivo de memoria. Esto se debe a que, el compilador implementará los árboles usando para los hijos "punteros" a los otros árboles ya creados, y no duplicando toda la estructura de los árboles hijos.

Completitud de la resolución de primer orden

Se demuestra que la resolución es un procedimiento de refutación completo también para la lógica de primer orden, lo que significa que si un conjunto de sentencias es insatisfacible, entonces la resolución siempre será capaz de derivar una contradicción.

Tomaremos como premisa que cualquier sentencia en lógica de primer orden se puede transformar en un conjunto de cláusulas en FNC. Nuestro objetivo, por lo tanto, es demostrar los siguiente: *si S es un conjunto de cláusulas insatisfacible, entonces la aplicación de un número finito de pasos de resolución sobre S nos dará una contradicción.*

Esbozamos la demostración que se encuentra en [1], la cual está basada en la demostración original de Robinson, con algunas simplificaciones:

1. Primero, observamos que si S es insatisfacible, entonces debe existir un subconjunto de instancias de base[5] de las cláusulas de S que también es insatisfacible (teorema de Herbrand).

2. Entonces apelamos al **teorema fundamental de la resolución** que establece que la resolución proposicional es completa para sentencias base.

3. Luego utilizamos el **lema de elevación** que muestra que para cualquier demostración mediante resolución proposicional que utiliza un conjunto de sentencias base, existe su correspondiente demostración mediante resolución de primer orden que utiliza las sentencias de primer orden de las que se obtuvieron las sentencias base.

[5]Una instancia de base de una cláusula se obtiene reemplazando toda las variables por elementos del universo de Herbrand, obteniendo de esta manera lo que se denomina una sentencia base (sin variables).

4.6. Sustitución

Para obtener resolventes a partir de cláusulas en la lógica de primer orden hay que hacer sustituciones de variables. Formalmente, una sustitución es un conjunto finito de la forma: $\{t_1/v_1, \cdots, t_n/v_n\}$ tal que:

1. cada v_i es una variable;
2. cada t_i es un término distinto de v_i;
3. para cada $i \neq j$, $v_i \neq v_j$.

Sea $\alpha = \{t_1/v_1, \cdots, t_n/v_n\}$ una sustitución y sea E una expresión, entonces decimos que $\text{SUBST}(\alpha,\ E)$ es la expresión obtenida a partir de E sustituyendo cada aparición de la variable v_i por el término t_i. Para abreviar, notamos esta aplicación de sustitución directamente como $E\alpha$. La expresión resultante se dice que es una instancia de E.

Importante: todas las sustituciones deben aplicarse en forma simultánea y no de manera secuencial. Por ejemplo, sea $E = P(x)$ y $\alpha = \{y/x, a/y\}$. La instancia $E\alpha$ resulta de aplicar sólo la primera sustitución, reemplazando la variable x por la variable y, pero no se aplica la segunda sustitución pues originalmente no hay ninguna variable y para reemplazar. Es decir, el resultado será $P(y)$ y no $P(a)$.

Composición de sustituciones

Sean $\alpha = \{t_1/x_1, \cdots, t_n/x_n\}$ y $\beta = \{u_1/y_1, \cdots, u_m/y_m\}$ dos sustituciones. Entonces la composición de α y β es la sustitución designada por $\alpha\beta$ que se obtiene a partir del conjunto:

$$\{t_1\beta/x_1, \cdots, t_n\beta/x_n, u_1/y_1, \cdots, u_m/y_m\}$$

suprimiendo en este orden (importante no alterar el orden):

1. todo elemento u_i/y_i tal que y_i sea igual a alguno de los x_j;
2. todo elemento $t_i\beta/x_i$ tal que $t_i\beta = x_i$.

Como la sustitución debe aplicarse totalmente de manera simultánea, resulta importante aplicar la regla 1, pues de lo contrario tendríamos dos sustituciones de la misma variable y no sabríamos cual aplicar. La regla 2 debe aplicarse para no tener sustituciones que reemplacen una variable por sí misma.

De esta manera, para cualquier expresión E y cualquier par de sustituciones α y β tenemos que $(E\alpha)\beta = E(\alpha\beta)$. Es decir, aplicar la sustitución compuesta es lo mismo que aplicar ambas sustituciones en el mismo orden. Como consecuencia, la composición de sustituciones resulta asociativa y la sustitución vacía resulta ser el elemento neutro tanto a derecha como a izquierda.

La aplicación y composición de sustituciones será fundamental para efectuar la unificación. Su implementación se encuentra en el archivo `Unification.hs`. El tipo de la sustitución es simplemente una lista de pares término y variable, como se puede ver en el fragmento de código a continuación.

```haskell
{- Sinonimo de tipo para una sustitucion -}
type Subst = [(Term, String)]

{- Aplica una sustitucion a una lista de terminos.
 Nota: toda la substitucion se aplica de manera "simultanea" -}
substitute :: Subst -> [Term] -> [Term]

{- Operador que compone dos sustituciones -}
(+-+) :: Subst -> Subst -> Subst
```

4.7. Unificación

En el procedimiento de prueba por resolución, para identificar pares complementarios de literales, hay que unificar a menudo dos o más expresiones. El conjunto S es un conjunto de cláusulas, y se interpreta como la conjunción de las mismas. Como todas las variables están cuantificadas universalmente, podemos "normalizar" los nombres de las variables para que dos cláusulas distintas no compartan ningún nombre de variable en común, sin alterar el valor de verdad del conjunto.

El hecho de que cláusulas distintas no tengan variables en común facilita la unificación. Sin embargo, en el método de resolución el conjunto de cláusulas S aumenta su tamaño con cada inserción de una nueva resolvente. Con lo cual el número de variables distintas que necesitaremos puede llegar a ser demasiado grande y engorroso de manejar. Por este motivo, permitimos que las distintas cláusulas compartan variables entre sí, pero al momento de intentar unificar dos de ellas lo primero que haremos será normalizar sus variables en las copias locales que se intentarán unificar; es decir, las cláusulas originales que tomamos del conjunto S mantendrán sus nombres de variables intactos. Este enfoque nos permite obtener una buena eficiencia computacional al evitar normalizar constantemente el conjunto de cláusulas.

Algoritmo de unificación

Una sustitución σ se dice que es un unificador para el par de literales l_1 y l_2 si y sólo si $l_1\sigma = l_2\sigma$. Además, se dice que es el unificador más general (umg) si y sólo si para cada unificador α, existe una sustitución β tal que $\sigma\beta = \alpha$.

El Algoritmo 2 permite hallar el unificador más general (umg), en caso de que sea posible, de un par de listas finitas de términos, las cuales llamaremos l_1 y l_2. Trabaja mediante la comparación, elemento por elemento, de las listas de entrada. La sustitución σ, se va construyendo a lo largo de todo el proceso y se utiliza para asegurar que las comparaciones posteriores sean consistentes con las ligaduras que previamente se han establecido. En la línea 7, se verifica si una variable pertenece a un término, $v \in t$. Esto es verdadero si la variable ocurre dentro del término, por ejemplo, $x \in f(D, g(y, x))$ pero $x \notin g(C, z)$.

Algorithm 2: UNIFY(l_1, l_2, σ) algoritmo de unificación

Input: l_1, l_2, dos listas de términos de igual longitud para unificar
σ, el umg hasta el momento (inicialmente la sustitución vacía ε)
Output: el unificador más general o una indicación de falla (no unificables)

if $l_1 = l_2 = \langle\rangle$ **then return** σ
else
  $\langle h_1 \rangle \frown t_1 \leftarrow l_1$ /* separo las listas en cabeza y cola */
  $\langle h_2 \rangle \frown t_2 \leftarrow l_2$
  if $h_1 = h_2$ **then** UNIFY(t_1, t_2, σ)
  else if h_1 *es una variable* $v \ \wedge h_2$ *es un término cualquiera* t *(o viceversa)* **then**
    if $v \in t$ **then return** *fallo*
    else
      $t_1 \leftarrow t_1\{t/v\}$ /* sustituyo en el resto de las listas */
      $t_2 \leftarrow t_2\{t/v\}$
      return UNIFY$(t_1, t_2, \sigma\{t/v\})$
    end
  else if h_1 *es una funcion* $f(xs) \ \wedge h_2$ *es una funcion* $g(ys)$ **then**
    if $f = g$ **then** /* tienen el mismo nombre de función */
      $\sigma' \leftarrow$ UNIFY(xs, ys, σ)
      if $\sigma' = fallo$ **then return** *fallo*
      else
        $t_1 \leftarrow t_1\sigma'$ /* sustituyo en el resto de las listas */
        $t_2 \leftarrow t_2\sigma'$
        return UNIFY$(t_1, t_2, \sigma\sigma')$
      end
    else return *fallo*
  else return *fallo*
end

Este algoritmo de unificación está implementado con una función exportada en el archivo `Unification.hs`. Esta función, denominada `unification`, toma dos listas de literales para unificar y devuelve posiblemente una sustitución, como se muestra en el siguiente fragmento de código.

```
{- Intenta unificar dos listas de terminos.
 Si las listas son unificables devuelve el unificador mas
 general correspondiente, sino devulve Nothing -}
unification :: [Term] -> [Term] -> Maybe Subst
```

4.8. Variante: resolución mediante conjunto soporte

La estrategia del conjunto soporte comienza por identificar un conjunto de sentencias denominado **conjunto soporte**. Cada resolución combina una sentencia del conjunto soporte con otra sentencia cualquiera y añade la resolvente al conjunto soporte. Si el conjunto soporte es relativamente pequeño respecto a la base de conocimiento, el espacio de búsqueda se reduce drásticamente.

Tenemos que ser cautos con este enfoque, porque una selección errónea del conjunto soporte hará que el algoritmo sea incompleto. Sin embargo, *si elegimos el conjunto soporte S de tal manera que el resto de sentencias sean conjuntamente satisfacibles, entonces la resolución mediante el conjunto soporte será completa*.

El enfoque que elegimos es el de utilizar la petición negada como el conjunto soporte, bajo la asunción que la base de conocimiento original es consistente. (Después de todo, si no es consistente, entonces el hecho de la petición que sigue es trivial.) La estrategia del conjunto soporte tiene la ventaja adicional de generar árboles de demostración que a menudo son fáciles de entender por personas, ya que éstas están orientadas al objetivo.

El programa implementado puede realizar la inferencia mediante cualquiera de los dos métodos, el original o usando conjunto soporte. En la siguiente sección se indica como utilizar cada uno y se muestra la particularidad de cada una de las salidas para un ejemplo concreto.

5. Interfaz y Ejecución

Se implementó una interfaz de texto para facilitar la carga de las sentencias y la ejecución de la resolución. La implementación se encuentra en el archivo `Main.hs` y se compone básicamente de un ciclo que espera comandos del usuario y produce resultados en archivos o en la salida estándar, hasta que se sale del programa con el comando `exit`. La lista de comandos disponibles y sus funciones se pueden ver ejecutando `help`.

Para la implementación del procesador de comandos se hizo uso del transformador de mónadas `StateT`, llevando como estado un par de listas de sentencias, `Pair [Form]`, y combinándolo con la mónada `IO()`, para la interacción con el exterior. Esto resultó muy interesante y clarificó la implementación. Una de las listas de fórmulas del estado es para representar a la base de conocimiento (KB) y la otra es para representar a la pregunta o conclusión (*query*). Los comandos de carga, agregan sentencias a estas listas, y luego los comando de acción, por ejemplo `run` o `skolem`, utilizan el estado para ejecutar las operaciones pertinentes y mostrar la salida al usuario o guardarla donde se indique. Además se ofrecen comandos de borrado y listado para ver qué sentencias están ya cargadas y modificarlas.

Para compilar el programa usamos el *Glasgow Haskell Compiler* o `ghc`, en la versión 6.12.1. Creamos un pequeño makefile que hace uso del make del mismo `ghc`. Luego de posicionarnos en la carpeta `foltp`[6] ejecutamos `make` y se debería generar la compilación como se muestra en la Figura 3, produciendo el programa `Resolucion`.

A continuación veremos un par de ejemplos de problemas prácticos.

[6] foltp es un acrónimo para First Orden Logic Theorem Prover.

```
usuario@usuario-PC:~/foltp$ make
ghc --make -O2 Foltp/Main.hs -o Resolucion
[ 1 of 12] Compiling Pretty            ( Pretty.hs, Pretty.o )
[ 2 of 12] Compiling Foltp.AST         ( Foltp/AST.hs, Foltp/AST.o )
[ 3 of 12] Compiling Foltp.Unification ( Foltp/Unification.hs, Foltp/Unification.o )
[ 4 of 12] Compiling Foltp.Clauses     ( Foltp/Clauses.hs, Foltp/Clauses.o )
[ 5 of 12] Compiling Parsing           ( Parsing.lhs, Parsing.o )
[ 6 of 12] Compiling Foltp.Parser      ( Foltp/Parser.hs, Foltp/Parser.o )
[ 7 of 12] Compiling Monads            ( Monads.hs, Monads.o )
[ 8 of 12] Compiling Foltp.Prenex      ( Foltp/Prenex.hs, Foltp/Prenex.o )
[ 9 of 12] Compiling Foltp.Skolem      ( Foltp/Skolem.hs, Foltp/Skolem.o )
[10 of 12] Compiling Foltp.Standardize ( Foltp/Standardize.hs, Foltp/Standardize.o )
[11 of 12] Compiling Foltp.Resolution  ( Foltp/Resolution.hs, Foltp/Resolution.o )
[12 of 12] Compiling Main              ( Foltp/Main.hs, Foltp/Main.o )
Linking Resolucion ...
usuario@usuario-PC:~/foltp$ ./Resolucion

 Bienvenido al demostrador automático de teoremas de Logica de Primer Orden
 Para ayuda, escriba 'help'

> 
```

Figura 3: Compilación y ejecución del programa `Resolucion`.

Primer ejemplo: De Morgan generalizado

El programa está preparado para poder resolver cualquier teorema de lógica de primer orden, sin necesidad de que sea una implicación. A continuación vemos el ejemplo de una de las formas alternativas del teorema de De Morgan generalizado.

Primero debemos cargar la expresión a demostrar con el comando `conclusion`. Luego si ejecutamos el comando `listQuery` veremos la negación de la fórmula ingresada, de la cual vamos a intentar probar su insatisfacibilidad. Por último, con el comando `run` se ejecuta el algoritmo de resolución y se muestra el árbol de refutación, en el caso de que el teorema sea válido.

Toda la ejecución y las salidas se muestran en la Figura 4.

```
>
> conclusion( ~[\forall x : P(x)] == [\exists x : ~P(x)] )
>
> listQuery
1: ~(~[\forall x : P(x)] == [\exists x : ~P(x)])
>
> run
 La implicación es VERDADERA

{}
│
├─ {P(x1)}
│  │
│  ├─ {P(x2), P(x4)}
│  │
│  └─ {P(x2), ~P(C1)}
│
└─ {~P(C0)}
   │
   ├─ {P(x4), ~P(C0)}
   │
   └─ {~P(C0), ~P(C1)}

> 
```

Figura 4: Prueba de uno de los teoremas de De Morgan generalizado.

Segundo ejemplo: Muerte de un gato

Ahora veamos un problema de implicación extraído de la práctica y su resolución con las dos variantes de resolución mencionadas en el trabajo. El problema que vamos a resolver es el siguiente:

Juan tiene un perro y Pedro tiene un gato. Todos los que tienen una mascota aman a los animales. Nadie que ama a los animales los mata. Juan, Pedro o María mataron a la gata de Luis que se llama Iris. Se pide que se pruebe que María mató a Iris.

En el archivo `Test/gato.kb` se encuentra la base de conocimiento del problema. Los comandos para resolver el problema con el algoritmo general y la salida se muestran en la Figura 5.

Como la base de conocimiento es consistente, podemos aplicar la variante con conjunto soporte que presenta una mejor eficiencia. Los comandos para resolver el problema con el algoritmo que utiliza un conjunto soporte y su salida correspondiente se muestran en la Figura 6.

6. Discusión y Conclusiones

Este trabajo esperaba generar un software que aplique todas las etapas del algoritmo de Robinson para comprobar la validez de una expresión de lógica de primer orden. Todas las etapas fueron implementadas satisfactoriamente, el proyecto ha dejado una prueba de concepto funcionando que podrá ser modificado para extenderse en software educativo. Enumerando nuestros objetivos originales:

- La etapa del parser permite que puedan ingresarse al programa expresiones lógicas sin necesidad de darles una normalización previa.
- La conversión a formas normales Prenex y de Skolem permiten observar a modo de asistencia las etapas intermedias del algoritmo de resolución.
- Hemos brindado una justificación adecuada a los teoremas ingresados por medio de la construcción del árbol de refutación correspondiente.
- Si bien la eficiencia siempre puede mejorarse consideramos que las respuestas se generan en tiempos razonables para el tipo de problemas que se está trabajando.

Una observación interesante es que al tener una base de conocimiento consistente, con mucha información irrelevante a la conclusión que se desea probar, la variante con conjunto soporte refleja con mayor énfasis su ventaja sobre el algoritmo original. Esto se debe a que desde el inicio intentará unificar con la conclusión, lo cual provoca que las cláusulas que no tienen información relativa a la prueba nunca se unifiquen entre sí, evitando la generación masiva de cláusulas innecesarias. Por eso se dice que esta variante está orientada al objetivo, similar al *backward chaining*.

Satisfechos con el rendimiento del proyecto enumeramos lo que consideramos futuro trabajo a modo de extensión:

- La construcción de una interfaz gráfica más amigable al usuario
- Mejorar la cota del tiempo de respuesta. Esto puede incluir la utilización de algunas heurísticas como, por ejemplo, la de ponderar las cláusulas prefiriendo las más cortas primero. Sin embargo, hay que tener siempre presente lo que pueda ocurrir con la completitud del método de inferencia.
- Buscar la manera de generar contraejemplos para las expresiones que no representan teoremas, esto puede requerir bastante trabajo.
- Buscar un enfoque más interactivo para mejorar la experiencia de aprendizaje en la etapa de resolución.

Referencias

[1] Stuart J. Russell and Peter Norvig, *Artificial Intelligence, A Modern Approach*, Pearson Education, Inc., (2003).

[2] Enrique P. Arís, Juan L. González y Fernando M. Rubio, *Lógica Computacional*, Thomson, (2005).

[3] John Harrison, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, (2009).

[4] Michael Huth and Mark Ryan, *Logic in Computer Science, Modeling and Reasoning about Systems*, Cambridge University Press, (2004).

[5] Graham Hutton, *Programming in Haskell*, University of Nottingham, Cambridge University Press, (2007).

[6] Bryan O'Sullivan, John Goerzen and Don Stewart, *Real World Haskell*, O'Reilly Media, Inc., (2009).

[7] Richard Bird, *Introducción a la Programación Funcional con Haskell*, University of Oxford, Pearson Education, S.A., (2000).

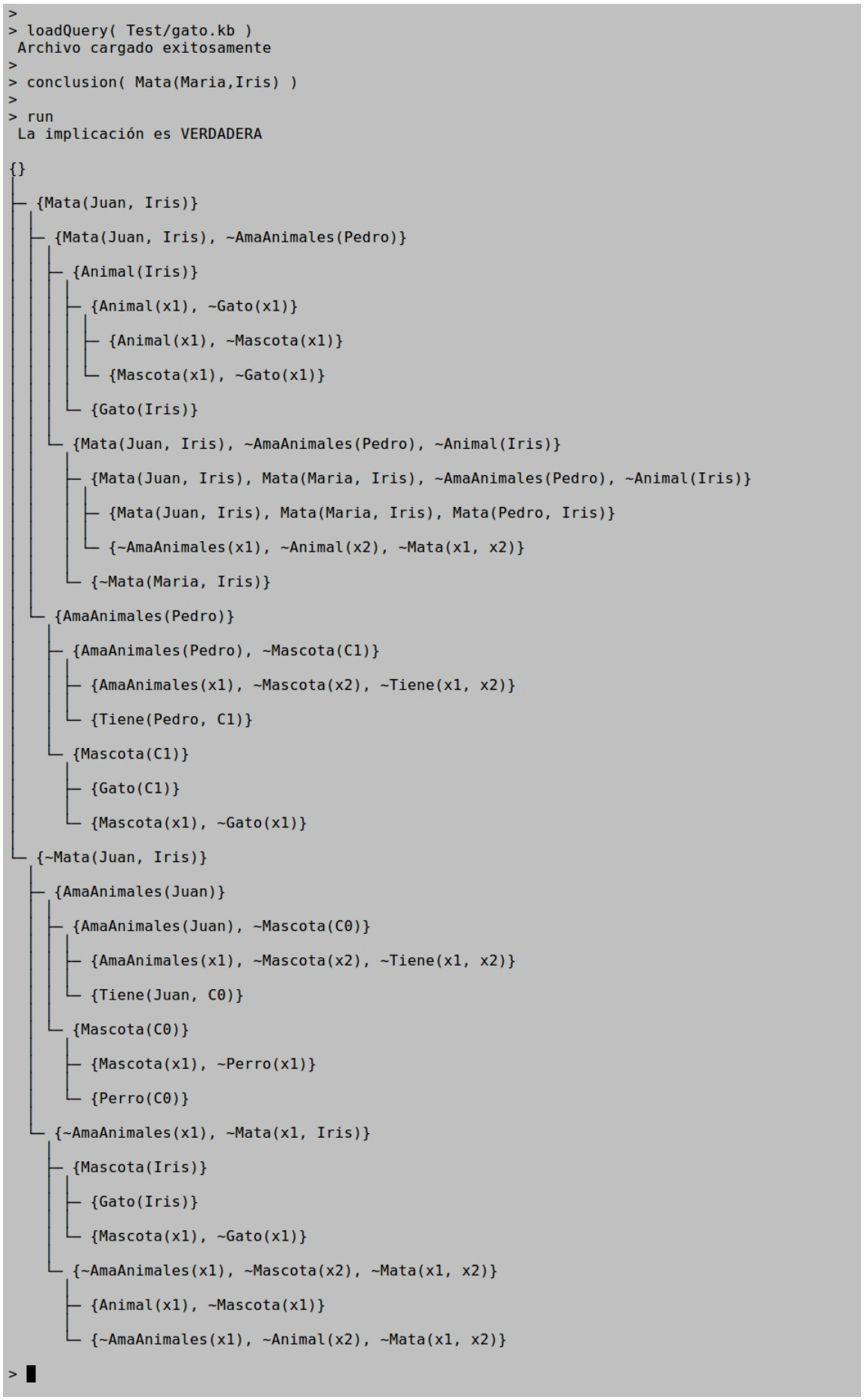

Figura 5: Prueba de la muerte de un gato sin conjunto soporte.

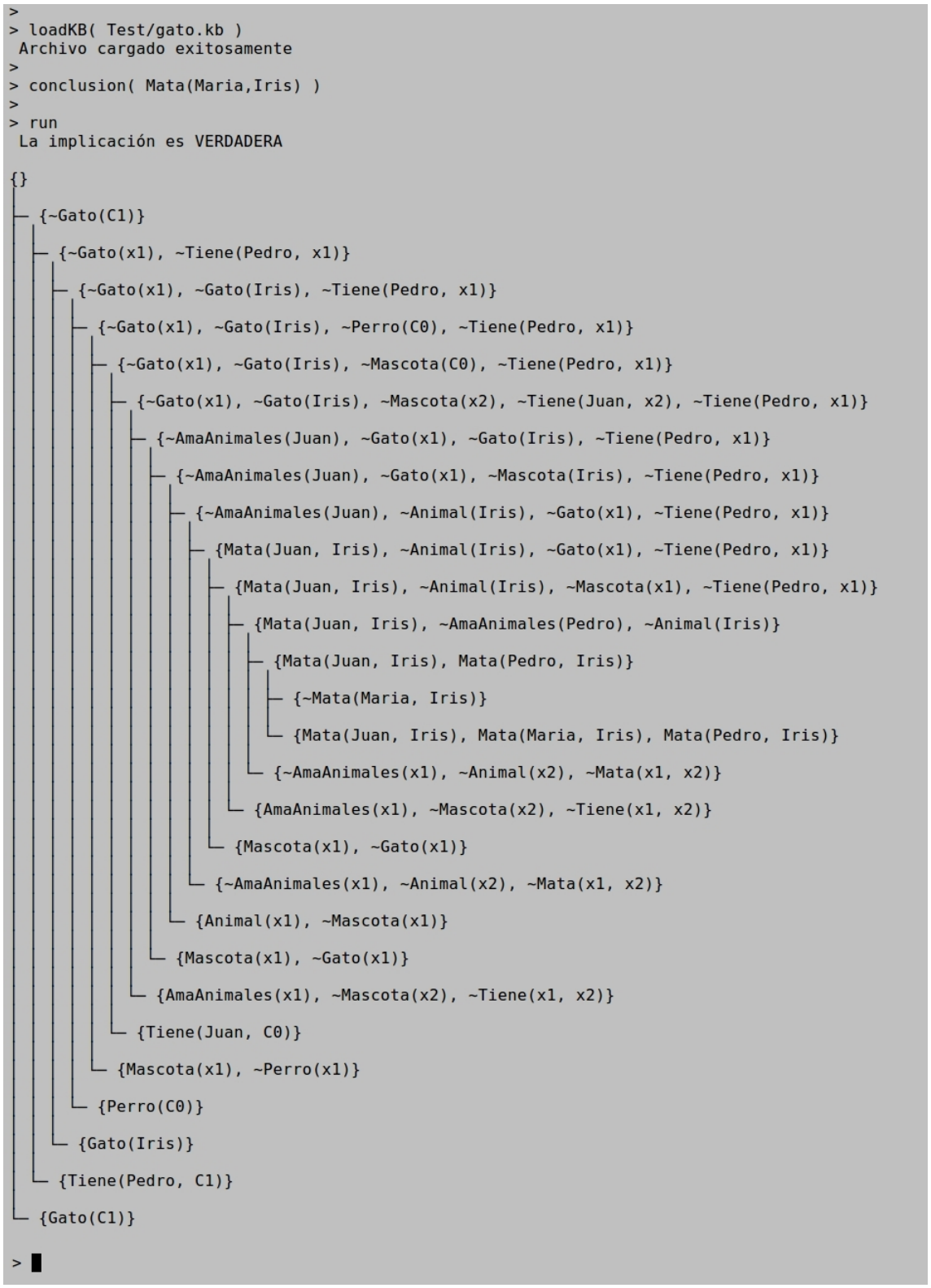

Figura 6: Prueba de la muerte de un gato usando conjunto soporte.