UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

Implementación de un demostrador automático de teoremas interactivo mediante el método de eliminación de modelos

PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2008-2009

Alumnos:
Delgado Muñoz, Agustín Daniel
Novillo Vidal, Álvaro
Pérez Morente, Fernando

Profesor director: López Fraguas, Francisco Javier

Índice

1. Resumen

1.1 Resumen en castellano

En este proyecto se ha desarrollado la herramienta de demostración automática interactiva de teoremas *DATEM* (**D**emostrador **A**utomático **I**nteractivo por **E**liminación de **M**odelos) en el lenguaje de programación lógica Prolog. Mediante esta herramienta es posible demostrar la validez de fórmulas lógicas a partir de teorías. Tanto las teorías como las fórmulas a demostrar se pueden introducir como conjuntos de cláusulas o como fórmulas de la lógica de primer orden. El sistema emplea el método de eliminación de modelos para intentar demostrar que una cierta fórmula es consecuencia lógica de un conjunto de premisas; si esto se consigue, además se obtiene una demostración de ello.

DATEM es una herramienta altamente configurable y que ofrece una gran versatilidad a la hora de efectuar sus demostraciones, permitiendo al usuario configurar el proceso de demostración a su gusto para obtener los mejores resultados. Además, cuenta con una interfaz gráfica amigable que facilita todas las acciones necesarias para llevar a cabo los procesos de demostración y la interpretación de los resultados obtenidos.

1.2 Resumen en inglés

In this project a tool for automatic interactive theorem proving using the model elimination method called *DATEM* (**D**emostrador **A**utomático **I**nteractivo por **E**liminación de **M**odelos) has been developed. This tool is written in the logic programming language Prolog. Using this application it is possible to prove the truth of logical theorems from a theory. Theories and theorems to prove can be written as set of clauses or using full first order logic formulas. This system employs the model elimination method to try to prove that a certain theorem is a logical consequence of a set of premises; if this goal is achieved the program shows an actual prove of that fact.

DATEM is a very customizable tool that offers great versatility to do demonstrations of theorems, allowing the user to custom the demonstration process to produce the best available results. Also it has a very friendly graphical user interface that makes it easy to perform all actions necessary to make the demonstration and the interpretation of the output.

1.3 Palabras clave

Demostración automática de teoremas, eliminación de modelos, cláusulas de Horn, lógica de primer orden, programación lógica, Prolog, XPCE.

2. Introducción

La ciencia moderna exige que todo resultado de cualquier tipo que se acepte universalmente debe de ser antes demostrado. Cada rama del conocimiento exige un tipo de demostración diferente. Por ejemplo, en medicina se exigen una serie de estrictos ensayos clínicos de diferentes niveles de complejidad antes de aceptarse un medicamento para curar una enfermedad; si este proceso no se lleva a cabo, el medicamento, aunque pueda parecer que tenga efectos muy beneficiosos para los enfermos, no se puede comercializar ni extender su uso si su eficacia y seguridad no se han visto probados en el restrictivo marco que marcan las leyes. En matemáticas, para que pueda admitirse que un enunciado es válido, hay que tener una demostración matemática, que no es más que un argumento irrefutable que a partir de hechos conocidos y demostrados permite deducir el teorema en cuestión. Por ello, las demostraciones juegan un papel imprescindible en toda ciencia moderna: un medicamento cuya eficacia no ha sido demostrada no tiene el margen mínimo de seguridad como para poder ser administrado libremente a los seres humanos; un teorema que no ha sido demostrado no es más que una conjetura, y no puede utilizarse para demostrar otros teoremas o para calcular nada. Una conjetura que no se ha podido demostrar, por muy cierta que parezca, ve limitada enormemente su utilidad. Por ello, es muy importante demostrar la certeza de todos los enunciados que se puedan llegar a proponer.

Hay muchas ciencias experimentales en las que es posible formalizar el conocimiento en forma de fórmulas lógicas. También existen métodos que se ha demostrado que son correctos y completos para demostrar que un cierto teorema se puede deducir a partir de una cierta teoría. Que sea correcto quiere decir que todo resultado que se deduzca a partir de una teoría será correcto siempre que ésta también lo sea. Que sea completo quiere decir que todo resultado correcto puede demostrarse a partir de la teoría. Si se tiene una teoría formalizada y un método que cumple las características anteriormente descritas, entonces se pueden demostrar los teoremas mediante la aplicación del método; de este modo, los pasos intermedios y el proceso de demostración en sí subyacen en la aplicación del método y los conocimientos acerca de su corrección que se poseen.

La demostración automática de teoremas es un campo de trabajo vigente desde la aparición de las primeras computadoras en el siglo XX. Por desgracia, los métodos existentes son muy difíciles de aplicar por parte de un ser humano debido a que requieren de muchísimas operaciones que hacen que sea una ardua tarea demostrar hasta los hechos más evidentes. Los seres humanos pueden producir demostraciones empleando la intuición o la experiencia, y en muchas ocasiones encontrarán demostraciones con un mayor sentido intuitivo que las que pueda encontrar la máquina. Sin embargo, con las modernas computadoras electrónicas es posible llevar a cabo estos procesos de manera automática, aunque las demostraciones calculadas pueden ser más difíciles de interpretar.

En este proyecto se implementa un demostrador automático según el método de eliminación de modelos, que es correcto y completo para las teorías escritas en forma clausal. Además, se ha diseñado un entorno que permite escribir las teorías, modificarlas y optimizar el método según las necesidades del usuario. Un usuario mínimamente experto podrá seleccionar las opciones necesarias para encontrar una demostración en el menor tiempo posible y que cumpla ciertas propiedades.

3. Objetivos

El principal objetivo de este proyecto ha sido diseñar e implementar un demostrador automático de teoremas que emplee el método de eliminación de modelos. Existen muchos demostradores automáticos comerciales y de calidad; este proyecto no pretendía competir en potencia o eficiencia con ellos. El objetivo principal ha sido desarrollar un sistema modesto pero de uso sencillo, que pudiera incluso emplearse eventualmente en la docencia como herramienta de apoyo para los estudiantes de lógica.

A continuación se listan todos los objetivos en los que se ha basado la realización del proyecto:

1. Desarrollar un demostrador automático de teoremas que empleara el método de eliminación de modelos, que está demostrado en diversos trabajos teóricos que es correcto y completo. El demostrador debía implementarse en el lenguaje de programación lógica Prolog, debido a que ofrece facilidades para la programación de este tipo de sistemas. Los teorías y los teoremas a demostrar tenían que estar formalizados en la forma clausal de la lógica de primer orden.
2. Mejorar el diseño original, que implementaba directamente el método de eliminación de modelos, para incrementar lo más posible la eficiencia del demostrador. En ningún caso se buscaba conseguir incrementos dramáticos de eficiencia, sino que la meta era proporcionar facilidades para llevar a cabo cierto tipo de demostraciones que con el método básico podían resultar demasiado costosas. Estas mejoras podían llevarse a cabo en algunos casos a costa de la completitud del método de demostración, conservándose siempre la corrección.
3. Permitir escribir las teorías y los teoremas a demostrar en lógica de primer orden. La forma clausal de la lógica es una forma restringida que requiere de un trabajo previo para escribir las teorías, que se escriben de forma mucho más natural cuando se pueden escribir fórmulas de cualquier tipo. Permitiendo introducir directamente todas las fórmulas en lógica de primer orden se incrementa la facilidad de uso del sistema y se ahorra trabajo al usuario.
4. Dar al usuario la posibilidad de configurar de manera sencilla en cada momento el sistema para que las demostraciones se lleven a cabo con las técnicas y parámetros que él elija. Todas las mejoras introducidas a partir del método básico debían ser configurables, de tal manera que se pudieran elegir las opciones más adecuadas para cada demostración; e incluso el usuario tendría que ser capaz de descubrir esa configuración óptima probando el comportamiento del sistema con distintas configuraciones.
5. Dotar al sistema de una interfaz gráfica amigable y sencilla de usar, con una ayuda explicativa y herramientas para la gestión de ficheros y la configuración del demostrador. De esta manera se buscaba facilitar el uso del sistema y adaptarse mejor a lo que se es-

pera de una herramienta docente: una interfaz amigable en la que los usuarios puedan poner en práctica los conocimientos adquiridos sin tener que realizar un gran esfuerzo en comprender el uso de la herramienta en sí.

Al final todos estos objetivos se han llevado a cabo en mayor o menor medida, consiguiéndose un sistema relativamente potente pero sencillo de utilizar.

4. Fundamentación teórica

4.1 Introducción

En este capítulo se abordan todos los conceptos teóricos que han sido necesarios para implementar el sistema. No se pretende dar una definición formal y rigurosa de todos ellos; para ese menester se pueden consultar innumerables libros que describen la lógica de primer orden y su forma clausal, y muchos artículos que describen en detalle, con demostraciones acerca de su corrección y completitud, el método de eliminación de modelos. Lo que aquí se pretende es describir de manera informal pero con rigor todo el trasfondo teórico en que se basa este trabajo, de modo que se pueda entender el sentido que tienen los métodos implementados y el alcance de las mejoras llevadas a cabo. La mayor parte de los conceptos que aquí se explican son intuitivamente muy sencillos y evidentes, especialmente para el lector familiarizado con la lógica matemática y la programación lógica; formalizarlos y demostrar todos los resultados que se han utilizado aumentaría en exceso la extensión y complejidad de este capítulo, sin aportar nada a lo que es la comprensión del sistema. El lector interesado en una visión más formal del trasfondo teórico de este proyecto siempre puede consultar el material al respecto que se lista en la bibliografía.

El capítulo se divide en dos grandes bloques. El primero (secciones 4.2 y 4.3) trata de la lógica de primer orden; en este bloque se describe en detalle sus sintaxis y su semántica, así como algunos resultados básicos acerca del proceso mediante el cual se llevan a cabo las deducciones, que es el objetivo principal del sistema implementado. También se define con precisión el subconjunto de la lógica de primer orden con el que se trabaja en el método de eliminación de modelos: la forma clausal; se describen con detalles métodos que permiten transformar fórmulas escritas en lógica de primer orden a fórmulas escritas en forma clausal.

En el segundo bloque se describe el método que se emplea para demostrar los teoremas, es decir, el método de eliminación de modelos.

4.2 Lógica de primer orden

4.2.1 Introducción

Una lógica estudia el modo de expresar conceptos de manera formal y proporciona mecanismos para llevar a cabo razonamientos con ellos. La lógica de primer orden que se emplea en este proyecto es una extensión de la lógica proposicional. La lógica proposicional es una de las lógicas matemáticas más básicas y sencillas, que permite escribir fórmulas que rela-

cionan enunciados simples; las fórmulas de la lógica proposicional no tienen la potencia suficiente para llevar a cabo razonamientos con ideas complejas.

En la lógica proposicional se pueden representar de manera simbólica conceptos sencillos, y luego, a partir de ellos, formar conceptos más complejos con las herramientas que proporciona. Los conceptos sencillos se representan mediante proposiciones, que pueden ser verdaderas o falsas. Los conceptos más complejos se forman como disyunciones, conjunciones, implicaciones o equivalencias de otros conceptos, que pueden ser a su vez sencillos o complejos. La certeza o falsedad de estos conceptos más complejos se puede calcular a partir de la certeza o falsedad de los conceptos más sencillos que los componen.

Una vez que se han representado en forma de fórmulas de la lógica proposicional una serie de conceptos, estas fórmulas se pueden manipular según las leyes de la lógica. Las leyes de la lógica son un conjunto de teoremas que permiten obtener información que no se tenía en la teoría original pero que se puede inferís a partir de ella. Una de las cosas que se puede hacer con estas fórmulas es demostrar si la certeza de un conjunto de ellas se puede deducir a partir de otras. En última instancia, ese es el objetivo de este proyecto.

La lógica que se emplea aquí es la lógica de primer orden. La lógica de primer orden permite expresar conceptos de manera más precisa que la lógica de proposiciones, pero los métodos de inferencia que trabajan con ella son más complejos.

Siempre que se trabaja con una lógica es imprescindible tener bien definido el universo de discurso. El universo de discurso es el conjunto de individuos, hechos, conceptos y entes abstractos de cualquier tipo que se nos pueda ocurrir sobre el que se van a escribir relaciones y propiedades. En la lógica de primer orden se representan los elementos del universo de discurso mediante términos. Los términos pueden ser directamente elementos del universo de discurso; pueden ser elementos indeterminados del mismo, que se representan mediante variables; y también pueden ser elementos obtenidos a partir de otros, aplicando funciones que permiten formar elementos a partir de otros.

Las fórmulas de la lógica de primer orden expresan características, ideas y propiedades que atañen a los elementos del universo de discurso. Las fórmulas más sencillas son los predicados, que expresan propiedades o relaciones de los términos. Luego, se pueden formalizar conceptos complejos de la misma manera que se hacía en la lógica proposicional. Además, la lógica de primer orden proporciona la capacidad de cuantificar variables, es decir, expresar mediante una fórmula compleja que una fórmula más simple se verifica para todos o para alguno de los posibles elementos del universo de discurso con los que se puede particularizar la variable cuantificada.

Una descripción completa y didáctica de la lógica de primer orden, con numerosos ejemplos y temas que aquí no se mencionan, se puede encontrar en (Hortalá, Leach Albert y Rodríguez Artalejo 2001) y en (Grassman y Tremblay 1998)

4.2.2 Sintaxis y semántica

En la lógica de primer orden hay dos tipos de elementos claramente diferenciados. Por un lado están los términos, que representan elementos del universo de discurso. Por otro lado están las fórmulas, que es la información que se maneja acerca de los términos. La sintaxis de algunos de los elementos de la lógica se ha restringido, empleando ciertas notaciones para escribir los símbolos de función, de predicado y de variable; estas restricciones no existen en la lógica de primer orden habitual, permitiéndose para estos elementos sintácticos cualesquiera conjuntos disjuntos que se elijan para representarlos; sin embargo, esta sintaxis se ha elegido para que sirva como convenio y así evitar tener que definir en cada caso el tipo de símbolo que se está manejando. La elección de estos convenios no ha sido arbitraria, sino que imita la que se emplea en el lenguaje de programación Prolog en el cual está programado el sistema.

Los términos se definen recursivamente de la siguiente manera:

1. Constantes
   Representan elementos concretos del universo de discurso. Se suelen representar mediante palabras que comienzan con letra minúscula. Por ejemplo: $pedro$, $cazo$, $calle$, $casa$ y $cero$.
2. Variables
   Representan elementos indeterminados del universo de discurso. Una variable puede tomar como valor cualquier elemento. Se suelen representar mediante palabras que comienzan con letras mayúsculas. Por ejemplo: X, Y y $Alumno$.
3. Símbolos de función
   Sirven para representar descripciones complejas de elementos del universo de discurso. Se pueden aplicar a otros términos para formar términos compuestos. El número de argumentos al que se aplica se llama aridad del símbolo de función. Las constantes se pueden considerar como símbolos de función de aridad cero. Su nombre se escribe mediante una palabra que empieza por letra minúscula, seguida de sus argumentos entre paréntesis y separados por comas. Por ejemplo, podemos tener un símbolo de función $pierna$ de aridad 1 que representa la pierna del término que tiene como argumento; así, $pierna(pedro)$ representa la pierna de Pedro. También podemos representar los números naturales mediante la constante cero y la función sucesor de aridad 1 denotada por el símbolo s; así, el número 1 se representa como $s(cero)$, el 2 como $s(s(cero))$ y así sucesivamente.

Por otro lado tenemos las fórmulas. Las fórmulas expresan propiedades y relaciones que atañen a los elementos del universo de discurso. La base de esta lógica es la certeza y la falsedad, y todas las fórmulas que enunciemos podrán tener uno de estos dos valores, que se llaman valores de verdad, dependiendo de la interpretación que se esté considerando.

Ahora se definen de manera recursiva las fórmulas. Expresamos la semántica de las mismas mediante el lenguaje natural, pero de forma rigurosa.

1. Fórmulas atómicas
   Se trata de las constantes lógicas y los predicados. Las constantes lógicas son cierto, que se representa mediante el símbolo ⊤ y falso, que se representa mediante el símbolo ⊥. Además, también están los predicados, que representan relaciones o propiedades de los términos. Se denotan mediante palabras que comienzan con letra minúscula y tienen asociada una aridad, es decir, el número de argumentos a los que se aplican. A los predicados de aridad 0 se les llama proposiciones. Los predicados de aridad mayor llevan escritos entre paréntesis y separados por comas sus argumentos, que deben ser términos bien construidos. Por ejemplo, se podría tener un predicado $padre$, de tal manera que $padre(pedro, juan)$ se interpreta como que Pedro es el padre de Juan, o un predicado mayor, de tal manera que se podría formalizar que el número 0 es menor que el número 1 escribiendo $menor(cero, s(cero))$. Para saber si un predicado es cierto o falso aplicado a una serie de argumentos, es necesaria una interpretación, que es una función que asigna uno de los dos valores de verdad a todos los predicados aplicados a términos cualesquiera; realmente el concepto de interpretación es más complejo, pues también involucra una función de estado para las variables, que permite sustituir el valor de cada variable por un término.

2. Fórmulas negadas.
   Si F es una fórmula de la lógica de primer orden, su negación también es una fórmula que se denota como $\neg F$. La negación de una fórmula F es cierta en todas las interpretaciones en las que la fórmula F es falsa.

3. Conjunción
   Si F y G son fórmulas de la lógica de primer orden, entonces la conjunción de ambas también lo es y se denota como $F \wedge G$. La conjunción de dos fórmulas F y G es cierta en todas las interpretaciones en las que F y G lo sean.

4. Disyunción
   Si F y G son fórmulas de la lógica de primer orden, entonces la disyunción de ambas también lo es y se denota como $F \vee G$. La disyunción de dos fórmulas F y G es cierta en todas las interpretaciones en las que F, G o ambas a la vez lo sean.

5. Implicación
   Si F y G son fórmulas de la lógica de primer orden, entonces la fórmula condicional que expresa que siempre que se cumpla una de ellas la otra también debe cumplirse es otra fórmula de la lógica de primer orden que se denota como $F \Rightarrow G$. A la fórmula F se le llama antecedente, y a G consecuente; la implicación es cierta en las interpretaciones en las que F es falsa, y en las que F es cierta y G también lo es.

6. Doble implicación
   Si F y G son fórmulas de la lógica de primer orden, entonces la fórmula de la equivalencia que expresa que siempre que se cumpla cualquiera de ellas la otra también debe

cumplirse también lo es y se denota como $F \Leftrightarrow G$. La equivalencia es cierta en las interpretaciones en las que F y G son ciertas o falsas a la vez.

7. Cuantificación existencial
   Si F es una fórmula de la lógica de primer orden y X es una variable, entonces todas las apariciones de la variable X en la fórmula F se pueden cuantificar existencialmente escribiendo la fórmula $\exists X.F$. Esta fórmula será cierta en todas las interpretaciones en las que la fórmula F sea cierta para alguna posible particularización de la variable X.

8. Cuantificación universal
   Si F es una fórmula de la lógica de primer orden y X es una variable, entonces todas las apariciones de la variable X en la fórmula F se pueden cuantificar universalmente escribiendo la fórmula $\forall X.F$. Esta fórmula será cierta en todas las interpretaciones en las que la fórmula F sea cierta para todas las posibles particularizaciones de la variable X.

Para completar este apartado, se dan algunos ejemplos de formalización de enunciados escritos en lenguaje natural a fórmulas de la lógica de primer orden. Se emplean símbolos de predicado y de función de significado intuitivo.

Enunciado	Formalización
Todos los perros roen huesos.	$\forall X.\big(perro(X) \Rightarrow roe(X, hueso)\big)$
El producto de cualquier número por 0 es 0.	$\forall X.\big(número(X) \Rightarrow igual(producto(X, cero), cero)\big)$
La liebre o el sombrerero mienten.	$miente(liebre) \vee miente(sombrerero)$
La abuela de Pedro sonríe cuando éste le visita.	$visita(pedro, abuela(pedro) \Rightarrow sonríe\big(abuela(pedro)\big)$

4.2.3. Otros conceptos

En este apartado se describen de manera superficial algunos conceptos de la lógica de primer orden que son imprescindibles para comprender algunos de los procesos que el sistema lleva a cabo. Estos son los conceptos relacionados con los posibles valores lógicos que pueden tomar las fórmulas según la interpretación que se considere y los relativos a la equivalencia entre fórmulas lógicas. También es esencial el concepto de consecuencia lógica, que es el núcleo en el que se basa el demostrador: al final lo que se intenta demostrar es que una fórmula es consecuencia lógica de una teoría

Una fórmula de la lógica de primer orden es lógicamente válida cuando es verdadera en todas las posibles interpretaciones. Así, por ejemplo, la fórmula $\exists X.p(X) \vee \neg\exists X.p(X)$ es lógicamente válida, ya que sea cual sea la interpretación considerada siempre será cierta. Si sucede al contrario y la fórmula es falsa en todas las interpretaciones, entonces se dice que la fórmula es contradictoria; un ejemplo de fórmula contradictoria es la fórmula $\exists X.p(X) \wedge \neg\exists X.p(X)$. Por último, si dependiendo de la interpretación una fórmula puede ser cierta o falsa, entonces se dice que la fórmula es una fórmula contingente; ejemplo de este último caso sería la fórmula $\exists X.p(X)$, que es cierta si en el universo de discurso existe algún término que hace cierto el predicado p y falsa en caso contrario.

Es muy importante el concepto de equivalencia lógica. Dos fórmulas son equivalentes cuando toman el mismo valor lógico en todas las interpretaciones. Cuando se traduce una fórmula escrita en lógica de primer orden a forma clausal, se busca conseguir un conjunto de fórmulas equivalente al original, de forma que las deducciones que se hagan a partir de ellas sean correctas para el conjunto de fórmulas originales. Por desgracia, y dadas las limitaciones de la forma clausal, esto no es posible. Se recurre entonces a una forma más débil de equivalencia: la equisatisfactibilidad. Dos fórmulas son equisatisfactibles si toman siempre el mismo valor de verdad para cualquier sustitución de sus variables por términos sin variables. Esto es así porque no puede garantizarse la equivalencia, al no existir los cuantificadores en la forma clausal de la lógica de primer orden.

Por último, es importante el concepto de consecuencia lógica. Se dice que una fórmula, a la que nos referiremos como conclusión, es consecuencia lógica de un conjunto de fórmulas, al que nos referiremos como conjunto de premisas o simplemente premisas, si cualquier interpretación que haga ciertas las premisas hace cierta la conclusión.

4.3 Forma clausal

4.3.1 Introducción

La forma cláusulada de la lógica de primer orden (o forma clausal) es un subconjunto de las fórmulas de la lógica de primer orden de gran interés en los campos de la programación lógica y la deducción automática. La principal ventajas de las fórmulas escritas en forma clausal es que admiten una lectura declarativa muy intuitiva, que puede trasladarse de manera inmediata a un algoritmo que puede programarse en un computador.

El método de demostración de eliminación de modelos que aquí se implementa admite como entrada fórmulas escritas en forma clausal; lo que pasa es que estas fórmulas no son las más adecuadas en el proceso de demostración; por ello se lleva a cabo un proceso de traducción inmediato a lo que es un subconjunto de la forma clausal: las cláusulas de Horn. El método de eliminación de modelos opera en última instancia con cláusulas de Horn, que son las más sencillas de interpretar de manera declarativa.

Como la forma clausal no es la manera más natural de expresar ciertas teorías, el sistema ofrece la posibilidad de escribir las fórmulas en lógica de primer orden. Por lo tanto, un nuevo paso de transformación es necesario, esta vez de forma general de la lógica de primer orden a forma clausal. En este paso no se consiguen fórmulas equivalentes, pero si equisatisfactibles, que garantizan que los resultados obtenidos son correctos en todo caso. La transformación a forma clausal, descrita en detalle más adelante, provoca que se pierda gran parte de la legibilidad de las fórmulas y causa que las demostraciones pierdan todo el sentido intuitivo que tienen cuando se escriben fórmulas directamente en forma clausal; este es el peaje que hay que pagar para poder ofrecer la posibilidad de escribir fórmulas en lógica de

primer orden, que puede ser una gran ventaja a la hora de simplificar la formalización de ciertas teorías.

En este capítulo se describe en detalle la forma clausal de la lógica de primer orden, la sintaxis de las cláusulas de Horn, y la transformación de fórmulas escritas en lógica de primer orden a forma clausal y de forma clausal a cláusulas de Horn. Todos estos métodos están implementados directamente en el sistema y son totalmente transparentes al usuario, cuyo único trabajo debe ser escribir sus teorías en el formato que más le convenga.

4.3.2 Forma clausal y cláusulas de Horn

En la forma clausal de la lógica de primer orden todas las fórmulas se escriben como una conjunción de cláusulas. Una cláusula es una disyunción de literales. Un literal es un símbolo de predicado, negado o sin negar, aplicado a tantos términos como indique su aridad. Por ejemplo, las siguientes fórmulas están escritas en forma clausal:

$$\big(p(X) \vee q(X)\big) \wedge \Big(\neg p\big(f(X,Y)\big) \vee q(Y) \vee p(Y)\Big)$$

$$p(X)$$

$$\big(q(X) \vee \neg p(X)\big)$$

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

Las cláusulas que componen una fórmula escrita en forma clausal pueden transformarse en otras equivalentes en forma de condicional, que admiten una interpretación declarativa muy intuitiva. Así, tomamos cualquier fórmula escrita en forma clausal y aplicamos la siguiente transformación, empleando leyes de equivalencia lógica:

$$\neg n_1 \vee \neg n_2 \vee \ldots \vee \neg n_k \vee p_1 \vee p_2 \vee \ldots \vee p_l \;\Rrightarrow \neg(n_1 \wedge n_2 \wedge \ldots \wedge n_k) \vee (p_1 \vee p_2 \vee \ldots \vee p_l)$$
$$\Rrightarrow (n_1 \wedge n_2 \wedge \ldots \wedge n_k) \Rightarrow (p_1 \vee p_2 \vee \ldots \vee p_l)$$

Para aplicar esta transformación ha sido necesario emplear varios resultados de equivalencia lógica, que no se demuestra aquí. Así pues, toda fórmula escrita en forma clausal, con k literales negativos y l postivos puede entender como que si se cumplen todos los literales negativos $n_1, n_2 \ldots n_k$ entonces se cumplirá uno de los positivos $p_1, p_2 \ldots p_k$.

Por ejemplo, aplicando este método a la fórmula $\big(p(X) \vee q(X)\big) \wedge \Big(\neg p\big(f(X,Y)\big) \vee q(Y) \vee pY$ tenemos:

$$\big(p(X) \vee q(X)\big) \wedge \Big(\neg p\big(f(X,Y)\big) \vee q(Y) \vee p(Y)\Big) \Rrightarrow p(X) \vee q(X) \wedge (\neg \Big(p\big(f(X,Y)\big)\Big) \vee$$
$$\big(q(Y) \vee p(Y)\big) \Rrightarrow \big(p(X) \vee q(X)\big) \wedge \Big(p\big(f(X,Y)\big) \Rightarrow \big(q(Y) \vee p(Y)\big)\Big)$$

Esto es una lectura intuitiva de los conjuntos de cláusulas, pero si el conjunto de literales positivos lo limitamos a que tenga un cardinal como mucho de uno, estamos definiendo las

cláusulas de Horn. El limitar este cardinal a 0 ó 1 no es arbitrario, ya que entonces las cláusulas pueden admitir únicamente las siguientes formas:

$\neg n_1 \vee \neg n_2 \vee \ldots \vee \neg n_k \vee p \Rightarrow (n_1 \wedge n_2 \wedge \ldots \wedge n_k) \Rightarrow p$

Esta es la forma que toman las cláusulas de Horn cuando hay un literal positivo. A estas fórmulas se les llama reglas, ya que siempre que se cumplen los literales negativos se cumple el positivo. Desde cierto punto de vista se ha definido un algoritmo: para hacer p, antes hay que hacer n_1, $n_2 \ldots n_k$. En el caso particular en el que no haya ningún literal negativo, entonces la fórmula es cierta siempre y se dice que lo que se está representando es un hecho, pues el literal positivo es siempre trivialmente cierto.

Cuando el cardinal del conjunto de literales positivos es 0, nos encontramos con la siguiente forma:

$\neg n_1 \vee \neg n_2 \vee \ldots \vee \neg n_k \Rightarrow \neg(n_1 \wedge n_2 \wedge \ldots \wedge n_k)$

Esto quiere decir que no puede suceder que todos los literales negativos sean ciertos a la vez. Estos son los que se conocen como objetivos, que provienen de la negación de la conclusión de la argumentación que se quiere demostrar. El sentido de esto viene de que los métodos de deducción automática intentan demostrar que una conclusión es consecuencia lógica de un conjunto de premisas demostrando que el conjunto de premisas junto con la negación de la conclusión son insatisfactibles, es decir, no hay ninguna interpretación que los hace ciertos.

4.3.3 De la lógica de primer orden a la forma clausal

En esta sección se explica con un ejemplo el método general que permite transformar fórmulas escritas en lógica de primer orden a forma clausal. Según se va explicando el método, se va desarrollando un ejemplo en el que se pueden visualizar todos los pasos que se explican. Como ya se mencionó anteriormente, las cláusulas resultantes no son equivalentes a la fórmula de partida, pero si son equisatisfactibles, lo que a efectos prácticos resulta igualmente útil. Este método no es exactamente el que se ha implementado en el sistema, ya que allí se han organizado de otra manera las transformaciones para que estas resultasen más sencillas de implementar.

Como ejemplo se va a pasar la fórmula escrita en lógica de primer orden $\forall X.\left(p(X) \wedge \exists Y.pX,Y \Rightarrow \neg qX.\right.$

El proceso se lleva a cabo en una serie de pasos, que se efectúan de manera secuencial. Los pasos son los siguientes:

1. El primer paso consiste en transformar la fórmula en otra equivalente de la forma $\exists X_1, \exists X_2, \ldots, \exists X_k, \forall X_{k+1}, \forall X_{k+2}, \ldots, \forall X_n F$, donde aquí F es una fórmula sin variables. Es decir, lo que se hace es sacar fuera de la fórmula todas las cuantificaciones. La

fórmula resultante de esta transformación se dice que está en forma normal prenexa, a $\exists X_1, \exists X_2, \dots, \exists X_k, \forall X_{k+1}, \forall X_{k+2}, \dots, \forall X_n$ se le llama prefijo y a F se le llama núcleo. Para llevar a cabo esta transformación se emplean sucesivamente algunos resultados de equivalencia lógica. También puede ser necesario algún renombramiento de variables.

En el ejemplo aplicamos este paso y pasamos a tener la fórmula $\forall X. \exists Y. \Big(p(X) \wedge \big(p(X,Y) => \neg q(X)\big)\Big)$, donde $\forall X. \exists Y.$ es el prefijo y $p(X) \wedge \big(p(X,Y) \Rightarrow \neg q(X)\big)$ es el núcleo.

2. Una vez se tiene una fórmula escrita en forma prenexa, el siguiente paso consiste en transformarla en otra equisatisfactible eliminando las cuantificaciones existenciales. En este paso no se puede conservar la equivalencia lógica. Al final, se parte de una fórmula en forma prenexa de la forma $\exists X_1, \exists X_2, \dots, \exists X_k, \forall X_{k+1}, \forall X_{k+2}, \dots, \forall X_n F$ y, tras aplicar el proceso de skolemización, se tiene una fórmula en forma de Skolem de la forma $\forall X_{k+1}, \forall X_{k+2}, \dots, \forall X_n F'$; en este formato la fórmula ya sólo tiene cuantificaciones universales en su prefijo y el núcleo es evidente que ha cambiado, pues si se eliminan las cuantificaciones existenciales es necesario llevar a cabo alguna transformación con la variable cuantificada que aparecía en el núcleo.. El proceso de Skolemización es sencillo; consiste en eliminar sucesivamente cada cuantificación existencial $\exists X_i$ sustituyendo todas las apariciones de la variable X_i en el núcleo de la fórmula por un símbolo de función f_i, que puede tener una aridad distinta de 0 en función del resto de cuantificadores del prefijo de la fórmula. Esta constante puede interpretarse intuitivamente como un testigo de la fórmula existencial: es decir, si la fórmula existencial se cumple para alguna asignación de la variable cuantificada, el símbolo es precisamente un término para el que se cumple.

   En el ejemplo se aplica este paso y así se tiene la fórmula $\forall X. \Big(p(X) \wedge \Big(p\big(X, c_0(X)\big) \Rightarrow \neg q(X)\Big)\Big)$, donde se ha introducido el símbolo de Skolem c_0 de aridad 1, que es el testigo de la fórmula existencial.

3. Transformar el núcleo de la fórmula que está en forma normal de Skolem a una fórmula equivalente en forma normal conjuntiva. Esto se consigue mediante algunos resultados de equivalencia lógica, siguiendo una serie de pasos de manera secuencial. Primero se eliminan las conectivas $\Rightarrow$ y $\Leftrightarrow$ mediante las equivalencias $F \Rightarrow G \equiv \neg F \vee G$ y $F \Leftrightarrow G \equiv \neg F \vee G$. Después, aplicando otra serie de reglas se completa la transformación.

   En el ejemplo se aplican diferentes reglas de equivalencia lógica al núcleo:

   $p(X) \wedge \Big(p\big(X, c_0(X)\big) \Rightarrow \neg q(X)\Big) \equiv p(X) \wedge \Big(\neg p\big(X, c_0(X)\big) \vee \neg q(X)\Big) \equiv \Big(p(X) \wedge$
   $\neg pX, c0X \vee pX \wedge \neg qX \equiv pX \vee \neg qX \wedge \neg pX, c0X \vee pX \wedge \neg pX, c0X \vee \neg qX$

4. Eliminar todas las cuantificaciones universales. Simplemente se eliminan las cuantificaciones universales, asumiéndose que todas las apariciones de variables en la fórmula resultante están cuantificadas universalmente de manera implícita.

4.3.4 De la forma clausal a las cláusulas de Horn

Ahora se explica el método que permite generar a partir de una cláusula general un conjunto de cláusulas de Horn equivalente. Este conjunto de cláusulas de Horn aporta la misma información que la cláusula general de la que se partió y se puede utilizar de una manera muy sencilla en un proceso de deducción.

En primer lugar, se consideran reglas escritas en forma general, de la forma $(n_1 \wedge n_2 \wedge ... \wedge n_k) \Rightarrow (p_1 \vee p_2 \vee ... \vee p_l)$. Entonces, se generan $k + l$ cláusulas de Horn, de la siguiente manera:

$$\neg p_2 \wedge ... \wedge \neg p_l \wedge n_1 \wedge n_2 \wedge ... \wedge n_k \Rightarrow p_1$$
$$\neg p_1 \wedge ... \wedge \neg p_l \wedge n_1 \wedge n_2 \wedge ... \wedge n_k \Rightarrow p_2$$
$$\vdots$$
$$\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_{l-1} \wedge n_1 \wedge n_2 \wedge ... \wedge n_k \Rightarrow p_l$$
$$\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_l \wedge n_2 \wedge ... \wedge n_k \Rightarrow \neg n_1$$
$$\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_l \wedge n_1 \wedge ... \wedge n_k \Rightarrow \neg n_2$$
$$\vdots$$
$$\neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_l \wedge n_1 \wedge n_2 \wedge ... \wedge n_{k-1} \Rightarrow \neg n_k$$

También es necesario llevar a cabo un proceso similar con los objetivos. Así, si se pretende generar el conjunto de cláusulas de Horn equivalentes a un objetivo de la forma $n_1 \wedge n_2 \wedge ... \wedge n_k$, se generan k cláusulas de Horn de la siguiente manera:

$$n_2 \wedge ... \wedge n_k \Rightarrow \neg n_1$$
$$n_1 \wedge ... \wedge n_k \Rightarrow \neg n_2$$
$$\vdots$$
$$n_1 \wedge n_2 \wedge ... \wedge n_{k-1} \Rightarrow \neg n_k$$

4.4 Eliminación de modelos

4.4.1 Introducción

En esta sección se explica el método de eliminación de modelos, que es el que se implementa en el demostrador. Este método parte de un conjunto de cláusulas de Horn que constituyen la teoría, al que se añaden las cláusulas de Horn resultantes de aplicar al objetivo el pro-

ceso descrito en el apartado 4.3.4. A partir de ahí se demuestra el objetivo mediante el conjunto de cláusulas disponibles, aplicando las reglas del método.

El método básico es correcto y completo, aunque algunas de las optimizaciones introducidas pueden hacer perder la completitud, aunque nunca la corrección.

Una descripción matemática del método de eliminación de modelos se puede encontrar en (Loveland, Mechanical theorem proving by model elimination 1968) y en (Loveland, A simplified format for the model elimination theorem-proving procedure 1969). Una descripción del método ya adaptado a Prolog encuentra en (Stickel 1984) y en (López Fraguas 1990).

4.4.2 Método de eliminación de modelos

El método de eliminación de modelos es un método de demostración; permite demostrar que una cláusula es la consecuencia lógica de un conjunto de cláusulas. El método de eliminación de modelos es muy adecuado para ser implementado en Prolog, gracias a que uno de sus pasos básicos para avanzar en las demostraciones se puede hacer corresponder con la unificación que ofrece Prolog. Realmente, la versión que aquí se explica está adaptada para que su implementación en Prolog sea directa.

El método parte de un conjunto de cláusulas, expresadas en forma de cláusulas generales de las que se explicaron en la sección 4.3.2 y un objetivo a demostrar, y genera como salida la demostración de cómo el objetivo se puede deducir a partir del conjunto de cláusulas. Además, en la salida se dan valores a las variables presentes en el objetivo; así pues, aparte de demostrarse la validez del objetivo, se obtienen los elementos del universo de discurso para los cuales se cumple. Todas las variables presentes en el objetivo se suponen cuantificadas existencialmente, y las presentes en las reglas están cuantificadas de manera universal.

El primer paso del demostrador consiste en transformar el conjunto de cláusulas de entrada en un conjunto de cláusulas de Horn, que son las que realmente se van a emplear a lo largo del proceso de demostración. Esta transformación se puede hacer en todo caso, mediante el procedimiento explicado en la sección 4.3.4. También hay que obtener reglas a partir del objetivo, mediante el tratamiento de objetivos descrito en el apartado 4.3.4. Una vez se ha obtenido este conjunto de reglas, se puede comenzar el proceso de demostración.

El uso de reglas procedentes del objetivo es imprescindible para obtener respuestas disyuntivas. Una respuesta disyuntiva es una respuesta que demuestra el objetivo sin asignar un valor a alguna de sus variables; en vez de eso se da un conjunto de valores para los que se ha demostrado el objetivo. La demostración debe interpretarse como que el objetivo es válido para uno de los valores que se devuelven para esa variable, sin saberse para cuál de ellos se cumple.

El proceso de demostración se hace empleando de manera sucesiva dos reglas: la expansión y la reducción. En cada paso de demostración se tiene una conjunción de literales que se busca demostrar, de la forma $p_1 \wedge ... \wedge p_k \wedge \neg n_1 \wedge ... \wedge \neg n_l$. Entonces se elije el primero de ellos como objetivo a demostrar y se comienza un proceso de demostración; si este proceso tiene éxito entonces se pasa a demostrar el siguiente literal de la conjunción, y así sucesivamente hasta que se han demostrado todos los literales que conforman el objetivo. Cada vez que se demuestra uno de los literales, se pueden ligar variables que aparecen con términos; es posible encontrar diferentes demostraciones para cada uno de los literales, y cada una de estas demostraciones puede verificarse para diferentes asignaciones a las variables que aparecen en ellos; es necesario probar todas las posibles asignaciones de variables, pues algunas pueden permitir que el proceso de demostración finalice con éxito y otras llevar a un punto en el que es imposible continuar.

La regla de expansión se aplica cuando se tiene un objetivo G y una regla en la base de reglas de la forma $L_1 \wedge ... \wedge L_n \Rightarrow G_1$, de forma que G unifique con G_1. Esta unificación puede entenderse como que existe una posible particularización de las variables de G de tal manera que G coincida con G_1. Pueden quedar variables libres después de este proceso de unificación, y también puede que alguna variable quede ligada a un término. Si esto ocurre, entonces el objetivo a demostrar pasa a ser $L_1 \wedge ... \wedge L_n$, donde se han sustituido todas las apariciones de las variables ligadas en la unificación al término al que se han ligado. A este nuevo objetivo se aplica nuevamente el proceso de eliminación de modelos.

La regla de reducción se aplica cuando se está demostrando un objetivo G y en la demostración, tras aplicarse una serie de reglas, se tiene que demostrar el objetivo $\neg G$. Entonces se puede considerar el objetivo G resuelto y continuar la demostración desde ese punto. Este proceso puede interpretarse como una demostración por reducción al absurdo.

La aplicación sistemática de estas reglas puede llevar a tres distintos tipos de situación. Por un lado, se puede llegar a un éxito al demostrarse cada uno de los literales que componen el objetivo. En este caso se ha demostrado el objetivo del que se partía y se ha hallado una ligadura para sus variables que hace que se cumpla y una demostración correspondiente. Por otro lado, puede ser que se apliquen reglas y nunca se llegue a demostrar el objetivo inicial ni a fallar; este caso se previene en el sistema implementado limitando el número de reglas que se pueden llegar a aplicar en una demostración, de la manera que se explicará en el capítulo correspondiente a la implementación del sistema. Finalmente, puede que se exploren todas las posibles demostraciones y no se halle una solución; en ese caso, no hay ninguna demostración para el objetivo del que se partía.

4.4.3 Optimizaciones

A continuación se describen algunas optimizaciones del método básico que se han llevado a cabo y se intenta dar una justificación intuitiva de su correcto funcionamiento.

1. Ancestro idéntico

Si al intentarse demostrar un objetivo G aparece otra vez ese objetivo G, se puede hacer fallar esa rama de la demostración. Esto es debido a que cualquier demostración del objetivo que pudiera hallarse tras su segunda aparición podría hallarse a partir de la primera, con menos recursos.

2. Ancestro complementario

   Si se está resolviendo un objetivo G como parte de la demostración de un objetivo $\neg G$, entonces puede descartarse el uso de pasos de expansión para dicho objetivo, ya que puede darse un paso de reducción que lo resolvería de manera más sencilla.

3. Lemas

   Es posible almacenar en una base de datos ciertos objetivos demostrados junto con su demostración. De esa manera, cuando estos objetivos aparecen se pueden demostrar automáticamente mediante la información almacenada.

5. Arquitectura del sistema

5.1. Lenguajes y herramientas utilizados

5.1.1. Prolog

Prolog (del francés ***Pro****gramation en* ***Log****iqué*) es un lenguaje de programación lógico y semi-interpretado diseñado en los años 70 en la Universidad de Aix-Marseille por A. Colmerauer y P. Roussel basándose en las ideas propuestas por R. Kowalski (Universidad de Edimburgo).

Obtuvo popularidad gracias a David H. D. Warren, que desarrolló un compilador que traducía Prolog en un conjunto de instrucciones de una máquina abstracta denominada Warren Abstract Machine (WAM), implementada de manera eficiente. Desde entonces, Prolog es un lenguaje muy usado en ámbitos académicos para la investigación, y en su momento fue promovido en Japón dentro del proyecto que buscaba desarrollar la "Quinta Generación de Computadores", gracias a su fundamentación matemática y al enfoque de programación que promulga.

El soporte matemático de Prolog está íntimamente ligado con la Lógica Matemática y especialmente con un subconjunto de la lógica de primer orden denominado "Cláusulas de Horn", las cuales pueden ser *hechos* (cláusulas incondicionales) o *reglas* (cláusulas condicionales), sobre las cuales se realiza un proceso de inferencia con la meta de verificar ciertos objetivos.

Entre los mecanismos en los que se basa Prolog destacan la unificación, la resolución, la vuelta atrás (backtracking), el reconocimiento de patrones, la gestión del indeterminismo, el occur check y la recursividad. En particular, el mecanismo de vuelta atrás permite la búsqueda de alternativas para satisfacer un cierto objetivo cuando se ha llegado a una situación en la que no se permite continuar con el proceso de inferencia, lo cual sirve de gran ayuda a la hora de construir un demostrador automático.

Algunas de las aplicaciones de Prolog más conocidas son la Inteligencia Artificial, los Sistemas Expertos, Compiladores, Demostradores de Teoremas, etc.

Dada la naturaleza del presente proyecto es inmediato pensar en Prolog como herramienta de trabajo fundamental para llevar a cabo su desarrollo, debido a que está orientado a la implementación de sistemas similares, gestiona el indeterminismo e incluye un mecanismo de unificación de variables, i.e., incluye herramientas imprescindibles y de gran ayuda para la implementación de un demostrador automático.

Se ha utilizado (Sterling y Shapiro 1986) como libro de referencia del lenguaje.

5.1.2. SWI-PROLOG

Hemos escogido SWI-Prolog fundamentalmente por tratarse de un estándar de Prolog de libre distribución. Desde la página web oficial de SWI-Prolog se pueden descargar distintas versiones para diferentes sistemas operativos y distintos complementos entre los que destaca un editor gráfico disponible en distintos idiomas (entre ellos el castellano).

www.swi-prolog.org

Dicho editor gráfico facilita las labores de programación y depuración, fundamentales para poder desarrollar un sistema complejo en Prolog.

Para su mejor compresión y su correcta utilización han sido necesarias frecuentes consultas a la guía de usuario de SWI-Prolog (Guía de usuario de SWI Prolog s.f.), disponible en Internet. También se puede acceder a los contenidos de esta ayuda a través de la consola de SWI-Prolog, mediante el predicado *help*/1, por ejemplo: *help(write)*.

En concreto, se ha hecho uso de la última versión estable a día de hoy, la 5.6.64. Además, esta distribución integra la librería XPCE que hemos utilizado para desarrollar la interfaz gráfica de nuestro sistema, y que se comenta con más detalle en el siguiente subapartado de la presente memoria.

5.1.3 XPCE/Prolog

XPCE es una herramienta, a caballo entre la programación lógica y la programación orientada a objetos, que facilita la creación de aplicaciones gráficas para diferentes lenguajes y especialmente enfocado a Prolog, lenguaje para el que proporciona una gran facilidad de uso, permitiendo producir un código muy homogéneo y comprensible.

XPCE/Prolog fue desarrollado por Jan Wielemaker y Anjo Anjewierden, de la Universidad de Ámsterdam, y es de libre distribución, estando plenamente disponible su código fuente a través de Internet. Se ha utilizado la versión 6.6.37, la última a día de hoy, que se incluye integrada en las últimas versiones de SWI-Prolog (librería PCE).

Su manual (Guía de usuario de XPCE s.f.) también disponible en la red, ha resultado de una utilidad relativa, y en la práctica se ha consultado con mucha mayor frecuencia el explorador de clases de XPCE, para conocer a fondo los métodos de las clases predefinidas que se han utilizado.

De entre las características de XPCE/Prolog, a continuación se enumeran las principales y las que han tenido una mayor repercusión a la hora de implementar la interfaz gráfica de usuario, aspectos que hacen de esta herramienta una extensión natural de Prolog.

- Los gráficos están definidos en C, en busca de la rapidez de respuesta necesaria en las interfaces gráficas, así como de la definición de una capa independiente de la plataforma en la que se ejecute.

- Proporciona un alto nivel de abstracción, haciendo transparentes detalles como el manejo de eventos o distintos aspectos gráficos como el alineado.
- XPCE/Prolog integra una herramienta gráfica para el diseño de interfaces y diversos mecanismos y primitivas de alto nivel para agilizar su generación.
- Proporciona todos los elementos semánticos comunes a la gran mayoría de lenguajes orientados a objetos, esto es, clases, objetos, métodos, herencia, etc.
- Permite al programador la creación de nuevas clases y objetos con Prolog. Las clases se definen mediante Prolog y los métodos se ejecutan en Prolog, permitiendo una cómoda depuración de errores y agilizando la modificación del código y su recompilación.
- La interfaz entre XPCE y Prolog es muy reducida y de fácil comprensión y manejo.

Las alternativas a XPCE/Prolog a la hora de implementar la interfaz gráfica de usuario pasaban por conectar Prolog con otro lenguaje ajeno al paradigma declarativo, principalmente el lenguaje imperativo Java. Para ello existen distintas herramientas que proporcionan una interfaz entre los dos lenguajes como son JPL o B-Prolog. En cualquier caso, debido a sus características y a las del sistema desarrollado, resulta mucho más apropiado el uso de XPCE, que garantiza una mayor claridad en el código generado y una integración natural con Prolog.

Como muestra de la capacidad de creación de aplicaciones gráficas que XPCE posee están el propio editor gráfico para SWI-Prolog y su depurador gráfico, que, además, han resultado determinantes durante el desarrollo del sistema por su potencia e intuitividad.

5.2 Organización del sistema

5.2.1 Visión de alto nivel del sistema

El sistema desarrollado puede dividirse en tres partes lógicas de alto nivel:

- Interfaz gráfica de usuario
- Motor de deducción
- Persistencia de datos

La interfaz de usuario está enfocada al uso de ventanas y formularios gráficos que, en conjunto, componen un entorno amigable e intuitivo para el usuario, con el fin de facilitar la comprensión de los resultados de las demostraciones, así como de permitir una gestión sencilla de los parámetros y de los datos de entrada del sistema. Físicamente, su código fuente se compone de una serie de ficheros de texto con sintaxis Prolog y XPCE/Prolog, que se describen más minuciosamente en la sección **5.2.2** de esta documentación**.**

El motor de deducción es la componente que se encarga de todo el proceso lógico de demostración de los objetivos introducidos por el usuario, y determinado por los valores de los parámetros del sistema. Implementa el método de eliminación de modelos y una serie de

optimizaciones con respecto al modelo básico. Su soporte físico consta de varios ficheros de texto con sintaxis Prolog cuyos detalles se exponen en la sección **5.3.3**.

Por último, se tiene la parte del sistema compuesta por los mecanismos que hacen posible la persistencia de la parte más relevante de los resultados obtenidos tras las demostraciones, así como de los datos de entrada y de otros datos auxiliares necesarios para el correcto funcionamiento de la aplicación y que son transparentes para el usuario. El soporte de la persistencia del sistema es una serie de archivos de texto; estos ficheros y su sintaxis se describen más adelante, en la sección **5.2.2**.

Las conexiones entre las distintas componentes del sistema se reducen a unas pequeñas interfaces que se describen a continuación, distinguiéndolas según cuáles sean las componentes que comunican y la dirección.

Interfaz GUI – Motor lógico

Las conexiones entre la interfaz gráfica de usuario y la componente que se ocupa del proceso de deducción constan de un pequeño catálogo de predicados Prolog. Se distinguirá a continuación entre la interfaz de una dirección y la de la otra.

GUI → Motor lógico

A través de la interfaz gráfica, el usuario introduce los datos necesarios para el desarrollo de la demostración, configura los distintos parámetros del sistema e inicia y aborta el proceso deductivo cuando él decida. Esta información es proporcionada al motor lógico al producirse determinados eventos mediante los predicados siguientes:

- ***solve(+G,-P,-A,-PD)***: Siendo *G* el objetivo a demostrar, *P* la demostración producida, *A* la respuesta para la que el objetivo es cierto y *PD* el número de pasos y la profundidad que se han requerido para alcanzar dicha demostración. Este predicado inicia el proceso deductivo para objetivos en forma clausal. La GUI provee al motor de demostración del objetivo introducido por el usuario.
- ***solve2(+G,-P,-R,-PD)***: Siendo *G* el objetivo a demostrar, *P* la demostración producida, *R* la lista de reglas en forma clausal producto de la transformación de un objetivo en sintaxis de lógica de primer orden y *PD* el número de pasos y la profundidad que se han requerido para alcanzar la demostración. Este predicado inicia el proceso deductivo para objetivos en sintaxis de lógica de primer orden. La GUI provee al motor de demostración del objetivo introducido por el usuario.
- ***reset***: Elimina de la base de Prolog todas las reglas y lemas creados en tiempo de ejecución durante la demostración.
- ***reset_all***: Elimina todas las reglas y lemas de la base de Prolog.
- ***update_options***: Modifica los valores almacenados para los parámetros del sistema de acuerdo con lo indicado por el usuario.

Motor lógico → GUI

El motor deductivo devuelve los resultados de las demostraciones a la GUI y también proporciona la configuración actual de los parámetros del sistema para ser mostrados por pantalla al usuario. Aquí se enumeran los predicados Prolog que constituyen esta interfaz:

- ***show_string(+Str)*:** Muestra la cadena *Str* por pantalla.
- ***show_options_configuration(+VO)*:** Siendo *VO* una variable global que referencia al formulario gráfico mediante el cual el usuario puede configurar los parámetros del sistema así como comprobar su estado actual. Precisamente, mediante este predicado se consulta el valor actual de cada parámetro y se muestra convenientemente en el formulario. El formulario aludido, *me_options_dialog* se describe con detalle en la sección **5.3.2.**

Interfaz GUI – Persistencia

La interfaz entre estas dos componentes se ocupa de los mecanismos de carga y guardado de ficheros de distintos tipos por parte del usuario. Seguidamente se detallan los predicados que asumen estas tareas, todas con la misma dirección, obviamente:

GUI → Persistencia

- ***open_file( )***: Abre y lee el contenido de un fichero de extensión *.me* , que se muestra en el editor superior de la interfaz gráfica. Para facilitar al usuario la labor de búsqueda y apertura del fichero deseado aparece una ventana con un explorador de directorios y ficheros. El contenido del fichero debe seguir la sintaxis en forma clausal; el sistema no comprobará esto en el momento de la carga del fichero, por lo que no se avisa al usuario de posibles errores sintácticos.
- ***open_file_first_order( )***: Al igual que el predicado anterior, abre y lee el contenido de un fichero de extensión *.me* , que se muestra en el editor superior de la interfaz gráfica. Para facilitar al usuario la labor de búsqueda y apertura del fichero deseado, aparece una ventana con un explorador de directorios y ficheros. El contenido del fichero debe seguir la sintaxis en forma de lógica de primer orden; el sistema no comprobará esto en el momento de la carga del fichero, por lo que no se avisa al usuario de posibles errores sintácticos.
- ***save_file( )***: Abre un fichero de extensión *.me* y escribe el texto que muestra el editor superior de la interfaz gráfica en él. Si el fichero no existe, se crea automáticamente. Para facilitar al usuario la labor de búsqueda y apertura del fichero deseado, aparece una ventana con un explorador de directorios y ficheros.
- ***save_last_proof( )***: Abre un fichero de extensión *.txt* y escribe el texto con los resultados obtenidos para la última demostración realizada en él. De nuevo, si el fichero no existe, se crea automáticamente. Para facilitar al usuario la labor de búsqueda y apertura del fichero deseado, aparece una ventana con un explorador de directorios y ficheros.

Motor deductivo – Persistencia

Para su correcto funcionamiento, el motor deductivo del sistema requiere de la creación, lectura y escritura de los ficheros de extensión *.me* con las reglas a cargar y de ficheros con los lemas guardados asociados a éstos, así como de otros ficheros de carácter auxiliar. En este apartado se exponen los detalles de los predicados que realizan esta labor, todas con la misma dirección, obviamente:

Motor deductivo → Persistencia

- ***me_consult(+File)***: Lee los términos Prolog contenidos en el fichero de nombre *File* de uno en uno, hasta el final del mismo, insertándose las reglas que representan estos términos dentro de la base de reglas de Prolog.
- ***generate_lemmas_file(+FileName,+FileLemma)***: Al cargarse un fichero de reglas, se crea otro con el mismo nombre y con extensión *.lemma* en el mismo directorio, en caso de no existir ya. *FileName* es el nombre del fichero de reglas y *FileLemmas* el del fichero asociado en el que se guardarán los lemas de las demostraciones.
- ***load_lemmas(+FileLemma)***: Se leen y se insertan los lemas contenidos en el fichero *FileLemmas* dentro de la base de Prolog.
- ***insert_Lemma(+G,+P,+A,+D,+N)***: Escribe un lema en el fichero de lemas asociado al fichero de reglas actualmente cargado, siendo *G* el objetivo demostrado, *P* la demostración, *A* la respuesta y *D* y ***N*** la profundidad y número de pasos requeridos, respectivamente.

Los predicados que interactúan con la componente que se ocupa de la persistencia de datos explicados anteriormente, tanto los predicados pertenecientes al módulo de la interfaz como los del motor deductivo, se apoyan en los mismos predicados Prolog de más bajo nivel; éstos son los siguientes:

- ***working_directory(-Old, +New)***: Cambia el directorio de trabajo al indicado por *New*. *Old* es el antiguo directorio de trabajo. Usando *working_directory(Dir,Dir)* se puede consultar el actual directorio de trabajo sin modificarlo.
- ***chdir(+Dir)***: Cambia el directorio de trabajo a la ruta indicada por *Dir*.
- ***open(+FileName,+Mode,-Stream)****:* Predicado predefinido de Prolog para la apertura de ficheros. *FileName* es el nombre del fichero, *Mode* indica qué operación se hará con éste, pudiendo adoptar los valores *read*, *write* o *append*, y devuelve un stream correspondiente a ese fichero. El cursor del stream se sitúa en la posición 0.
- ***close(+Stream)****:* Cierra el fichero abierto asociado con el stream indicado.
- ***write(+Stream,+String)***: Escribe la cadena de caracteres en un stream dado asociado a un fichero. El fichero debe estar abierto en modo de escritura. La escritura sustituirá el contenido actual del fichero o se añadirá por el final, según cuál fuera el modo de apertura (*read* o *append*, respectivamente).

- ***read(+Stream,+String)***: Lee el término Prolog del stream dado, situado en la posición indicada por su cursor, asociado a un fichero que debe haber sido abierto en modo de lectura anteriormete. El cursor avanza una posición.
- ***get(@finder, +File, +Mode, +tuple('File Names', ext)***: Método de la clase finder definida en la librería *find_file* para apertura, lectura y escritura de ficheros**.** Hace aparecer una ventana con un explorador de directorios para que el usuario pueda encontrar y elegir cómodamente el fichero deseado. *File* es el nombre del fichero a tratar; *Mode* es la operación que se realizará con el fichero indicado, *open* para su apertura y *save* para su guardado; con *tuple*(*'File Names'*, *ext*) se limita el tipo de ficheros que mostrará el explorador a los de las extensiones aquí indicadas y, obviamente, a los directorios.
- ***send(+E,+ Mode,+FileName)***: *Mode* puede adoptar los valores *load* o *save*. En el primer caso, se carga el contenido del fichero de texto de nombre *FileName* en el objeto *E* de la clase *Editor* predefinida en la librería *pce*. En el segundo, se guarda el contenido del objeto *E* en el fichero indicado.

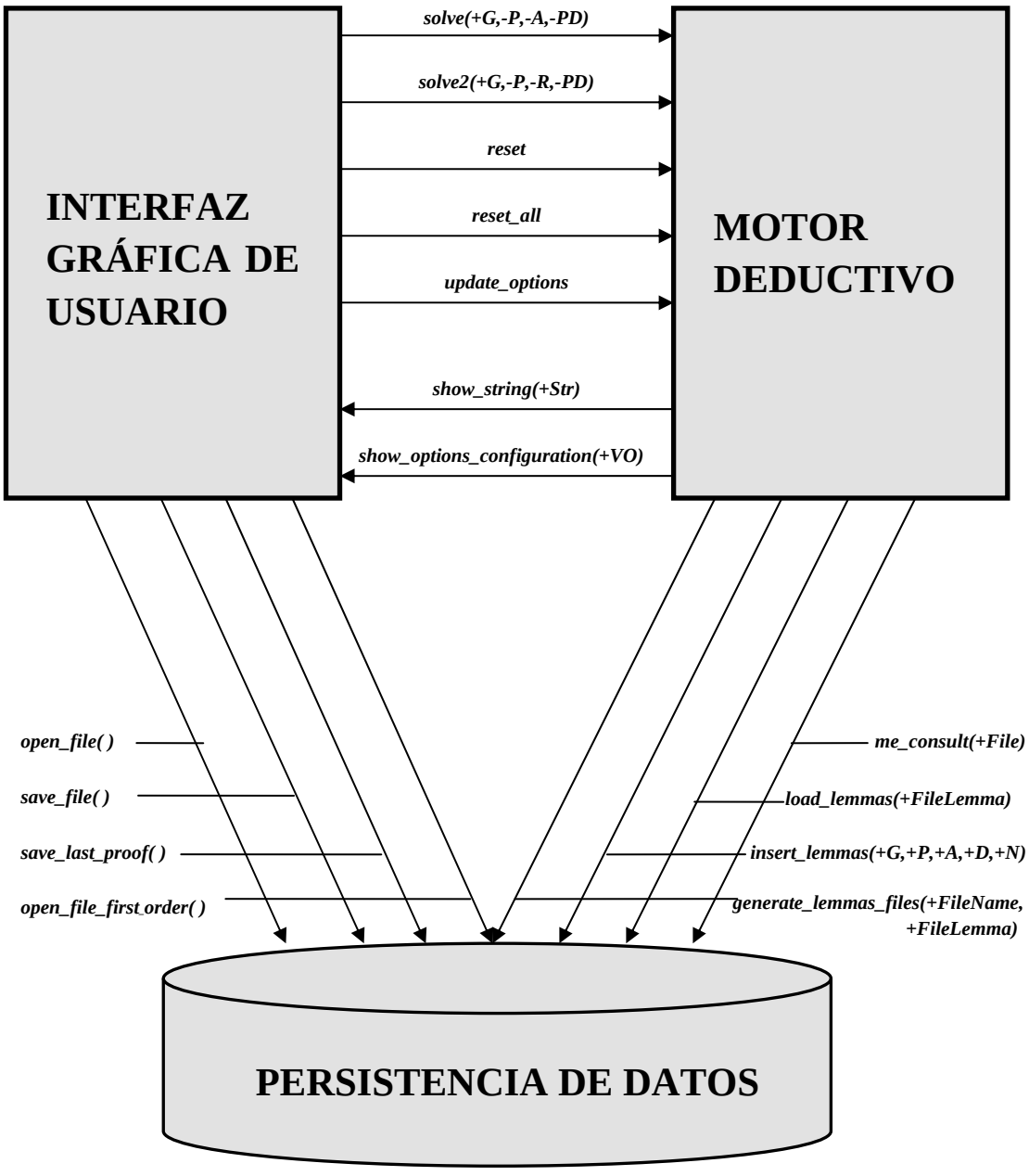


Fig. 1 Diagrama de los módulos básicos del sistema y las interfaces entre ellos

5.2.2 Estructura de los ficheros

Ficheros de código fuente

El sistema se compone de 10 ficheros de código fuente con la extensión **.pl* propia de los programas escritos en Prolog. El nombre de todos los ficheros comienza con el prefijo "me" por Model Elimination (Eliminación de Modelos), el método en el que se basa nuestro demostrador automático.

A continuación se hará una breve descripción de cada uno de los ficheros que componen el sistema:

- **meLoader.pl**: Se trata de un cargador que compila y ejecuta todos los ficheros del sistema tras lanzarlo sobre la consola de Prolog.

- **meFormMain.pl**: Contiene el código fuente de la interfaz gráfica principal del sistema, implementada con la librería XPCE

- **meFormOptions.pl**: Implementa el formulario de opciones en el que el usuario del sistema puede configurar los diferentes parámetros del método de búsqueda y/o activar o desactivar los diferentes mecanismos de optimización del método ME que permite el programa. Este formulario es accesible desde la ventana principal del sistema mediante *Herramientas → Opciones*.

- **meFormDirectories.pl**: Se trata de un formulario que permite al usuario modificar la ruta de su directorio de trabajo. En dicha ruta, el sistema generará distintos ficheros auxiliares que se explican con detalle en el siguiente apartado. Es accesible desde la ventana principal del sistema mediante *Archivo → Cambiar directorio de trabajo*.

- **meFormRules.pl**: Contiene el código del formulario que permite al usuario ver las reglas cargadas (activas) en el sistema. Permite al usuario activar/desactivar dichas reglas. Se accede a él desde *Herramientas → Reglas*.

- **meFormLemmas.pl**: Incluye el código del formulario que permite al usuario ver los lemas cargados en el sistema. Además permite que el usuario active/desactive dichos lemas. Se accede a él desde *Herramientas → Lemas*.

- **meSolve.pl**: Implementa el mecanismo de demostración de eliminación de modelos teniendo en cuenta distintos parámetros que rigen la búsqueda, y diferentes optimizaciones del método de eliminación de modelos que el usuario haya configurado en el formulario de opciones.

- **meLemmas.pl**: Incluye funciones auxiliares para el manejo de una de las optimizaciones del método de eliminación de modelos: lemas y lemas intermedios. Dichas funciones sirven para guardar lemas en archivos de lemas asociados a los archivos de las teorías (ver apartado Ficheros de E/S), calcular los parámetros que forman parte de la es-

tructura de lemas que guardamos en dichos ficheros y para cargar los lemas del fichero asociado en la base de Prolog.

- **meUtils.pl**: Contiene una colección de funciones auxiliares orientadas a la salida por pantalla de las demostraciones y las teorías en los editores de texto de la ventana principal del sistema, así como para distintos tratamientos de las reglas de los objetivos y de la teoría previos a las demostraciones.

- **meClauses.pl**: Implementa el código necesario para realizar la conversión de fórmulas generales de la lógica de primer orden a fórmulas en forma clausal para poder aplicar sobre ellas el método eliminación de modelos. Incluye una implementación del Algoritmo de Skolem usado como paso previo a la conversión de dichas fórmulas generales de la lógica de primer orden a forma clausal.

Ficheros auxiliares

Para su correcto funcionamiento, el sistema hace uso de una serie de ficheros auxiliares que, de no existir ya dentro del directorio de trabajo, se crearán en el momento en el que se les requiera. El directorio de trabajo es seleccionable por el usuario mediante la interfaz gráfica, como se explica en la sección **5.3.2.**; éste debe asegurarse de que posee derechos de escritura y lectura en el directorio elegido. La creación y la utilización de estos fichero auxiliares es, como es lógico, transparente para el usuario.

Los ficheros auxiliares del sistema se listan y se detallan a continuación:

- **primerOrdenAux.txt**: Fichero en el que se apoya el proceso de transformación de un conjunto de fórmulas de la lógica de primer orden en otro de reglas en forma clausal. Tras realizarse la transformación, las reglas en forma clausal obtenidas se escriben en este fichero auxiliar de texto, con el fin de facilitar la carga de este contenido en el editor superior de la interfaz gráfica de usuario.

  La aplicación no da al usuario la posibilidad de guardar en un fichero *.me* estas nuevas reglas resultantes de la transformación a forma clausal; en cualquier caso, el usuario puede abrir el fichero aquí explicado y hacer con su contenido lo que desee. El contenido del fichero será eliminado y sustituido, por un nuevo texto de la misma naturaleza, la próxima vez que se produzca una carga de un fichero *.me* con fórmulas de la lógica de primer orden, con su consecuente conversión a reglas de forma clausal.

- **salidaAux.txt**: Fichero auxiliar cuya razón de ser es la de facilitar la exhibición de la cadena de caracteres volcada a la salida estándar de Prolog cuando se use la línea de comandos de la GUI en el modo de emulador de consola de Prolog. Para ello se establece un stream asociado a este fichero como salida estándar. Al finalizar la operación, el contenido del fichero se carga en el editor inferior de la interfaz gráfica de usuario, y se reestablece la salida estándar a su estado habitual.

El usuario puede acceder al contenido del fichero y darle el uso que desee, pero esta información suele carecer de interés. El contenido del fichero será eliminado y sustituido, por un nuevo texto de la misma naturaleza, la próxima vez que se haga el mismo uso de la línea de comandos.

- **auxfile.txt**: Este fichero se usa de manera análoga a *primerOrdenAux.txt* pero durante el proceso de transformación de una fórmula de la lógica de primer orden introducida como objetivo a demostrar en la línea de comandos de la interfaz; debe estar activada la opción de demostración de objetivos en forma de lógica de primer orden.

  De nuevo, el usuario puede acceder a sus contenidos; no obstante, éstos tienen escaso valor informativo. Cada vez que se realice una demostración de este tipo se eliminará la información previa del fichero y se sustituirá por la nueva.

Ficheros de E/S

En este apartado nos referiremos al conjunto de ficheros que recibe y genera el sistema para consultas del usuario.

El usuario debe proporcionar al sistema ficheros que incluyan teorías, esto es, conjuntos de axiomas, escritos en forma clausal o en lógica de primer orden respetando la notación usada por el programa y que se explica con detalle en el Manual de Usuario que éste incluye. Dichos ficheros deberán tener la extensión **.me* (por *model elimination*). Para cargar dichos ficheros al sistema se ha de seleccionar *Archivo* → *Cargar Clausal* si todos los axiomas de la teoría se han escrito en formato clausal, o *Archivo* → *Cargar Primer Orden* si alguno de los axiomas de la teoría está escrito en formato de la lógica de primer orden, respetando la notación usada por el sistema. Al cargarse el fichero, el sistema lanza una ventana emergente que informa al usuario de que el fichero se ha cargado de manera satisfactoria.

Una vez que el usuario ha cargado un fichero con una teoría (sea File.me), dicha teoría se muestra en el editor de texto superior de la ventana principal y automáticamente el sistema generará un archivo con el mismo nombre pero con extensión **.lemma* (File.lemma) en el que se irán guardando los lemas que el usuario haya ido demostrando siempre que esté activada la opción de guardado de los lemas. Este nuevo fichero se creará en el mismo directorio en el que se encontraba el fichero de la teoría.

Cuando el usuario quiera cargar un fichero con una teoría que ya tenga asociado un fichero de lemas, ambos ficheros deberán estar en el mismo directorio para que el sistema cargue automáticamente los lemas guardados en éste último.

El usuario podrá consultar el fichero de lemas File.lemma, y observará que se guardan en una estructura del siguiente estilo:

lema(G,P,A,D,S): *G* es el lema demostrado por el usuario; *P* es la demostración del lema G, se guarda siguiendo una cierta estructura que indica en cada paso de demostración la técnica de deducción que se ha utilizado; *A* es la lista finita de términos que cumplen el lema *G*, que

puede ser vacía; *D* indica la profundidad de búsqueda en la que se obtuvo la demostración *P*; *S* , por su parte, indica el número de pasos de la demostración *P*.

Los parámetros *D* y *S* pueden variar dependiendo de la configuración de opciones y activación de técnicas de demostración que haya hecho el usuario. Por ello, es factible que para un mismo lema, el sistema devuelva una demostración con menor número de pasos u obtenida con menor profundidad que otra demostración del mismo lema realizada con anterioridad. En dicho caso, el sistema usa la política de guardar la demostración obtenida a menor profundidad y/o con menor número de pasos. Se recuerda que el guardado de los lemas es una opción configurable por el usuario. En la sección **6**, en la que se evalúan los resultados para distintos casos de prueba, se presentarán diferentes ejemplos en los que se aprecia el rendimiento del sistema y el valor de los parámetros *D* y *S* en dependencia de las opciones activadas por el usuario.

Una vez cargado un fichero **.me*, el programa permite al usuario poder modificar su teoría en el editor de texto superior ubicado en la ventana principal, y guardar dichos cambios en un fichero de texto (de extensión **.me*). Para ello el usuario elegirá *Archivo* → *Guardar .me*.

Las demostraciones de los objetivos aparecen en el editor de texto inferior de la ventana principal. El usuario tiene la posibilidad de guardar la información obtenida en dicho editor para la última demostración realizada seleccionando *Archivo* → *Guardar última prueba*. El fichero se guardará con la extensión **.txt*.

5.3 Implementación del sistema

5.3.1 Parametrización del sistema. Opciones.

Unos de los principales objetivos durante el diseño de la aplicación fue el de desarrollar una herramienta flexible y altamente configurable, dando la capacidad de decisión al usuario en la mayor cantidad de elementos posible. En lo referente a las demostraciones, se han incluido una serie de parámetros, los cuales determinaran los resultados de éstas, así como la información mostrada por pantalla. Esto permite al usuario probar distintas combinaciones de valores para estos parámetros, comparar resultados, buscar configuraciones óptimas y, de este modo y mediante la información devuelta por el sistema, poder sacar conclusiones propias que, de haber enfocado la aplicación desde una óptica menos exigente para el usuario, nunca podría llegar a obtener.

No obstante, pese a esta motivación, no se ha sobrecargado el sistema con opciones de carácter más secundario que ya podrían crear confusión e, incluso, abrumar al usuario. Buscando este equilibrio se tomó la decisión de dejar a rigor del usuario una serie de opciones, que se pueden dividir en dos clases, y que se exponen con detalle a continuación.

Parámetros de la demostración

Son los parámetros que influyen en el proceso lógico de demostración de un objetivo. Dentro de estos parámetros se pueden distinguir, también, tres tipos diferenciados, los que se refieren a las optimizaciones del sistema, las que repercuten en el mecanismo de búsqueda en profundidad limitada y, por último, otras opciones que afectan al proceso de demostración.

Los resultados producidos pueden llegar a ser drásticamente diferentes dependiendo de la configuración establecida, y no es difícil encontrar ejemplos para los que con una combinación de valores encuentra rápidamente la demostración, mientras que con otro se obtiene un rendimiento notablemente peor o, incluso, no se llega a concluir la demostración nunca.

Dentro del primer grupo, opciones que afectan al uso de las optimizaciones del motor de deducción (en la sección **5.3.3.** se explican pormenorizadamente estas optimizaciones), se encuentran las siguientes:

- **Uso de reglas del objetivo**: Si esta opción está activada, al iniciar una demostración se introducirán en la base de Prolog una serie de reglas generadas a partir del objetivo a demostrar.
- **Uso de ancestros complementarios**: Si esta opción está activada, para cada subobjetivo expandido, se comprobará si existe algún ancestro suyo complementario, esto es, su negación.
- **Uso de ancestros idénticos**: Si esta opción está activada, para cada subobjetivo expandido, se comprobará si existe algún ancestro suyo idéntico a él.
- **Uso de lemas persistentes**: Indica si se usarán o no los lemas persistentes durante la demostración.
- **Guardado de lemas persistentes**: Indica si se da la posibilidad al usuario de guardar los lemas de tipo persistente generados tras una demostración.
- **Auto-guardado de lemas persistentes:** Determina si el guardado de lemas persistentes se produce automáticamente, sin pedir confirmación al usuario. Para ello, debe estar activada la opción de guardado de lemas.
- **Uso de lemas intermedios:** Indica si se generarán y se usarán o no los lemas intermedios durante la demostración.

En cuanto a los parámetros que influyen en el mecanismo de búsqueda en profundidad escalonada usado para las demostraciones (en la sección **5.3.3.** se describen con detalle las características y la implementación del mecanismo de búsqueda), se permite al usuario la elección de los siguientes valores:

- **Factor de pasos:** Este parámetro determina el número máximo de pasos que el demostrador utilizará para cada profundidad: *nº de pasos = profundidad * factor*. Puede adoptar cualquier valor numérico natural.

- **Profundidad límite:** El valor de este parámetro limita la profundidad que se puede alcanzar como máximo el mecanismo de búsqueda escalonada, se podría decir que indica cuál es el último escalón. Si se alcanza esta cota, la demostración finaliza, dándose por fallida. Puede adoptar cualquier valor numérico natural. El valor 0 implica una búsqueda sin límite de profundidad; no obstante, este mismo efecto se puede obtener en la práctica otorgando valores altos a este parámetro, ya que el espacio de búsqueda aumentará enormemente en cada nivel de profundidad para objetivos que no sean extremadamente sencillos.

Finalmente, se exponen los parámetros que, sin ser de ninguno de los grupos diferenciados anteriormente, influyen de algún modo en el proceso deductivo:

- **Uso de reducción**: Esta opción indica si estará habilitados o no el uso de pasos de reducción.
- **Unificación con occur-check**: Si esta opción está activada, las unificaciones que se deriven del proceso de demostración de un objetivo se realizarán con occur check.
- **Tipo de objetivo:** El usuario debe indicar, antes de proceder a su demostración, qué tipo de objetivo ha introducido en la línea de comandos, es decir, si esta expresado en forma clausal o es una fórmula de la lógica de primer orden (siempre siguiendo las sintaxis correctas).

Parámetros de la salida

Permiten al usuario del sistema elegir qué resultados se le mostrarán por pantalla durante y tras una demostración. Estas opciones sólo tienen un efecto superficial, pues, sean cuales sean sus valores, la aplicación seguirá generando internamente los mismos resultados, independientemente de que se muestren o no.

- **Mostrar traza**: Si esta opción está activada, durante la demostración de un objetivo se generará y se mostrará una traza informativa, que indica la profundidad, el tiempo y el número de pasos usado por el demostrador para esa profundidad.
- **Mostrar demostración**: Si esta opción está activada, al finalizar la demostración de un objetivo se mostrará la información de las inferencias realizadas para concluir dicha demostración.

El usuario puede gestionar externamente estos parámetros a través de la interfaz gráfica, de una manera intuitiva y ágil. Internamente, la manera en que el sistema almacena y modifica estos valores es mediante variables globales de Prolog y las operaciones asociadas a estos, como son *nb_getval* y *nb_setval.*

5.3.2 Interfaz gráfica de usuario

Elementos de diseño

Como ya se ha venido contando, algunas de las principales motivaciones que determinaron el diseño y el desarrollo de este sistema fueron las de conseguir una aplicación flexible y

con un entorno intuitivo y amigable para el usuario. Estos objetivos han sido los que decidieron los distintos elementos de diseño de la interfaz gráfica de usuario, para los cuales se puede establecer varias categorías bien diferenciadas:

- Los que componen el entorno principal de la aplicación, esto es, la línea de comandos de entrada y los editores de texto que componen la salida del sistema, en los que se exponen los resultados que se van obteniendo.
- Los relacionados con el manejo de ficheros, esto es, apertura, carga y guardado.
- Los relacionados con la configuración de los parámetros del sistema.
- Los relacionados con la gestión de las reglas y los lemas a usar.
- Los relacionados con los contenidos de ayuda y otras informaciones.

La ventana principal de la aplicación se compone de los siguientes elementos y funcionalidades:

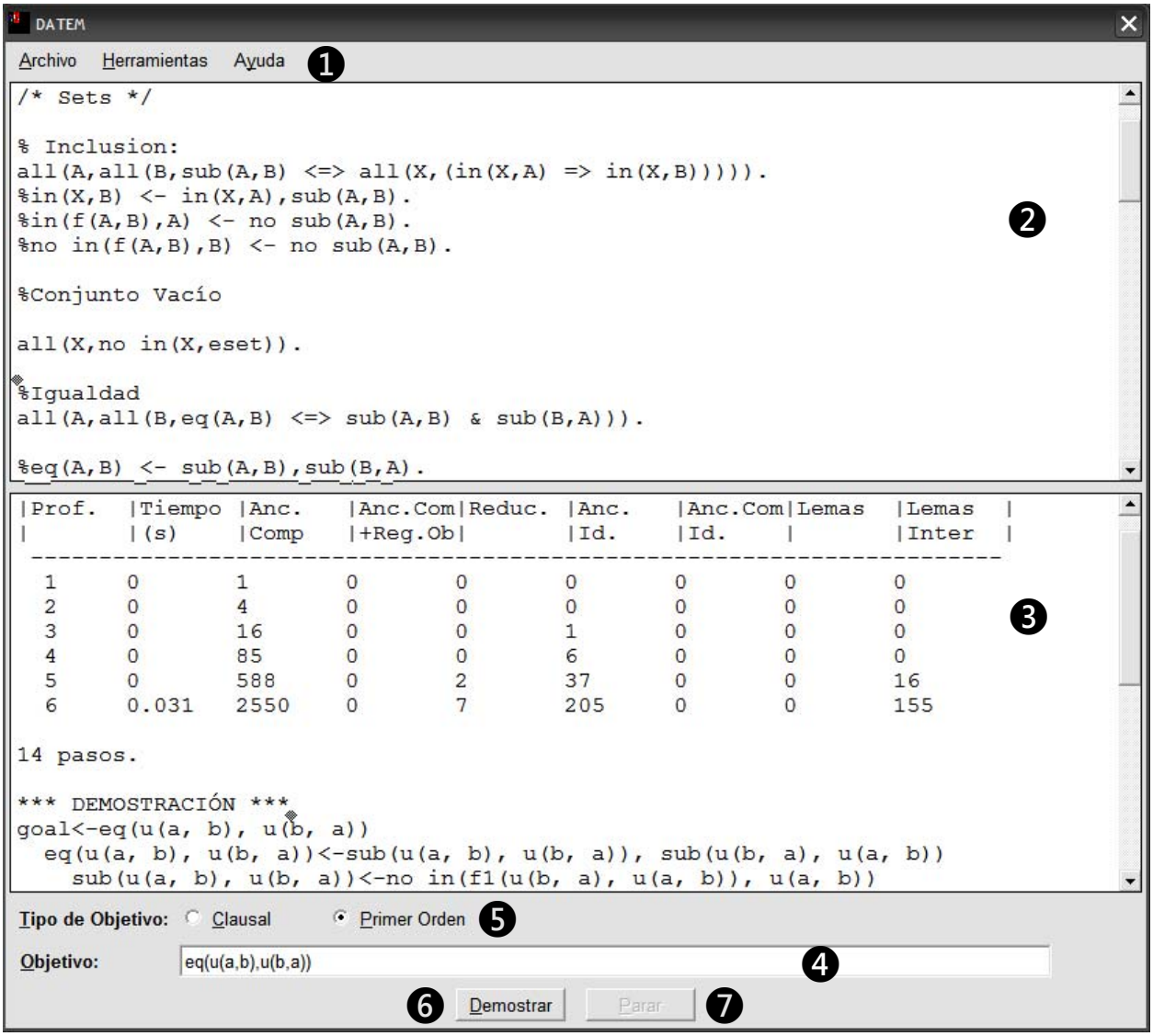


Fig. 2 Ventana principal de la interfaz gráfica de usuario

❶ Menú principal del sistema: Mediante él, el usuario puede acceder a las distintas herramientas y funcionalidades de la aplicación. Se compone de tres submenús: el primero agrupa las funciones asociadas con el manejo de ficheros; el segundo da acceso a las herramientas de gestión de reglas y lemas, así como a la permite la gestión de los parámetros del sistema; en tercer lugar está el submenú por el que se puede llegar a la ayuda y a la información básica de la aplicación.

❷ Editor de texto superior: En él se muestra el contenido del último fichero cargado. Tanto si éste fue cargado como fichero con reglas en forma clausal o como fichero con fórmulas de la lógica de primer orden, se muestra su contenido sin ninguna modificación. El texto de este elemento gráfico es editable, pudiéndose, además, volcar a un fichero, como se explicará más adelante.

❸ Editor de texto inferior: En él se muestran los resultados que se van generando durante una demostración o bien puede funcionar como salida estándar de Prolog cuando se está utilizando la línea de comandos a modo de consola de Prolog.

❹ Línea de comandos: Campo de texto en el que el usuario debe introducir el objetivo a demostrar o bien un predicado Prolog precedido por el carácter '/'.

❺ Menú de selección del tipo de objetivo a demostrar: El usuario debe indicar la naturaleza del objetivo que va a demostrar, esto es, si está en formal clausal o es una fórmula de la lógica de primer orden. Esto es independiente del tipo de fichero cargado.

❻ Botón de inicio de demostración: Al presionarlo, se da comienzo al proceso de demostración. En el caso de que el texto de la línea de comandos comience con el carácter '/', se interpretará el resto como un predicado Prolog, funcionando como la consola de Prolog. El botón permanecerá bloqueado desde el comienzo hasta el final de una demostración. Al finalizar una demostración, aparecerán una serie de mensajes informativos y el usuario podrá elegir entre concluir la búsqueda o buscar más soluciones y también puede decidir si se guardan los lemas derivados de la demostración para el objetivo probado; para esto último debe estar activada la opción de guardado de lemas y desactivada la de autoguardado de lemas.

❼ Botón de aborto de demostración: Al presionarlo, se interrumpe abruptamente el proceso de demostración activo. El botón sólo permanecerá desbloqueado mientras haya un proceso de demostración activo.

A las funcionalidades de apertura, carga y guardado de fichero se accede mediante los elementos del primer submenú del menú principal, llamado *Archivo*.

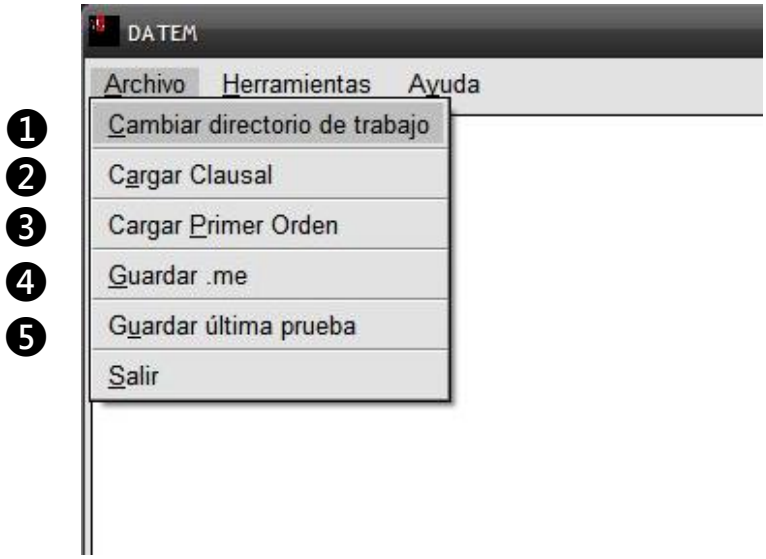


Fig. 3 Submenú *Archivo*

❶ Cambio del directorio de trabajo: Se muestra un explorador para seleccionar el directorio sobre el que se crearán los ficheros temporales necesarios para el correcto funcionamiento del programa. El usuario debe asegurarse de tener los privilegios suficientes para crear, leer y modificar ficheros en el directorio indicado.

❷ Cargar fichero con reglas en forma clausal: Se muestra un explorador (predefinido para el sistema operativo sobre el que se esté ejecutando la aplicación) para abrir un fichero *.me* con texto en forma clausal. El contenido se mostrará automáticamente en el editor de texto superior de la interfaz principal. De no ser así, el usuario deberá comprobar que el fichero cumple las reglas léxicas y sintácticas de la forma clausal.

❸ Cargar fichero con fórmulas de la lógica de primer orden: Se muestra un explorador (predefinido para el sistema operativo sobre el que se esté ejecutando la aplicación) para abrir un fichero *.me* con texto en forma de primer orden. El contenido se mostrará automáticamente en el editor de texto superior de la interfaz principal. De no ser así, el usuario deberá comprobar que el fichero cumple las reglas léxicas y sintácticas de la lógica de primer orden.

❹ Guardar fichero *.me*: Se muestra un explorador (predefinido para el sistema operativo sobre el que se esté ejecutando la aplicación) para guardar el contenido actual del editor superior de la interfaz principal en el fichero con extensión *.me* que se indique. El contenido se guarda directamente, sin realizar ninguna comprobación de su corrección léxica ni sintáctica.

❺ Guardar última prueba: Se muestra un explorador (predefinido para el sistema operativo sobre el que se esté ejecutando la aplicación) para guardar el texto de salida generado en la última demostración; este texto puede contener la traza de la demostración, la prueba, ambas o ninguna, según estén o no activadas las opciones correspondientes.

Haciendo uso del segundo submenú del menú principal, llamado *Herramientas*, el usuario puede acceder a las funcionalidades relacionadas con la configuración de los parámetros del sistema, sí como a las de gestión de las reglas activas y los lemas activos, esto es, que se utilizarán durante las demostraciones.

Fig. 4 Submenú *Herramientas*

❶ Herramienta para la selección de reglas: Esta herramienta permite al usuario activar y desactivar, de una en una o todas a la vez, las reglas producidas en la carga del fichero *.me* actual, que se muestran en dos listas con elementos seleccionables.

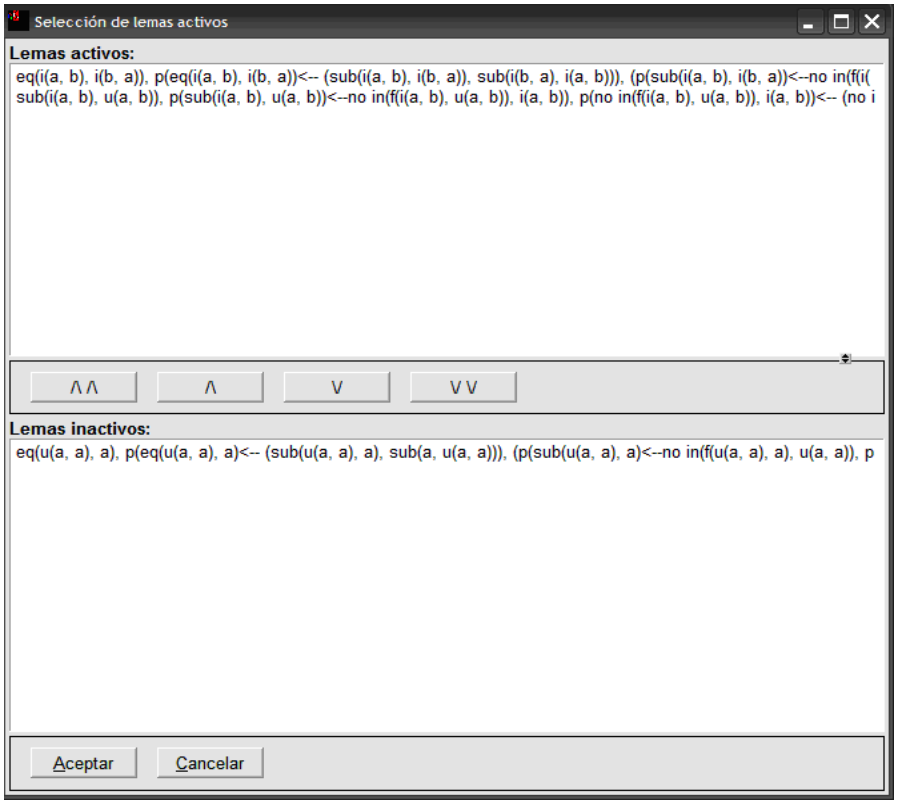


Fig. 5 Ventana de selección de reglas

❷ Herramienta para la selección de lemas: Esta herramienta permite al usuario activar y desactivar, de uno en uno o todos a la vez, los lemas persistentes producidos en las demostraciones relativas al fichero *.me* actual, que se muestran en dos listas con elementos seleccionables.

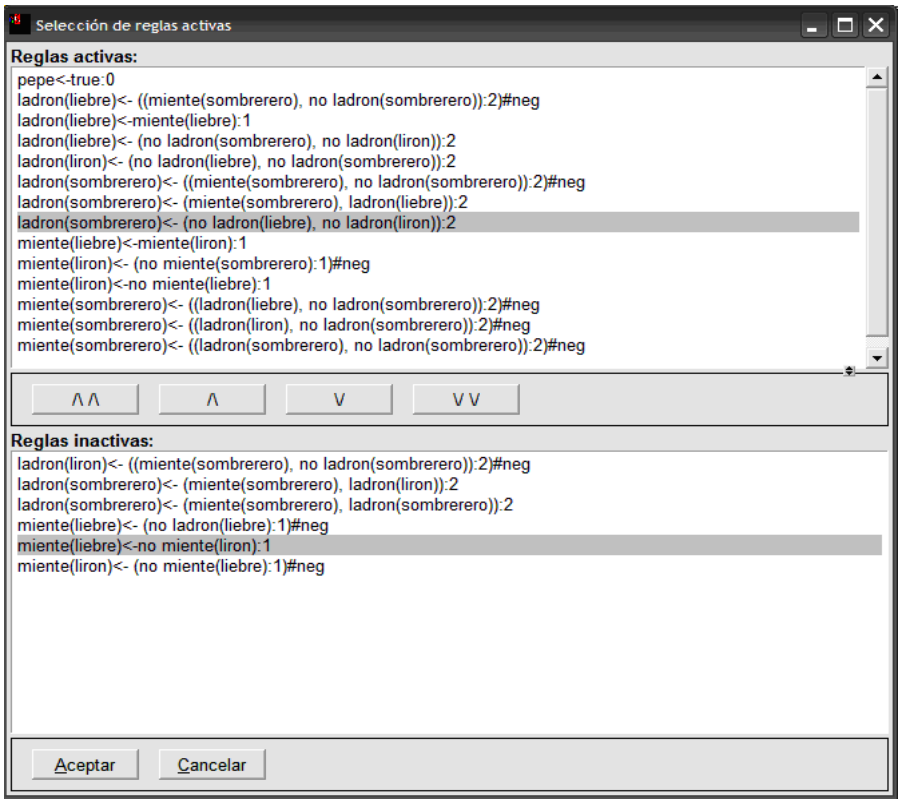


Fig. 6 Ventana de selección de lemas

❸ Formulario para la gestión de los parámetros del sistema: Permite al usuario configurar los parámetros del sistema de una manera ágil, rápida e intuitiva.

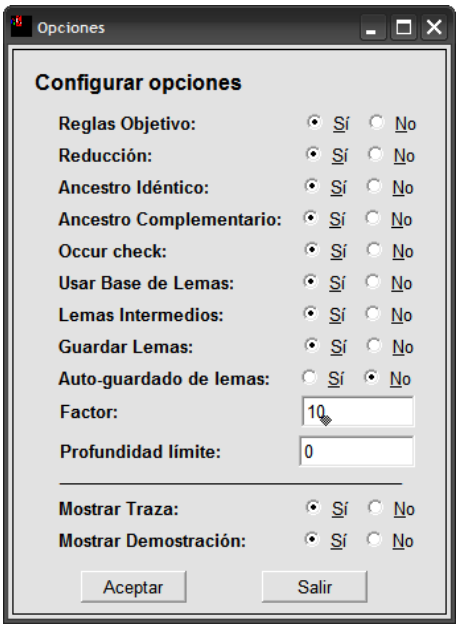


Fig. 7 Ventana de configuración de parámetros del sistema

Finalmente, mediante el tercer submenú, el de nombre *Ayuda*, se tiene acceso a los contenidos de la ayuda y a la información básica del sistema.

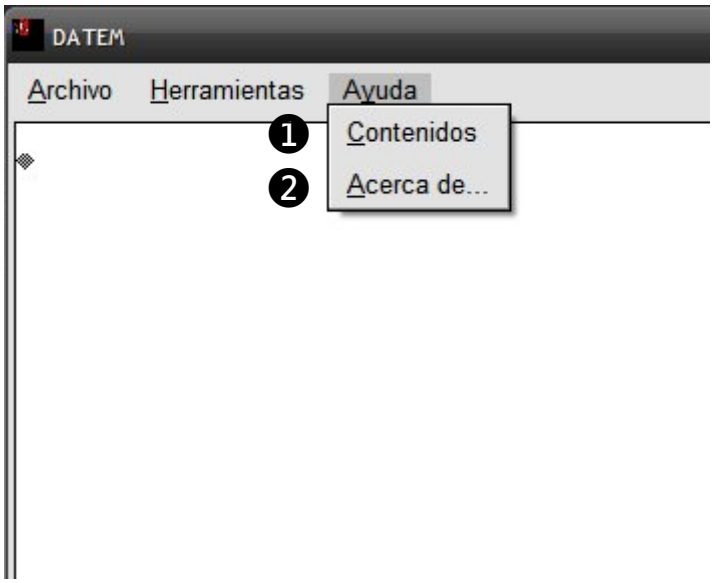


Fig. 8 Submenú *Ayuda*

❶ Contenidos de la ayuda: Se accede a una serie de documentos *.html* navegables que pretenden exponer brevemente las principales características del sistema, resolver las principales dudas que al usuario le puedan surgir a la usar de utilizar esta aplicación y que éste consiga rápidamente una comprensión sobre el manejo de las distintas herramientas y fundamentos teóricos básicos de los que hace uso el sistema.

❷ Información básica del sistema: Se muestra información sobre el personal desarrollador de la aplicación, sobre la coordinación de su proceso de implementación y sobre la versión del sistema.

Implementación

Para desarrollar la interfaz gráfica descrita previamente se ha elegido la herramienta XPCE/Prolog, cuyas principales características se explican en la sección **5.1.3**. Respecto a la implementación de la GUI mediante XPCE, en primer lugar se distinguirá entre los elementos visuales y los elementos funcionales. En cuanto a los primeros, nos limitaremos a realizar una breve descripción de su estructura, centrándonos más en la implementación de las funciones que ofrece la interfaz y en los métodos usados para ello.

El elemento visual de mayor importancia de la interfaz es la ventana principal. Para su implementación se ha creado una nueva clase XPCE llamada *me_interface* , que hereda de la clase *frame*, predefinida para XPCE. En ella se añade un objeto de la clase *tool_kit*, en la cual, a su vez, se añaden los distintos elementos de la ventana, que, como se explicó anteriormente, son menús, editores de texto, botones y un campo de texto; para ello se crean instancias de las clases predefinidas de XPCE *popop*, *menu*, *editor*, *text_item* y *button*.

Para desarrollar las herramientas gráficas de gestión de reglas y lemas activos, ambas equivalentes desde el punto de vista visual, no se requirió la creación de una nueva clase, utilizándose la ya mencionada clase *frame*, en la que se añaden dos listas de texto (clase *browser*) y una serie de botones (clase *button*).

Sin embargo, para la creación del formulario para la configuración de los parámetros del sistema, debido a la cantidad de elementos visuales a incluir, se decidió utilizar la herramienta de XPCE para la generación y edición de ventanas (XPCE dialog editor), en su versión 0.8. Gracias a esta utilidad, la creación del formulario mencionado resultó rápida y sencilla, pero se pudo comprobar que no resultaba tan operativa a la hora de desarrollar los distintos elementos visuales descritos anteriormente, debido a la extensión y otras características de las plantillas de código generado por el editor.

El formulario al que nos referimos consta de una serie de menús de selección (objetos de la clase *menu*), campos de texto (*text_item*), etiquetas de texto (*label*) y botones (*button*), todos ellos incluidos en un contenedor de la clase *dialog*.

En cuanto a la implementación de las funcionalidades de las que está dotada la interfaz, se destacan las siguientes:

- Lanzamiento de demostraciones e interrupción de demostraciones. Hilos Swi-Prolog
- Salida de datos
- Gestión de reglas y lemas activos
- Gestión de parámetros
- Gestión de ficheros
- Inicio de la interfaz gráfica de usuario

Lanzamiento de demostraciones e interrupción de demostraciones. Hilos Swi-Prolog

El mecanismo de inicio de demostraciones es la funcionalidad esencial de la interfaz, perfectamente complementado con el de interrupción del proceso de deductivo por parte del usuario. En primer lugar, se distingue entre los casos en que el objetivo, introducido por el usuario en la línea de comandos, comienza por el carácter '/' y cuando no. De darse esta situación, la cadena de caracteres restante (sin el '/') es interpretado como un predicado Prolog a comprobar, actuando el sistema como la consola de Prolog; en el otro caso, se tratará de iniciar la demostración. Antes de ello se realiza una comprobación previa para determinar si el objetivo introducido es un término Prolog correcto; de no ser así, el usuario es aviso convenientemente mediante una ventana informativa. De cumplirse estas restricciones, el objetivo a demostrar se captura a través de la línea de comandos.

Para iniciar la demostración, en el caso de ser un objetivo en forma clausal, se llama al predicado *solve*/4; de tratarse de una fórmula de la lógica de primer orden, el proceso de demostración se lanzaría mediante el predicado *solve2*/4. En ambos casos, el objetivo captura-

do se pasa como argumento del predicado. El funcionamiento interno de éstos, que ya no pertenecen al ámbito de la interfaz, se explica pormenorizadamente en la sección **5.3.3** de esta documentación.

Por lo visto hasta ahora, el mecanismo de inicio de un proceso de demostración no implica ninguna complicación reseñable, pero si se tiene en cuenta que en multitud de ocasiones, ya sea porque la demostración requiere demasiado tiempo o porque, simplemente, el objetivo no es demostrable, se hacía esencial la inclusión de un mecanismo por el cual el usuario pudiera abortar el proceso deductivo y retomar el control de la aplicación. Solucionar esta problemática no es en absoluto trivial y requirió el uso de hilos.

SWI-Prolog, en sus últimas versiones, integra una serie de predicados para la gestión de hilos, de los que se han utilizados los mencionados seguidamente:

- ***thread_create(:Goal, -Id, +Options)*:** Crea un nuevo hilo en el que se ejecuta el objetivo *Goal*. Si se crea con éxito, *Id* devuelve el identificador del hilo. En ningún caso se ha hecho uso de opción alguna a la hora de crear hilos en este sistema. Si el lector desea conocer más información acerca de este predicado, puede consultar la ayuda de SWI-Prolog.
- ***in_pce_thread(:Goal)*:** Asumiendo que XPCE se está ejecutando en el hilo principal, mediante este predicado, los hilos secundarios pueden hacer llamadas al hilo de XPCE. Esto es muy útil, por tanto, para aplicaciones con una importante componente gráfica como es la que nos ocupa.
- ***thread_signal(+ThreadId, :Goal)*:** Fuerza al hilo cuyo identificador es *ThreadId* a ejecutar el objetivo *Goal*. De nuevo, para una información más exhaustiva, se puede consultar la ayuda de SWI-Prolog.
- ***throw(+Exception)*:** Lanza una excepción de tipo *Exception*; conlleva el efecto lateral de la finalización inmediata del hilo sobre el que se ejecuta. Pese a que este no es un predicado, aparentemente, muy relacionado con el manejo de hilos, esta es la manera recomendada en la ayuda de SWI-Prolog para finalizar un hilo explícitamente, desaconsejando el uso del predicado *thread_join*/2.

Una vez conocidos estos conceptos, se pueden explicar con precisión la forma en que se inician y finalizan los procesos de demostración. Primeramente, se lanza un hilo en el que se ejecuta un predicado llamado *solve_thread*/4:

- ***solve_thread(+G,+S,+FN,+Sel)***: *G* es el objetivo que se pretende demostrar, *S* es la lista de parámetros del sistema junto con sus valores actuales, *FN* es el nombre del fichero cargado actualmente; *Sel* indica el tipo de objetivo (clausal o lógica de primer orden). Mediante este predicado se llama a *solve* (o *solve2*, según cuál sea el tipo del objetivo). Hay que tener en cuenta un factor adicional, y es que el nuevo hilo no tendrá accesibles los valores de las variables globales del hilo padre (ni de ningún otro hilo), por lo que se hace un tratamiento explícito de los valores de los parámetros del sistema, pasándolos como argumento.

En el caso de que la demostración finalice con éxito, el hilo creado muere automáticamente, sin necesidad de realizar ninguna operación de comprobación.

Mientras una demostración está activa, si el usuario lo cree conveniente, puede, presionando el botón correspondiente, desencadenar la interrupción inmediata del proceso, quedando nuevamente la interfaz a su plena disposición para realizar nuevas demostraciones. El encargado de esta labor es el predicado *abort_thread*/0:

- ***abort_thread***: Mediante *thread_signal*/2 y *throw*/1 se lanza una excepción en el hilo en el cual se está ejecutando la demostración. Como resultado, el hilo muere, interrumpiéndose de este modo el proceso deductivo. El usuario es informado convenientemente, y la interfaz vuelve a permitir realizar nuevas demostraciones.

Salida de datos

Durante una demostraciones van produciendo una serie de resultados; concretamente, los datos que forman la traza que informa acerca de las profundidades, número de pasos, tiempos, etc., se generan a lo largo de todo el proceso, sin embargo, los resultados relativos a las inferencias realizadas para obtener la demostración final tan sólo se muestran explícitamente al finalizar todo el proceso deductivo. Por lo tanto, se ha necesitado de una comunicación continua entre el hilo que ejecuta la demostración y el hilo en el que se está ejecutando la interfaz, de modo que se vayan haciendo llegar en cada momento los resultados que se generan a la GUI, para ser mostrados al usuario inmediatamente.

Un problema que surgió fue que, si se ejecutaba *in_pce_thread*/1 todas las veces que se generaba un resultado, el uso intensivo que se hacía de la escritura en el editor de la interfaz, hacía que ésta permaneciera bloqueada continuamente para el usuario. Es por ello que se usa un buffer en el que se van almacenando estos resultados, siendo añadidos al contenido del editor cada un cierto tiempo.

Los predicados que realizan estas acciones son éstos:

- ***show_string(+Str)***: Añade la cadena de caracteres *Str* al buffer de salida. Cada un cierto número de veces que se ejecuta este predicado, se realiza el volcado del contenido del buffer al editor de texto de la interfaz.
- ***show_aux(+Str)***: Añade la cadena de caracteres *Str* al contenido actual del editor de texto.
- ***show_final***: Añade el contenido del buffer aun no volcado al editor. Se usa exclusivamente cuando la demostración finaliza, ya sea con o sin éxito, para mostrar el contenido restante del buffer.
- ***clear_output***: Borra el contenido del editor de texto inferior.

Además de la información explicitada en el editor de salida, la interfaz gráfica ofrece otro tipo de informaciones y requerimientos al usuario al finalizar satisfactoriamente una demostración. En primer lugar, se muestra una ventana emergente en la que se informa de que se ha encontrado una demostración exitosa y dando la posibilidad al usuario de finalizar defini-

tivamente, o bien seguir buscando otras posibles demostraciones. Cuando finalmente se opte por no buscar nuevas soluciones, y en el caso de estar activada la opción de guardado de lemas persistentes y desactivado su guardado automático.

Gestión de parámetros

Como ya se dicho, el usuario puede consultar el estado actual de los parámetros del sistema y modificar sus valores mediante un formulario gráfico accesible desde el menú principal. Al abrirse el formulario, los valores actuales de los parámetros son consultados y los elementos visuales de la ventana se modifican de modo que el usuario pueda conocer con un simple vistazo la configuración actual. Estas acciones son llevadas a cabo por los predicados *show_options_dialog*/0 y *show_options_configuration*/1:

- ***show_options_dialog***: Muestra por pantalla el objeto de la clase *dialog* que implementa el formulario de configuración de los parámetros del sistema
- ***show_options_configuration(+Dialog)***: Modifica el aspecto de los elementos visuales del formulario *Dialog* de modo que se corresponda con los valores actuales de los parámetros del sistema.

Una vez que el usuario ha modificado los valores de los parámetros según su criterio, puede abandonar el formulario aceptando o cancelando los cambios. En el caso de cancelarlos, los valores permanecerán igual que cuando se abrió el formulario. Si se acepta, el predicado *update_options*/0 se ocupa de guardar los nuevos valores:

- ***update_options***: Almacena los nuevos valores de los parámetros del sistema en las variables globales correspondientes.

Gestión de reglas y lemas activos

Para la implementación de las herramientas de gestión de reglas y de lemas activos, que permite al usuario elegir qué reglas y lemas se usarán en las demostraciones sucesivas, se ha seguido un patrón totalmente análogo, por lo que se detallarán los elementos de sólo una de ellas. En concreto, para el desarrollo de la herramienta para la elección de reglas activas, el tratamiento intermedio de las reglas se hace mediante listas de cadenas de caracteres y mediante la inserción y eliminación de reglas dentro de la base de Prolog.

Al lanzarse la ventana, accediendo a ella a través del menú principal, se cargan las actuales reglas activas (*rule*/2) e inactivas (*inactive_rule*/2) convenientemente, para ser mostradas al usuario. Se encargan de ello los predicados *rule_management_dialog*/0 y *init_rules*/0:

- ***rule_management_dialog***: Muestra la ventana para la elección de reglas activas e inactivas.
- ***init_rules***: Accede a la base de reglas de Prolog para capturar todas las reglas activas y las inactivas que contiene y modifica los elementos visuales de la ventana convenientemente. Para la gestión posterior de las reglas, se crean tres listas de cadenas de caracteres, una relativa a las reglas activas (*active_rule_list*), otra a las reglas inactivas (*inacti-*

ve_rule_list) y otra para ambas (*rule_list*), que servirán para recuperar el estado inicial en caso de que se requiera. Además, se crea una copia de cada una de las dos primeras listas anteriores (*act_active_rule_list*, *act_inactive_rule_list*), que serán las que contengan el estado actual, con las modificaciones realizadas por el usuario.

Mediante la selección de elementos de las listas gráficas y la activación de los diferentes botones de la ventana, se ejecutan los distintos predicados que permiten la activación o desactivación de reglas; éstos son *activate_rule*/2, *disactivate_rule*/2, *activate_all_rules*/2 y *disactivate_all_rules*/2:

- ***activate_rule(+Browser1,+Browser2)***: *Browser1*y *Browser2* son objetos de la clase *browser* de XPCE, y son las listas de texto de la ventana gráfica, la primera muestra las reglas activas y la segunda las inactivas. El predicado elimina de la base de Prolog la regla inactiva asociada al elemento seleccionado de *Browser2* y lo añade como regla activa, actualizando convenientemente el contenido de los elementos visuales.
- ***disactivate_rule(+Browser1,+Browser2)***: De nuevo, *Browser1*y *Browser2* son objetos de la clase *browser* de XPCE, y son las listas de texto de la ventana gráfica, la primera muestra las reglas activas y la segunda las inactivas. El predicado elimina de la base de Prolog la regla activa asociada al elemento seleccionado de *Browser1* y lo añade como regla inactiva, actualizando convenientemente el contenido de los elementos visuales.
- ***activate_all_rule(+Browser1,+Browser2)***: Realiza la misma operación que *activate_rule*/2, pero no sólo con un posible elemento seleccionado de *Browser2*, sino con todos. Es decir, todas las reglas pasa a estar activas.
- ***disactivate_all_rule(+Browser1,+Browser2)***: Realiza la misma operación que dis*activate_rule*/2, pero no sólo con un posible elemento seleccionado de *Browser1*, sino con todos. Es decir, todas las reglas pasa a estar inactivas.

Gestión de ficheros

Para la apertura, carga y guardado de ficheros se usan los predicados *working_directory*/2, *chdir*/1, *open*/3, *close*/1, *write*/2, *read*/2, predefinidos para Prolog, y *get*/5, *send*/3, para los objetos XPCE; todos ellos fueron ya descritos en la sección **5.2.1.**

Inicio de la interfaz gráfica de usuario

El lanzamiento de la interfaz gráfica se realiza mediante el predicado *exec*/0, que es ejecutado al iniciarse el sistema, al compilar el fichero meLoader.pl.

5.3.3 Motor de Deducción

Paso de LPO a forma clausal

El paso de fórmulas escritas en lógica de primer orden a forma clausal se lleva a cabo en una serie de pasos sucesivos. La implementación de cada uno de estos pasos asume que se han llevado a cabo los anteriores, por lo que la ejecución de estos predicados no es conmutativa. Los pasos son los siguientes:

1. Expansión de las macros sintácticas

   Se expanden las macros sintácticas de la lógica de primer orden.

2. Eliminación de las fórmulas condicionales

   Se eliminan las implicaciones y las dobles implicaciones mediante leyes de equivalencia lógica.

3. Introducción de negaciones

   Se introducen las negaciones lo más posible dentro de la fórmula empleando las leyes de equivalencia lógica.

4. Skolemización

   Se eliminan las cuantificaciones existenciales mediante un proceso de Skolemización.

5. Eliminación de las cuantificaciones universales

   Se eliminan las cuantificaciones universales.

6. Paso a forma normal conjuntiva

   Se pasa la fórmula a forma normal conjuntiva empleando leyes de equivalencia lógica.

7. Extracción de cláusulas

   Se extraen las cláusulas y se forma la lista de ellas.

Este proceso se encuentra descrito con todo nivel de detalle en (Clocksin y Mellish 1981).

Métodos de búsqueda

Prolog, para explorar el espacio de búsqueda de un objetivo (que se puede ver como un árbol) mediante unificaciones sucesivas con las reglas introducidas en su base de reglas usa el método de búsqueda en profundidad, lo cual implica una serie de consecuencias inapropiadas para nuestro sistema, como se explicará a continuación. También se plantearán las características, ventajas e inconvenientes que implicaría la utilización de la búsqueda en anchura y las razones por las que se ha optado por implementar el método de búsqueda en profundidad escalonada.

Primero en profundidad

Método para recorrer los nodos de un árbol o de un grafo, pero nos centraremos en el primer caso, que es el que aquí nos interesa. Partiendo del nodo raíz, se visita el primer árbol hijo del nodo actual; una vez recorrido, se visitan los demás árboles hijo en un orden establecido, realizando vuelta atrás.

Cómo ya se dijo anteriormente, es el método utilizado por Prolog, por lo que la principal ventaja de su uso sería que no se requeriría una gestión explicita del método de búsqueda, delegando en Prolog para ello.

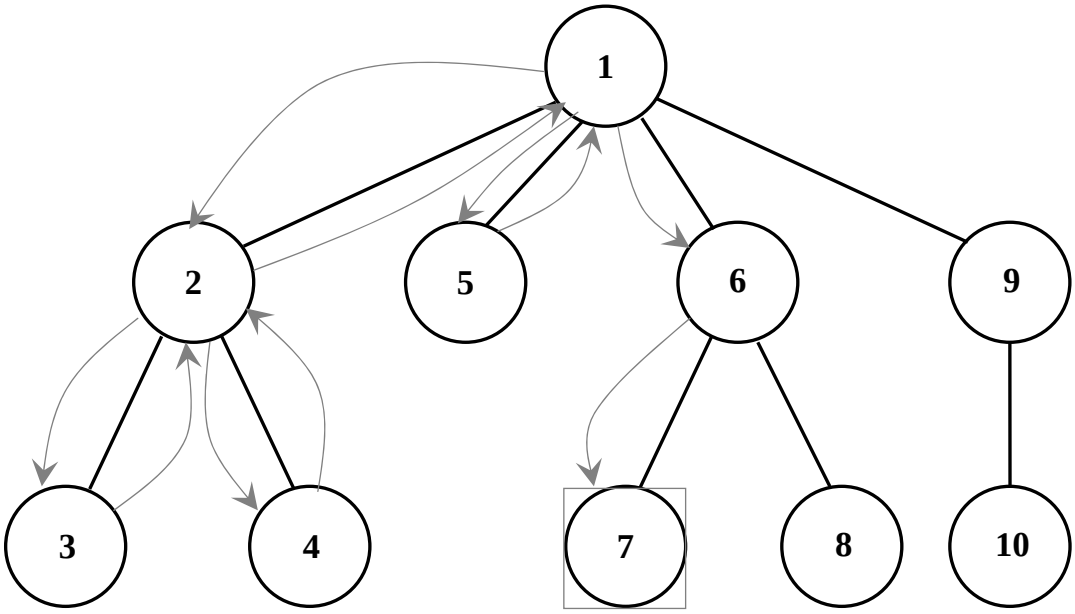


Fig. 9 Ejemplo de exploración en profundidad

Si en el árbol no hay ramas infinitas, se garantiza la exploración de todos los nodos del espacio de búsqueda; sin embargo, de existir ramas infinitas, la exploración puede no acabar, incluso habiendo soluciones, por lo que no asegura la completitud. Además, no se garantiza que la primera solución encontrada sea la de profundidad mínima. Estas razones hacen totalmente indeseable su uso en nuestro sistema.

Primero en anchura

Método para recorrer los nodos de un árbol o de un grafo, pero de nuevo nos limitaremos al primer supuesto. Partiendo del nodo raíz, se visitan todos sus nodos hijos; posteriormente, se visitan todos los nodos hijos de éstos, y así sucesivamente. Es decir, primero se exploran el nodo de profundidad 1, más tarde los de profundidad 2, luego los de 3, etc., hasta alcanzar la máxima profundidad, de existir ésta.

Aun habiendo ramas infinitas, el método garantiza la obtención de cualquier solución, es decir, es completo. Además, obviamente, encuentra las soluciones por orden creciente de profundidad. Estas características del método de búsqueda en anchura hacen mucho más apropiado su uso que el de profundidad, sin embargo, requeriría una gestión explicita bastante costosa, especialmente en uso de memoria.

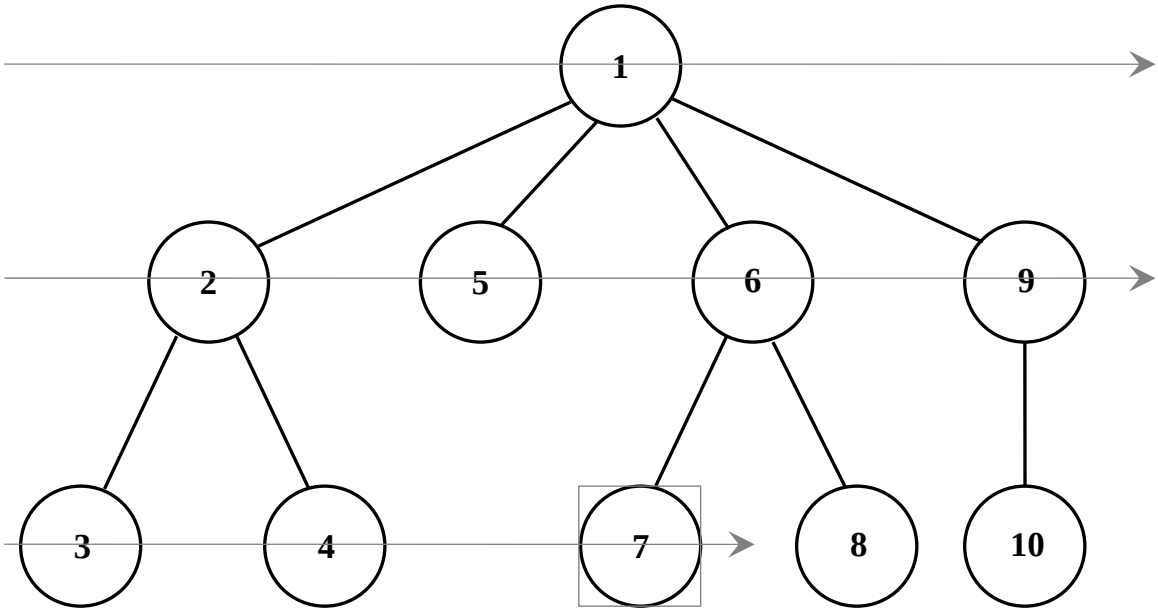


Fig. 10 Ejemplo de exploración en anchura

Profundidad escalonada

Consiste, básicamente, en la realización de sucesivas búsquedas en profundidad con un límite de profundidad determinado, es decir, la búsqueda finaliza en los nodos de esa profundidad y no se visitan los de profundidad superior; se llama en profundidad escalonada porque, en las sucesivas búsquedas, se va incrementando la profundidad máxima mencionada. Por ejemplo, se comienza con un límite de profundidad de 1, a continuación se realiza una búsqueda con una profundidad máxima de 2, luego 3, 4, etc.

A efectos de resultados, su comportamiento es exactamente el mismo que el de la búsqueda en anchura, sin embargo tiene unas cualidades añadidas que la hacen la mejor opción para su uso en el sistema desarrollado, como son la sencillez de su implementación y el reducido uso de memoria que requiere. Por lo tanto, el método de búsqueda en profundidad escalonada reúne las ventajas de los de profundidad y anchura.

En cuanto a la eficiencia, cabe aclarar que el impacto negativo que tiene sobre ella el hecho de repetir en cada iteración todo el trabajo anterior es poco significativo. Suponiendo un factor de ramificación b, el número de nodos en la profundidad (n+1) está en $\Theta(b^{n+1})$, mientras que el de nodos para los que se repite el trabajo anterior está en $\Theta(b^{n})$.

Implementación del método de búsqueda en profundidad escalonada

El mecanismo de búsqueda implementado, integrado con el método de eliminación de modelos, es una variante de la estrategia de búsqueda en profundidad escalonada. En él,

además de la profundidad, se incluye un parámetro adicional, el del número de pasos disponible, que se va repartiendo sucesivamente a los subobjetivos. El usuario dispone de dos parámetros configurables por él para influir en el comportamiento del proceso de búsqueda, son la profundidad máxima límite y el factor. Cabe señalar que una profundidad máxima límite de valor 0 establece una búsqueda sin profundidad límite.

En el nivel más alto, se incrementa la profundidad máxima y se realiza una nueva búsqueda, a no ser que se haya alcanzado la profundidad máxima límite; en tal caso, la demostración finalizaría, obviamente, sin éxito. Al objetivo principal se le asigna un número máximo de pasos de *P * factor*, siendo *P* la profundidad máxima de la búsqueda actual y factor el parámetro configurable por el usuario anteriormente mencionado. Se incluye un fragmento simplificado del código que ilustra este proceso:

```
/*
solve(+G,-P,+LA,+V,-R,+D,-I_F:DAux) <=> G es el objetivo a demostrar, P es la demostración, LA es la lista de ancestros, V es la lista de variables usadas en G, R es una lista de valores para las variables de V que hacen cierto G, D es la diferencia entre la profundidad máxima de la búsqueda actual, la profundidad de G respecto al objetivo principal, I_F es el número exacto de pasos que se ha requerido, en el caso de conseguirlo, para demostrar G
*/

solve(G,P,LA,V,R,D,I_F:DAux) :-
    get_val(factor),
    I is D*factor,                  % I es el numero máximo de pasos
    solve(G,P,LA,V,R,D,I,F)
    ;
    get_val(depth_limit,DL),        % DL es la profundidad máxima límite
    (DL == 0
     ->
     S is D + 1,
     solve(G,P,LA,V,R,S,I_F:DAux)
     ;
     (D < DL
       ->
       S is D + 1,
       solve(G,P,LA,V,R,S,I_F:DAux)
        ;
       assert(finished)
     )
   ).
```

En el nivel intermedio, y suponiendo que el subobjetivo actual *Q* consta de un número N de literales *Q1', Q2',…, QN'*, en primer lugar, se comprueba que no se haya excedido la profundidad máxima de la búsqueda actual y que se dispone de un número de pasos superior a

N; de no ser así se daría fin a la demostración del subobjetivo actual. Si se cumplen las condiciones anteriores, se reparte el número de pasos entre los diferentes literales de un subobjetivo, del modo siguiente:, al primero se le asignan I – N +1 pasos, siendo I los pasos disponibles para resolver *Q*; Para el conjunto restante *Q2', Q3', … , QN'*, se actúa de igual manera, disponiendo para ello de M + N -1 pasos, siendo M los pasos que sobraron para demostrar *Q1'*. De nuevo, se incluye un extracto simplificado de la parte del código que realiza esta tarea:

```
/*
solve((+G,-P,+LA,+V,-R,+D,+I,-F) <=> G es el objetivo a demostrar, P es la demostración, LA es la lista de ancestros, V es la lista de variables usadas en G, R es una lista de valores para las variables de V que hacen cierto G, D es la diferencia entre la profundidad máxima de la búsqueda actual y la profundidad de G respecto al objetivo principal, I es el número de pasos disponibles para demostrar G y F es el número de pasos que han sobrado para demostrar G. G se resuelve por 'model elimination' en I - F pasos, y con profundidad no superior a D (D puede contemplarse como la iteración del operador a la que se encuentra la solución
*/

solve(true:_,nil,_,V,V,D,I,I) :-
        !,D >= 0,I >= 0.            %Comprobaciones de profundidad y número de pasos
solve((A,B):N,(PA,PB),LA,V,R,D,I,F) :-
        !,D > 0,I >= N,
        I1 is I-N+1,                % I1 es el nº máximo de pasos disponibles para A
        solve(A:1,PA,LA,V,V1,D,I1,M),
        M1 is M+N-1,                % M1 es el No. de pasos que quedan para B
        N1 is N-1,
        solve(B:N1,PB,LA,V1,R,D,M1,F).

solve(A:1,PA,LA,V,FV,D,I,F) :-
        D > 0,I > 0,
        solve_one(A,PA,LA,V,FV,D,I,F).
```

En el nivel más bajo, se intenta resolver un subobjetivo compuesto de un solo literal, mediante el método de eliminación de modelos junto con las optimizaciones desarrolladas, comprobando en cada paso intermedio que aun se tienen pasos disponibles y actualizando el número de pasos restante convenientemente, así como la profundidad actual. En caso de finalizar con éxito una demostración, se devuelve el número de pasos no utilizado, para ponerlos a disposición de sus subobjetivos hermanos.

Método de eliminación de modelos

El método de eliminación de modelos se implementa en el fichero meSolve.pl apoyado con las funciones del fichero meUtils.pl cuyos códigos se adjuntan en el apéndice B y que se

recomiendan leer paralelamente para comprender de manera satisfactoria lo que se expone en este apartado de la memoria.

La implementación se lleva a cabo mediante una serie de predicados denominados *solve*, de distintas aridades, cuyo propósito consiste en, dado un objetivo *Goal* (aquello que el usuario quiere demostrar), se devuelva su demostración *Proof*, una posible lista de respuestas *Answer* (términos que cumplen el objetivo), la profundidad *Depth* en la que se obtuvo la demostración *Proof*, y el número de pasos *Steps* de la misma:

solve(+Goal,-Proof,-Answer,-Steps:Depth)

El método utiliza una búsqueda escalonada basada en diferentes parámetros: La profundidad actual en la que se lleva a cabo la búsqueda, y un factor (por defecto con valor 10) definido por el usuario en el menú de configuración de opciones. Estos parámetros nos dan un número máximo de pasos/recursos en los que el sistema aplicará el motor de deducción para cada nivel de profundidad, obtenido de la siguiente forma:

*Nº recursos = profundidad_actual * factor*

El usuario puede fijar en el menú de configuración de opciones una profundidad máxima (por defecto ilimitada, lo cual se indica con el valor 0), de forma que si se llega a dicha profundidad sin haber obtenido una demostración del objetivo, el sistema de deducción para e informa al usuario de que a dicha profundidad (y con los demás parámetros configurados por el usuario), el sistema no ha encontrado una demostración de dicho objetivo.

Una vez que se agotan los recursos para un cierto nivel de profundidad, el método se llama recursivamente con el siguiente nivel de profundidad para intentar obtener una demostración del objetivo, pasando a otra función solve de mayor aridad, definida como sigue:

solve(+Goal:Length,-Proof,+List_of_Ancestors,+Vars,-FinalVars,+Depth,
+First,+Last)

Goal se resuelve por eliminación de modelos en *First* - *Last* pasos, y con profundidad no superior a *Depth* (que puede contemplarse como la iteración del operador a la que se encuentra la solución).

Dicha función, mira la forma del objetivo o subobjetivo, distinguiendo si es trivial (*true*), en cuyo caso para, o si es un único término, en cuyo caso llama al último nivel de predicados *solve* que explicaremos a continuación, o si dicho objetivo/subobjetivo se compone de varios términos, en cuyo caso, se llama recursivamente para cada uno de esos términos para, finalmente, aplicar el último nivel de predicados *solve*.

El predicado *solve* de más bajo nivel utiliza otro, denominado *solve_one*, que se aplica a objetivos/subobjetivos compuestos por un único literal. Dicha función intenta resolver dichos subobjetivos con el número de recursos disponibles, usando para ello las técnicas de deducción que aplica el método de eliminación de modelos, más las optimizaciones que

hemos incluido en el sistema: reglas del objetivo, comprobación de existencia de ancestro complementario o idéntico y uso de lemas persistentes e intermedios.

Dichas técnicas y optimizaciones se explican con detalle en el siguiente apartado de la memoria. En resumen podríamos decir que las técnicas basadas en lemas consisten en usar para una demostración resultados demostrados con anterioridad, ya sean de forma persistente (lemas), o resultados probados en el transcurso de una demostración (lemas intermedios).

El resto de técnicas se basan en el concepto de ancestro, que son los distintos términos generados durante una demostración a partir de un objetivo o subobjetivo compuesto por un único término.

En particular el uso de las reglas del objetivo suele implicar la devolución de respuestas disyuntivas.

En el caso de los objetivos escritos con la notación de la lógica de primer orden, antes de aplicar el proceso descrito atrás se procede como sigue:

Dado un objetivo *G* escrito en lógica de primer orden se genera la fórmula *G => goal*, dónde *goal* es un nuevo predicado sin argumentos. Seguidamente se añade dicha fórmula a la teoría, aplicando su conversión a formato clausal y generando las cláusulas Prolog correspondientes. Finalmente se lanza como objetivo la prueba *<-- goal* aplicando el proceso explicado anteriormente.

En este caso, el sistema no admite respuestas disyuntivas.

Optimizaciones

Para incrementar la potencia del sistema y aumentar la eficiencia de los procesos de demostración, se han añadido una serie de mejoras al método básico de eliminación de modelos; todas ellas pueden ser activadas o desactivadas por el usuario. Se pueden distinguir dos tipos de optimizaciones; en primer lugar, las que tienen como intención provocar la poda de esa rama lo antes posible, finalizando la demostración para el subobjetivo correspondiente; por otro lado, están las que gestionan el uso de los lemas y los lemas intermedios, esto es, objetivos anteriormente demostrados, durante un proceso anterior o durante el actual, respectivamente.

Antes de explicar la base teórica y la implementación del primer grupo de mejoras, hay que mencionar el uso de una estructura auxiliar, llamada lista de ancestros, y que resulta esencial para la aplicación de tales optimizaciones. La lista de ancestros de un subobjetivo Q_i es la lista con todos los subobjetivos que se encuentran en el camino desde el propio Q_i hasta el objetivo principal, esto es, hasta la raíz del árbol de demostración; es por esta razón por la que se les llama ancestros. La gestión de la lista de ancestros resulta trivial en Prolog, por lo que no se ha considerado relevante incluir ningún texto ilustrativo.

Poda por ancestro idéntico

Cuando se intenta resolver un subobjetivo compuesto por un único literal Q, se comprueba si ya se encontraba ese mismo subobjetivo Q dentro de la lista de su lista de ancestros. En ese caso, se provoca el fallo y se poda esa rama. Esto es debido a que cualquier demostración del objetivo que pudiera hallarse tras su segunda aparición podría hallarse a partir de la primera, con menos recursos. Se añade seguidamente un extracto simplificado del código que lo implementa:

```
solve_one(A,_,LA,_,_,_,_,_) :-
        identical_ancestor(A,LA),
        !,fail.

identical_ancestor(A,LA) :-
        get_val(id_anc,on),!,          % Se comprueba si está activada la optimización
        strict_member(A,LA).
```

Poda por ancestro complementario idéntico

Cuando se intenta resolver un subobjetivo compuesto por un único literal Q, se comprueba si ya se encontraba ese mismo subobjetivo complementado $\neg Q$ dentro de la lista de su lista de ancestros. De ser así, puede descartarse el uso de pasos de expansión para dicho objetivo, ya que puede darse un paso de reducción que resolvería el problema de manera más sencilla.. Se muestra su implementación en un fragmento simplificado del código:

```
solve_one(A,p(A <-- B,PB),LA,V,FV,D,I,F) :-
        program_rule_occurs(A,B:N),
        \+ identical_comp_ancestor(A,LA),
        D1 is D-1,S is I-1,
        solve(B:N,PB,[A|LA],V,FV,D1,S,F).

solve_one(A,p(A <-- B ** g,PB),LA,V,FV,D,I,F) :-
        goal_rule_occurs(A,B:N,V1),
        \+ identical_comp_ancestor(A,LA),
        D1 is D-1,S is I-1,
        solve(B:N,PB,[A|LA],V or V1,FV,D1,S,F).

identical_comp_ancestor(A,LA) :-
        get_val(id_comp_anc,on),!,     % Se comprueba si está activada la optimización
        negation(A,No_A),
        strict_member(No_A,LA).
```

Uso de las reglas del objetivo

En el momento de iniciar la demostración de un objetivo dado C_1, C_2,…, C_n, se introducen las siguientes reglas, a las que llamamos reglas del objetivo por motivos obvios, dentro de la base teórica:

$$\neg C_1 \leftarrow C_2, C_3,\ldots, C_n$$
$$\neg C_2 \leftarrow C_1, C_3,\ldots, C_n$$
$$\vdots$$
$$\neg C_n \leftarrow C_1, C_2,\ldots, C_{n-1}$$

Uso de lemas

Las optimizaciones basadas en lemas se dividen en 2 tipos fundamentales: lemas persistentes (o simplemente lemas), y lemas intermedios.

Los lemas (persistentes) son objetivos o subobjetivos compuestos por un único término (en caso de que el usuario hubiera puesto un objetivo compuesto por n términos distintos, se crearían n lemas, uno por cada término), que se guardaron anteriormente en el fichero de lemas (ver apartado **5.2.2**) asociado a una teoría, para que el sistema recuerde dichos resultados.

Como se explicó en el apartado **5.2.2**, dichos lemas tienen la siguiente estructura:

lema(G,P,A,D,S): *G* es el lema demostrado por el usuario; *P* es la demostración del lema G, se guarda siguiendo una cierta estructura que indica en cada paso de demostración la técnica de deducción que se ha utilizado; *A* es la lista finita de términos que cumplen el lema *G*, que puede ser vacía; *D* indica la profundidad de búsqueda en la que se obtuvo la demostración *P*; *S* , por su parte, indica el número de pasos de la demostración *P*.

Este mecanismo permite al usuario ir construyendo demostraciones de objetivos de forma incremental con pruebas cada vez más complejas que impliquen probar de nuevo dichos lemas como pasos previos, acelerando así el proceso de demostración de esos nuevos objetivos porque el sistema recuerda las demostraciones de dichos lemas.

Puesto que según haya parametrizado el usuario los factores de búsqueda de demostraciones, se encuentran pruebas a mayor o menor profundidad o con menor o mayor número de pasos, el sistema sigue la política de guardar aquellas demostraciones de lemas obtenidas a menor profundidad y/o a menor número de pasos. El sistema no guarda aquellos lemas que contengan respuestas disyuntivas.

Los lemas intermedios son resultados que el motor de deducción ha ido probando en el transcurso de una demostración. Este mecanismo permite acelerar el método de demostración puesto que dichos resultados pueden aparecer varias veces durante la búsqueda, evitando así, probarlos varias veces.

Cuando esta opción está activada por el usuario, en cada nueva demostración se crea una base de lemas intermedios vacía, que irá conteniendo los subobjetivos que se han ido demostrando en la función *solve_one* explicada en el apartado **5.3.3**.

A diferencia de los lemas, los lemas intermedios no se pueden guardar de manera persistente puesto que son resultados que solo tienen sentido en el contexto de una cierta demostración, pero no de forma general.

En el apartado **6** se exponen una serie de pruebas en las que se puede ver de forma evidente el rendimiento en cuanto a tiempo y recursos (número de pasos y profundidad) que ofrece el mecanismo de los lemas intermedios.

6. Ejemplos y pruebas

En este apartado se expondrán distintos ejemplos del funcionamiento del sistema basados en una serie de ficheros de teoría que contiene axiomas escritos en notación clausal, y en notación de la lógica de primer orden.

Para poder observar el rendimiento del sistema, en los ejemplos se supondrá siempre activa la opción de muestra de la traza de las demostraciones, que nos indica el tiempo que el sistema tarda en responder.

Prueba 1

El siguiente fichero de teoría presenta un problema basado en el libro *Alicia en el país de las maravillas* del lógico y matemático británico Lewis Carroll.

```
no miente(liebre) <- no ladron(liebre).
no miente(sombrerero) <- ladron(liebre),no ladron(sombrerero).
no miente(sombrerero) <- ladron(sombrerero),no ladron(sombrerero).
no miente(sombrerero) <- ladron(liron),no ladron(sombrerero).
no miente(liron) <- no miente(liebre).
no miente(liron) <- no miente(sombrerero).
miente(liebre),miente(liron).
ladron(liebre),ladron(sombrerero),ladron(liron).
```

Seguidamente vemos la traza de una demostración usando reglas del objetivo, y otra demostración usando la técnica de reducción, que puede verse como una demostración por reducción al absurdo. El objetivo a demostrar será: ladron(X)

La demostración usando reglas del objetivo queda como sigue:

```
 ---------------------------------------------------------
|Prof.|Tiempo|Anc.|Anc.Com|Reduc.|Anc.|Anc.Com|Lemas|Lemas|
|     |(s)   |Comp|+Reg.Ob|      |Id. |Id.    |     |Inter|
 ---------------------------------------------------------
  1    0      7    0       0      0    0       0      0
  2    0      19   2       0      0    0       0      0

3 pasos.

*** DEMOSTRACIÓN ***

ladron(liebre)<-no ladron(sombrerero), no ladron(liron)

  no ladron(sombrerero)<-true  *** REGLA DEL OBJETIVO

  no ladron(liron)<-true  *** REGLA DEL OBJETIVO

Answer = [liebre]or[sombrerero]or[liron]
```

La demostración mediante la técnica de reducción es la siguiente:

```
-------------------------------------------------------
|Prof.|Tiempo|Anc.|Anc.Com|Reduc.|Anc.|Anc.Com|Lemas|Lemas|
|     |(s)   |Comp|+Reg.Ob|      |Id. |Id.    |     |Inter|
 -------------------------------------------------------
  1    0      7     0      0      0     0      0      0
  2    0      22    0      0      0     0      0      0
  3    0      46    0      0      0     0      0      0
  4    0      49    0      1      0     0      0      0

4 pasos.

*** DEMOSTRACIÓN ***

ladron(liebre)<-miente(liebre)

  miente(liebre)<-miente(liron)

    miente(liron)<-no miente(liebre)

      no miente(liebre) *** REDUCCIÓN

Answer = [liebre]
```

Como puede observarse, la opción de uso de reglas del objetivo nos permite hallar respuestas disyuntivas, que recogen todos los elementos del universo de discurso que cumplen el objetivo que hemos demostrado, mientras que la técnica de reducción nos devuelve una respuesta con uno de esos posibles valores.

Prueba 2

Los siguientes ejemplos están basados en dos ficheros de teoría, uno con axiomas en forma clausal, y el otro con axiomas escritos en lógica de primer orden, basados en una serie de conceptos básicos de la teoría elemental de conjuntos.

En primer lugar supondremos el siguiente fichero de teoría en el que todos los axiomas están escritos en forma clausal:

```
/* Sets */

% Inclusion:

in(X,B) <- in(X,A),sub(A,B).
in(f(A,B),A) <- no sub(A,B).
no in(f(A,B),B) <- no sub(A,B).

%Igualdad
eq(A,B) <- sub(A,B),sub(B,A).
sub(A,B) <- eq(A,B).
sub(B,A) <- eq(A,B).

% Union
in(X,u(A,B)) <- in(X,A).
in(X,u(A,B)) <- in(X,B).
in(X,A),in(X,B) <- in(X,u(A,B)).

% Intersection
in(X,i(A,B)) <- in(X,A),in(X,b).
in(X,A) <- in(X,i(A,B)).
in(X,B) <- in(X,i(A,B)).
```

```
% Complementation
in(X,c(A)) <- no in(X,A).
no in(X,A) <- in(X,c(A)).


% Diferencia
in(X,dif(A,B)) <- in(X,A), no in(X,B).
in(X,A) <- in(X,dif(A,B)).
no in(X,B) <- in(X,dif(A,B)).

% Conjunto Vacío
no in(X,eset).

% Partes de un conjunto
in(X,p(A)) <- sub(X,A).
sub(X,A) <- in(X,p(A)).
```

Para ilustrar la importancia del parámetro FACTOR, demostremos que la unión de conjuntos es conmutativa tomando FACTOR=10 y FACTOR=2. En ambos casos activaremos la opción de lemas intermedios, cuya relevancia en el rendimiento del sistema la veremos en posteriores ejemplos.

Demostrar $A \cup B = B \cup A \quad \forall A, B$

Objetivo: eq(u(a,b),u(b,a))

Factor =10

Prof.	Tiempo (s)	Anc. Comp	Anc.Com +Reg.Ob	Reduc.	Anc. Id.	Anc.Com Id.	Lemas	Lemas Inter
1	0	1	0	0	0	0	0	0
2	0	7	0	0	0	0	0	0
3	0	41	0	0	1	0	0	0
4	0	333	0	1	3	0	0	3
5	0.031	2435	0	2	28	0	0	70
6	0.046	5242	0	6	63	0	1	159

```
17 pasos.

*** DEMOSTRACIÓN ***

eq(u(a, b), u(b, a))<-sub(u(a, b), u(b, a)), sub(u(b, a), u(a, b))

  sub(u(a, b), u(b, a))<-no in(f(u(a, b), u(b, a)), u(a, b))

    no in(f(u(a, b), u(b, a)), u(a, b))<-no in(f(u(a, b), u(b, a)), a), no
in(f(u(a, b), u(b, a)), b)

      no in(f(u(a, b), u(b, a)), a)<-no in(f(u(a, b), u(b, a)), u(b, a))

        no in(f(u(a, b), u(b, a)), u(b, a))<-no sub(u(a, b), u(b, a))

          no sub(u(a, b), u(b, a)) *** REDUCCIÓN

      no in(f(u(a, b), u(b, a)), b)<-no in(f(u(a, b), u(b, a)), u(b, a))

         *** LEMA INTERMEDIO

        no in(f(u(a, b), u(b, a)), u(b, a))<-no sub(u(a, b), u(b, a))
```

```
          no sub(u(a, b), u(b, a)) *** REDUCCIÓN
  sub(u(b, a), u(a, b))<-no in(f(u(b, a), u(a, b)), u(b, a))
    no in(f(u(b, a), u(a, b)), u(b, a))<-no in(f(u(b, a), u(a, b)), b), no
in(f(u(b, a), u(a, b)), a)
      no in(f(u(b, a), u(a, b)), b)<-no in(f(u(b, a), u(a, b)), u(a, b))
         *** LEMA INTERMEDIO
        no in(f(u(b, a), u(a, b)), u(a, b))<-no sub(u(b, a), u(a, b))
          no sub(u(b, a), u(a, b)) *** REDUCCIÓN
      no in(f(u(b, a), u(a, b)), a)<-no in(f(u(b, a), u(a, b)), u(a, b))
         *** LEMA INTERMEDIO
        no in(f(u(b, a), u(a, b)), u(a, b))<-no sub(u(b, a), u(a, b))
          no sub(u(b, a), u(a, b)) *** REDUCCIÓN
```

Factor = 2

Prof.	Tiempo (s)	Anc. Comp	Anc.Com +Reg.Ob	Reduc.	Anc. Id.	Anc.Com Id.	Lemas	Lemas Inter
1	0	1	0	0	0	0	0	0
2	0	7	0	0	0	0	0	0
3	0	41	0	0	1	0	0	0
4	0	333	0	1	3	0	0	3
5	0.031	2435	0	2	28	0	0	70
6	0.046	3911	0	3	48	0	0	115
7	0.093	7350	0	6	85	0	0	157
8	1.015	72848	0	9	790	0	0	1104
9	2.515	160523	0	11	1765	0	0	3417

```
17 pasos.
*** DEMOSTRACIÓN ***
eq(u(a, b), u(b, a))<-sub(u(a, b), u(b, a)), sub(u(b, a), u(a, b))
   *** LEMA INTERMEDIO
  sub(u(a, b), u(b, a))<-no in(f(u(a, b), u(b, a)), u(a, b))
    no in(f(u(a, b), u(b, a)), u(a, b))<-no in(f(u(a, b), u(b, a)), a), no
in(f(u(a, b), u(b, a)), b)
      no in(f(u(a, b), u(b, a)), a)<-no in(f(u(a, b), u(b, a)), u(b, a))
        no in(f(u(a, b), u(b, a)), u(b, a))<-no sub(u(a, b), u(b, a))
          no sub(u(a, b), u(b, a)) *** REDUCCIÓN
      no in(f(u(a, b), u(b, a)), b)<-no in(f(u(a, b), u(b, a)), u(b, a))
         *** LEMA INTERMEDIO
        no in(f(u(a, b), u(b, a)), u(b, a))<-no sub(u(a, b), u(b, a))
          no sub(u(a, b), u(b, a)) *** REDUCCIÓN
  sub(u(b, a), u(a, b))<-no in(f(u(b, a), u(a, b)), u(b, a))
```

```
    no in(f(u(b, a), u(a, b)), u(b, a))<-no in(f(u(b, a), u(a, b)), b), no
in(f(u(b, a), u(a, b)), a)
       *** LEMA INTERMEDIO
      no in(f(u(b, a), u(a, b)), b)<-no in(f(u(b, a), u(a, b)), u(a, b))
         *** LEMA INTERMEDIO
        no in(f(u(b, a), u(a, b)), u(a, b))<-no sub(u(b, a), u(a, b))
          no sub(u(b, a), u(a, b)) *** REDUCCIÓN
      no in(f(u(b, a), u(a, b)), a)<-no in(f(u(b, a), u(a, b)), u(a, b))
         *** LEMA INTERMEDIO
        no in(f(u(b, a), u(a, b)), u(a, b))<-no sub(u(b, a), u(a, b))
          no sub(u(b, a), u(a, b)) *** REDUCCIÓN
```

Como se puede apreciar, tomando factor=10, obtenemos una demostración a profundidad 6 obtenida en 0.046 segundos, mientras que tomando factor =2, obtenemos una demostración a profundidad 9 y en 2.515 segundos.

Prueba 3

El siguiente ejemplo se toma de un fichero de teoría con los axiomas escritos con la notación de la lógica de primer orden. El fichero es el siguiente:

```
/* Sets */
% Inclusion:
all(A,all(B,sub(A,B) <=> all(X,(in(X,A) => in(X,B))))).

%Conjunto Vacío
all(X,no in(X,eset)).

%Igualdad
all(A,all(B,eq(A,B) <=> sub(A,B) & sub(B,A))).

% Union
all(A,all(B,all(X,in(X,u(A,B)) <=> in(X,A) # in(X,B)))).

%% Intersection
all(A,all(B,all(X,in(X,i(A,B)) <=> in(X,A) & in(X,B)))).

%% Complementation
in(X,c(A)) <- no in(X,A).
no in(X,A) <- in(X,c(A)).

%% Diferencia
all(A,all(B,all(X,in(X,dif(A,B)) <=> no in(X,B) & in(X,A)))).

%% Partes de un conjunto
all(A,all(X,in(X,p(A)) <=> sub(X,A))).
```

Para ilustrar la efectividad del mecanismo de los lemas intermedios, vamos a demostrar que la intersección de conjuntos es asociativa.

Demostrar $X \cap (Y \cap Z) = (X \cap Y) \cap Z \quad \forall X, Y, Z$

Objetivo: all([X,Y,Z],eq(i(X,i(Y,Z)),i(i(X,Y),Z)))

Usando lemas intermedios:

Prof.	Tiempo (s)	Anc. Comp	Anc.Com +Reg.Ob	Reduc.	Anc. Id.	Anc.Com Id.	Lemas	Lemas Inter
1	0	1	0	0	0	0	0	0
2	0	4	0	0	0	0	0	0
3	0	16	0	0	1	0	0	0
4	0	85	0	0	6	0	0	0
5	0	589	0	2	37	0	0	16
6	0.031	3720	0	6	280	0	0	197
7	0.265	18917	0	9	1750	0	0	1378
8	0.671	44402	0	9	4984	0	0	3847

26 pasos.

*** DEMOSTRACIÓN ***

goal<-eq(i(f2, i(f3, f4)), i(i(f2, f3), f4))

eq(i(f2, i(f3, f4)), i(i(f2, f3), f4))<-sub(i(f2, i(f3, f4)), i(i(f2, f3), f4)), sub(i(i(f2, f3), f4), i(f2, i(f3, f4)))

sub(i(f2, i(f3, f4)), i(i(f2, f3), f4))<-in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), i(i(f2, f3), f4))

in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), i(i(f2, f3), f4))<-in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), i(f2, f3)), in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), f4)

in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), i(f2, f3))<-in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), f2), in(f1(i(i(f2, f3), f4), i(f2, i(f3, f4))), f3)

*** LEMA INTERMEDIO

.

.

.

in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), i(f2, f3))<-in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), i(i(f2, f3), f4))

*** LEMA INTERMEDIO

in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), i(i(f2, f3), f4)) *** REDUCCIÓN

in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), f4)<-in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), i(i(f2, f3), f4))

*** LEMA INTERMEDIO

```
            in(f1(i(f2, i(f3, f4)), i(i(f2, f3), f4)), i(i(f2, f3), f4))<-no
sub(i(i(f2, f3), f4), i(f2, i(f3, f4)))

              no sub(i(i(f2, f3), f4), i(f2, i(f3, f4))) *** REDUCCIÓN
```

Sin usar lemas intermedios:

```
-------------------------------------------------------
|Prof.|Tiempo|Anc.  |Anc.Com|Reduc.|Anc.|Anc.Com|Lemas|Lemas|
|     |(s)   |Comp  |+Reg.Ob|      |Id. |Id.    |     |Inter|
 -------------------------------------------------------
  1    0      1      0       0      0     0      0     0
  2    0      4      0       0      0     0      0     0
  3    0      16     0       0      1     0      0     0
  4    0      85     0       0      6     0      0     0
  5    0.015  629    0       12     35    0      0     0
  6    0.062  4016   0       169    278   0      0     0
  7    0.343  23323  0       1324   1898  0      0     0
  8    1.781  120020 0       8190   11270 0      0     0
  9    4.828  299517 0       20715  29108 0      0     0


36 pasos.

*** DEMOSTRACIÓN ***

goal<-eq(i(f5, i(f6, f7)), i(i(f5, f6), f7))

  eq(i(f5, i(f6, f7)), i(i(f5, f6), f7))<-sub(i(f5, i(f6, f7)), i(i(f5, f6),
f7)), sub(i(i(f5, f6), f7), i(f5, i(f6, f7)))

    sub(i(f5, i(f6, f7)), i(i(f5, f6), f7))<-in(f1(i(i(f5, f6), f7), i(f5, i(f6,
f7))), i(i(f5, f6), f7))

      in(f1(i(i(f5,  f6),  f7),  i(f5,  i(f6,  f7))),  i(i(f5,  f6),  f7))<-
in(f1(i(i(f5, f6), f7), i(f5, i(f6, f7))), i(f5, f6)), in(f1(i(i(f5, f6), f7),
i(f5, i(f6, f7))), f7)

                                        .

                                        .

                                        .

            in(f1(i(f5, i(f6, f7)), i(i(f5, f6), f7)), i(f5, f6))<-in(f1(i(f5,
i(f6, f7)), i(i(f5, f6), f7)), i(i(f5, f6), f7))

              in(f1(i(f5, i(f6, f7)), i(i(f5, f6), f7)), i(i(f5, f6), f7))<-no
sub(i(i(f5, f6), f7), i(f5, i(f6, f7)))

                no sub(i(i(f5, f6), f7), i(f5, i(f6, f7))) *** REDUCCIÓN

          in(f1(i(f5,  i(f6,  f7)),  i(i(f5,  f6),  f7)),  f7)<-in(f1(i(f5,  i(f6,
f7)), i(i(f5, f6), f7)), i(i(f5, f6), f7))

            in(f1(i(f5, i(f6, f7)), i(i(f5, f6), f7)), i(i(f5, f6), f7))<-no
sub(i(i(f5, f6), f7), i(f5, i(f6, f7)))

              no sub(i(i(f5, f6), f7), i(f5, i(f6, f7))) *** REDUCCIÓN
```

Usando lemas intermedios, hemos demostrado la asociatividad de la intersección a profundidad 8 en 0.671 segundos, mientras que sin utilizar éste mecanismo, se ha demostrado a profundidad 9 en 4.125 segundos. Además la demostración con lemas intermedios tiene 26 pasos frente a los 36 de la demostración sin lemas intermedios.

Prueba 4

En los siguientes ejemplos omitiremos las demostraciones puesto que con los ejemplos previos, el lector ya ha podido observar el aspecto de las respuestas ofrecidas por el sistema, todas ellas, en formato clausal. No obstante, se incluyen las trazas por su valor informativo.

Demostraremos una de las leyes de De Morgan:

$$\text{Demostrar } \forall X,Y : \overline{X \cup Y} = \overline{X} \cap \overline{Y}$$

Objetivo: all([X,Y],eq(c(u(X,Y)),i(c(X),c(Y))))

Usando lemas intermedios:

Prof.	Tiempo (s)	Anc. Comp	Anc.Com +Reg.Ob	Reduc.	Anc. Id.	Anc.Com Id.	Lemas	Lemas Inter
1	0	1	0	0	0	0	0	0
2	0	4	0	0	0	0	0	0
3	0	16	0	0	1	0	0	0
4	0	83	0	0	6	0	0	0
5	0	568	0	2	36	0	0	16
6	0.046	3547	0	6	272	0	0	198
7	0.25	17482	0	9	1626	0	0	1300
8	0.625	40374	0	9	4547	0	0	3505

21 pasos.

La demostración se obtiene a profundidad 8 en 0.625 segundos y con 21 pasos.

Sin usar lemas intermedios:

Prof.	Tiempo (s)	Anc. Comp	Anc.Com +Reg.Ob	Reduc.	Anc. Id.	Anc.Com Id.	Lemas	Lemas Inter
1	0	1	0	0	0	0	0	0
2	0	4	0	0	0	0	0	0
3	0	16	0	0	1	0	0	0
4	0.015	83	0	0	6	0	0	0
5	0.015	608	0	12	34	0	0	0
6	0.062	3790	0	176	272	0	0	0
7	0.312	21037	0	1292	1790	0	0	0
8	1.625	104637	0	7389	10118	0	0	0
9	4.343	261762	0	18651	26100	0	0	0

25 pasos.

La demostración se obtiene a profundidad 9 en 4.343 segundos y con 25 pasos.

Prueba 5

Como último ejemplo demostramos la siguiente propiedad distributiva, usando como parámetro el uso de la optimización por ancestro idéntico:

Demostrar: $\forall X, Y, Z : X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$

Objetivo: all([X,Y,Z],eq(i(X,u(Y,Z)),u(i(X,Y),i(X,Z))))

Comprobando ancestros idénticos:

```
-------------------------------------------------------------
|Prof.|Tiempo|Anc.   |Anc.Com|Reduc.|Anc.|Anc.Com|Lemas|Lemas|
|     |(s)   |Comp   |+Reg.Ob|      |Id. |Id.    |     |Inter|
 -------------------------------------------------------------
  1    0      1      0       0      0     0      0     0
  2    0      4      0       0      0     0      0     0
  3    0      16     0       0      1     0      0     0
  4    0      86     0       0      6     0      0     0
  5    0      605    0       2      37    0      0     16
  6    0.046  3908   0       6      286   0      0     203
  7    0.296  20009  0       9      1828  0      0     1454
  8    1.375  83735  0       11     9019  0      0     7220

24 pasos.
```

La demostración se obtiene a profundidad 8 en 1.375 segundos y con 24 pasos.

Sin comprobar ancestros idénticos:

```
-------------------------------------------------------------
|Prof.|Tiempo|Anc.    |Anc.Com|Reduc.|Anc.|Anc.Com|Lemas|Lemas|
|     |(s)   |Comp    |+Reg.Ob|      |Id. |Id.    |     |Inter|
 -------------------------------------------------------------
  1    0      1       0       0      0    0       0     0
  2    0      4       0       0      0    0       0     0
  3    0      18      0       0      0    0       0     0
  4    0      130     0       0      0    0       0     0
  5    0.015  1401    0       2      0    0       0     16
  6    0.218  18466   0       6      0    0       0     413
  7    3.14   243529  0       9      0    0       0     7127
  8    35.468 2622091 0       19     0    0       0     88530

24 pasos.
```

La demostración se obtiene a profundidad 9 en 35.468 segundos y en el mismo número de pasos. En este caso, la comprobación de ancestros ha resultado ser un factor crítico en la eficiencia; en general, su impacto sobre el tiempo de demostración es muy positivo.

7. Conclusiones

Los objetivos marcados al inicio del proceso de desarrollo del proyecto, que fueron expuestos en la sección **3**, se han realizado con éxito:

- El núcleo del sistema es un intérprete escrito en Prolog.
- El sistema tiene la capacidad de ajustar diferentes parámetros y opciones cuyo valor influye en el comportamiento y rendimiento del sistema, como hemos ilustrado con ejemplos en el apartado anterior.
- El programa incluye una serie de herramientas auxiliares como, por ejemplo, el paso a forma clausal de fórmulas de la lógica de primer orden, pretty-printing de las demostraciones…
- El sistema incluye una GUI de manejo fácil e intuitivo, implementada en XPCE/Prolog.

El programa está orientado a usuarios que manejan conceptos básicos y notación de la lógica de predicados y la lógica de primer orden, dejándoles a estos el deber de construir ficheros de teoría con axiomas válidos. Además, el sistema está fuertemente parametrizado y automatizado, por lo que podría servir para fines pedagógicos.

Como recogemos en el apartado **5.1**, el uso de Prolog y XPCE nos ha ayudado a la hora de construir un sistema con estas características, a pesar de la poca documentación existente para utilizarlos, especialmente para XPCE.

A continuación, enumeramos una serie de mejoras futuras que nuestro sistema podría incluir:

- Añadir nuevos métodos de búsqueda y optimizaciones al sistema: Búsqueda guiada por combinación de parámetros con pesos relativos, etc.
- Compilación a Prolog del intérprete que permita mejorar la eficiencia del programa.
- Generación de ficheros LaTeX para las demostraciones de los objetivos.

8. Bibliografía

Clocksin, William F., and Christopher S. Mellish. *Programming in Prolog.* Berlín: Springer, 1981.

Grassman, Winfried Karl, and Jean Paul Tremblay. *Matemática discreta y lógica matemática.* Prentice Hall, 1998.

Guía de usuario de SWI Prolog. http://www.swi-prolog.org/download/stable/doc/SWI-Prolog-5.6.59.pdf.

Guía de usuario de XPCE. http://www.swi-prolog.org/download/xpce/doc/userguide/userguide.pdf.

Hortalá, María Teresa, Javier Leach Albert, y Mario Rodríguez Artalejo. *Matemática discreta y lógica matemática.* Madrid: Editorial Complutense, 2001.

López Fraguas, Francisco Javier. "A prolog metainterpreter for model elimination." 1990.

Loveland, Donald W. "A simplified format for the model elimination theorem-proving procedure." 1969.

Loveland, Donald W. "Mechanical theorem proving by model elimination." 1968.

Sterling, Leon, and Ehud Shapiro. *The art of prolog.* Londres: The MIT press, 1986.

Stickel, Mark E. "A prolog technology theorem prover." 1984.

Apéndice A. Manual de usuario

ÍNDICE

1. Introducción al Sistema

1.1. Ejecución del Programa:

El sistema requiere tener cualquier versión de la consola de SWI-PROLOG. Este software puede obtenerse de manera gratuita en el siguiente link:

www.swi-prolog.org/

El fichero meLoader.pl es un cargador que compila y ejecuta el sistema automáticamente. Para ejecutar el programa, el usuario deberá ejecutar dicho archivo en la consola de Prolog.

1.2. Model-Elimination:

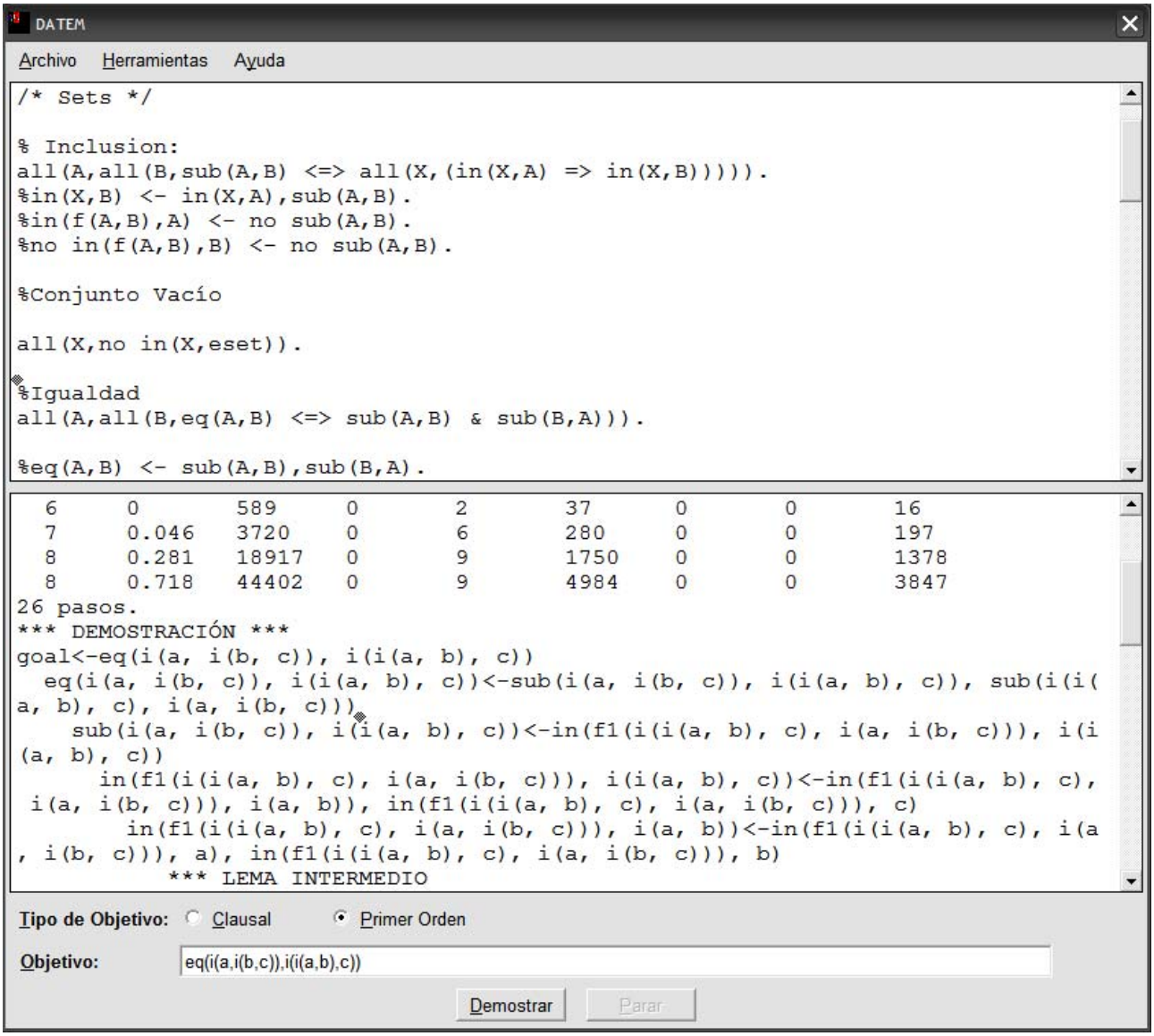
El sistema consiste en un demostrador automático de teoremas basado en la técnica de eliminación de modelos (Model-Elimination) desarrollada por el profesor Donald W. Loveland en los años 60. Model-Elimination es aplicable a la lógica de primer orden en forma clausal, pero el sistema admite fórmulas de lógica de primer orden generales que previamente se convertirán a forma clausal. La sintaxis de ambos tipos de fórmulas se explica con detalle en el punto **2** de la ayuda.

Además, el sistema admite distintos parámetros y opciones admisibles por la técnica de eliminación de modelos como el uso de reglas del objetivo, el uso de la técnica de reducción, el factor influyente en la búsqueda de demostraciones, la profundidad límite y la gestión de lemas (resultados ya probados por el usuario) y lemas intermedios (resultados previos pro-

bados en el transcurso de una demostración) que se explican con detalle en la sección **5** de la ayuda.

1.3. Introducción a la Interfaz y Uso del Programa:

La pantalla principal del programa es la siguiente:



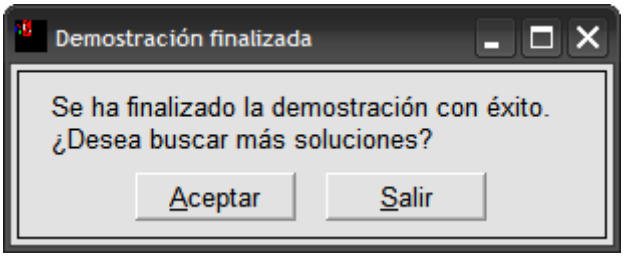
En la pestaña *Archivo*, el usuario puede escoger el directorio de trabajo en el que se irán guardando distintos archivos generados por el sistema en el transcurso de las distintas demostraciones.

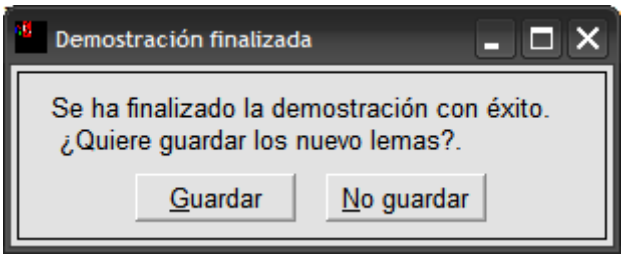
Se distinguen dos editores de texto. En el editor superior se mostrará la teoría (conjunto de axiomas escritos con la notación explicada en el punto **2** de esta ayuda) cargada por el usua-

rio (*Archivo* → *Cargar Clausal* o *Archivo*→*Cargar Primer Orden* según sean las fórmulas de dicha teoría). El usuario puede modificar su teoría y posteriormente guardarla en un archivo **.me* (*Archivo* → *Guardar .me*).

Una vez cargada una teoría, el usuario escribirá un objetivo que quiera demostrar indicando su tipo (Clausal o Primer Orden) y pulsará el botón *Demostrar*. Una vez acabada la demostración de un objetivo, en el editor de texto inferior se podrán ver los pasos de demostración y la traza. Además, el usuario tiene la posibilidad de guardar la demostración en un fichero de texto mediante la opción *Archivo* → *Guardar última prueba*. El botón *Parar* aborta una demostración.

Al finalizar una demostración exitosamente, aparecerán ventanas emergentes que permitirán al usuario poder seguir buscando nuevas demostraciones para el objetivo, así como guardar los lemas generados (siempre y cuando esté activada la opción de guardado de lemas, y desactivada la de su guardado automático). Se muestran a continuación las ventanas mencionadas:





En la pestaña *Herramientas*, el usuario puede configurar los parámetros y opciones usados por el método de búsqueda de demostraciones que se explican en el punto **5** de la ayuda. Dichas modificaciones pueden afectar en el rendimiento del sistema. En dicha pestaña también se ofrece la posibilidad de poder ver las reglas y lemas cargados por el programa.

La pestaña de *Ayuda* ofrece información sobre la versión y los creadores del programa y el vínculo a este manual.

En la sección **4** de la ayuda se ofrece más información sobre el uso de la Interfaz.

2. Forma clausal

Tanto las teorías como los objetivos pueden introducirse al sistema en formato clausal. Para ello debe elegirse esta opción a la hora de seleccionar la opción del menú para cargar la teoría o con el selector del modo de entrada al introducir los objetivos.

El elemento básico de las fórmulas es el literal, que es un símbolo de predicado aplicado a una serie de términos. Un literal se representa mediante un identificador que empieza por una letra minúscula cuyos argumentos se encuentran entre paréntesis separados por comas. Los términos pueden ser una constante, una variable o un símbolo de función aplicado a su vez a una serie de términos. Las constantes se representan mediante palabra que empiezan por una letra mayúscula, las variables mediante palabras que empiezan con una letra minúscula y los símbolos de función se representan igual que las constantes, pero llevan entre paréntesis a continuación sus argumentos separados por comas; se pueden considerar las constantes como símbolos de función de aridad 0. También se pueden emplear operadores como símbolo de predicado o función, siempre que se haga la declaración correspondiente en la sesión de Prolog en la que se ejecute el demostrador con anterioridad.

Se pueden escribir diferentes tipos de fórmulas. Las reglas constan de una cabeza y un cuerpo, y son elementos condicionales La cabeza de las reglas está formada por un literal negado o sin negar, siendo un literal un símbolo de predicado aplicado a una serie de términos, siguiendo la sintaxis habitual de Prolog. Los predicados negados se escriben mediante el operador unario “no”. Todas las variables que aparezcan estarán cuantificadas de manera implícita de forma existencial. También se pueden incluir hechos en el programa, que son predicados seguidos de un punto.

Por ejemplo, una regla con el significado de que si un individuo es alto y es deportista entonces juega al baloncesto, se podría escribir así:

$$juega(X, baloncesto) < -alto(X), deportista(X)$$

Una consulta que intente descubrir si existe algún individuo que juegue al baloncesto y sea bajo se escribiría:

$$juega(X, baloncesto), no\ alto(X)$$

El hecho de que Pedro juega al baloncesto se expresaría de la siguiente forma:

$$juega(pedro, baloncesto)$$

3. Formato de primer orden

Tanto las teorías como los objetivos pueden introducirse al sistema en formato de primer orden. Para ello debe elegirse esta opción a la hora de seleccionar la opción del menú para cargar la teoría o con el selector del modo de entrada al introducir los objetivos, que debe estar marcando la opción de primer orden.

El elemento básico de las fórmulas es el predicado, que es un símbolo de predicado aplicado a una serie de términos. Un predicado se representa mediante un identificador que empieza

por una letra minúscula cuyos argumentos se encuentran entre paréntesis separados por comas. Los términos pueden ser una constante, una variable o un símbolo de función aplicado a su vez a una serie de términos. Las constantes se representan mediante palabra que empiezan por una letra mayúscula, las variables mediante palabras que empiezan con una letra minúscula y los símbolos de función se representan igual que las constantes, pero llevan entre paréntesis a continuación sus argumentos separados por comas; se pueden considerar las constantes como símbolos de función de aridad 0. También se pueden emplear operadores como símbolo de predicado o función, siempre que se haga la declaración correspondiente en la sesión de Prolog en la que se ejecute el demostrador con anterioridad. Por una limitación del demostrador, no se puede utilizar el mismo nombre de variable en dos puntos distintos de una fórmula aunque estos pertenezcan a ámbitos diferentes.

Las fórmulas básicas de la lógica de primer orden son los símbolos de predicado. A partir de ellos, y mediante las conectivas y cuantificaciones habituales se pueden construir todas las posibles fórmulas. A partir de aquí, suponemos que f, f_1 y f_2 son fórmulas válidas de la lógica de primer orden.

La negación de la fórmula f se representa como $\sim f$ y será válida en aquellas interpretaciones en las que f no lo sea.

La disyunción de las fórmulas f_1 y f_2 se representa como $f_1 \# f_2$ y será válida en aquellas interpretaciones en las que al menos una de ellas sea válida.

La conjunción de las fórmulas f_1 y f_2 se representa como $f_1 \& f_2$ y será válida en aquellas interpretaciones en las que ambas fórmulas sean válidas.

La fórmula condicional $f_1 => f_2$es válida siempre que f_1 sea falsa o si f_2 es verdadera siendo f_1 verdadera. Es la implicación de la lógica de primer orden.

La fórmula bicondicional $f_1 <=> f_2$es válida siempre que f_1 y f_2 es sean verdaderas o falsas a la vez. Es la doble implicación o equivalencia de la lógica de primer orden.

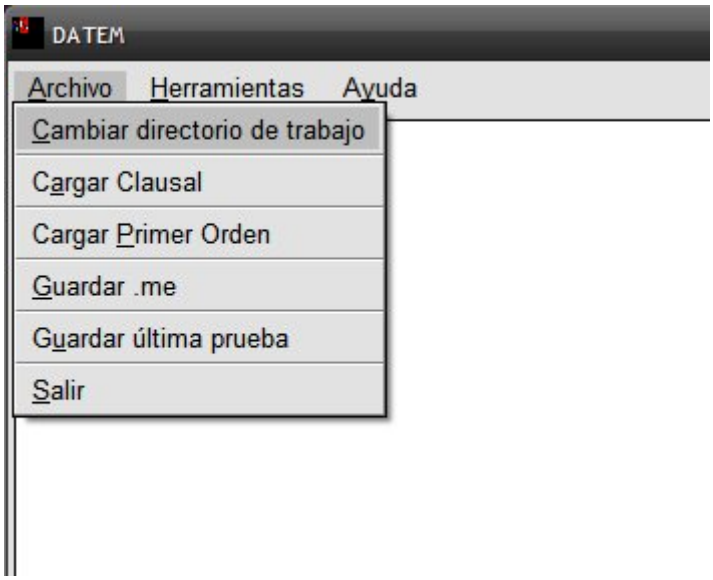
Además, las variables que aparecen en las fórmulas deben estar cuantificadas. Ello se consigue mediante los símbolos especiales de aridad 2 all y $exists$. Si se escribe $all(X, f)$ se están cuantificando universalmente todas las apariciones de la variable X en la fórmula f, mientras que si lo que se escribe es $exists(X, f)$, lo que se está haciendo es cuantificarlas de manera existencial. También se pueden cuantificar a la vez un conjunto de variables mediante las macros que se definen a continuación:

$all([X, Y, Z, \dots], f)$ equivale a $all(X, all(Y, all(Z, \dots, f)))$

$exists([X, Y, Z, \dots], f)$ equivale a $exists(X, all(Y, all(Z, \dots, f)))$

4. Manejo de la Interfaz

4.1. Manejo de Ficheros



Cambiar directorio de trabajo: Se muestra un explorador para seleccionar el directorio sobre el que se crearán los ficheros temporales necesarios para el correcto funcionamiento del programa. El usuario debe asegurarse de tener los privilegios suficientes para crear, leer y modificar ficheros en el directorio indicado.

Cargar clausal: Se muestra un explorador para abrir un fichero ***.me*** con texto en forma clausal. El contenido se mostrará automáticamente en el editor de texto superior de la interfaz principal. De no ser así, comprobar que el fichero cumple las reglas léxicas y sintácticas de la forma clausal.

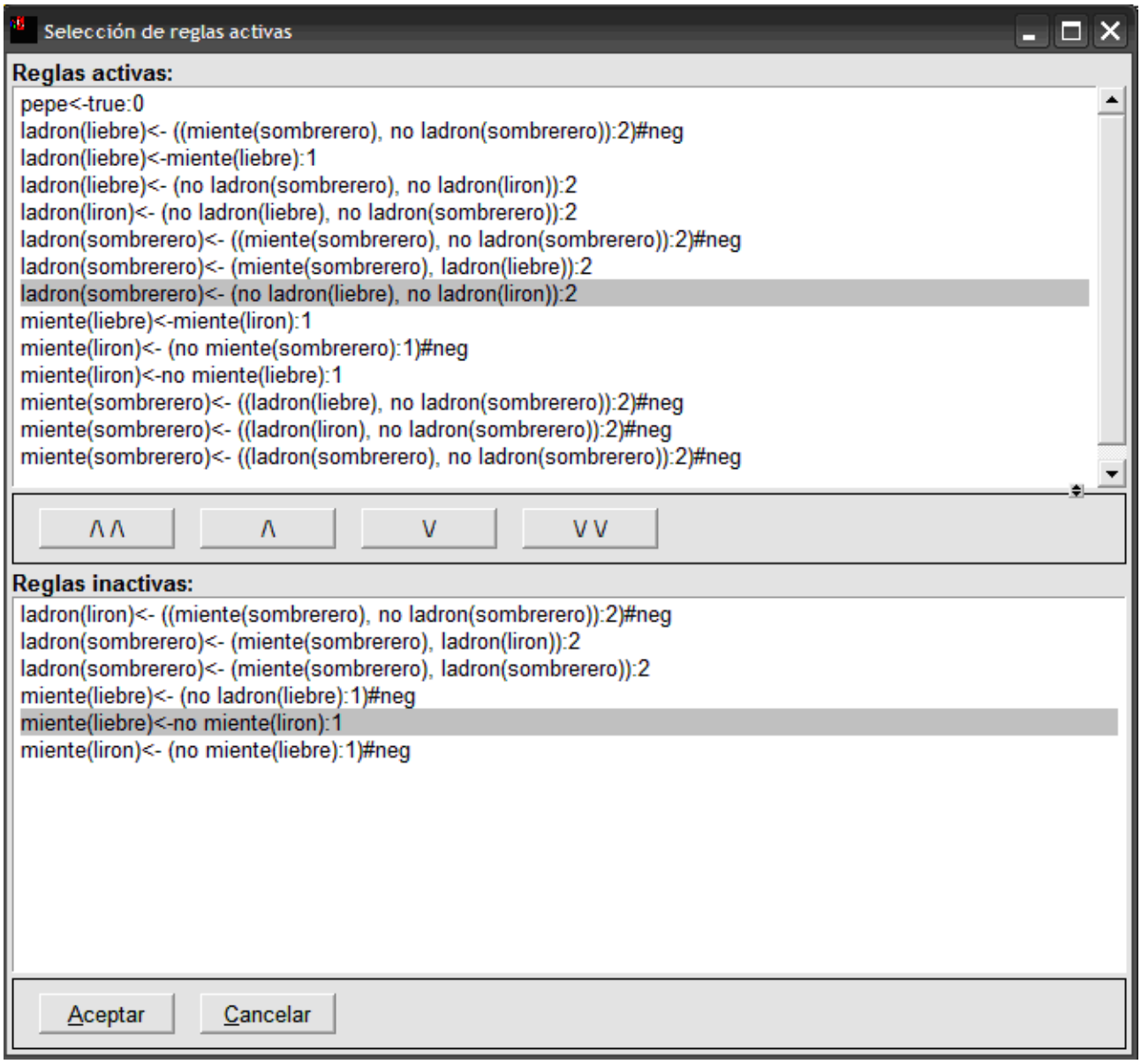
Cargar primer orden: Se muestra un explorador para abrir un fichero *.me* con texto en forma de primer orden. El contenido se mostrará automáticamente en el editor de texto superior de la interfaz principal. De no ser así, comprobar que el fichero cumple las reglas léxicas y sintácticas de la lógica de primer orden.

Guardar .me: Se muestra un explorador para guardar el contenido actual del editor superior de la interfaz principal en el fichero con extensión *.me* que se indique. El contenido se guarda directamente, sin realizar ninguna comprobación de su corrección léxica ni sintáctica.

Guardar última prueba: Se muestra un explorador para guardar el texto de salida generado en la última demostración; este texto puede contener la traza de la demostración, la prueba, ambas o ninguna, según estén o no activadas las opciones correspondientes.

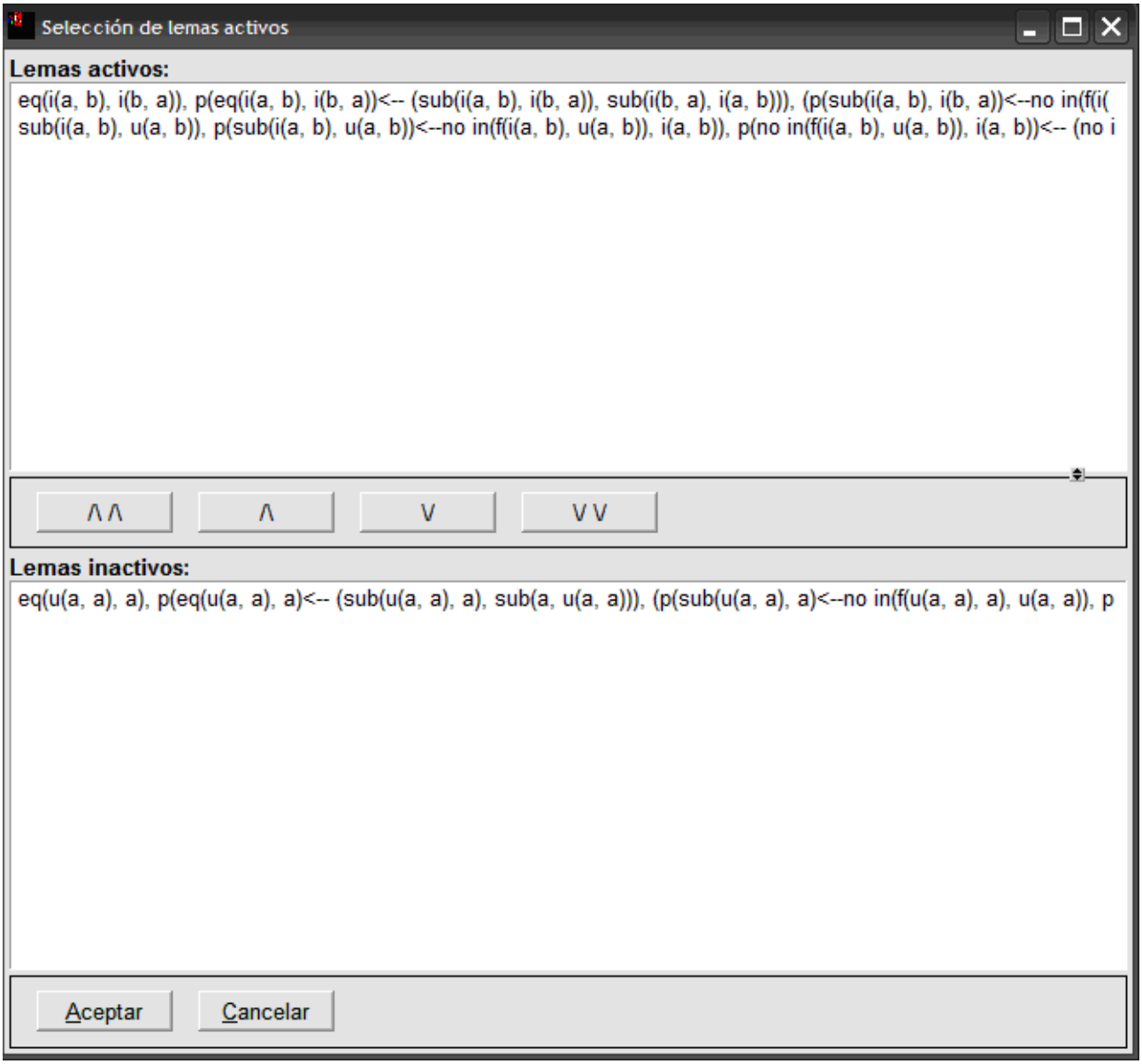
4.2. Elección de Reglas

Esta herramienta permite al usuario activar y desactivar, de una en una o todas a la vez, las reglas producidas en la carga del fichero *.me* actual, que se muestran en dos listas con elementos seleccionables.



4.3. Elección de Lemas

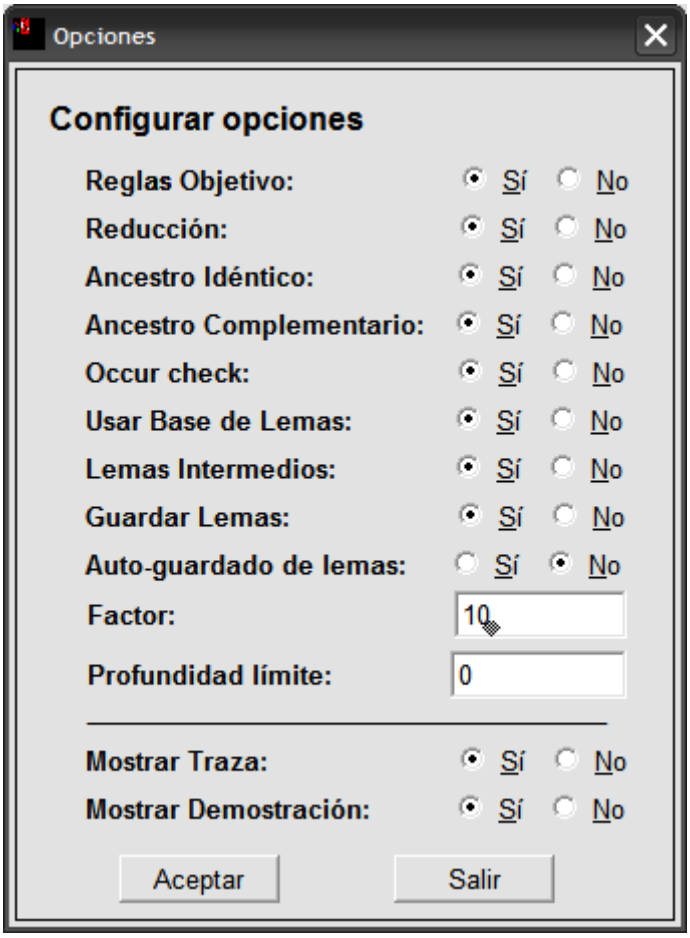
Esta herramienta permite al usuario activar y desactivar, de una en una o todas a la vez, los lemas persistentes producidos en las demostraciones relativas al fichero *.me* actual, que se muestran en dos listas con elementos seleccionables.



4.4. Configuración de Opciones

Permite activar o desactivar las opciones que afectan al comportamiento del demostrador y a la salida generada por éste.

Los parámetros factor y profundidad límite son valores numéricos enteros positivos.



5. Opciones

5.1. Mostrar Traza

Si esta opción está activada, durante la demostración de un objetivo se generará y se mostrará una traza informativa, que indica la profundidad, el tiempo y el número de pasos usado por el demostrador para esa profundidad. También se muestran los contadores que indican los distintos recursos utilizados en el proceso deductivo.

5.2. Mostrar Demostración

Si esta opción está activada, al finalizar la demostración de un objetivo se mostrará la información de las inferencias realizadas para concluir dicha demostración.

5.3. Occur Check

Si esta opción está activada, las unificaciones que se deriven del proceso de demostración de un objetivo se realizarán con occur check.

5.4. Factor

Este parámetro determina el número máximo de pasos que el demostrador utilizará para cada profundidad: *nº de pasos = profundidad * factor*.

5.5. Reglas del Objetivo

En el momento de iniciar la demostración de un objetivo dado C_1, C_2,…, C_n, se introducen las siguientes reglas, a las que llamamos reglas del objetivo por motivos obvios, dentro de la base teórica:

$$\neg C_1 \leftarrow C_2, C_2,\ldots, C_n$$

$$\neg C_2 \leftarrow C_1, C_3,\ldots, C_n$$

.

.

$$\neg C_n \leftarrow C_1, C_2,\ldots, C_n$$

5.6. Reducción

Esta opción indica si estarán habilitados o no los pasos de reducción.

5.7. Ancestros Idénticos

Si esta opción está activada, para cada subobjetivo expandido, se comprobará si existe algún ancestro suyo idéntico.

5.8. Ancestros Complementarios

Si esta opción está activada, para cada subobjetivo expandido, se comprobará si existe algún ancestro suyo complementario, esto es, su negación.

5.9. Lemas y Lemas Intermedios

Los lemas son demostraciones de objetivos realizadas anteriormente, y de las cuales se guarda toda la información útil. Su objetivo es evitar en lo posible la repetición de demostraciones para objetivos ya probados anteriormente. Se distinguen dos tipos de lemas:

- **Lemas persistentes**: se generan al finalizar la demostración de un objetivo introducido por el usuario, incluyendo información sobre ésta. Tal información se guarda en un fichero de lemas asociado al fichero de reglas **.me** cargado actualmente; este fichero se llama igual que el fichero de reglas, pero con la extensión *.lemma*.
- **Lemas intermedios**: se generan dinámicamente durante una demostración, guardando información de las demostraciones de subobjetivos ciertos. No se genera ninguna persistencia de ellos, siendo eliminados tras al finalizar el proceso de demostración principal.

5.9.1. Uso de la base de lemas

Indica si se usarán o no los lemas persistentes durante la demostración.

5.9.2. Guardado de lemas

Indica si se da la posibilidad al usuario de guardar los lemas de tipo persistente generados tras una demostración.

5.9.3. Auto-guardado de lemas

Indica si el guardado de lemas persistentes se produce automáticamente, sin pedir confirmación al usuario. Para ello, debe estar activada la opción de guardado de lemas.

5.9.4. Uso de lemas intermedios

Indica si se generarán y se usarán o no los lemas intermedios durante la demostración.

Apéndice B. Código fuente

meLoader.pl

```
/*
Compila todos los archivos del proyecto y ejecuta el programa
*/
:- compile(meUtils),
   compile(meClauses),
   compile(meSolve),
   compile(meFormMain),
   compile(meFormOptions),
   compile(meFormRules),
   compile(meFormLemmas),
   compile(meFormDirectories),
   compile(meLemmas),
   exec.
```

meSolve.pl

```
/*
Este fichero incluye todos los predicados que implementan los mecanismos de demostración
(método de eliminación de modelos y sus optimizaciones) y el mecanismo de búsqueda en
profundidad escalonada
*/

/*
solve2(+Goal,-Proof,+Rules,-Steps:Depth) <=> Goal es el objetivo a demostrar expresado
como fórmula de primer orden, Proof es la demostración, Rules es la lista de reglas en for-
ma clausal en la que se transforma Goal, Steps y Depth son el número exacto de pasos que
se ha requerido, en el caso de conseguirlo, para demostrar G y la profundidad máxima,
respectivamente
*/
solve2(Goal,Proof,Rules,Steps:Depth):-
      findall(rule(X,Y),rule(X,Y),Previous),
      calculate_rules(Goal => goal,Rules),
      insert_rules(Rules),
      solve(goal,Proof,_,Steps:Depth),
      define_rules(Previous).
```

```
/*
```
solve2(+Goal,-Proof,-Answer,-Steps:Depth) <=> Goal es el objetivo a demostrar expresado como fórmula de primer orden, Proof es la demostración, Answer es la lista de valores para los que se ha demostrado Goal, Steps y Depth son el número exacto de pasos que se ha requerido, en el caso de conseguirlo, para demostrar G y la profundidad máxima, respectivamente

```
*/
solve(Goal,Proof,Answer,Steps:Depth)
  :- reset,
     process_goal(Goal,V,N),
     show_trace_head,
     solve(Goal:N,Proof,[],V,Answer,1,Steps:Depth),
     write_steps(Steps),
     show_results(Proof,Answer).


/*
```
solve(+G,-P,+LA,+V,-R,+D,-I_F:DAux) <=> G es el objetivo a demostrar, P es la demos tración, LA es la lista de ancestros, V es la lista de variables usadas en G, R es una lista de valores para las variables de V que hacen cierto G, D es la diferencia entre la profundidad máxima de la búsqueda actual, la profundidad de G respecto al objetivo principal, I_F es el número exacto de pasos que se ha requerido, en el caso de conseguirlo, para demostrar G

```
*/
solve(G,P,LA,V,R,D,I_F:DAux):-
     write_trace(D),  % Profundidad,tiempo y contadores
     factor_is(FACTOR),
     I is D*FACTOR,   % I es el numero maximo de pasos
     solve(G,P,LA,V,R,D,I,F),
     write_trace_fin(D),
     DAux is D,
     I_F is I-F

     ;

     get_val(depth_limit,DL),
     (DL==0
      ->
      S is D+1,
      solve(G,P,LA,V,R,S,I_F:DAux)
      ;
      (D<DL
```

```
          ->
          S is D+1,
          solve(G,P,LA,V,R,S,I_F:DAux)
           ;
             show_depth_failure,
             assert(finished)
       )
      ).

/*
solve((+G,-P,+LA,+V,-R,+D,+I,-F) <=> G es el objetivo a demostrar, P es la demostración, LA es la lista de ancestros, V es la lista de variables usadas en G, R es una lista de valores para las variables de V que hacen cierto G, D es la diferencia entre la profundidad máxima de la búsqueda actual y la profundidad de G respecto al objetivo principal, I es el número de pasos disponibles para demostrar G y F es el número de pasos que han sobrado para demostrar G. G se resuelve por 'model elimination' en I - F pasos, y con profundidad no superior a D (D puede contemplarse como la iteración del operador a la que se encuentra la solución
*/
solve(true:_,nil,_,V,V,D,I,I)
   :- !,D >= 0,I >= 0.

solve((A,B):N,(PA,PB),LA,V,R,D,I,F)
   :- !,D > 0,I >= N,
      I1 is I-N+1,  % I1 es el No. maximo de pasos disponibles para A
      solve(A:1,PA,LA,V,V1,D,I1,M),
      M1 is M+N-1,  % M1 es el No. de pasos que quedan para B
      N1 is N-1,
      solve(B:N1,PB,LA,V1,R,D,M1,F).

solve(A:1,PA,LA,V,FV,D,I,F)
   :- D > 0,I > 0,
      solve_one(A,PA,LA,V,FV,D,I,F),
      S is I-F,
      insert_intermediate_lemma(A,PA,FV,D,S),!.

/*
solve_one(+A,-PA,+LA,+V,-FV,+D,+I,-F) <=> A es el objetivo simple a resolver, PA es la demostración de A, LA es la lista de ancestros de A, V es la lista de variables usadas en A, FV es una lista de valores para las variables de V que hacen cierto A, D es la diferencia entre la profundidad máxima de la búsqueda actual y la profundidad de A respecto al objetivo principal, I es el número de pasos disponibles para demostrar A y F es el número de
```

pasos que han sobrado para demostrar G. Se busca una demostración para A, usando la reducción y las distintas optimizaciones del método básico de eliminación de modelos desarrolladas (poda por ancestro idéntico y por ancestro idéntico complementario)
*/

```
solve_one(A,lemma(PA),_,_V,V1,D,I,F):-
      get_val(lemmas,on),
      lemma(A,PA,V1,D1,S),
      D >= D1,
      I >= S,
      F is I-S,
      ctr_inc(c5,_),!.

solve_one(A,PA ** il,_,_V,V1,D,I,F):-
      get_val(interlemmas,on),
      interlemma(A,PA,V1,D1,S),
      D >= D1,
      I >= S,
      F is I-S,
      ctr_inc(c6,_),!.

solve_one(A,_,LA,_,_,_,_,_)
   :-  identical_ancestor(A,LA),
       ctr_inc(c3,_),
       !,fail.

solve_one(A,reduced(A),LA,V,V,_D,I,F)
   :-  reduction(A,LA),
       ctr_inc(c2,_),F is I-1.

solve_one(A,p(A <-- B,PB),LA,V,FV,D,I,F)
   :- program_rule_occurs(A,B:N),
      \+ identical_comp_ancestor(A,LA),ctr_inc(c0,_),
      D1 is D-1,S is I-1,
      solve(B:N,PB,[A|LA],V,FV,D1,S,F).

solve_one(A,p(A <-- B ** g,PB),LA,V,FV,D,I,F)
   :- goal_rule_occurs(A,B:N,V1),
      \+ identical_comp_ancestor(A,LA),ctr_inc(c1,_),
      D1 is D-1,S is I-1,
      solve(B:N,PB,[A|LA],V or V1,FV,D1,S,F).
```

```
/*
reduction(+A,+LA) <=> A es un objetivo y LA es su lista de ancestros. Se realiza un paso de reducción, si está activado.
*/
reduction(A,LA)
   :- get_val(reduction,on),!,
      complementary_ancestor(A,LA).

/*
identical_ancestor(+A,+LA) <=> A es un objetivo y LA es su lista de ancestros. Se busca un ancestro idéntico de A
*/
identical_ancestor(A,LA)
   :- get_val(id_anc,on),!,
      strict_member(A,LA).

/*
identical_comp_ancestor(+A,+LA) <=> A es un objetivo y LA es su lista de ancestros. Se busca un ancestro complementario idéntico de A
*/
identical_comp_ancestor(A,LA)
   :- get_val(id_comp_anc,on),!,
      negation(A,No_A),
      strict_member(No_A,LA),
      ctr_inc(c4,_).

/*
complementary_ancestor(+A,+LA) <=> A es un objetivo y LA es su lista de ancestros. Se busca un ancestro complementario de A
*/
complementary_ancestor(A,LA)
   :- negation(A,No_A),
     %member(No_A,LA).
     member_occurs(No_A,LA).
```

meUtils.pl

```
/*
Este fichero incluye los predicados auxiliares que proporcionan las distintas utilidades que requiere el demostrador para el tratamiento previo de reglas y para la exposición de los resultados
*/
```

```
:- dynamic regobj/0,ctr/2,
        factor/1, depth_limit/1 , reduction/1,
        id_anc/1,id_comp_anc/1,goal_rules/1,proof/1,
        rule/2,regobj/2, inactive_rule/2,inactive_lemma/5,
        value/2,
        lemma/5,
        interlemma/5,
        finished/0,
        no_pop_ups/0 .


:- op(1005,xfx,'<-').
:- op(500,fy,no).
:- op(500,yfx,or).
:- op(505,xfx,'#').
:- op(505,xfx,'<--').
:- op(510,xfx,'**').

/*
Gestion de valores globales (para opciones, flags y contadores)
*/

set(X,Y) :- nb_setval(X,Y).
get_val(X,Y) :- nb_getval(X,Y).

/*
  Selección de opciones
  mediante el predicado set(+Option,+Status), donde Option puede ser:

  trace_on,
  proof,
  goal_rules,
  reduction,
  id_anc,
  id_comp_anc,
  factor,
  depth_limit,
  lemmas,
  save_lemmas,
  occurCheck,
  interlemmas,
```

```
  autolemmas

  y Status puede ser:

  on,
  off
*/

/*
  Lista de opciones y consulta de settings
*/
options([trace_on,proof,goal_rules,reduction,id_anc,id_comp_anc,factor,depth_limit,
        lemmas,save_lemmas,occurCheck,interlemmas,autolemmas]).
settings(S) :- setof((X,Y),(options(O),member(X,O),get_val(X,Y)),S).


/*
Opciones por defecto
*/
:- set(occurCheck,on).
:- set(lemmas,on).
:- set(save_lemmas,on).
:- set(interlemmas,on).
:- set(autolemmas,off).
:- set(trace_on,on).
:- set(proof,on).
:- set(goal_rules,on).
:- set(reduction,on).
:- set(id_anc,on).
:- set(id_comp_anc,on).
:- set(factor,10). % factor se usa para establecer el máximo numero de pasos N
                   % correspondiente a una profundidad máxima D
                   % N = D*factor
                   % Poner factor 1 es dejarlo al estilo mepret, sin tener en cuenta la prof. D

:- set(depth_limit,0).

/*
 Actualización de contadores
 Los contadores son:

 c0 --> numero de ancestro complementario
```

```
 c1 --> numero de ancestro complementario + Regla del objetivo
 c2 --> numero de reducción
 c3 --> numero de ancestro identico
 c4 --> numero de ancestro complementario idéntico
 c5 --> numero de lemas usados
 c6 --> numero de lemas intermedios usados
*/
ctr_inc(Counter,_)
   :- get_val(Counter,Value),
      Value1 is Value+1,
      set(Counter,Value1).
ctr_set(Counter,Value)
   :- set(Counter,Value).
ctr_is(Counter,Value)
   :- get_val(Counter,Value).

/*
Actualización del factor
*/
factor_is(F) :- get_val(factor,F).



/*
calculate_rules(+Goal,-Rules) <=> Goal es un fórmula de la lógica de primer orden y Rules es una lista con las reglas en forma clausal en las que se transforma dicha fórmula
*/
calculate_rules(Goal,Rules):-
      findall(rule(X,Y),rule(X,Y),Previous),
      retractall(rule(_,_)),
      working_directory(Dir,Dir),
      string_concat(Dir,'auxfile.me',FicheroAux),
      translate(Goal,Clauses),
      export_me(Clauses,FicheroAux),
      me_consult(FicheroAux),
      findall(rule(X,Y),rule(X,Y),Rules),
      define_rules(Previous).

/*
define_rules(+Rules) <=> Rules es una lista de reglas. Se eliminan todas las reglas rule de la base de Prolog y se insertan las contenidas en la lista anterior
*/
define_rules(Rules):-
```

```
        retractall(rule(_,_)),
        insert_rules(Rules).

/*
insert_rules(+Rules) <=> Inserta en la base de Prolog las reglas contenidas en la lista Rules
*/
insert_rules([]).
insert_rules([X|Xs]):-
        assert(X),
        insert_rules(Xs).


program_rule(no O,Body : N)
   :- rule(O,(Body : N) # neg).
program_rule(O,Body : N)
   :- \+ (O = no _),rule(O,Body : N).

goal_rule(no O,Body : N,Vars)
   :- get_val(goal_rules,on),
     regobj(O ,((Body : N) # neg) : Vars).
goal_rule(O,Body : N,Vars)
   :- get_val(goal_rules,on),
     \+ (O = no _),regobj(O , (Body : N) : Vars).

program_rule_occurs(no O,Body : N)
   :- rule(O1,(Body : N) # neg),unification(O,O1).
program_rule_occurs(O,Body : N)
   :- \+ (O = no _),rule(O1,Body : N),unification(O,O1).

goal_rule_occurs(no O,Body : N,Vars)
   :- get_val(goal_rules,on),
     regobj(O1 ,((Body : N) # neg) : Vars),unification(O,O1).
goal_rule_occurs(O,Body : N,Vars)
   :- get_val(goal_rules,on),
     \+ (O = no _),regobj(O1 , (Body : N) : Vars),unification(O,O1).

/* Ver reglas para manejo de los programas y objetivos */

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

```
strict_member(X,[Y|_]) :- X == Y,!.
strict_member(X,[_|L]) :- strict_member(X,L).

member_occurs(X,[Y|_]) :- unification(X,Y).
member_occurs(X,[_|L]) :- member_occurs(X,L).


/*
me_consult(+L) <=> L es una lista de nombres de ficheros. Carga las reglas de los ficheros de la lista
*/
me_consult([]) :- !.
me_consult([File1|L])
  :-  me_consult(File1),!,
      me_consult(L).

/*
me_consult(+File) <=> File es un nombres de fichero. Carga las reglas del fichero
*/
me_consult(File)
  :- op(1005,xfx,<-),
     op(500,fy,no),
     open(File,read,H),
     repeat,
     read(H,Rule),
     process_rule(Rule),
     Rule = end_of_file,
     close(H).


/*
process_rule(+Term) <=> Carga de la reglaque representa Term, leído desde un fichero. Utiliza el predicado auxiliar process_rule/3

  Las reglas son de la forma
  Cabeza <- Cuerpo. ,   o bien
  Cabeza.

  Cabeza y Cuerpo pueden ser de la forma
  Lit1,...,LitN,  siendo LitK un  tomo o un  tomo negado, con
  la forma  no Atomo.
```

```
   Cada regla da lugar a tantas reglas estilo Prolog como literales
   contenga. Estas reglas se almacenan con la forma
   Atom :- (Lit1,...,LitN) : N   o bien
   Atom :- (Lit1,...,LitN) : N # neg, en reglas para 'no Atom'
*/

process_rule(end_of_file) :- !.
process_rule((Head <- Body))
    :- !,listlits(Head,Headl),
       listlits(Body,Bodyl),
       length(Headl,N1),
       length(Bodyl,N2),
       N is N1+N2-1,
       process_rule(Headl,Bodyl,N).
process_rule(Head)
    :- listlits(Head,Headl),
       length(Headl,N),
       M is N-1,
       process_rule(Headl,[],M).

process_rule(Headl,Bodyl,N)
    :- delete(Lit,Headl,Rest),
       negation_l(Rest,NegRest),
       append(NegRest,Bodyl,FinalBodyl),
       assert_rule(Lit,FinalBodyl,N),
       fail.  % fuerza el backtracking

process_rule(Headl,Bodyl,N)
    :- delete(Lit,Bodyl,Rest),
       negation(Lit,NegLit),
       negation_l(Headl,NegHeadl),
       append(NegHeadl,Rest,FinalBodyl),
       assert_rule(NegLit,FinalBodyl,N),
       fail.  % fuerza el backtracking

process_rule(_,_).


/*
assert_rule(+Head,+Body,+N) <=> Se inserta en la base de Prolog la regla con cabeza
Head, cuerpo Body y de N literales
*/
```

```
assert_rule(no Head,[],0)
    :- !,assert(rule(Head,(true : 0) # neg)).
assert_rule(Head,[],0)
    :- !,assert(rule(Head,true : 0)).
assert_rule(no Head,Bodyl,N)
   :- !,listlits(Body,Bodyl),
      assert(rule(Head,(Body : N) # neg)).
assert_rule(Head,Bodyl,N)
   :- listlits(Body,Bodyl),
      assert(rule(Head,Body : N)).

/*
 listlits(?A,?B) <=> A es (T1,T2,...,TN) y B es [T1,T2,...,TN]
*/

listlits(Literal,[Literal])
   :- \+ (Literal = (_,_)).
listlits((Literal,Rest),[Literal|Restl])
   :- listlits(Rest,Restl).

negation(no Lit,Lit) :- !.
negation(Lit,no Lit).

negation_l([],[]).
negation_l([X|Y],[NX|NY])
   :- negation(X,NX),
      negation_l(Y,NY).

append([],L,L).
append([X|R],L,[X|L1]) :- append(R,L,L1).

delete(X,[X|L],L).
delete(X,[Y|L],[Y|L1]) :- delete(X,L,L1).


/*
 GOAL RULES
 Las reglas que provienen del objetivo se almacenan como terminos Prolog bajo la clave
'regobj'. Se crean tantas regla como literales tenga el objetivo.
Los terminos son de la forma
    (Atom :- (Lit1,...,LitN) : N) : Variables, o bien
    (Atom :- (Lit1,...,LitN) : N # neg) : Variables, siendo
```

Variables una lista de variables Prolog identicas a las que aparezcan en la regla. Es ecesario para construir respuestas disjuntivas.
*/

```
process_goal(Goal,Vars,N)
  :- variables(Goal,Vars),
     listlits(Goal,Goall),
     length(Goall,N),
     M is N-1,
     assert_goal_rules(Goall,Vars,M).

assert_goal_rules(Goall,Vars,M)
  :- get_val(goal_rules,on),
     delete(Literal,Goall,Bodyl),
     negation(Literal,NegLit),
     assert_one_goal_rule(NegLit,Bodyl,Vars,M),
     fail.

assert_goal_rules(_,_,_).

assert_one_goal_rule(no Atom ,[],Vars,0)
  :- !,assert(regobj(Atom , ((true:0) # neg):Vars)).
assert_one_goal_rule(Atom ,[],Vars,0)
  :- !,assert(regobj(Atom , (true:0):Vars)).

assert_one_goal_rule(no Atom,Bodyl,Vars,M)
  :- !,listlits(Body,Bodyl),
     assert(regobj(Atom, ((Body:M) # neg):Vars)).
assert_one_goal_rule(Atom,Bodyl,Vars,M)
  :- listlits(Body,Bodyl),
     assert(regobj(Atom,(Body:M):Vars)).

/*
variables(+Goal,-Vars) <=>  Vars es una lista con las variables contenidas en el objetivo Goal
*/
variables(Goal,Vars)
  :- setof(X,appears_var(X,Goal),Vars),!.
variables(_Goal,[]).

appears_var(X,Goal):- Goal =.. [_|Args],
                      appears_var_l(X,Args).
```

```
appears_var_l(X,[X|_]) :- var(X),!.
appears_var_l(X,[Y|_]) :- \+ (atomic(Y)),appears_var(X,Y).
appears_var_l(X,[_|R]) :- appears_var_l(X,R).

/*
```
reset <=> Reinicia la base de lemas intermedios, elimina las reglas del objetivo generadas y reinicia los contadores.
```
*/
reset :- retractall(interlemma(_,_,_,_,_)),
        erase_goal_rules,
        reset_time,
        ctr_set(c0,0),
        ctr_set(c1,0),
        ctr_set(c2,0),
        ctr_set(c3,0),
        ctr_set(c4,0),
        ctr_set(c5,0),
        ctr_set(c6,0).

/*
```
erase_goal_rules <=> Elimina las reglas del obejtivo de la base de Prolog
```
*/
erase_goal_rules :- abolish(regobj/2).

/*
```
reset_time <=> Reinicia el contador de tiempo
```
*/
reset_time :- statistics(cputime,T),ctr_set(time,T).

/*
```
reset_rules <=> Elimina todas las reglas cargadas (rule/2) de la base de Prolog
```
*/
reset_rules :- abolish(rule/2).

/*
```
write_trace(+Ini) <=> Se muestra por pantalla la traza con los datos asociados con cada escalón de profundidad de la búsqueda:

|*Prof.* |*Tiempo(s)*|*Anc. Comp.*|*Anc. Comp.*|*Reducciones*|*Anc. Id.*|*Anc.Comp.*|*Lemas*|*Lemas*|
| | | |*+ Reg. Obj*| | |*Id.* | |*Inter* |

Son, por orden:


```
```

- La profundidad máxima actual
- El tiempo usado en la búsqueda del escalón actual, en segundos
- Número de podas por ancestro complementario
- Número de podas por ancestro complementario junto con reglas del objetivo
- Número de pasos de reducción
- Número de podas por ancestro idéntico
- Número de podas por ancestro complementario idéntico
- Número de veces que se ha usado un lema
- Número de veces que se ha usado un lema intermedio
*/

```
write_trace(1):-!.

write_trace(Ini):-
        get_val(trace_on,on),!,
        show_string('\n'),
        Ini2 is Ini-1,
        write_depth(Ini2),
        show_string('\t'),
        time,
        show_string('\t'),
        counters,
        nl.
write_trace(_).

write_trace_fin(D):-
                D1 is D+1,
                write_trace(D1).
```

/*
show_trace_head :-Muestra por pantalla la cabecera informativa de la traza
*/

```
show_trace_head:-
               get_val(trace_on,T),
               (T == on
                ->
                  show_string('\n ------------------------------------------------------------------------'),
                  show_string('\n|Prof\t|Tiempo\t|Anc.\t|Anc.Com|Reduc\t|Anc\t
                                |Anc.Com|Lemas\t|Lemas\t|'),
                  show_string('\n|    \t|(s)   \t|Comp\t|+Reg.Ob|      \t|Id. \t|Id.    |     \t|Inter\t|'),
                  show_string('\n ------------------------------------------------------------------------')
                  ;
```

```
                true
               ).

/*
write_depth(+Ini) <=> Muestra por pantalla la profundidad Ini
*/
write_depth(Ini)
     :- show_string('  '),
        show_string(Ini).

/*
write_steps(+Steps) <=> Muestra por pantalla el número de pasos Steps
*/
write_steps(Steps):-
      (finished
       ->
        true
       ;
        show_string('\n\n'),
        show_string(Steps),
        show_string(' pasos.'),
        show_string('\n')
      ).

/*
show_results(+Proof,+Answer) <=> Proof es una demostración y Answer es una lista de
valores para los que se consigue la demostración. Se muestran los resultados de una de-
mostración por pantalla.
*/
show_results(Proof,Answer):-
      (finished
         ->
           retract(finished),
           assert(no_pop_ups)
       ;
          write_proof(Proof),
      write_answer(Answer)
      ).

/*
show_depth_failure <=> Se muestra el mensaje informativo de fin de demostración fallida
debido a haberse alcanzado la profundidad límite
```

```
*/
show_depth_failure:-
        show_string('\n\n       --- Se ha alcanzado la profundidad máxima alcanzada ---'),
        show_string('\n                 --- La demostración ha fallado ---'),
        show_string('\n_____________________________________________________________
                        _____________________________\n').

/*
write_proof(+P) <=> Muestra por pantalla el contenido de la demostración P, con las inferencias realizadas
*/
write_proof(P)
   :- get_val(proof,on),
     !,
     show_string('\n*** DEMOSTRACIÓN ***\n'),
     pretty_print(P,0).
write_proof(_).

/*
write_answer(+Answer) <=> Answer es una lista de valores para los que se consigue una demostración. Se muestran pantalla.
*/
write_answer(Answer):-
        show_string('Valores de las variables = '),show_string(Answer),
        show_string('\n_____________________________________________________
                        _____________________________________\n').

/*
pretty_print(+Term,+Tab) <=> Escritura estructurada de los resultados de las demostraciones Tab es el número de tabulaciones con las que se inicia la línea de texto
*/
pretty_print(nil,_):-!.
pretty_print((A,B),Tab):-
                !,
                pretty_print(A,Tab),
                pretty_print(B,Tab).
pretty_print(p(A,B),Tab):-
                !,
                pretty_print(A,Tab),
                Tab1 is Tab + 2,
                pretty_print(B,Tab1).
pretty_print(A <-- B ** g,Tab):-
```

```
                !,
                spaces(Tab),
                show_string((A <- B)),
                show_string('  *** REGLA DEL OBJETIVO\n').
pretty_print(A <-- B,Tab):-
                !,
                spaces(Tab),
                show_string((A <- B)),
                show_string('\n').
pretty_print(reduced(A),Tab):-
                !,
                spaces(Tab),
                show_string(A),
                show_string(' *** REDUCCIÓN\n').
pretty_print(lemma(A),Tab):-
                !,
                spaces(Tab),
                show_string(' *** LEMA\n'),
                pretty_print(A,Tab).
pretty_print(A ** il,Tab):-
                !,
                spaces(Tab),
                show_string(' *** LEMA INTERMEDIO\n'),
                pretty_print(A,Tab).
pretty_print(A,Tab):-
                !,
                spaces(Tab),
                show_string(A),
                show_string('\n').

/*
spaces(+N) <=> Se muestra un número N de espacios en blanco
*/
spaces(0) :- !.
spaces(N) :- N > 0, show_string(' '),N1 is N-1,spaces(N1).

/*
time <=> Se actualiza el contador de tiempo y se muestra su valor por pantalla
*/
time :-
        statistics(cputime,T),
        ctr_is(time,T1),
```

```
        Time is T-T1,
        precision(3,Time,Time1),
        show_string(Time1).

/*
precision(+N,+R,-R4) <=> Expresa el número real R con una precisión de N decimales, el
nuevo número es R4.
*/
precision(N,R,R4):-
            N1 is 10^N,
            R2 is R*N1,  R3 is floor(R2),
            R4 is R3/N1.

/*
counters <=> Se muestran los contadores por pantalla
*/
counters
    :- ctr_is(c0,N0),show_string(N0),show_string('\t'),
       ctr_is(c1,N1),show_string(N1),show_string('\t'),
       ctr_is(c2,N2),show_string(N2),show_string('\t'),
       ctr_is(c3,N3),show_string(N3),show_string('\t'),
       ctr_is(c4,N4),show_string(N4),show_string('\t'),
       ctr_is(c5,N5),show_string(N5),show_string('\t'),
       ctr_is(c6,N6),show_string(N6).

/*
unification(?O,?O1) <=> Se comprueba si está activada la unificación con occur check. Se
realiza la unificación de O y O1 en consecuencia.
*/
unification(O,O1):-
                    get_val(occurCheck,OC),
                    (OC == on ->
                       unify_with_occurs_check(O,O1)
                     ;
                       O = O1
                    ).
```

meLemmas.pl

```
/*
Se implementan todas las utilidades necesarias para el manejo de lemas persistentes y le-
mas intermedios
```

```
*/

/*
save_lemmas(+Mode) <=> Mode puede ser on u off. En el primer caso, se insertan en un
fichero los lemas en forma clausal. En el segundo, en forma de la lógica de primer orden.
*/
save_lemmas(on) :-
        nb_getval(current_goal,G),
        nb_getval(current_proof,P),
        nb_getval(current_answer,A),
        listlits(G,Aux),
        listlits(P,AuxP),
        insert_lemmas(Aux,AuxP,A).

save_lemmas(off) :-
        nb_getval(current_goal,G),
        nb_getval(current_proof,P),
        nb_getval(current_rules,R),
        insert_lemmasFO(G,P,R).

/*
insert_lemmas(+G,+P,+A) <=> G es un objetivo en forma clausal, P una demostración de
G y A la lista de valores que hacen válida la demostración P. Inserta los lemas correspon-
dientes a G, P y A, de literal en literal de G. Utiliza la función auxiliar in-
sert_lemma(+G,+P,+A,+D,+N), que escribe cada lema en el fichero correspondiente.
*/
insert_lemmas([],_,_).
insert_lemmas([X|Xs],[P|Ps],A):-
        calculate_parameters(P,D,S),
        insert_lemma(X,P,A,D,S),
        insert_lemmas(Xs,Ps,A).

insert_lemma(_,_,_ or _,_,_):-!.
insert_lemma(G,_,_,_,_):-lemma(G,_,_,_,_),!.
insert_lemma(G,P,A,D,N):-assert(lemma(G,P,A,D,N)),
                nb_getval(current_file_name,FileName),
                generate_lemmas_file(FileName,FileLemma),
                open(FileLemma,append,Stream),
                write(Stream,lema(G,P,A,D,N)),
                write(Stream,'.'),
                write(Stream,'\n'),
                close(Stream).
```

```
/*
```
insert_lemmasFO(+G,+P,+R) <=> G es un objetivo expresado como fórmula de la lógica de la lógica de primer orden, P es su demostración y R son las reglas en las que se transforma G. Se insertan los lemas correspondientes en el fichero de lemas asociado

```
*/
insert_lemmasFO(G,P,R):-
        nb_getval(current_file_name,FileName),
        generate_lemmas_file(FileName,FileLemma),
        open(FileLemma,append,Stream),
        write(Stream,'%'),
        write(Stream,lemaPO(G,P,R)),
        write(Stream,'.'),
        write(Stream,'\n'),
        close(Stream).

/*
```
calculate_parameters(+P,-D,-S) <=> Obtiene la profundidad D y el número de pasos S de una demostración P

```
*/
calculate_parameters((A,B),D,S):-
    calculate_parameters(A,D1,S1),calculate_parameters(B,D2,S2),D is max(D1,D2),S is S1+S2.
calculate_parameters(p(_<-- true,nil),1,0).
calculate_parameters(p(_<--_,B),D,S):-
    calculate_parameters(B,D1,S1),D is D1+1,S is S1+1.
calculate_parameters((p(_<--_,B)),D,S):-
    calculate_parameters(B,D1,S1),D is D1+1,S is S1+1.
calculate_parameters(lemma(A),D,S):-
    calculate_parameters(A,D,S).
calculate_parameters(A ** il,D,S):-
    calculate_parameters(A,D,S).
calculate_parameters(_,1,1).
calculate_parameters(no _,1,1).


/*
```
load_lemmas(+FileLemma) <=> Carga los lemas contenidos en el fichero de lemas de nombre FileLemma. Se insertan en la base de Prolog.

```
*/
load_lemmas(FileLemma):-
    retractall(lemma(_,_,_,_,_)),
```

```
        retractall(inactive_lemma(_,_,_,_,_)),
        open(FileLemma,read,Str),
        repeat,
        read(Str,Lemma),
        process_lemma(Lemma),
        Lemma = end_of_file,
        close(Str).

/*
process_lemma(+Term) <=> Inserta un lema leído de un fichero en la base de lemas; esto
es, se inserta en la base de Prolog un término de la forma lema(G,P,A,D,N),siendo G el
objetivo demostrado, P una demostración de G, A la lista de valores de las variable de G
para los que se consigue la demostración P y D y N la profundidad y el número de pasos
que se necesitaron.
*/
process_lemma(end_of_file):-!.
process_lemma((lema(G,P,A,D,N))):-assert(lemma(G,P,A,D,N)).


/*
generate_lemmas_file(+FileName,-FileLemma) <=> FileLemma es el archivo de Lemas
(de extensión *.lemma) asociado al archivo FileName (de extensión *.me). Si no existe ya
FileLemma, se crea.
*/
generate_lemmas_file(FileName,FileLemma):-
                string_length(FileName,N),
                NAux is N-2,
                sub_string(FileName,0,NAux,2,Aux),
                string_concat(Aux,'lemma',FileLemma),
                open(FileLemma,update,Stream),
                close(Stream).

/*
insert_intermediate_lemma(+G,+P,+A,+D,+S) <=> Inserta en tiempo de ejecición de la-
demostración un lema intermedio en la base de lemas intermedios (se aserta en la base de
prolog un término de la forma interlemma(G,P,A,D,S), siendo G el subobjetivo demostrado,
P una demostración de G, A la lista de valores de las variable de G para los que se consi-
gue la demostración P y D y N la profundidad y el número de pasos que se necesitaron).
Evita repetición de Lemas.
*/
insert_intermediate_lemma(G,_,A,D,_):-
        interlemma(G,_,A,D1,_),D>=D1.
```

```
insert_intermediate_lemma(G,P,A,D,S):-
        interlemma(G,_,A,D1,_),D1>D,assert(interlemma(G,P,A,D,S)),!.
insert_intermediate_lemma(G,P,A,D,S):-
        assert(interlemma(G,P,A,D,S)),!.
```

meClauses.pl

```
/*
```
Fichero que permite manejar fórmulas en lógica de primer orden y proporciona un predicado para transformarlas a forma clausal, así como otra serie de utilidades. La sintaxis de estas fórmulas es la siguiente:
Si P y Q son fórmulas, entonces ~P, P # Q, P & Q, P => Q y P <=> Q son fórmulas que representan respectivamente la negación, la disyunción, la conjunción, la implicación y la doble implicación, todas ellas definidas de forma estándar según la semántica habitual de la lógica de primer orden. Se puede cuantificar una variable X de manera existencial en una fórmula F mediante la sintaxis exists(X,F), mientras que para cuantificarla universalmente se escribe all(X,F). Existen macros que permiten cuantificar varias variables a la vez; por ejemplo, para cuantificar universalmente las variables X, Y y Z en la formula F habría que escribir all([X,Y,Z],F). Las fórmulas básicas son cualquier símbolo de predicado aplicado a una serie de términos, escritos al estilo de Prolog.
```
*/

/*
*Operadores lógicos*
*/
:- op(200,fx,~).     %negación
:- op(400,xfy,#).    %disyunción
:- op(400,xfy,&).    %conjunción
:- op(700,xfy,=>).   %implicación
:- op(700,xfy,<=>).  %doble implicación

/***********
```
translate(+X,-Clauses) <=> Al transformar la fórmula en lógica de primer orden X a forma clausal se obtiene la lista de cláusulas Clauses. El proceso se lleva a cabo secuencialmente, partiendo de una fórmula escrita en sintaxis de primer orden y con las macros sintácticas expandidas y dando como resultado la mencionada lista de cláusulas.
```
**********/
translate(X,Clauses) :-
        expand(X,Xe),
        implout(Xe,X1),
        negin(X1,X2),
        skolem(X2,X3),
```

```
        univout(X3,X4),
        conjn(X4,X5),
        clausify(X5,[],Clauses).

/***********
```
expand(+X,-Xt) <=> Xt es la fórmula resultante de expandir las macros sintácticas contenidas en la fórmula X. Las macros permitidas son:
all([X,Y,Z,...],F) equivale a all(X,all(Y,all(Z,...,F)))
exists([X,Y,Z,...],F) equivale a exists(X,all(Y,all(Z,...,F)))
```
***********/
expand(X<=>Y,X1<=>Y1) :-
      !,
      expand(X,X1),
      expand(Y,Y1).

expand(X=>Y,X1=>Y1) :-
      !,
      expand(X,X1),
      expand(Y,Y1).

expand(X&Y,X1&Y1) :-
      !,
      expand(X,X1),
      expand(Y,Y1).

expand(X#Y,X1#Y1) :-
      !,
      expand(X,X1),
      expand(Y,Y1).

expand(~X,~X1) :-
      expand(X,X1).

expand(all(X,B),all(X,B1)) :-
      var(X),
      !,
      expand(B,B1).

expand(all([X|Xs],B),all(X,B1)) :-
      var(X),
      !,
      expand(all(Xs,B),B1).
```

```
expand(all([],B),B1) :-
      !,
      expand(B,B1).

expand(exists(X,B),exists(X,B1)) :-
      var(X),
      !,
      expand(B,B1).

expand(exists([X|Xs],B),exists(X,B1)) :-
      var(X),
      !,
      expand(exists(Xs,B),B1).

expand(exists([],B),B1) :-
      !,
      expand(B,B1).

expand(X,X):-
      term(X).

/***********
```
term(+X) <=> X es un término correcto, sin apariciones de los símbolor reservados para conectivas y cuantificaciones.
```
**********/
term(X):-
      var(X),
      !.


term(X):-
      atomic(X),
      !.

term(X):-
      functor(X,Y,_),
      Y \== all,
      Y \== exists,
      X \== (<=>),
      X \== (=>),
      X \== (&),
```

```
      X \== (#),
      X \== (~),
      X =.. [_|Args],
      term_list(Args).

/***********
term_list(+X) <=> X es una lista de términos bien formados.
**********/
term_list([]).

term_list([X|Xs]):-
       term(X),
       term_list(Xs).


/***********
implout(+F,-F1) <=> F1 es la fórmula resultante de eliminar las implicaciones de la fórmula F.
Se utilizan las siguientes leyes de equivalencia lógica:
P <=> Q es equivalente a (P & Q) # (~P & ~Q)
P => Q es equivalente a ~P # Q
**********/

implout((P<=>Q),((P1&Q1)#(~P1 & ~Q1))):-
       !,
       implout(P,P1),
       implout(Q,Q1).

implout((P=>Q),(~P1#Q1)):-
       !,
       implout(P,P1),
         implout(Q,Q1).

implout(all(X,P),all(X,P1)):-
     !,
     implout(P,P1).

implout(exists(X,P),exists(X,P1)):-
       !,
       implout(P,P1).

implout((P & Q),(P1&Q1)) :-
```

```
        !,
        implout(P,P1),
        implout(Q,Q1).

implout((P#Q),(P1#Q1)) :-
        !,
        implout(P,P1),
        implout(Q,Q1).

implout((~P),(~P1)) :-
        !,
        implout(P,P1).

implout(P,P).

/***********
```
neg y negin meten las negaciones lo mas dentro posible de la formula.
negin recorre recursivamente la fórmula, introduciendo las negaciones hacia el interior.
neg
```
***********/
negin((~P),P1):-
     !,
     neg(P,P1).

negin(all(X,P),all(X,P1)):-
     !,
     negin(P,P1).

negin(exists(X,P),exists(X,P1)):-
     !,
     negin(P,P1).

negin((P & Q),(P1 & Q1)) :-
     !,
     negin(P,P1),
     negin(Q,Q1).

negin((P # Q),(P1 # Q1)) :-
     !,
     negin(P,P1),
     negin(Q,Q1).
```

```
negin(P,P).

/***********
```
Niega una fórmula e introduce las negaciones con llamadas recursivas a negin.
```
***********/
neg((~P),P1):-
   !,
   negin(P,P1).

neg(all(X,P),exists(X,P1)):-
   !,
   neg(P,P1).

neg(exists(X,P),all(X,P1)):-
   !,
   neg(P,P1).

neg((P & Q),(P1 # Q1)):-
   !,
   neg(P,P1),
   neg(Q,Q1).

neg((P # Q),(P1 & Q1)):-
   !,
   neg(P,P1),
   neg(Q,Q1).

neg(P,(~P)).

/**********
```
skolem(+X,-X1) <=> X1 es la fórmula resultante de eliminar las cuantificaciones existenciales de la fórmula X medianteel proceso de skolemización. La fórmula X no tiene condicionales y tiene las negaciones introducidas dentro. Para completar el proceso, se utiliza el predicado auxiliar skolem1, que lleva cuenta de las variables encontradas para generar las constantes de Skolem correctas.
```
***********/
skolem(X,X1):-
    skolem1(X,X1,[]).

skolem1(X,_,_):-
     var(X),
     !.
```

```
skolem1(all(X,P),all(X,P1),Vars):-
      !,
      skolem1(P,P1,[X|Vars]).

skolem1(exists(X,P),P2,Vars):-
      !,
      gen_sym(f,F),
      Sk =.. [F|Vars],
      %subst(X,Sk,P,P1),
      X = Sk,
      skolem1(P,P2,Vars).

skolem1((P # Q), (P1 # Q1), Vars) :-
      !,
      skolem1(P,P1,Vars),
      skolem1(Q,Q1,Vars).

skolem1((P & Q), (P1 & Q1),Vars):-
      !,
      skolem1(P,P1,Vars),
      skolem1(Q,Q1,Vars).

skolem1(P,P,_).

/***********
```

gen_sym(+Root,-Atom) <=> Atom es una constante de Skolem generada a partir de una raiz Root, añadiendo un nuevo número a la raiz en cada llamada.

```
***********/
gen_sym(Root,Atom):-
      get_num(Root,Num),
      atom_chars(Root, Name1),
      number_chars(Num,Name2),
      append(Name1,Name2,Name),
      atom_chars(Atom,Name).

/***********
```

get_num(Root,Num) <=> Num es el número actual para la raíz Root.

```
***********/
get_num(Root,Num):-
       retract(current_num(Root,Num1)),
       !,
       Num is Num1 + 1,
```

```
        asserta(current_num(Root,Num)).

get_num(Root,1) :- asserta(current_num(Root,1)).

/************
```
univout(+X,-X1) <=> X1 es la fórmula resultante de eliminar las cuantificaciones universales de la fórmula X. Como se está pasando la fórmula a forma clausal y allí todas las variables están cuantificadas universalmente de manera implícita, basta con eliminar los predicados correspondientes a las cuantificaciones universales.
```
***********/
univout(all(_,P),P1):-
        !,
        univout(P,P1).

univout((P & Q),(P1 & Q1)) :-
        !,
        univout(P,P1),
        univout(Q,Q1).

univout((P # Q),(P1 # Q1)) :-
        !,
        univout(P,P1),
        univout(Q,Q1).

univout(P,P).

/************
```
conjn(X,Xt) <=> Xt es la fórmula resultante de distribuir la & respecto de la #. El predicado auxiliar conjn1 es el encargado de aplicar las reglas de la equivalencia lógica necesarias para llevar a cabo este paso.
```
***********/
conjn((P # Q),R):-
    !,
    conjn(P,P1),
    conjn(Q,Q1),
    conjn1((P1 # Q1),R).

conjn((P & Q),(P1 & Q1)):-
    !,
    conjn(P,P1),
    conjn(Q,Q1).
```

```
conjn(P,P).

conjn1(((P & Q) # R),(P1 & Q1)) :-
      !,
      conjn((P # R), P1),
      conjn((Q # R), Q1).

conjn1((P # (Q & R)),(P1 & Q1)):-
      !,
      conjn((P # Q),P1),
      conjn((P # R),Q1).

conjn1(P,P).

/***********
```
clausify(F,L1,L2) <=> L2 es la lista resultante de añadir a la lista L1 las clausulas obtenidas de F. Genera clausulas de la forma cl([A],[B]). [A] es la lista de literales positivos y [B] la de los negativos.
```
***********/
clausify((P & Q),C1,C2):-
       !,
       clausify(P,C1,C3),
       clausify(Q,C3,C2).

clausify(P,Cs,[cl(A,B)|Cs]):-
      inclause(P,A,[],B,[]),
      !.

clausify(_,C,C).

/***********
```
inclause(F,A1,A,B1,B) <=> A1 es la lista de literales positivos y B1 la lista de literales negativos resultante de extraer los literales de F, siendo A los literales positivos extraídos anteriormente y B los negativos.
```
***********/
inclause((_&_),_,_,_) :- write('Fallo').

inclause((P#Q),A,A1,B,B1):-
       !,
       inclause(P,A2,A1,B2,B1),
       inclause(Q,A,A2,B,B2).
```

```
inclause((~P),A,A,B1,B):-
        notin(P,B),
        !,
        putin(P,B,B1).

inclause(P,A1,A,B,B):-
        notin(P,A),
        !,
        putin(P,A,A1).

inclause(_,A,A,B,B).

/***********
notin(X,L) <=> X no es un elemento de L.
***********/
notin(X,[Y|_]) :-
    X==Y,
    !,
    fail.

notin(X,[_|L]):-
    !,
    notin(X,L).

notin(_,[]).

/***********
putin(X,L1,L2) <=> L2 es el resultado de añadir X a L1, si X no pertenece a L1.
***********/
putin(X,[],[X]):-
    !.

putin(X,[Y|L],[Y|L]):-
    X==Y,
    !.

putin(X,[Y|L],[Y|L1]):-
    putin(X,L,L1).

/***********
```

transformer(+File,-AuxFile) <=> Se transforman las reglas contenidas en el fichero File a forma clausal y se incorporan a la base de reglas los resultados de la transformación. Se emplea como fichero auxiliar de la transformación el ficher AuxFile.
**********/

```
transformer(File,AuxFile):-
        working_directory(Dir,Dir),
        string_concat(Dir,'primerordenAux.me',AuxFile),
        generate_me(File,AuxFile).
```

/**********
generate_me(+Input,-Output) <=> En el fichero de ruta Output aparece la transformación a forma clausal de las fórmulas de primer orden que hay en el fichero de ruta Input.
**********/

```
generate_me(Input,Output):-
    me2_consult(Input,Clauses),
    export_me(Clauses,Output).
```

/**********
me2_consult(+File,-Clauses) <=> Clauses es la lista de cláusulas resultante de transformar a forma clausal las fórmulas escritas en lógica de primer orden contenidas en el fichero File.
**********/

```
me2_consult(File, Clauses):-
    open(File,read,H),
    read_clauses(H,Clauses),
    close(H).
```

/**********
export_me(+Clauses,-File) <=> Se escribe en el fichero de ruta File las cláusulas de la lista de cláusulas Clauses.
**********/

```
export_me(Clauses,File):-
    open(File,write,_,[]),
    tell(File),
    write_clauses(Clauses),
    told.
```

/**********
read_clauses(+Handler,-Clauses) <=> Clauses es la lista de cláusulas que se obtienen a partir del manejador de ficheros Handler. Para ello se van leyendo fórmulas escritas en lógica de primer orden del fichero y se van transformando a forma clausal. El predicado

auxiliar read_clauses1 tiene un argumento más para ir acumulando la lista de cláusulas generada.
**********/
```
read_clauses(Handler,Clauses):-
        read_clauses1(Handler,[],Clauses).

read_clauses1(Handler,L,L1):-
        read_rule(Handler,R),
        process_rule(Handler,R,L,L1).
```

/**********
process_rule(+Handler,+Rule,+L,-L1) <=> L1 es la lista resultante de acumular todas las reglas obtenidas a partir de transformar las reglas leídas del manejador Handler partiendo de la lista inicial L. Se termina todo el proceso de lectura al procesar la marca de fin de fichero end_of_file.
**********/
```
process_rule(_,R,L,L):-
        R = end_of_file,
        !.

process_rule(H,R,L,L1):-
        translate(R,Clauses),
        insert(Clauses,L,L2),
        read_clauses1(H,L2,L1).
```

/**********
read_rule(+Handler,-Rule) <=> Rule es la primera regla leída del manejador Handler. Si el fichero está vacío se leerá end_of_file.
**********/
```
read_rule(Handler,Rule):-
        read(Handler,Rule).
```

/**********
insert(L1,L2,L3) <=> L3 es la lista resultante de insertar todos los elementos de L1 en L2, sin repeticiones.
**********/
```
insert([],L,L):-
      !.

insert([X|Xs],L,L1):-
      insert_element(X,L,L2),
      insert(Xs,L2,L1).
```

```
/***********
```
insert_element(E,L1,L2) <=> L2 es la lista resultante de insertar el elemento E al final de L1, siempre que E no perteneciese a L1.

```
***********/
insert_element(N,[],[N]):-
        !.

insert_element(N,[M|Ms],[M|Ms]):-
        N==M,
        !.

insert_element(N,[M|Ms],[M|Ms1]):-
        insert_element(N,Ms,Ms1).

/***********
```
write_clauses(List) <=> Se escriben las cláusulas de la lista List en la salida estándar.

```
***********/
write_clauses([]).
write_clauses([X|Xs]):-
                write_clause(X),
                nl,
                write_clauses(Xs).

/***********
```
write_clause(+Clause) <=> Se escribe en formato de model elimination la cláusula Clause en la salida estándar.

```
***********/
write_clause(cl(P,[])) :-
                !,
                write_list(P),
                write('.').

write_clause(cl([],_)) :-
                !.

write_clause(cl(P,N)):-
                write_list(P),
                write(' <- '),
                write_list(N),
                write('.').
```

```
/***********
write_list(L) <=> Se escribe por la salida estándar la lista de literales L.
***********/
write_list([X]):-
        !,
        write_literal(X).

write_list([X|Xs]):-
        write_literal(X),
        write(','),
        write_list(Xs).

/***********
write_literal(+Literal) <=> Se escribe el literal Literal en la salida estándar, según el formato de model elimination.
***********/
write_literal(~X):-
         !,
         write('no '),
         write_literal(X).

write_literal(X):-
         write(X).
```

meFormMain.pl

```
/*
Se define la ventana principal de la GUI, así como sus funcionalidades
*/

:- use_module(library(pce)).
:- use_module(library(toolbar)).
:- use_module(library(find_file)).

/*
Clase me_interface: Ventama principal de la interfaz gráfica de usuario
*/
:- pce_begin_class(me_interface, frame,
              "Model Elimination").

initialise(B) :->
     send_super(B, initialise, 'DATEM'),
```

```
        send(B,can_resize,false),
        send(B, append, new(D, tool_dialog(B))),
        send(B, fill_dialog(D)).

/*
fill_dialog(+B,+D) <=> B es la ventana, objeto de la clase frame de XPCE, D es un contenedor de objetos XPCE, de la clase dialog. Crea los elementos de la interfaz gráfica principal
 */
fill_dialog(B, D:dialog) :->
        send_list(D, append,
                [ new(F, popup(archivo)),
                  new(V, popup(herramientas)),
                  new(H, popup(ayuda)),
                  new(I,editor(@default, 80, 16)),
                  new(J,editor(@default, 80, 16)),
                  new(TO,menu(goal_type_menu, toggle)),
                  new(T,text_item(objetivo,'',message(D?demostrar_member, execute))),
                  new(_,button(demostrar, message(@prolog, read_command))) ,
                  new(P,button(parar, message(@prolog, abort_thread)))
                ]),

        send(T,label, 'Objetivo:            '),
        send(TO,label,'Tipo de Objetivo:'),
        send_list(TO,append,
                 [ menu_item('on', @default, 'Clausal', @off, @nil, '\\es'),
                   menu_item(off, @default, 'Primer Orden', @off, @nil, '\\en')
                 ]),
        send(TO,multiple_selection,@off),
        send(P,active,false),
        send(T,length,90),

        nb_setval(abort_button,P),
        nb_setval(editor1,I),
        nb_setval(editor2,J),
        nb_setval(goal_text_item,T),

        send_list(F, append,
                [
                  menu_item('Cambiar directorio de trabajo',
                          message(@prolog, dir_viewer)),
                  gap,
```

```
            menu_item('Cargar Clausal',
                    message(B, open_file)),
            gap,
            menu_item('Cargar Primer Orden',
                    message(B, open_file_first_order)),
            gap,
            menu_item('Guardar .me',
                    message(B, save_file)),
            gap,
            menu_item('Guardar última prueba',
                    message(B, save_last_proof)),
            gap,
            menu_item(salir,
                    message(B, destroy))
          ]),
    send_list(V, append,
          [
            menu_item(reglas,
                    message(@prolog, rule_management_dialog)),
            menu_item(lemas,
                    message(@prolog, lemma_management_dialog)),
            menu_item(opciones,
                    message(@prolog, configure_options))
          ]),
    send_list(H, append,
          [  menu_item(contenidos,
                    message(B, show_contents)),
             menu_item('Acerca de...',
                    message(B, about))
          ]).

/*
about(+B) <=> Muestra la información sobre el personal responsable del desarrollo del
sistema y de la versión de éste
*/
about(_B) :->
    send(@display, inform,
        'Proyecto Sistemas Informáticos\n\\n\
         Agustín Daniel Delgado Muñoz\n\
         Álvaro Novillo Vidal \n\
         Fernando Pérez Morente\n\\n\
         Facultad de Informática\n\
```

```
        Universidad Complutense de Madrid').

/*
show_contents(+B) <=> Da acceso a los contenidos de ayuda del sistema
*/
show_contents(_B) :->
        win_shell(open,'ayuda.htm',normal).

:- pce_group(action).


/*
open_file(+B) <=> Muestra una ventana de exploración para la apertura de un fichero de
extensión .me con sintaxis de forma clausal
*/
open_file(_B) :->
      "Cargar archivo"::
      get(@finder, file, open,
         tuple('Model elimination files', me),
         FileName),
      nb_getval(editor1,E),

     update_file_name(FileName),
     generate_lemmas_file(FileName,FileLemma),
     %Crea Fichero de Lemmas por defecto.
     %Los archivos de Lemas tienen el mismo nombre que los *.me pero con extensión
       *.lemma
     %En caso de existir el archivo de lemas, simplemente se abre y se cierra.
    /* open(FileLemma,update,Stream),
     close(Stream),      */
     file_name_extension(_, Ext, FileName),
      (   Ext == me
      ->  send(E,load,FileName),
         load_clausal(FileName),
         load_lemmas(FileLemma)
      ).

/*
open_file_first_order(+B) <=> Muestra una ventana de exploración para la apertura de
un fichero de extensión .me con sintaxis de Lógica de Primer Orden
*/
open_file_first_order(_B) :->
```

```
        "Cargar archivo primer orden"::
        get(@finder, file, open,
            tuple('Model elimination files', me),
            FileName),
        nb_getval(editor1,E),
         update_file_name(FileName),
       generate_lemmas_file(FileName,FileLemma),
       %Crea Fichero de Lemmas por defecto.
       %Los archivos de Lemas tienen el mismo nombre que los *.me pero con extensión
          *.lemma
       %En caso de existir el archivo de lemas, simplemente se abre y se cierra.
       open(FileLemma,update,Stream),
       close(Stream),
        file_name_extension(_, Ext, FileName),
        (   Ext == me
        ->  send(E,load,FileName),
            load_first_order(FileName)
        ).

/*
save_file(+B) <=> Permite guardar el contenido del editor de entrada de la interfaz gráfica en el fichero indicado, con extensión .me
*/
save_file(_B) :->
        "Guardar archivo"::
        get(@finder, file, save,
            tuple('Model elimination files', me),
            FileName),
        nb_getval(editor1,E),
        file_name_extension(_, Ext, FileName),
        (   Ext == me
        ->  send(E,save,FileName)
        ).

/*
save_last_proof(+B) <=> Permite guardar la última prueba realizada en el fichero indicado, con extensión .txt
*/
save_last_proof(_B) :->
        "Guardar archivo"::
        get(@finder, file, save,
            tuple('Text Files', txt),
```

```
        FileName),
    save_textfile(FileName).


:- pce_end_class(me_interface).


/*
read_command <=> Lee el comando introducido por el usuario. Distingue entre dos modos:
   - Modo de ejecución de predicado Prolog, cuando la cadena introducida introducida
     comienza por '/'. La interfaz funciona como la consola de Prolog
   - Modo de demostración del objetivo, si no empieza por '/'. Se crea un nuevo hilo donde
     se realizará el proceso de demostración
En ambos casos se comprueba que se ha introducido un término Prolog sintácticamente correcto
*/
read_command :-
        nb_getval(goal_text_item,T),
        nb_getval(editor2,E),
        nb_getval(me_interface_var,I),
        get(T, selection,Goal),
        string_to_atom(Goal,At),
        string_to_list(Goal,ListaCar),

        (nth0(0, ListaCar, 47)
          ->
          working_directory(Dir,Dir),
          string_concat(Dir,'salidaAux.txt',FicheroAux),

          length(ListaCar,Longitud),
          Longitud2 is Longitud-1,
          sub_atom(At, 1, Longitud2,_,At2),
          catch(catch(atom_to_term(At2,G2,_),_E3,throw(invalid)),_E4,
                (send(@display,inform,'Introduzca un predicado Prolog válido.\n'),fail)),

          open(FicheroAux,write,_,[]),
          tell(FicheroAux),
          term_variables(G2,Vars),
          call(G2),
          write(Vars),
          told,
```

```
            send(E,load,FicheroAux)
          ;
            catch(catch(atom_to_term(At,G,_),_E,throw(invalid)),_E2,
                  (send(@display,inform,'Introduzca un objetivo válido.\n'),fail)),
            settings(S),
            nb_getval(current_file_name,FN),
            nb_setval(last_proof,''),
            get(I,member,tool_dialog,TD),
            get(TD,member,goal_type_menu,TO),
            get(TO, selection,Sel),
            nb_setval(goal_type,Sel),
            thread_create(solve_thread(G,S,FN,Sel),Id,[]),
            nb_setval(id_solve_thread,Id),
            unlock_button(parar),
            lock_button(demostrar)
        ).


/*
solve_thread(G,S,FN,Sel) <=> G es el objetivo a demostrar, S es la lista con los valores de
los parámetros del sistema, FN es el nombre del fichero cargado actualmente y Sel indica el
tipo de objetivo que se está intentando demostrar. Se ejecu ta la demostración del objetivo,
distinguiendo si éste está en forma clausal o es una fórmula de la lógica de primer orden.
*/
solve_thread(G,S,FN,Sel) :-
    nb_setval(proof_excerpt,''),
    nb_setval(refresh,0),

    load_options(S),
    nb_setval(current_file_name,FN),
    ( Sel == on
     ->
     catch(solve(G,P1,A1,_),E,(fail))
    ;
     catch(solve2(G,P1,R1,_),E,(fail)),
     in_pce_thread(nb_setval(current_rules,R1))
    ),
    in_pce_thread(nb_setval(current_goal,G)),
    in_pce_thread(nb_setval(current_proof,P1)),
    in_pce_thread(nb_setval(current_answer,A1)),

    show_final,
```

```
    ( E == 'abortado'
     ->
      write('Demostración abortada')
     ;
    (no_pop_ups
      ->
        retract(no_pop_ups)
      ;
       popup_messages
      )
    ),
    reset,

    in_pce_thread(lock_button(parar)),
    in_pce_thread(unlock_button(demostrar)).

/*
popup_messages <=> Muestra ventanas informativas al finalizar una demostración con éxito. Se requiere la intervención del usuario para decidir si se buscan más soluciones y si se guardan los lemas generados
*/
popup_messages :-
    in_pce_thread(solve_more_dialog),
    catch(waiting_answer,E3,(true)),
    E3 \== more,

    in_pce_thread(solve_info_dialog).

/*
waiting_answer <=> Bloquea la interfaz. La espera termina al producirse una excepción
*/
waiting_answer :-
        repeat,
        fail.


/*
load_options(+S) <=> S es la lista con los valores actuales de los prámetros del sistema. Inicializa las variables globales del hilo que se requieran mediante una lista con los nombres y valores de éstas.
*/
load_options(S) :-
```

```
        forall(member((X,Y),S),nb_setval(X,Y)).

/*
abort_thread <=> Se fuerza la muerte por excepción del hilo que ejecuta una demostración. Sólo puede haber una demostración activa a la vez.
*/
abort_thread :-
        nb_getval(id_solve_thread,Id),
        thread_signal(Id,throw(abortado)),
        reset,
        lock_button(parar),
        unlock_button(demostrar),
        show_simple('\n\n                ---- Demostración Abortada ----        '),
        show_simple('\n_____________________________________________________
                        _____________________________\n').


/*
unlock_button(+B) <=> Desbloquea el botón B indicado de la interfaz gráfica
*/
unlock_button(B) :-
        nb_getval(me_interface_var,I),
        get(I,member,tool_dialog,D),
        get(D,member,B,P),
        send(P,active,true).

/*
lock_button(+B) <=> Bloquea el botón B indicado de la interfaz gráfica
*/
lock_button(B) :-
        nb_getval(me_interface_var,I),
        get(I,member,tool_dialog,D),
        get(D,member,B,P),
        send(P,active,false).

/*
solve_more_dialog <=> Muestra la ventana informativa que permite al usuario buscar más soluciones o no
*/
solve_more_dialog :-
            new(D,dialog('Demostración finalizada')),
            send(D,append(label(informacion,'Se ha finalizado la demostración con éxito.
```

```
                \n¿Desea buscar más soluciones?'))),
        send(D,append(button(aceptar,message(@prolog,solve_more,D)))),
        send(D,append(button(salir,message(@prolog,solve_nomore,D)))),
        send(D,open).

/*
solve_more(+D) <=> Lanza una excepción de tipo more, para salir de la espera y buscar más posibles demostraciones
*/
solve_more(D) :-
        nb_getval(id_solve_thread,Id),
        thread_signal(Id,throw(more)),
        send(D,destroy).

/*
solve_nomore(+D) <=> Lanza una excepción de tipo nomore, para salir de la espera y no buscar más posibles demostraciones
*/
solve_nomore(D) :-
        nb_getval(id_solve_thread,Id),
        thread_signal(Id,throw(nomore)),
        send(D,destroy).

/*
solve_info_dialog <=> Según estén o no activados el guardado y el guardado automático de lemas persistente, se muestra o no la ventana informativa para el guardado voluntario de los lemas generados en una demostración
*/
solve_info_dialog :-
          nb_getval(save_lemmas,S),
          nb_getval(autolemmas,AS),
         (S == on
          ->
            (AS == on
             ->
             solve_info_dialog_autolemmas_on
             ;
             solve_info_dialog_lemmas_on
            )
          ;
           solve_info_dialog_lemmas_off
         ).
```

```
/*
solve_info_dialog_autolemmas_on <=> Ventana informativa que se muestra tras una demostración en el caso de que el autoguardado de lemas esté activado
*/
solve_info_dialog_autolemmas_on :-
        save_lemmas_top(D),
        new(D,dialog('Demostración finalizada')),
        send(D,append(label(informacion,'Se ha finalizado la demostración con éxito.
            \nSe han guardado los nuevos lemas.'))),
        send(D,append(button(salir,message(D,destroy)))),
        send(D,open).

/*
solve_info_dialog_lemmas_on <=> Ventana informativa que se muestra tras una demostración en el caso de que el autoguardado de lemas esté desactivado y el guardado de lemas activado
*/
solve_info_dialog_lemmas_on :-
        new(D,dialog('Demostración finalizada')),
        send(D,append(label(info,'Se ha finalizado la demostración con éxito. \n ¿Quiere guardar los nuevo lemas?.'))),
        send(D,append(button(guardar,message(@prolog,save_lemmas_top,D)))),
        send(D,append(button(no_guardar,message(D,destroy)))),
        send(D,open).

/*
save_lemmas_top(+D) <=> D es la ventana informativa de guardado de lemas. Se ejecuta el predicado para el guardado de lemas y se cierra la ventana
*/
save_lemmas_top(D) :-
        nb_getval(goal_type,TG),
        save_lemmas(TG),
        send(D,destroy).

/*
solve_info_dialog_lemmas_off <=> Ventana informativa que se muestra tras una demostración en el caso de que el guardado de lemas esté desactivado
*/
solve_info_dialog_lemmas_off :-
        new(D,dialog('Demostración finalizada')),
        send(D,append(label(info,'Se ha finalizado la demostración con éxito.'))),
        send(D,append(button(salir,message(D,destroy)))),
```

```
        send(D,open).


/*
configure_options <=> Se muestra la ventana de configuración de los parámetros del sis-
tema
*/
configure_options :-
        init_options,
        show_options_dialog.



%-------------------------      FICHEROS      ------------------------------

/*
load_clausal(+FileName) <=> Carga de el fichero de reglas en forma clausal de nombre
FileName
*/
load_clausal(FileName):-
        reset_all,
        me_consult(FileName),send(@display, inform,'Archivo cargado satisfactoriamente').

/*
load_first_order(+FileName) <=> Carga de el fichero de fórmulas de la Lógica de Primer
Orden de nombre FileName
*/
load_first_order(FileName):-
        reset_all,
        transformer(FileName,FicheroAux),
        me_consult(FicheroAux),send(@display, inform,'
                        Archivo cargado satisfactoriamente').


/*
save_textfile(+FileName) <=> Guarda el contenido del string que contiene la prueba para
la última demostración realizada en el fichero de nombre FileName
*/
save_textfile(FileName):-
        nb_getval(last_proof,Str),
        string_to_file(FileName,Str).
```

```
%----------------------    SALIDA POR PANTALLA      ------------------------

/*
show_string(+Str) <=> Str es una cadena de caracteres. Se almacena el texto de salida en
un buffer y se actualiza en pantalla 1 de cada 100 veces. Para ejecutar desde el hilo de de-
mostración
*/
show_string(Str):-
        write(Str),
        nb_getval(refresh,R),
        nb_getval(proof_excerpt,E),
        (atom(Str)
          ->
         string_concat(E,Str,E2)
         ;
         term_to_atom(Str,At),
         string_concat(E,At,E2)
        ),

        (R >= 100
          ->
         in_pce_thread(show_aux(E2)),
         nb_setval(proof_excerpt,''),
         nb_setval(refresh,0)
         ;
         nb_setval(proof_excerpt,E2),
         R2 is R+1,
         nb_setval(refresh,R2)
        ).

/*
show_aux(+Str) <=> Concatena el string Str en el texto mostrado por pantalla
*/
show_aux(Str):-
        nb_getval(editor2,E),
        string_to_atom(Str,At),
        send(E,append,At),
        update_last_proof(At).
```

```
/*
show_simple(+Str) <=> Concatena el string Str en el texto mostrado por pantalla. Para usar desde el hilo de PCE
*/
show_simple(Str) :-
        write(Str),
        nb_getval(editor2,Edi),
        string_concat('',Str,E2),
        string_to_atom(E2,At),
        send(Edi,append,At),
        update_last_proof(At).

/*
show_final <=> Concatena el contenido del buffer aun no mostrado al texto exhibido por pantalla
*/
show_final :-
        nb_getval(proof_excerpt,E),
        in_pce_thread(show_aux(E)),
        nb_setval(proof_excerpt,'').

/*
clear_output <=> Limpia el contenido del editor de texto de salida de la interfaz gráfica
*/
clear_output:-
         nb_getval(editor2,E),
         send(E,clear).



%----------------------------  UTILIDADES  -------------------------------

/*
update_last_proof(+Str) <=> Añade cadena de caracteres Str al final de la variable que guarda la última prueba realizada
*/
update_last_proof(Str):-
                nb_getval(last_proof,UP),
                string_concat(UP,Str,UP1),
                nb_setval(last_proof,UP1).

/*
```

```
update_file_name(+FileName) <=> Actualiza la variable que guarda el fichero .me cargado actualmente
*/
update_file_name(FileName):-
                nb_setval(current_file_name,FileName).

/*
string_to_file(+FileName,+Str) <=> Realiza la escritura del string Str en el fichero de nombre FileName
*/
string_to_file(FileName,Str):-
        open(FileName,write,H),
        write(H,Str),
        close(H).

/*
reset_all <=> Elimina de la base de Prolog todas las reglas y lemas introducidos
*/
reset_all :-
        reset,
        retractall(rule(_,_)),
        retractall(inactive_rule(_,_)),
        retractall(lemma(_,_,_,_,_)),
        retractall(interlemma(_,_,_,_,_)),
        retractall(inactive_lemma(_,_,_,_,_)).

%----------------------------  INICIO  --------------------------------

/*
exec <=> Lanza la interfaz gráfica e inicializa variables globales
*/
exec:-
    new(I,me_interface),
    nb_setval(me_interface_var,I),
    nb_setval(last_proof,''),
    nb_setval(current_file_name,''),
    reset_all,
    send(I,open).
```

meFormOptions.pl

```
/*
```

Se define la herramienta visual, y los métodos que utiliza, para la configuración de los distintos parámetros del sistema
*/

/*
init_options <=> Muestra la ventana para la configuración de los parámetros del sistema
*/

```
init_options:-
   retractall(dialog(_,_)),
   assert(
    dialog(me_options_dialog,
     [ object       :=
        Opciones,
       parts        :=
        [ Opciones              :=
           dialog('Opciones'),
          ReglasObjetivo        :=
           menu(reglasObjetivo, toggle),
          Aplicar               :=
           button(aceptar),
          Salir                 :=
           button(salir),
          AncestroIdentico      :=
           menu(ancestroIdentico, toggle),
          AncestroComplementario :=
           menu(ancestroComplementario, toggle),
          Reduccion             :=
           menu(reduccion, toggle),
          Name1                 :=
           label(name, 'Configurar opciones'),
          Factor                :=
           text_item(factor),
          ProfundidadLimite             :=
           text_item(prof_limite),
          Name2                 :=
           label(name, '_____'),
          MostrarTraza          :=
           menu(mostrarTraza, toggle),
          MostrarPrueba         :=
           menu(mostrarPrueba, toggle),
          BaseLemas         :=
           menu(baseLemas, toggle),
```

```
    GuardarLemas        :=
      menu(guardarLemas, toggle),
    OccurCheck          :=
      menu(occurCheck, toggle),
    LemasIntermedios    :=
     menu(interlemmas, toggle),
    AutoLemas           :=
      menu(autolemmas, toggle)
  ],
modifications :=
  [ ReglasObjetivo      :=
      [ label           :=
          'Reglas Objetivo:               ',
        multiple_selection :=
          @off,
        reference       :=
          point(0, 16),
        append          :=
          [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
            menu_item('off', @default, 'No', @off, @nil, '\\en')
          ]
      ],
    AncestroIdentico    :=
      [ label           :=
          'Ancestro Idéntico:             ',
        multiple_selection :=
          @off,
        reference       :=
          point(0, 16),
        append          :=
          [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
            menu_item('off', @default, 'No', @off, @nil, '\\en')
          ]
      ],
    AncestroComplementario :=
      [ label           :=
          'Ancestro Complementario: ',
        multiple_selection :=
          @off,
        reference       :=
          point(0, 16),
        append          :=
```

```
    [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
      menu_item(off, @default, 'No', @off, @nil, '\\en')
    ]
  ],
Reduccion             :=
  [ label             :=
      'Reducción:                    ',
    multiple_selection :=
      @off,
    reference         :=
      point(0, 16),
    append            :=
      [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
        menu_item(off, @default, 'No', @off, @nil, '\\en')
      ]
  ],
Name1                 :=
  [ font := @helvetica_bold_14
  ],
Factor                :=
  [
    label     := 'Factor:                          ',
    selection := 10,
    type      := int,
    length    := 6
  ],

ProfundidadLimite             :=
  [
    label     := 'Profundidad límite:          ',
    selection := 0,
    type      := int,
    length    := 6
  ],
MostrarTraza          :=
  [ label             :=
      'Mostrar Traza:                ',
    multiple_selection :=
      @off,
    reference         :=
      point(0, 16),
    append            :=
```

```
        [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
          menu_item(off, @default, 'No', @off, @nil, '\\en')
        ]
    ],
  MostrarPrueba       :=
    [ label           :=
        'Mostrar Demostración:      ',
      multiple_selection :=
        @off,
      reference       :=
        point(0, 16),
      append          :=
        [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
          menu_item(off, @default, 'No', @off, @nil, '\\en')
        ]
    ],

  BaseLemas       :=
    [ label           :=
        'Usar Base de Lemas:        ',
      multiple_selection :=
        @off,
      reference       :=
        point(0, 16),
      append          :=
        [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
          menu_item(off, @default, 'No', @off, @nil, '\\en')
        ]
    ],
  GuardarLemas      :=
    [ label           :=
        'Guardar Lemas:             ',
      multiple_selection :=
        @off,
      reference       :=
        point(0, 16),
      append          :=
        [   menu_item('on',message(@prolog,unlock_autolemmas),   'Sí',   @off,   @nil,
'\\es'),
          menu_item(off, message(@prolog,lock_autolemmas), 'No', @off, @nil, '\\en')
        ]
    ],
```

```
  OccurCheck         :=
    [ label            :=
       'Occur check:                  ',
      multiple_selection :=
       @off,
      reference        :=
       point(0, 16),
      append           :=
       [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
         menu_item(off, @default, 'No', @off, @nil, '\\en')
       ]
    ],
  LemasIntermedios         :=
    [ label            :=
       'Lemas Intermedios:          ',
      multiple_selection :=
       @off,
      reference        :=
       point(0, 16),
      append           :=
       [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
         menu_item(off, @default, 'No', @off, @nil, '\\en')
       ]
    ],

    AutoLemas        :=
    [ label            :=
       'Auto-guardado de lemas:   ',
      multiple_selection :=
       @off,
      reference        :=
       point(0, 16),
      append           :=
       [ menu_item('on', @default, 'Sí', @off, @nil, '\\es'),
         menu_item(off, @default, 'No', @off, @nil, '\\en')
       ]
    ]
  ],
layout       :=
  [
    area(Name1,
         area(15, 13, 236, 22)),
```

```
        area(ReglasObjetivo,
            area(34, 44, 270, 22)),
        area(Reduccion,
            area(34, 68, 269, 22)),
        area(AncestroIdentico,
            area(34, 92, 268, 22)),
        area(AncestroComplementario,
            area(34, 116, 271, 22)),
        area(OccurCheck,
            area(34, 140, 275, 22)),
        area(BaseLemas,
            area(34, 164, 275, 22)),
        area(LemasIntermedios,
            area(34, 188, 275, 22)),
        area(GuardarLemas,
            area(34, 212, 275, 22)),
        area(AutoLemas,
            area(34, 236, 275, 22)),
        area(Factor,
            area(34, 260, 270, 20)),
        area(ProfundidadLimite,
            area(34, 290, 270, 20)),
        area(Name2,
            area(34, 310, 236, 22)),
        area(MostrarTraza,
            area(34, 334, 273, 22)),
        area(MostrarPrueba,
            area(34, 358, 275, 22)),
        area(Aplicar,
            area(51, 392, 80, 24)),
        area(Salir,
            area(188, 392, 80, 24))
      ],
      behaviour     :=
      [ Aplicar := [ message := message(@prolog,update_options)],
        Salir := [ message := message(Opciones, destroy)]
      ]
    ])
     )
     .
/*
```

```
*lock_autolemmas <=> Bloquea el menú para la elección del guardado automático de lemas persistentes*
*/
lock_autolemmas :-
        nb_getval(me_options_dialog_var,O),
        get(O,member,autolemmas,AL),
        send(AL, selection,off),
        send(AL,active,false).

/*
*lock_autolemmas <=> Desbloquea el menú para la elección del guardado automático de lemas persistentes*
*/
unlock_autolemmas :-
        nb_getval(me_options_dialog_var,O),
        get(O,member,autolemmas,AL),
        send(AL,active,true).

/*
*update_options <=> Se llevan a cabo las modificaciones de los valores de los parámetros del sistema realizadas por el usuario a través de la interfaz gráfica*
*/
update_options:-
         update_reduction,
         update_goal_rules,
         update_identical_ancestors,
         update_complementary_ancestors,
         update_factor,
         update_depth_limit,
         update_proof_option,
         update_trace_option,
         update_lemma_base_option,
         update_save_lemma,
         update_occur_check,
         update_intermediate_lemmas,
         update_lemma_autosave,
         nb_getval(me_options_dialog_var,O),
         send(O,destroy).

update_reduction:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, reduccion,F),
```

```
        get(F, selection,S),
        set(reduction,S).

update_identical_ancestors:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, ancestroIdentico,F),
        get(F, selection,S),
        set(id_anc,S).

update_complementary_ancestors:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, ancestroComplementario,F),
        get(F, selection,S),
        set(id_comp_anc,S).

update_goal_rules:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, reglasObjetivo,F),
        get(F, selection,S),
        set(goal_rules,S).

update_factor:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, factor,F),
        get(F, selection,S),
        set(factor,S).

update_depth_limit:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, prof_limite,F),
        get(F, selection,S),
        set(depth_limit,S).

update_proof_option:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, mostrarPrueba,F),
        get(F, selection,S),
        set(proof,S).

update_trace_option:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, mostrarTraza,F),
```

```
        get(F, selection,S),
        set(trace_on,S).

update_lemma_base_option:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, baseLemas,F),
        get(F, selection,S),
        set(lemmas,S).

update_save_lemma:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, guardarLemas,F),
        get(F, selection,S),
        set(save_lemmas,S).

update_occur_check:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, occurCheck,F),
        get(F, selection,S),
        set(occurCheck,S).

update_intermediate_lemmas:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, interlemmas,F),
        get(F, selection,S),
        set(interlemmas,S).

update_lemma_autosave:-
        nb_getval(me_options_dialog_var,O),
        get(O, member, autolemmas,F),
        get(F, selection,S),
        set(autolemmas,S).

/*
show_options_dialog <=> Se muestra el formulario para la modificación de los parámetros del sistema por parte del usuario
*/
show_options_dialog:-
    make_dialog(O,me_options_dialog),
    send(O,can_resize,false),
    nb_setval(me_options_dialog_var,O),
    show_options_configuration(O),
```

```
        send(O, open).

/*
show_options_configuration(+O) <=> Modifica los elementos del formulario acorde con el
estado actual de los parámetros del sistema
*/
show_options_configuration(O):-
        get(O, member, reduccion, R),
        nb_getval(reduction, RAct),
        send(R, selection, RAct),

        get(O, member, ancestroIdentico, AI),
        nb_getval(id_anc, AIAct),
        send(AI, selection, AIAct),

        get(O, member, ancestroComplementario, AC),
        nb_getval(id_comp_anc, ACAct),
        send(AC, selection, ACAct),

        get(O, member, reglasObjetivo, RO),
        nb_getval(goal_rules, ROAct),
        send(RO, selection, ROAct),

        get(O, member, factor, F),
        nb_getval(factor, FAct),
        send(F, selection, FAct),

        get(O, member, prof_limite, DL),
        nb_getval(depth_limit, DLAct),
        send(DL, selection, DLAct),

        get(O, member, mostrarPrueba, P),
        nb_getval(proof, PAct),
        send(P, selection, PAct),

        get(O, member, mostrarTraza, T),
        nb_getval(trace_on, TAct),
        send(T, selection, TAct),

        get(O, member, baseLemas, BL),
        nb_getval(lemmas, BLAct),
        send(BL, selection, BLAct),
```

```
 get(O, member, guardarLemas, GL),
nb_getval(save_lemmas, GLAct),
send(GL, selection, GLAct),

 get(O, member, occurCheck, OC),
nb_getval(occurCheck, OCAct),
send(OC, selection, OCAct),

 get(O, member, interlemmas, ILL),
nb_getval(interlemmas, ILLAct),
send(ILL, selection, ILLAct),

 get(O, member, autolemmas, ALL),
nb_getval(autolemmas, ALLAct),
send(ALL, selection, ALLAct),

(GLAct == on
  ->
  unlock_autolemmas
  ;
  lock_autolemmas)
.
```

meFormRules.pl

```
/*
Se define la herramienta visual, y los métodos y estructuras utilizadas, para la selección de
reglas activos
*/

/*
rule_management_dialog <=> Se muestra la ventana para la gestión de reglas activas
*/
rule_management_dialog :-
        new(F, frame('Selección de reglas activas')),

        send(F, append(new(B1, browser(size := size(90,15))))),
        send(new(D1, dialog), below(B1)),
        send(new(B2,browser(size := size(90,15))),below(D1)),
        send(new(D2, dialog), below(B2)),
```

```
    send(B1,label('Reglas activas:')),
    send(B2,label('Reglas inactivas:')),

    send(D1, append(button(activar_todas,message(@prolog, activate_all_rules,B1,B2)))),
    send(D1, append(button(activar,message(@prolog, activate_rule,B1,B2)))),
    send(D1, append(button(desactivar,message(@prolog, disactivate_rule,B1,B2)))),
    send(D1, append(button(desactivar_todas,message(@prolog,
                                disactivate_all_rules,B1,B2)))),
    send(D2, append(button(aceptar,message(@prolog,exit_rules_management,F)))),
    send(D2, append(button(cancelar,message(@prolog,undo_rules,F)))),

    get(D1,member,activar,Act),
    get(D1,member,desactivar,Dac),
    get(D1,member,activar_todas,TAct),
    get(D1,member,desactivar_todas,TDac),
    send(Act,label(' /\\ ')),
    send(Dac,label(' \\/ ')),
    send(TAct,label(' /\\ /\\ ')),
    send(TDac,label(' \\/ \\/ ')),

    init_rules,
    show_rules(B1,B2),

    send(F, open).

/*
init_rules <=> Captura las actuales reglas activas y las inactivas y actualiza el contenido
de las listas gráficas convenientemente. Se crea una lista de string con la información de
todas las reglas activas, otra para las inactivas y otra para todas, para poder recuperar el
estado inicial si se llega a requerir. Además, se crea una copia para la lista de reglas acti-
vas y otra para la de reglas inactivas para permitir el tratamiento intermedio de activacio-
nes y desactivaciones de reglas
*/
init_rules :-
    findall((X1,Y1),(rule(X1,Y1);inactive_rule(X1,Y1)),L1),
    sort(L1,L4),
    nb_setval(rule_list,L4),
    nb_setval(act_rule_list,L4),
    findall((X2,Y2),rule(X2,Y2),L2),
    sort(L2,L5),
    nb_setval(active_rule_list,L5),
    nb_setval(act_active_rule_list,L5),
```

```
        findall((X3,Y3),inactive_rule(X3,Y3),L3),
        sort(L3,L6),
        nb_setval(inactive_rule_list,L6),
        nb_setval(act_inactive_rule_list,L6),
        retractall(rule(_,_)),
        forall(member((Y4,Z4),L5),assert(rule(Y4,Z4))),
        retractall(inactive_rule(_,_)),
        forall(member((Y5,Z5),L6),assert(inactive_rule(Y5,Z5))).

/*
show_rules(+B1,+B2) <=> B1 es la lista gráfica para mostrar las reglas actualmente activas, B2 es la lista gráfica para mostrar las reglas actualmente inactivas. Actualiza el contenido de B1 y B2 convenientemente
*/
show_rules(B1,B2) :-
            send(B1,clear),
            send(B2,clear),
            show_active_rules(B1),
            show_inactive_rules(B2).

/*
show_active_rules(+B) <=> B es la lista gráfica para mostrar las reglas actualmente activas. Se actualizan sus elementos según el contenido de la lista que guarda la información de las reglas actualmente activas
*/
show_active_rules(B) :-
                new(C,chain),
                findall(X <- Y,rule(X,Y),L),
                forall(member(Z,L),(term_to_atom(Z,At),send(C,append,At))),
                send(B,members(C)).

/*
show_inactive_rules(+B) <=> B es la lista gráfica para mostrar las reglas actualmente inactivas. Se actualizan sus elementos según el contenido de la lista que guarda la información de las reglas actualmente inactivas
*/
show_inactive_rules(B) :-
                new(C,chain),
                findall(X <- Y,inactive_rule(X,Y),L),
                forall(member(Z,L),(term_to_atom(Z,At),send(C,append,At))),
                send(B,members(C)).
```

```
/*
```
activate_all_rules(+B1,+B2) <=> B1 es la lista gráfica para mostrar las reglas actualmente activas, B2 es la lista gráfica para mostrar las reglas actualmente inactivas. Todas las reglas pasan a estar activas y se actualiza el contenido de B1 y B2 convenientemente

```
*/
activate_all_rules(B1,B2) :-
        nb_getval(rule_list,L),
        nb_setval(act_active_rule_list,L),
        nb_setval(act_inactive_rule_list,[]),
        retractall(rule(_,_)),
        retractall(inactive_rule(_,_)),
        forall(member((Y,Z),L),assert(rule(Y,Z))),
        show_rules(B1,B2).

/*
```
disactivate_all_rules(+B1,+B2) <=> B1 es la lista gráfica para mostrar las reglas actualmente activas, B2 es la lista gráfica para mostrar las reglas actualmente inactivas. Todas las reglas pasan a estar inactivas y se actualiza el contenido de B1 y B2 convenientemente

```
*/
disactivate_all_rules(B1,B2) :-
        nb_getval(rule_list,L),
        nb_setval(act_active_rule_list,[]),
        nb_setval(act_inactive_rule_list,L),
        retractall(rule(_,_)),
        retractall(inactive_rule(_,_)),
        forall(member((Y,Z),L),assert(inactive_rule(Y,Z))),
        show_rules(B1,B2).


/*
```
activate_rule(+B1,+B2) <=> B1 es la lista gráfica para mostrar las reglas actualmente activas, B2 es la lista gráfica para mostrar las reglas actualmente inactivas. La regla asociada al elemento de B2 seleccionado pasa a estar activa y se actualiza el contenido de B1 y B2 convenientemente

```
*/
activate_rule(B1,B2) :-
         get(B2,selection,S),
         get(S,key,K),
         string_to_atom(K,K2),
         atom_to_term(K2,Term,_),
         nb_getval(act_inactive_rule_list,L1),
```

```
                nb_getval(act_active_rule_list,L3),
                forall( (member((X,Y),L1),subsumes(X<-Y,Term),subsumes(Term,X<-Y)) ,
                        (Rule = (X,Y),nb_setval(act_rule,Rule))),
                nb_getval(act_rule,Rule),
                subtract(L1,[Rule],L2),
                append(L3,[Rule],L4),
                sort(L2,L5),
                sort(L4,L6),
                nb_setval(act_inactive_rule_list,L5),
                nb_setval(act_active_rule_list,L6),
                retractall(inactive_rule(_,_)),
                forall(member((Y1,Z1),L5),assert(inactive_rule(Y1,Z1))),
                retractall(rule(_,_)),
                forall(member((Y2,Z2),L6),assert(rule(Y2,Z2))),
                show_rules(B1,B2).

/*
disactivate_rule(+B1,+B2) <=> B1 es la lista gráfica para mostrar las reglas actualmente
activas, B2 es la lista gráfica para mostrar las reglas actualmente inactivas. La regla aso-
ciada al elemento de B1 seleccionado pasa a estar inactiva y se actualiza el contenido de
B1 y B2 convenientemente
*/
disactivate_rule(B1,B2) :-
                get(B1,selection,S),
                get(S,key,K),
                string_to_atom(K,K2),
                atom_to_term(K2,Term,_),
                nb_getval(act_active_rule_list,L1),
                nb_getval(act_inactive_rule_list,L3),
                forall( (member((X,Y),L1),subsumes(X<-Y,Term),subsumes(Term,X<-Y)) ,
                      (Rule = (X,Y),nb_setval(act_rule,Rule))),

                nb_getval(act_rule,Rule),
                subtract(L1,[Rule],L2),
                append(L3,[Rule],L4),
                sort(L2,L5),
                sort(L4,L6),
                nb_setval(act_active_rule_list,L5),
                nb_setval(act_inactive_rule_list,L6),

                retractall(rule(_,_)),
                forall(member((Y1,Z1),L5),assert(rule(Y1,Z1))),
```

```
                retractall(inactive_rule(_,_)),
                forall(member((Y2,Z2),L6),assert(inactive_rule(Y2,Z2))),

                show_rules(B1,B2).

/*
undo_rules(+F) <=> F es la ventana para la gestión de reglas activas. Se restaura el estado inicial y se cierra la ventana
*/
undo_rules(F) :-
        retractall(rule(_,_)),
        retractall(inactive_rule(_,_)),
        nb_getval(active_rule_list,Lact),
        nb_getval(inactive_rule_list,Liac),
        forall(member((Y1,Z1),Lact),assert(rule(Y1,Z1))),
        forall(member((Y2,Z2),Liac),assert(inactive_rule(Y2,Z2))),
        exit_rules_management(F).

/*
exit_rules_management(+F) <=> F es la ventana para la gestión de reglas activas. Se eliminan todas las variables creadas para la gestión de las reglas activas y se cierra la ventana
*/
exit_rules_management(F) :-
                nb_delete(rule_list),
                nb_delete(active_rule_list),
                nb_delete(inactive_rule_list),
                nb_delete(act_active_rule_list),
                nb_delete(act_inactive_rule_list),
                send(F,destroy).
```

meFormLemmas.pl

```
/*
Se define la herramienta visual, y los métodos y estructuras utilizadas, para la selección de lemas activos
*/

/*
lemma_management_dialog <=> Se muestra la ventana para la gestión de lemas activos
*/
```

```
lemma_management_dialog :-
        new(F, frame('Selección de lemas activos')),

        send(F, append(new(B1, browser(size := size(90,15))))),
        send(new(D1, dialog), below(B1)),
        send(new(B2,browser(size := size(90,15))),below(D1)),
        send(new(D2, dialog), below(B2)),

        send(B1,label('Lemas activos:')),
        send(B2,label('Lemas inactivos:')),

        send(D1,append(button(activar_todas,message(@prolog,activte_all_lemmas,B1,B2)))),
        send(D1, append(button(activar,message(@prolog, activate_lemma,B1,B2)))),
        send(D1, append(button(desactivar,message(@prolog, disactivate_lemma,B1,B2)))),
       send(D1,append(button(desactivar_todas,message(@prolog,
                                    disactivate_all_lemmas,    B1,B2)))),
        send(D2, append(button(aceptar,message(@prolog,exit_lemmas_management,F)))),
        send(D2, append(button(cancelar,message(@prolog,undo_lemmas,F)))),

        get(D1,member,activar,Act),
        get(D1,member,desactivar,Dac),
        get(D1,member,activar_todas,TAct),
        get(D1,member,desactivar_todas,TDac),
        send(Act,label(' /\\ ')),
        send(Dac,label(' \\/ ')),
        send(TAct,label(' /\\ /\\ ')),
        send(TDac,label(' \\/ \\/ ')),

        init_lemmas,
        show_lemmas(B1,B2),

        send(F, open).

/*
init_lemmas <=> Captura los actuales lemas activos y los inactivos y actualiza el contenido de las listas gráficas convenientemente. Se crea una lista de string con la información de todas los lemas activos, otra para los inactivos y otra para todos, para poder recuperar el estado inicial si se llega a requerir. Además, se crea una copia para la lista de lemas activos y otra para la de lemas inactivos para permitir el tratamiento intermedio de activaciones y desactivaciones de lemas
*/
init_lemmas :-
```

```
        findall((X1,Y1,Z1,S1,D1),(lemma(X1,Y1,Z1,S1,D1);
                inactive_lemma(X1,Y1,Z1,S1,D1)),L1),
        sort(L1,L4),
        nb_setval(lemma_list,L4),
        nb_setval(act_lemma_list,L4),
        findall((X2,Y2,Z2,S2,D2),lemma(X2,Y2,Z2,S2,D2),L2),
        sort(L2,L5),
        nb_setval(active_lemma_list,L5),
        nb_setval(act_active_lemma_list,L5),
        findall((X3,Y3,Z3,S3,D3),inactive_lemma(X3,Y3,Z3,S3,D3),L3),
        sort(L3,L6),
        nb_setval(inactive_lemma_list,L6),
        nb_setval(act_inactive_lemma_list,L6),
        retractall(lemma(_,_,_,_,_)),
        forall(member((X4,Y4,Z4,S4,D4),L5),assert(lemma(X4,Y4,Z4,S4,D4))),
        retractall(inactive_lemma(_,_,_,_,_)),
        forall(member((X5,Y5,Z5,S5,D5),L6),assert(inactive_lemma(X5,Y5,Z5,S5,D5))).

/*
show_lemmas(+B1,+B2) <=> B1 es la lista gráfica para mostrar los lemas actualmente
activos, B2 es la lista gráfica para mostrar los lemas actualmente inactivos. Actualiza el
contenido de B1 y B2 convenientemente
*/
show_lemmas(B1,B2) :-
          send(B1,clear),
          send(B2,clear),
          show_active_lemmas(B1),
          show_inactive_lemmas(B2).

/*
show_active_lemmas(+B) <=> B es la lista gráfica para mostrar los lemas actualmente
activos. Se actualizan sus elementos según el contenido de la lista que guarda la informa-
ción de los lemas actualmente activos
*/
show_active_lemmas(B) :-
              new(C,chain),
              findall((X,Y,Z,S,D),lemma(X,Y,Z,S,D),L),
              forall(member(M,L),(term_to_atom(M,At),send(C,append,At))),
              send(B,members(C)).

/*
```

show_inactive_lemmas(+B) <=> B es la lista gráfica para mostrar los lemas actualmente inactivos. Se actualizan sus elementos según el contenido de la lista que guarda la información de los lemas actualmente inactivos
*/
```
show_inactive_lemmas(B) :-
        new(C,chain),
        findall((X,Y,Z,S,D),inactive_lemma(X,Y,Z,S,D),L),
        forall(member(M,L),(term_to_atom(M,At),send(C,append,At))),
        send(B,members(C)).
```

/*
activate_all_lemmas(+B1,+B2) <=> B1 es la lista gráfica para mostrar los lemas actualmente activos, B2 es la lista gráfica para mostrar los lemas actualmente inactivos. Todas los lemas pasan a estar activos y se actualiza el contenido de B1 y B2 convenientemente
*/
```
activate_all_lemmas(B1,B2) :-
        nb_getval(lemma_list,L),
        nb_setval(act_active_lemma_list,L),
        nb_setval(act_inactive_lemma_list,[]),
        retractall(lemma(_,_,_,_,_)),
        retractall(inactive_lemma(_,_,_,_,_)),
        forall(member((X,Y,Z,S,D),L),assert(lemma(X,Y,Z,S,D))),
        show_lemmas(B1,B2).
```

/*
disactivate_all_lemmas(+B1,+B2) <=> B1 es la lista gráfica para mostrar los lemas actualmente activos, B2 es la lista gráfica para mostrar los lemas actualmente inactivos. Todos los lemas pasan a estar inactivos y se actualiza el contenido de B1 y B2 convenientemente
*/
```
disactivate_all_lemmas(B1,B2) :-
        nb_getval(lemma_list,L),
        nb_setval(act_active_lemma_list,[]),
        nb_setval(act_inactive_lemma_list,L),
        retractall(lemma(_,_,_,_,_)),
        retractall(inactive_lemma(_,_,_,_,_)),
        forall(member((X,Y,Z,S,D),L),assert(inactive_lemma(X,Y,Z,S,D))),
        show_lemmas(B1,B2).
```

/*
activate_lemma(+B1,+B2) <=> B1 es la lista gráfica para mostrar los lemas actualmente activos, B2 es la lista gráfica para mostrar los lemas actualmente inactivos. El lema aso-

ciado al elemento de B2 seleccionado pasa a estar activo y se actualiza el contenido de B1 y B2 convenientemente
*/
```
activate_lemma(B1,B2) :-
        get(B2,selection,Sel),
        get(Sel,key,K),
        string_to_atom(K,K2),
        atom_to_term(K2,Term,_),
        nb_getval(act_inactive_lemma_list,L1),
        nb_getval(act_active_lemma_list,L3),
        forall( (member((X,Y,Z,S,D),L1),Term == (X,Y,Z,S,D)) ,
                (Lemma = (X,Y,Z,S,D),nb_setval(act_lemma,Lemma))),
        nb_getval(act_lemma,Lemma),
        subtract(L1,[Lemma],L2),
        append(L3,[Lemma],L4),
        sort(L2,L5),
        sort(L4,L6),
        nb_setval(act_inactive_lemma_list,L5),
        nb_setval(act_active_lemma_list,L6),

        retractall(inactive_lemma(_,_,_,_,_)),
        forall(member((X1,Y1,Z1,S1,D1),L5),
        assert(inactive_lemma(X1,Y1,Z1,S1,D1))),

        retractall(lemma(_,_,_,_,_)),
        forall(member((X2,Y2,Z2,S2,D2),L6),assert(lemma(X2,Y2,Z2,S2,D2))),

        show_lemmas(B1,B2).
```

/*
disactivate_lemma(+B1,+B2) <=> B1 es la lista gráfica para mostrar los lemas actualmente activos, B2 es la lista gráfica para mostrar los lemas actualmente inactivos. El lema asociado al elemento de B1 seleccionado pasa a estar inactivo y se actualiza el contenido de B1 y B2 convenientemente
*/
```
disactivate_lemma(B1,B2) :-
        get(B1,selection,Sel),
        get(Sel,key,K),
        string_to_atom(K,K2),
        atom_to_term(K2,Term,_),
        nb_getval(act_active_lemma_list,L1),
        nb_getval(act_inactive_lemma_list,L3),
```

```
                forall( (member((X,Y,Z,S,D),L1),Term == (X,Y,Z,S,D)) ,
                        (Lemma = (X,Y,Z,S,D),nb_setval(act_lemma,Lemma))),
                nb_getval(act_lemma,Lemma),
                subtract(L1,[Lemma],L2),
                append(L3,[Lemma],L4),
                sort(L2,L5),
                sort(L4,L6),
                nb_setval(act_active_lemma_list,L5),
                nb_setval(act_inactive_lemma_list,L6),

                retractall(lemma(_,_,_,_,_)),
                forall(member((X1,Y1,Z1,S1,D1),L5),assert(lemma(X1,Y1,Z1,S1,D1))),

                retractall(inactive_lemma(_,_,_,_,_)),
                forall(member((X2,Y2,Z2,S2,D2),L6),
                        assert(inactive_lemma(X2,Y2,Z2,S2,D2))),

                show_lemmas(B1,B2).

/*
undo_lemmas(+F) <=> F es la ventana para la gestión de lemas activos. Se restaura el
estado inicial y se cierra la ventana
*/
undo_lemmas(F) :-
        retractall(lemma(_,_,_,_,_)),
        retractall(inactive_lemma(_,_,_,_,_)),
        nb_getval(active_lemma_list,Lact),
        nb_getval(inactive_lemma_list,Liac),
        forall(member((X1,Y1,Z1,S1,D1),Lact),assert(lemma(X1,Y1,Z1,S1,D1))),
        forall(member((X2,Y2,Z2,S2,D2),Liac),assert(inactive_lemma(X2,Y2,Z2,S2,D2))),
        exit_lemmas_management(F).

/*
exit_lemmas_management(+F) <=> F es la ventana para la gestión de lemas activos. Se
eliminan todas las variables creadas para la gestión de los lemas activos y se cierra la ven-
tana
*/
exit_lemmas_management(F) :-
                nb_delete(lemma_list),
                nb_delete(active_lemma_list),
                nb_delete(inactive_lemma_list),
                nb_delete(act_active_lemma_list),
```

```
        nb_delete(act_inactive_lemma_list),
        send(F,destroy).
```

meFormDirectories.pl

```
/*
Se define la herramienta visual, y los métodos utilizados por ella, para la selección del directorio de trabajo
*/

/*
dir_viewer <=> Muestra el explorador para la elección del directorio de trabajo
*/
dir_viewer :-
    new(F, frame('Directorio de trabajo')),
    send(F, append(new(B, browser(size := size(30,20))))),
    send(new(D1, dialog), above(B)),
    send(new(D2, dialog), below(B)),
    working_directory(Dir,Dir),
    nb_setval(dirInicial,Dir),
    dir_view(B),
    send(D1, append(button(subir,message(@prolog,dir_up,B)))),
    send(D1, append(button(entrar,message(@prolog, dir_down,B?selection?key,B)))),

    send(D1,append(new(M,menu(cycle,cycle)))),
    send(M,label,'Unidad'),
    send_list(M,append,[
        menu_item(c, message(@prolog,change_device,B,'C:/'), 'C:', @off, @nil, '\\ec'),
        menu_item(d, message(@prolog,change_device,B,'D:/'), 'D:', @off, @nil, '\\ed'),
        menu_item(e, message(@prolog,change_device,B,'E:/'), 'E:', @off, @nil, '\\ee'),
        menu_item(f, message(@prolog,change_device,B,'F:/'), 'F:', @off, @nil, '\\ef'),
        menu_item(g, message(@prolog,change_device,B,'G:/'), 'G:', @off, @nil, '\\eg'),
        menu_item(h, message(@prolog,change_device,B,'H:/'), 'H:', @off, @nil, '\\eh'),
        menu_item(i, message(@prolog,change_device,B,'I:/'), 'I:', @off, @nil, '\\ei'),
        menu_item(j, message(@prolog,change_device,B,'J:/'), 'J:', @off, @nil, '\\ej'),
        menu_item(k, message(@prolog,change_device,B,'K:/'), 'K:', @off, @nil, '\\ek'),
        menu_item(l, message(@prolog,change_device,B,'L:/'), 'L:', @off, @nil, '\\el'),
        menu_item(m, message(@prolog,change_device,B,'M:/'), 'M:', @off, @nil, '\\em'),
        menu_item(n, message(@prolog,change_device,B,'N:/'), 'N:', @off, @nil, '\\en'),
        menu_item(o, message(@prolog,change_device,B,'O:/'), 'O:', @off, @nil, '\\eo'),
        menu_item(p, message(@prolog,change_device,B,'P:/'), 'P:', @off, @nil, '\\ep'),
        menu_item(q, message(@prolog,change_device,B,'Q:/'), 'Q:', @off, @nil, '\\eq'),
```

```
            menu_item(r, message(@prolog,change_device,B,'R:/'), 'R:', @off, @nil, '\\er'),
            menu_item(s, message(@prolog,change_device,B,'S:/'), 'S:', @off, @nil, '\\es'),
            menu_item(t, message(@prolog,change_device,B,'T:/'), 'T:', @off, @nil, '\\et'),
            menu_item(u, message(@prolog,change_device,B,'U:/'), 'U:', @off, @nil, '\\eu'),
            menu_item(v, message(@prolog,change_device,B,'V:/'), 'V:', @off, @nil, '\\ev'),
            menu_item(w, message(@prolog,change_device,B,'W:/'), 'W:', @off, @nil, '\\ew'),
            menu_item(x, message(@prolog,change_device,B,'X:/'), 'X:', @off, @nil, '\\ex'),
            menu_item(y, message(@prolog,change_device,B,'Y:/'), 'Y:', @off, @nil, '\\ey'),
            menu_item(z, message(@prolog,change_device,B,'Z:/'), 'Z:', @off, @nil, '\\ez')
        ]),

        send(D2, append(button(aceptar,message(F, destroy)))),
        send(D2, append(button(cancelar,message(@prolog, dir_undo,F)))),

        get(D1,member,subir,S),
        get(D1,member,entrar,E),
        send(S,label('/..')),
        send(E,label('/->')),
        send(S,width(25)),
        send(E,width(25)),

        send(F, open).

/*
dir_up(+Browser) <=> Se cambia el directorio de trabajo al directorio padre del actual.
Se actualiza el contenido de la lista gráfica Browser con los nombres de los directorios
contenidos en el nuevo directorio de trabajo
*/
dir_up(Browser) :-
    chdir(..),
    dir_view(Browser).

/*
dir_down(+DirNuevo,+Browser) <=> Se cambia el directorio de trabajo al de ruta Dir-
Down. Se actualiza el contenido de la lista gráfica Browser con los nombres de los directo-
rios contenidos en el nuevo directorio de trabajo
*/
dir_down(DirNuevo,Browser) :-
        working_directory(Dir,Dir),
        string_concat(Dir,DirNuevo,Dir1),
        working_directory(_,Dir1),
        dir_view(Browser).
```

```
/*
dir_view(+Browser) <=> Se actualiza el contenido de la lista gráfica Browser con los
nombres de los directorios contenidos en el actual directorio de trabajo
*/
dir_view(Browser) :-
        working_directory(Dir,Dir),
        send(Browser,label(Dir)),
        send(Browser, members(directory(Dir)?directories)).

/*
dir_undo(+F) <=> Cambia el directorio de trabajo al que lo era cuando se abrió la ventana y se cierra la ventana
*/
dir_undo(F) :-
        nb_getval(dirInicial,Dir),
        chdir(Dir),
        send(F,destroy).

/*
change_device(+Browser,+D) <=> Cambia de unidad indicada con la variable D y actualiza el contenido de la lista Broser
*/
change_device(Browser,D) :-

catch(working_directory(_,D),_,(send(Browser,clear),send(Browser,label(D)),fail)),
                dir_view(Browser).
```

Autorización

Los autores de este proyecto abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Delgado Muñoz, Agustín Daniel

Novillo Vidal, Álvaro

Pérez Morente, Fernando