Universidad Nacional del Sur

Tesis de Magister en Ciencias de la Computación

La Programación en Lógica Rebatible su definición teórica y computacional

Alejandro Javier García

Bahía Blanca Argentina

1997

Universidad Nacional del Sur

Tesis de Magister en Ciencias de la Computación

La Programación en Lógica Rebatible su definición teórica y computacional

Alejandro Javier García

Bahía Blanca Argentina

1997

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Magister en Ciencias de la Computación, de la Universidad Nacional del Sur, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias de la Computación, durante el período comprendido entre el 1 de junio de 1994 y el 19 de junio de 1997, bajo la dirección del Dr. Guillermo R. Simari, Profesor Titular del Departamento de Ciencias de la Computación.

Lic. Alejandro Javier García
ccgarcia@criba.edu.ar
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca, 19 de Junio de 1997.

Agradecimientos

En primer lugar, quiero expresar mi sincero agradecimiento a mi director, Guillermo Simari, quien me introdujo en la investigación en Ciencias de la Computación, me enseñó a querer esta actividad y me permitió comenzar mi formación bajo su dirección. Esta tesis no hubiera sido posible sin su apoyo y la dedicación que supo ofrecerme.

Agradezco a mis compañeros del Departamento de Ciencias de la Computación, y del Grupo de Investigación en Inteligencia Artificial, por su amistad, el entorno de trabajo creado, y su buena predisposición para confrontar y debatir ideas. Agradezco también a los investigadores citados en esta tesis, a quienes en su mayoría no conozco personalmente, pero que a través de sus trabajos han permitido que esta investigación sea posible; especialmente a David H. D. Warren, y Hassan Aït-Kaci. También agradezco a los integrantes del jurado que evaluaron este trabajo, Ronald P. Loui y Enrique Forradelas, por sus valiosos comentarios. Agradezco al Consejo Nacional de Investigaciones Científicas y Técnicas de la República Argentina, quien confió en mi otorgándome el financiamiento necesario para realizar esta investigación.

Finalmente quiero agradecer especialmente a Cori, por su apoyo incondicional, y permitirme compartir con ella la alegría de realizar este trabajo. Quiero dedicar esta tesis a mis padres en agradecimiento por su cariño, dedicación y preocupación, y por la confianza que han depositado en mi, pero fundamentalmente por el apoyo que me han dado siempre para que pueda trabajar en lo que me gusta.

Resumen

La programación en lógica se ha convertido en uno de los principales exponentes de la programación declarativa. Sin embargo, aunque se la ha utilizado como herramienta de representación de conocimiento, presenta limitaciones para adaptarse al razonamiento del sentido común. Paralelamente al desarrollo de la programación en lógica, los investigadores en Inteligencia Artificial han continuado produciendo nuevos formalismos, con el fin de obtener mejores métodos de razonamiento y de representación de conocimiento. Uno de estos formalismos es la argumentación rebatible, donde se pone especial énfasis en la noción de argumento. Un argumento para una conclusión C constituye una pieza de razonamiento tentativa que un agente inteligente está dispuesto a aceptar para explicar C. Si el agente adquiriese luego nueva información, la conclusión C junto con el razonamiento que la produjo podrían quedar invalidados.

Como los agentes inteligentes tienden a razonar en una forma rebatible, sería sumamente interesante disponer de un paradigma de programación donde conclusiones previas puedan ser refutadas ante la presencia de mayor información. Por lo tanto, uno de los objetivos perseguidos en esta tesis es definir un lenguaje de programación en lógica rebatible como una extensión de la programación en lógica convencional. Este nuevo lenguaje captura aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Permite además la representación de información incompleta y potencialmente inconsistente, y utiliza los conceptos de la argumentación rebatible a fin de poder decidir entre metas contradictorias. La respuesta a una consulta involucra el análisis de sus argumentos, y contraargumentos, con el fin de obtener una justificación.

El segundo objetivo perseguido en esta tesis, es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. La Maquina Abstracta de Warren o WAM (Warren's Abstract Machine), se ha convertido en el estándar *de facto*

para la implementación de compiladores para el lenguaje Prolog. Por lo tanto, se diseñó una nueva máquina abstracta, llamada JAM (Justification Abstract Machine) como una extensión de la WAM. La arquitectura de la JAM estará formada al igual que en la WAM por un conjunto de intrucciones, una estructura de memoria, y un conjunto de registros, que permitirán construir argumentos, contraargumentos y generar una justificación para las consultas.

De la misma forma que la programación en lógica significó una evolución de los lenguajes de programación, la utilización de la argumentación rebatible como paradigma de programación, podría permitir un nuevo avance.

Índice General

Capítulo 1

Introducción

Desde su concepción, la lógica ha estado ligada al razonamiento científico. La lógica provee un lenguaje preciso para la representación de nuestro conocimiento y para la formulación explícita de nuestras hipótesis y objetivos. Introduce además los fundamentos necesarios para obtener conclusiones a partir de nuestras premisas, establecer la consistencia de nuestras creencias y verificar la validez de nuestros argumentos.

También desde su aparición, las computadoras han sido utilizadas como una herramienta científica. Los lenguajes de programación se han desarrollados para escribir en una notación cercana a la utilizada por los científicos, programas que solucionen o formalicen los problemas de los investigadores. Al igual que la lógica, los lenguajes de programación necesitan de una descripción precisa y explícita de nuestro conocimiento, y objetivos.

La tarea de hacer explícito nuestro conocimiento no es trivial. Sin embargo, la formalización del mismo mediante el uso de la lógica, permite analizar los problemas desde una mejor posición. Lo mismo debería ocurrir con los lenguajes de programación. Pero lamentablemente, el desarrollo de los lenguajes de programación se vió afectado en sus comienzos por severas restricciones de índole tecnológica, produciendo un marcado esfuerzo de diseño desde el punto de vista técnico y no desde la óptica de los usuarios. En los últimos años, esta situación ha cambiando, con la aparición de nuevos paradigmas de programación, más cercanos al lenguaje natural.

La programación en lógica

Aunque la lógica a sido usada como una herramienta para diseñar tanto computadoras como lenguajes de programación, el uso de la lógica directamente como un lenguaje de programación es muy reciente [Kowa74, CKRP73, Green69, Hayes73, BatMel73, Rousel75]. La *programación en lógica* surgió al principios de la década del 70, a raíz de los trabajos que se habían desarrollado en prueba automática de teoremas, y en inteligencia artificial. Fue introducida por Kowalsky [Kowa74] y Colmerauer [CKRP73], y en 1972 Roussel [Rousel75] implementó el primer intérprete del lenguaje PROLOG.

La idea central en la programación en lógica, es el uso de la lógica de primer orden (o al menos un subconjunto de ella), como un lenguaje de programación. Siguiendo la idea de Kowalsky, un algoritmo consiste de dos componentes disjuntas: *la lógica*, y *el control*. La componente lógica es la que establece *que* es lo que debe resolverse, mientas que la componente de control establece *como* debe resolverse. Esta separación permite al programador concentrarse en la componente lógica de su algoritmo, dejando el control para que sea resuelto por el sistema de programación en lógica. Un programa lógico [Lloyd87] está constituido por un conjunto finito de axiomas que representa nuestro conocimiento y las hipótesis del problema a resolver, y una sentencia meta que representa el objetivo que queremos alcanzar. La ejecución de un programa lógico implica la demostración de la sentencia meta a partir de los axiomas dados.

Sistemas de Razonamiento Rebatible

Paralelamente al desarrollo de la programación en lógica, los investigadores en Inteligencia Artificial (IA) han continuado produciendo nuevos formalismos, con el fin de obtener mejores métodos de razonamiento y de representación de conocimiento. Tanto la implementación de métodos de razonamiento, como la representación de conocimiento, siguen siendo problemas abiertos, y constituyen áreas centrales dentro de investigación en IA, ya que cualquier avance en estas áreas, redundará en un beneficio casi imnediato para muchas otras áreas de las Ciencias de la Computación.

Aunque la lógica clásica ha sido utilizada desde sus inicios como una formalización del razonamiento humano, razonar implica mecanismos mucho más complejos que los que nos provee la lógica clásica. Por ejemplo, una de las características de nuestro razonamiento es que la aparición de nueva evidencia, puede invalidar conclusiones obtenidas

con anterioridad. Sin embargo, las lógicas de primer orden, cumplen con la propiedad de *monotonicidad*: " si S es un conjunto de sentencias, y p se deduce de S ($S \vdash p$), entonces para cualquier sentencia f, se cumple que $S \cup \{f\} \vdash p$." Es decir, en la lógica clásica, agregar nuevos axiomas o hipótesis a la teoría no invalida viejos teoremas, y por lo tanto no es posible desechar conclusiones obtenidas con anterioridad.

Los sistemas de razonamiento no monótonos (SRNM) carecen de la propiedad de monotonía, y han sido desarrollados con el objetivo de intentar solucionar los problemas que acarrea la utilización de la lógica clásica como herramienta para formalizar conocimiento. Los SRNM más importantes que se han desarrollados son: *razonamiento rebatible* [Pollock87, Nute87], *lógica default* [Reiter80], *lógica no-monotónica* [McDDoy80], *lógica autoepistémica* [Moore85], *circunscripción* [McCar80], y la *argumentación rebatible* [SimLou92, Prakken, Vrees93, Dung93a, LinSho89, GCS93b].

La argumentación rebatible es una formalización del razonamiento rebatible, donde se pone especial énfasis en la noción de *argumento*. Un argumento para una conclusión C constituye una pieza de razonamiento tentativa que un agente inteligente está dispuesto a aceptar para explicar C. Si el agente adquiriese luego nueva información, la conclusión C junto con el razonamiento que la produjo podrían quedar invalidados. De esta manera aparece la no monotonía en el sistema.

En un sistema de argumentación rebatible, una conclusión C será aceptada como una nueva creencia, cuando exista un argumento que sea una *justificación* de C. El proceso de obtención de una justificación para C, involucra la construcción de un argumento $\mathcal{A}$, para C, que no esté derrotado. Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, llamado *árbol de dialéctica*, donde $\mathcal{A}$ es la raíz, y cada nodo tiene por hijos a sus derrotadores. En el árbol de dialéctica, argumentos y contraargumentos son comparados utilizando un criterio de preferencia. La argumentación rebatible puede definirse independientemente del criterio de preferencia entre argumentos, no obstante, en este trabajo se asumirá que se utiliza el criterio de *especificidad* [Loui87, Poole85a, GarSim96b].

Extensiones de la programación en lógica

La programación en lógica se ha convertido en uno de los principales exponentes de la programación declarativa. Sin embargo, aunque se la ha utilizado como herramienta de representación de conocimiento, presenta limitaciones para adaptarse al razonamiento rebatible. Como los agentes inteligentes tienden a razonar en una forma rebatible, sería sumamente interesante disponer de un paradigma de programación donde conclusiones previas puedan ser refutadas ante la presencia de mayor información. De la misma forma que la programación en lógica significó una evolución de los lenguajes de programación, la utilización de la argumentación rebatible como paradigma de programación, podría permitir un nuevo avance.

En los últimos años se han desarrollado extensiones de la programación en lógica que incorporan algunos aspectos del razonamiento rebatible. Por ejemplo, Gelfond y Lifschitz en su trabajo *Logic Programming with Classical Negation* [GelLif90] introdujeron los programas lógicos extendidos (PLE) que permiten la representación de la negación clásica en los programas, y la derivación de literales negados. Los PLE presentan ventajas importantes para representar conocimiento, pero lamentablemente cuando dos literales complementarios (por ej. P y $\neg P$) pueden derivarse de un PLE, el programa se torna "contradictorio", y entonces pueden derivarse todos los literales del programa. Como habitualmente el conocimiento maneja información tentativa (defaults), y la representación de información tentativa en la mayoría de los casos lleva a bases de conocimiento inconsistentes, entonces el programa casi siempre derivará todo el lenguaje. Este es uno de los puntos que trataron de resolver Kowalski y Sadri en [KowSad90] y Inoue en [Inoue91].

El trabajo de Kowalsky y Sadri *Logic Programming with Exceptions* [KowSad90] utiliza los PLE para poder representar información negativa, y los extiende para la representación de excepciones a reglas generales. Por ejemplo, los pingüinos son una excepción a la regla "las aves vuelan". La idea que proponen es representar las reglas generales con cláusulas de la forma "$V \leftarrow A$"y las excepciones con cláusulas "$\neg V \leftarrow P$", luego a la hora de calcular la respuesta, las excepciones tienen prioridad sobre las reglas. Lamentablemente la forma en que diferencian una regla de una excepción, es que las primeras tienen un consecuente positivo y las últimas un consecuente negado. Esto representa una restricción muy grande, y trae aparejado confusiones, ya que la cláusula "los mamíferos no vuelan" representaría una excepción a la regla "los murciélagos vuelan".

El trabajo de Katsumi Inoue *Extended Logic Programming with Default Assumptions* [Inoue91] extiende el trabajo de Gelfond y Litfschitz para poder trabajar con información potencialmente inconsistente. También ataca el problema de los programas contradictorios, pero lo hace separando la información absoluta, de la información tentativa o hipotética. De esta forma un "sistema de conocimiento" es un par (T, H) donde T es un conjunto de hechos que son verdaderos en el dominio en cuestión, y H es un conjunto de hipótesis. La tarea del sistema es hallar un subconjunto E de H tal que el conjunto de respuestas de $T \cup E$ sea consistente. El enfoque de Inoue se asemeja en algunas cosas a un sistema de argumentación rebatible. No obstante, presenta dificultades cuando se tiene que elegir entre dos explicaciones contradictorias, ya que no tiene definido un criterio de preferencia.

La programación en lógica rebatible

Uno de los objetivos de esta tesis es definir un lenguaje de programación en lógica rebatible como una extensión de la programación en lógica convencional. De esta forma, se podrán capturar los aspectos antes citados del razonamiento rebatible, y solucionar los problemas de las extensiones mencionadas anteriormente. Este nuevo lenguaje capturará aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Permitirá además la representación de información incompleta y potencialmente inconsistente, y utilizará los conceptos de la *argumentación rebatible* a fin de poder decidir entre metas contradictorias.

Los *programas lógicos rebatibles* (PLR), estarán formados por dos tipos de cláusulas: las *cláusulas de programa extendido*, y las *cláusulas de programa rebatible*. De esta forma, un programa estará formado por dos conjuntos disjuntos de cláusulas: uno que representa conocimiento estricto (seguro), y otro que contiene información rebatible (tentativa). Un argumento será definido como un subconjunto particular de un PLR, y una vez construído, se podrán utilizar los conceptos de *justificación* y *derrota* de la argumentación rebatible para obtener la respuesta de la consulta efectuada.

Los programas lógicos rebatibles permitirán utilizar dos tipos de negación: la *negación por falla finita* (`not`), y la *negación clásica* ($\sim$). Esto permitirá representar información incompleta, y potencialmente inconsistente. Como los PLR incorporarán algunos conceptos que no estaban presentes en los sistemas de argumentación rebatible (como la negación

por falla, y las presuposiciones), se extenderá el criterio de especificidad para comparar correctamente los argumentos obtenidos.

Una máquina abstracta para la programación en lógica rebatible

Como se dijo antes, el primer intérprete de PROLOG fue desarrollado por el equipo de Colmerauer en 1972, en la Universidad de Marseille, como resultado de un sistema de demostración de teoremas especializado para el procesamiento de lenguaje natural [CKRP73]. Luego entre 1973 y 1975 fue desarrollado en esa misma universidad, un intérprete de PROLOG implementado en el lenguaje Fortran [BatMel73].

Pero en 1977 David H. D. Warren, diseñó una *máquina abstracta* para la ejecución de PROLOG, que consiste de una arquitectura de memoria y un conjunto de instrucciones [Warren77, Warren83]. Este diseño se denominó posteriormente Maquina Abstracta de Warren o WAM (Warren's Abstract Machine) y se ha convertido en el estándar *de facto* para la implementación de compiladores para el lenguaje PROLOG [Aït91].

La ventaja de tener una máquina abstracta (MA) es que dado un programa P en un lenguaje $\mathcal{L}$, el programa P puede ser traducido a instrucciones de la MA, y luego ejecutar sobre la arquitectura de la máquina virtual dichas instrucciones. La MA puede implementarse como una máquina virtual, o directamente sobre una máquina real, o diseñarse un nuevo procesador con la arquitectura de la MA diseñada. En el caso de la WAM, estas tres alternativas han sido desarrolladas [VanRoy90].

El segundo objetivo perseguido en esta tesis, es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. La nueva máquina abstracta (que se llamará JAM), estará diseñada como una extensión de la WAM. La arquitectura de la JAM estará formada al igual que en la WAM por un conjunto de intrucciones, una estructura de memoria, y un conjunto de registros, que permitirán construir argumentos, contraargumentos y generar el árbol de dialéctica necesario para obtener una justificación.

Además del desarrollo de la nueva máquina abstracta, se implementó un compilador que toma como entrada un programa lógico rebatible (PLR), y produce como salida un programa en instrucciones de la JAM. Para realizar este trabajo, se combinaron técnicas convencionales de compilación, como el análisis recursivo descendente, con técnicas especiales para la programación en lógica. De esta forma, cada cláusula de programa puede compilarse por separado e integrarse a un único código ejecutable. Esto permite que el

programa este dividido en varios archivos, o agregar nuevas cláusulas al programa sin necesidad de recompilar el programa entero. El compilador además permite la generación de código para predicados predefinidos del lenguaje. El compilador construído puede utilizarse también para traducir programas en PROLOG a código WAM, y podría modificarse fácilmente para producir compiladores para otras extensiones de la programación en lógica. Finalmente, la nueva máquina abstracta fue implementada como una máquina virtual, y sobre ella se implementó un intérprete para programas lógicos rebatibles.

Organización de la tesis

Esta tesis está organizada de la siguiente manera:

Capítulo 2: Está dedicado integramente a introducir los conceptos fundamentales de la programación en lógica, y las extensiones mencionadas antes.

Capítulo 3: Se describrirá en detalle el sistema de argumentación rebatible propuesto por Simari y Loui [SimLou92], el cuál es utilizado para desarrollar el mecanismo de inferencia de la programación en lógica rebatible.

Capítulo 4: Se define la programación en lógica rebatible: se presenta el lenguaje, su semántica, se define el método de inferencia, se definen los dos tipos de negación, y se desarrolla una extensión del criterio de especificidad, para los nuevos elementos del lenguaje.

Capítulo 5: Se presenta el diseño de un conjunto de algoritmos, que permite construir por encadenamiento hacia atrás de cláusulas de programa, argumentos y contrargumentos, utilizando únicamente la información contenida en los argumentos involucrados. De esta forma, el costo de obtener una justificación estará relacionado únicamente con los argumentos que forman el árbol de dialéctica, y será independiente del tamaño del programa. Estos algoritmos serán utilizados para la implementación de las instrucciones de la nueva máquina abstracta.

Capítulo 6: Está dedicado integramente a la presentar la Máquina Abstracta de Warren (WAM). Se presenta su arquitectura, el funcionamiento de sus instrucciones, y la forma en que se implementan. Se desarrolla la implementación de un compilador para programas lógicos, que tiene como salida instrucciones (WAM). Se muestra además cómo implementar un intérprete sobre esta máquina virtual.

Capítulo 7: Presenta el desarrollo de la nueva máquina abstracta para la programación en lógica rebatible, llamada JAM (Justification Abstract Machine). Se describe cómo se construyen argumentos, contraargumentos, y cómo se obtinene una justificación para una consulta de un programa lógico rebatible. Se extiende el compilador desarrollado para poder compilar programas lógicos rebatibles, y se implementa un intérprete que corre sobre la máquina virtual desarrollada.

Capítulo 8: Finalmente se detallan los resultados y conclusiones obtenidas.

Capítulo 2

La programación en lógica y algunas de sus extensiones

Aunque la lógica a sido usada como una herramienta para diseñar tanto computadoras como lenguajes de programación, el uso de la lógica directamente como un lenguaje de programación es muy reciente [Kowa74, CKRP73, Green69, Hayes73, BatMel73, Rousel75]. La programación en lógica surgió al principio de la década del 70, a raíz de los trabajos que se habían desarrollado en prueba automática de teoremas, y en inteligencia artificial. Fue introducida por Kowalsky [Kowa74] y Colmerauer [CKRP73], y en 1972 Roussel [Rousel75] implementó en Algol-W, el primer intérprete del lenguaje Prolog.

La principal idea en la programación en lógica, es el uso de la lógica de primer orden (o al menos un subconjunto de ella), como un lenguaje de programación. Siguiendo la idea de Kowalsky, un algoritmo consiste de dos componentes disjuntas: la lógica, y el control. La componente lógica es la que establece *que* es lo que debe resolverse, mientas que la componente de control establece *como* debe resolverse. La separación de estas dos componentes permite al programador concentrarse en la componente lógica de su algoritmo, dejando el control para que sea resulto por el sistema de programación en lógica.

Un programa lógico [Lloyd87] está consitituido por un conjunto finito de axiomas que representa nuestro conocimiento y las hipótesis del problema a resolver, y una sentencia meta que representa el objetivo que queremos alcanzar. La ejecución de un programa lógico implica la demostración de la sentencia meta a partir de los axiomas dados.

El objetivo de este capítulo es introducir los principales conceptos de la programación en lógica, y desarrollar algunas extensiones que han sido producidas recientemente, a fin de incorporar nuevas características al lenguaje. Dichas extensiones son analizadas en detalle indicando las ventajas y desventajas que presentan.

2.1 El lenguaje de la programación en lógica

A continuación se describirá brevemente el lenguaje de la programación en lógica. En esta tésis se asumirá que el lector posee conocimiento de programación en lógica, y del lenguaje Prolog. Se incluirán en esta sección sólo las definiciones necesarias para el resto de los capítulos. Un desarrollo completo sobre la programación en lógica se encuentra en [Lloyd87] y [Das92], de donde fueron extraídas estas definiciones. Una descripción detallada del lenguaje Prolog puede encontrase en [SteSha86, Bratko87].

Definición 2.1.1 : *Términos*
En el lenguaje de la programación en lógica un término tiene la siguiente definición resursiva:

1. Una variable es un término.
2. Una constante es un término.
3. Si f es un símbolo de functor, de aridad n, y $t_1, \ldots, t_n$ son términos, entonces $f(t_1, \ldots, t_n)$ es un término.

□

Como convención tipográfica para los términos, las variables serán denotadas con una mayúscula inicial, y tanto las constantes como los símbolos de functores serán denotados con una minúscula inicial. También es usual denotar f/n para indicar que el functor f tiene aridad n.

Definición 2.1.2 : *Átomos y Literales*
Si $t_1, \ldots, t_n$ son términos, un *átomo, (predicado, o formula atómica)* de aridad n, será denotado $p(t_1, \ldots, t_n)$ donde el símbolo de predicado p se denotará con una minúscula inicial.
Un *literal* es un átomo, o la negación de un átomo. □

Definición 2.1.3 : *Clásula de programa definido*
Una cláusula de programa definido es una cláusula de la forma:

$$A \leftarrow B_1, \ldots, B_n.$$

que contiene un único átomo A como consecuente, llamado *cabeza* de la cláusula, y una conjunción de átomos $B_1, \ldots, B_n$ como antecedente, llamada *cuerpo* de la cláusula. □

La semántica informal de $A \leftarrow B_1, \ldots, B_n.$ según Lloyd [Lloyd87] es: "si para cada asignación de cada variable, $B_1, \ldots, B_n$ es verdadero, entonces A es verdadero."

Definición 2.1.4 : *Programa definido*
Un programa definido, es un conjunto finito de cláusulas de programa definido. □

Definición 2.1.5 : *Meta definida*
Una meta definida es una cláusula de la forma:

$$\leftarrow B_1, \ldots, B_n.$$

donde cada B_i se llamará *submeta*. □

A lo largo de la tesis, el símbolo "$\sim$" representará la negación clásica, y el símbolo "`not`" la negación por falla. Dos literales serán complementarios con respecto a la negación clásica, si uno es la negación del otro. Para indicar el complemento del literal l se denotará $\overline{l}$. Por lo tanto, $\overline{a} = \sim a$, y $\overline{\sim a} = a$. La complementariedad está definida siempre sobre la negación clásica "$\sim$".

Definición 2.1.6 : *Clásula de programa normal* [Lloyd87]
Una cláusula de programa normal es una cláusula de la forma:

$$A \leftarrow L_1, \ldots, L_n.$$

Donde A es un átomo, y los $L_1, \ldots, L_n$ son literales. En el caso de los programas normales, se utiliza la negación por falla `not` para los literales. □

Un *programa normal* es un conjunto finito de cláusulas de programa normal. La programación en lógica tradicional, utiliza programas normales, y por ende no es posible escribir una cláusula como la siguiente: "$\sim$`vuela` $\leftarrow$ `mamifero`".

2.2 Extensiones de la programación en lógica

La programación en lógica ha sido utilizada ampliamente en sistemas de representación de conocimiento. Esto ha llevado a considerar extensiones que permitan una mayor expresividad en el lenguaje. Cada una de estas extensiones ha incorporado ventajas, pero también han traido aparejado problemas de índole conceptual y computacional. En 1990 Gelfond y Lifschitz en *Logic Programs with Classical Negation* [GelLif90] extendieron la programación en lógica para permitir la utilización de la negación clásica en los programas lógicos. En ese mismo año, Kowalski y Sadri motivados por este trabajo, y por su investigación en representación de conocimiento con restricciones de integridad, extendieron el trabajo de Gelfond y Lifschitz en *Logic Programs with Exceptions* [KowSad90], introduciendo la posibilidad de representar excepciones dentro de un programa lógico. Un año mas tarde, Katsumi Inoue, en *Extended Logic Programs with Default Assumptions* [Inoue91] incorpora hipótesis default a los programas lógicos extendidos de Gelfond y Lifschitz.

En esta sección es describirán las extensiones mencionadas, mostrando las soluciones que incorpora cada una de ellas, así como también los problemas y fallas que presentan. En los capítulos siguientes se mostrará como pueden solucionarse estos problemas utilizando la *programación en lógica rebatible*.

El trabajo de Gelfond y Lifschitz [GelLif90] consiste en incorporar a los programas lógicos, además de la negación por falla finita (`not`), la negación clásica ($\sim$). De esta forma podría escribirse la cláusula "$\sim$`vuela` $\leftarrow$ `mamifero, not murcielago`". Para ello definen los programas lógicos extendidos (PLE), y describen una semántica para dichos programas basada en el método de los modelos estables [GelLif88]. Muestran además cómo puede representarse conocimiento más facilmente cuando está disponible la negación clásica, y cómo eliminarla con un simple preprocesador antes de correr el programa. Por último establecen que los PLE son un caso especial de las *teorías default* definidas por Reiter [Reiter80]. La sección 2.3 está dedicada a analizar este trabajo.

Los PLE presentan ventajas importantes a la hora de representar conocimiento. Pero lamentablemente cuando dos literales complementarios (por ej. p y $\sim p$) pueden derivarse de un PLE el *conjunto de respuestas* del programa es igual a todo el lenguaje. Como habitualmente el conocimiento maneja información tentativa (defaults), y la representación de información tentativa en la mayoria de los casos lleva a bases de conocimiento incon-

sistentes, en la mayoria de los casos el programa derivará todo el lenguaje. Este es uno de los puntos que trataron de resolver Kowalski y Sadri en [KowSad90], e Inoue en [Inoue91].

El trabajo de Kowalsky y Sadri [KowSad90] utiliza los PLE para poder representar información negativa, y los extiende para la representación de excepciones a reglas generales. Por ejemplo, los pingüinos son una excepción a la regla "los pajaros vuelan". La idea que proponen es representar las reglas generales con cláusulas de la forma "$V \leftarrow P$"y las excepciones con cláusulas "$\sim V \leftarrow Q$", luego a la hora de calcular la respuesta, las excepciones tienen prioridad sobre las reglas. Lamentablemente la forma en que diferencian una regla de una excepción, es que las primeras tienen un consecuente positivo y las últimas un consecuente negado. Esto representa una restricción muy grande, y trae aparejado confusiones, ya que la cláusula "los mamiferos no vuelan" representaría una excepción a la regla que "los murcielagos vuelan" (ver ejemplo 2.4.1). Todo lo referido a este trabajo, esta desarrollado en la sección 2.4.

El trabajo de Katsumi Inoue [Inoue91] es un desarrollo que extiende el trabajo de Gelfond y Litfschitz para poder trabajar con información potencialmente inconsistente. También ataca el problema de los conjuntos de respuestas incosistentes, pero lo hace separando la información absoluta, de la información tentativa o hipotética. De esta forma un "sistema de conocimiento" es un par (T, H) donde T es un conjunto de hechos que son verdaderos en el dominio en cuestión, y H es un conjunto de hipótesis. La tarea del sistema es hallar un subconjunto E de H tal que el conjunto de respuestas de $T \cup E$ sea consistente. En la sección 2.5 se muestra que el enfoque de Inoue es muy bueno y se asemeja en algunas cosas a un sistema de argumentación rebatible (ver capítulo 3). No obstante, presenta dificultades cuando se tiene que elegir entre dos explicaciones contradictorias.

2.3 La programación en lógica extendida con la negación clásica

En esta sección se analizará el trabajo *Logic Programs with Classical Negation* de Gelfond y Lifschitz [GelLif90], indicándose las mejoras y problemas que este presenta.

Gelfond y Lifschitz comienzan su trabajo diciendo :

> "Una importante limitación de la programación en lógica tradicional como herramienta de representación de conocimiento (en comparación con la lógica

> clásica) es que la programación en lógica no nos permite trabajar directamente con información incompleta. Una teoría clásica consistente, particiona el conjunto de sentencias en tres partes: una sentencia es demostrable, refutable, o indecidible. Un programa lógico particiona el conjunto de consultas básicas en solo dos partes: una respuesta para una consulta es "YES" o de lo contrario "NO". Esto ocurre porque la semántica declarativa tradicional de la programción en lógica, automáticamente aplica la hipótesis de mundo cerrado (CWA) a todos los predicados. Lo cuál no da lugar a sentencias indecidibles, que formalmente corresponden a la incompletitud de la información en las teorías axiomáticas clásicas. Para solucionar esta limitación, proponemos considerar programas lógicos extendidos (PLE) que continenen la negación clásica (denotada con '$\sim$'), además de la negación por falla (denotada con '`not`'). "

Contrariamente a un programa normal, un PLE puede incluir explitamente información negativa. Lo cuál permite distinguir entre una consulta que falla porque no tiene éxito, y una consulta que falla porque su negación si tiene éxito.

2.3.1 Sintaxis

Definición 2.3.1 : *Programa lógico extendido* [GelLif90]
Una programa lógico extendido (PLE) es un conjunto finito de cláusulas de la forma $L_0 \leftarrow L_1, \ldots, L_m, not\ \ L_{m+1}, \ldots, not\ \ L_n$, donde $n \geq\ m \geq\ 0$ y cada L_i es un literal, que puede ser un átomo o un átomo negado con "$\sim$". □

Por ejemplo $\Pi_1 = \{\ \sim q \leftarrow not\ \ p\ \}$ es un PLE, cuyo significado sería: la falla de la prueba de p es una prueba para $\sim q$.

2.3.2 Semántica y deducción

Para un PLE, se definirá un *conjunto de respuestas* que será el conjunto de literales básicos que puede derivar dicho PLE. Gelfond y Lifschitz aclaran en su trabajo que la semántica de los PLE trata a una cláusula con variables como una abreviatura del conjunto de sus instancias básicas, y que por ende es suficiente definir el conjunto de respuestas para un PLE sin variables.

Definición 2.3.2 : *Conjunto de respuestas (preliminar)* [GelLif90]
Sea Π un PLE sin variables que no contine el operador `not`. Sea *Lit* el conjunto de literales básicos del lenguaje de Π. El conjunto de respuestas de un programa Π que no contiene negación por falla (denotado $\alpha(\Pi)$), es el menor subconjunto S de *Lit* que verifique:
1. para toda regla $L_0 \leftarrow L_1, \ldots L_n$ de Π, si $L_1, \ldots L_n \in S$, entonces $L_0 \in S$;
2. si S contiene un par de literales complementarios $(A, \sim A)$, entonces $S = Lit$. □

Definición 2.3.3 : *Conjunto de respuestas* [GelLif90]
Sea ahora Π un PLE cualquiera, sin variables. Para TODO conjunto $S \subset Lit$, sea Π^S el PLE obtenido borrando en Π :
1. toda cláusula que tenga una fórmula *not* L en su cuerpo, con $L \in S$ y
2. todas las fórmulas de la forma *not* L en los cuerpos de las reglas restantes.
Claramente Π^S no contiene el operador `not`, y para él si se tiene definido el conjunto de respuestas. Por lo tanto si el conjunto de respuestas de Π^S coincide con S, entonces S es el conjunto de respuestas de Π. En otras palabras (las de Gelfond y Lifschitz) el conjunto de respuestas queda caracterizado por $S = \alpha(\Pi^S)$. □

La definición anterior esta motivada en los siguientes hechos: si S es el conjunto de literales básicos que un agente cree que son verdaderos, entonces una submeta *not* L con $L \in S$ no podrá dar éxito jamás, por lo tanto se puede eliminar la regla que la tenga en su cuerpo. Por otro lado cuando $L \notin S$, *not* L siempre dará éxito, luego es posible eliminar *not* L del cuerpo de la regla.

Observación 1: para calcular el conjunto de respuestas de un PLE que contiene `not`, primero se debe fijar un S, luego calcular Π^S y luego verificar que el conjunto de respuestas de Π^S sea igual al S inicial.

Observación 2: según esta definición, un programa que contiene un `not`, nunca tiene un conjunto de respuestas que es igual a *Lit*.

Observación 3: puede ocurrir, y de hecho pasará en la mayoría de los casos en que un programa tenga un `not`, que no exista conjunto de respuestas (ver ejemplo 2.5.1).

Ejemplo 2.3.1 :

1. Sea $\Pi_2 = \{ \sim P \leftarrow \text{TRUE} \ ; \ P \leftarrow \sim Q \}$ [1] entonces $\alpha(\Pi_2) = \{\sim P\}$.

[1] Gelfons y Lifschitz utilizan una cláusula de la forma "`P` ← " para indicar una regla incondicional (un hecho), a fin de evitar confusiones en la lectura, en este trabajo se utilizará "`P` ← TRUE".

2. Sea $\Pi_3 = \{\ Q \leftarrow \sim P\ ;\ \sim P \leftarrow \text{TRUE}\ \}$ como no contiene el operador `not` entonces $\alpha(\Pi_3) = \{Q, \sim P\}$.

3. Sea $\Pi_1 = \{\ \sim Q \leftarrow not\ \ P\ \}$ para ver que $\{\sim Q\}$ es un conjunto de respuestas de Π_1, se construye $\Pi_1^{\{\sim Q\}}$. Este PLE contiene solo la regla $\sim Q \leftarrow \text{TRUE}$ (el resultado de borrar *not* P). Luego, como el conjunto de respuestas correctas de $\Pi_1^{\{\sim Q\}}$ es $\{\sim Q\}$ entonces $\alpha(\Pi_1) = \{\sim Q\}$.

4. Sea $\{\ \Pi_4 = A \leftarrow not\ \ B;\ B \leftarrow not\ \ R\ \}$ aquí $S = \{\ A, B\}$ no es un conjunto de respuestas, pues el conjunto de respuestas de Π^S es $\{B\}$ y no coincide con S. En cambio $S = \{B\}$ si lo es.

5. Sea $\Pi_5 = \{\ P \leftarrow not\ \ P\}$. Este programa no tiene un conjunto de respuestas.

□

Según Gelfond y Lifschitz, la semántica de "$\leftarrow$" y "$\sim$" no es contrapositiva, *i.e.*, asignan diferente significado a las cláusulas "$\sim Q \leftarrow P$" y "$\sim P \leftarrow Q$". El motivo de esto es que cada cláusula es tomada como una regla de inferencia y no como un condicional.

El problema de la contraposición de la implicación no había aparecido antes porque un programa normal (además de no poseer la negación clásica) no permite cláusulas con negación en el consecuente. Por lo tanto no existe dentro del lenguaje de la programación en lógica tradicional la posibilidad de representar la contrapositiva de "$A \leftarrow B$".

Ejemplo 2.3.2 : Dado $\Pi = \{\ P \leftarrow not\ \ Q\ ;\ \sim P \leftarrow not\ \ R\ ;\ A \leftarrow B, C, D, E\ \}$. De aquí se pueden deducir los literales A, B, C, D y E, lo cuál no tiene sentido. Mientras que tendría más sentido haber elegido uno de los literales "P" o "$\sim P$ " para el conjunto de respuestas, y evitar tener todo el conjunto de literales. Más adelante se verá que Kowalski y Sadri intentaron solucionar esto, aunque de una forma poco exitosa. □

Por último Gelfond y Lifschitz resaltan este hecho más aún en la siguiente proposición:

> "*Todo programa contradictorio (i.e., su conjunto de respuestas tiene un par de literales complementarios) tiene exactamente un conjunto de respuestas : Lit*. Lo cuál muestra que la aproximación a la negación usada es diferente a la aproximación *paraconsistente* de [BlaSub89]".

2.3.3 Representación de conocimiento

Los siguientes son algunos ejemplos dados por Gelfond y Lifschitz para mostrar la utilidad de los PLE. También se muestra otrs ejemplo donde se analizan algunos problemas de la representación.

Ejemplo 2.3.3 : [GelLif90]
Si se quiere indicarle a un conductor de un colectivo de escolares que "cruce la vía cuando no viene el tren", no se debería utilizar la regla "`cruce` $\leftarrow$ `not tren`", ya que esta cláusula significa "cruce la vía cuando no sabe nada acerca del tren". Por la vida de los escolares debería usarse la regla "`cruce` $\leftarrow \sim$ `tren`". Gelfond y Lifschitz señalan que este ejemplo fue inventado por John McCarthy. □

Ejemplo 2.3.4 : [GelLif90]
El conjunto de vértices de un grafo dirigido puede definirse de la siguiente forma:

$$\sim\texttt{terminal(X)} \leftarrow \texttt{arco(X,Y)}$$
$$\texttt{terminal(X)} \leftarrow \texttt{not } \sim\texttt{terminal(X)}$$

□

Ejemplo 2.3.5 : [GelLif90]
La hipótesis de mundo cerrado puede escribirse como: $\sim P(X) \leftarrow$ `not` $P(X)$.
Para algunos predicados también puede ser apropiado tener la cláusula: $P(X) \leftarrow$ `not` $\sim P(X)$. □

Ejemplo 2.3.6 : Sea Π el siguiente PLE:

`vuela` $\leftarrow$ `ave`.
$\sim$ `vuela` $\leftarrow$ `pingüino`.
`ave` $\leftarrow$ `pingüino`.
`pingüino` $\leftarrow$ TRUE.
`corre` $\leftarrow$ `mamifero,hambre,comida,cerca`.

A partir de este programa, es posible deducir los literales *mamifero, comida, cerca, hambre, corre* como efecto colateral de la contradicción que implica creer que un pingüino vuela y no vuela al mismo tiempo. La solución aquí sería poder elegir entre respuestas contradictorias. □

2.3.4 Computación

Al final de su trabajo de Gelfond y Lifscithz agregan:

> "la respuestas a una consulta q pueden ahora ser tres: YES, NO, o UNKNOWN, dependiendo si el conjunto de respuestas contiene q, $\sim q$, o ninguno de los dos".

Pero luego demuestran y establecen un algoritmo por medio del cuál un PLE puede transformarse en un programa normal. Esto es, se reemplaza toda aparición de un predicado $\sim P(\ldots)$ por un predicado $P'(\ldots)$. La transformación (según [GelLif90] proposición 2) garantiza que el significado del programa transformado sigue siendo la misma a la del extendido, *i.e.*, tienen el mismo conjunto de respuestas.

Lo que en realidad están haciendo es tratar en forma sintáctica al $\sim$, de esta forma $\sim P$ es lo mismo que P'. La idea que presentan es transformar con un preprocesador todo PLE en un programa normal y luego utilizar un sistema tradicional de programación en lógica. Pero no está dicho en el trabajo como es que el sistema calcula los conjuntos de respuestas, y responde YES, NO, o UNKNOWN.

2.4 Representación de reglas y excepciones

A continuación se desarrolla la propuesta de Kowalski y Sadri [KowSad90], que según sus propias palabras tiene por objetivo: "extender la programación en lógica para que pueda manejar razonamiento default, permitiendo la representación explícita de excepciones". Para ello utilizan los PLE definidos por Gelfond y Lifschitz, modificando la definición de conjunto de respuestas. Además dicen:

> " La contribución de este trabajo es doble, describimos una semántica para programación en lógica con reglas y excepciones, y también presentamos una transformación que elimina la distinción entre reglas y excepciones usando negación por falla".

La idea que proponen es representar las reglas generales con cláusulas de la forma "$V \leftarrow P$"y las excepciones con cláusulas "$\sim V \leftarrow Q$", luego al momento de derivar una

respuesta, las excepciones tienen prioridad sobre las reglas. Lamentablemente la forma en que diferencian una regla de una excepción, es que las primeras tienen un consecuente positivo y las últimas un consecuente negado. Esto restringe el lenguaje de forma artificial, y trae aparejadas confusiones, ya que en el ejemplo 2.4.1, la cláusula "los mamiferos no vuelan" es una excepción a la regla que "los murcielagos vuelan".

Ejemplo 2.4.1 :

$$\sim\texttt{vuela} \leftarrow \texttt{mamifero}$$
$$\texttt{vuela} \leftarrow \texttt{murcielago}$$

□

Ejemplo 2.4.2 : [KowSad90]

$$\texttt{vuela} \leftarrow \texttt{ave}$$
$$\sim\texttt{vuela} \leftarrow \texttt{pingüino}$$

□

Además existe otro problema, los autores manifiestan que en el programa del ejemplo 2.4.2 la cláusula " $\sim$`vuela` $\leftarrow$ `pingüino`" es una excepción a la regla "`vuela` $\leftarrow$ `ave`" y el vínculo entre ellas es que tienen literales complementarios en la conclusión. Pero el programa no posee la regla "`ave` $\leftarrow$ `pingüino`" por lo tanto el programa no posee ninguna información de que los pingüinos sean aves. Esto no solo restringe el lenguaje de representación de conocimiento sino que establece vinculos poco claros entre las reglas y sus excepciones. Esto puede verse en el ejemplo 2.4.3 donde las vacas son una excepción a de las aves que vuelan.

Ejemplo 2.4.3 :

$$\texttt{vuela} \leftarrow \texttt{ave}$$
$$\sim\texttt{vuela} \leftarrow \texttt{vaca}$$

□

En la sección 2.2 dicen:

> "La intuición detrás del razonamiento con reglas y excepciones, es que las excepciones tienen más alta prioridad que las reglas. Esto es, si aparece una contradicción entre una regla y una excepción, la excepción es la que se toma como válida. Para capturar esto, modificamos la semántica de conjunto de respuestas para que las contradicciones potenciales entre reglas y excepciones sean abolidas prefiriendo los literales negados clásicamente ($\sim A$) a los positivos (A)".

Para ello redefinen el conjunto de respuestas (definición 2.3.3) agregando que hay que borrar toda regla de Π que tenga una conclusión A (positiva), si es el caso que $\sim A \in S$

Definición 2.4.1 : *Conjunto-e de respuestas* [KowSad90]
Sea Π un PLE cualquiera, sin variables. Para TODO conjunto $S \subset Lit$, sea ${}^S\Pi$ el PLE obtenido borrando en Π :

1. toda cláusula que tenga una fórmula *not* L en su cuerpo, con $L \in S$ y
2. todas las fórmulas de la forma *not* L en los cuerpos de las reglas restantes.
3. toda cláusula que tenga una conclusión positiva A con $\sim A \in S$

Claramente ${}^S\Pi$ no contiene el operador `not`, y para él si se tiene definido el conjunto de respuestas. Por lo tanto si el conjunto de respuestas de ${}^S\Pi$ coincide con S, entonces S es el conjunto de respuestas de Π. El conjunto S es un conjunto-e de respuestas sssi $S = \alpha({}^S\Pi)$. □

En el ejemplo 2.4.1, el conjunto-e de respuestas sería $S = \{ \sim\text{vuela} \}$ que no es lo que debería ocurrir.

2.4.1 Transformación de los programas

Una contribución interesante del trabajo de Kowalski y Sadri, es la transformación de los programas lógicos extendidos con excepciones, en programas normales mediante un algoritmo muy sencillo. Esto es:

- aplican la transformación de Gelfond y Lifschitz, y luego,

- a toda cláusula c con conclusión A, si existe una excepción (claúsula con conclusión $\sim A$), agregan `not` $\sim A$ a la lista de condiciones de c

- por último reemplazan $\sim A$ por la lista de condiciones de la excepción.

Ejemplo 2.4.4 : Dado Π:

$$\texttt{vuela} \leftarrow \texttt{ave}$$
$$\sim\texttt{vuela} \leftarrow \texttt{pingüino}$$

luego de la transformación queda:

$$\texttt{vuela} \leftarrow \texttt{ave, not pingüino}$$
$$\sim\texttt{vuela} \leftarrow \texttt{pingüino}$$

□

Obviamente todos los problemas mencionados antes se trasladan a la transformación ya que el ejemplo 2.4.1 queda la regla: " `no vuela` ← `murcielago, not mamifero`", y en el ejemplo 2.4.3 queda la regla: "`vuela` ← `ave, not vaca`".

2.5 Representación de hipótesis default

En esta sección se detalla la propuesta de Katsumi Inoue en *Extended Logic Programming with Default Assumptions*. Inoue, expande la idea de Gelfond y Lifschitz, separando dentro del lenguaje las cláusulas que son siempre verdaderas, de las que respresentan información tentativa (rebatible). Muestra además como este nuevo sistema se comporta bien para el razonamiento por default y la abducción. Luego se explica una transformación que puede aplicarse al sistema de conocimiento definido, con la cuál se obtiene un PLE equivalente. Por último hay un análisis sobre programación lógica abductiva, resolución por incosistencia, y prioridades.

2.5.1 Sistema formal

Definición 2.5.1 : *Sistema de conocimiento* [Inoue91]
Un sistema de conocimiento $\mathcal{K}$ esta representado por un par (T,H), donde:

1. Tanto T como H son PLE,
2. T representa un conjunto de *hechos* que se saben que son ciertos en el dominio, y
3. H representa un conjunto de *hipótesis* que se espera que sean ciertas.

□

La tarea principal de $\mathcal{K}$ es encontrar un subconjunto E de H tal que el conjunto de respuestas de $T \cup E$ sea consistente. Según Inoue, usando este mecanismo, pueden realizarse dos tipos de razonamiento:

1. Razonamiento default: encontrar un subconjunto maximal E de H, tal que $T \cup E$ sea consistente, donde el conjunto de respuestas de $T \cup E$ corresponderá al conjunto de literales que pertenecen a una extensión.
2. Abducción: Encontrar una explicación $E \subseteq H$ de una fórmula O tal que (i) $T \cup E$ sea consistente y (ii) O sea derivable a partir de $T \cup E$.

Inoue agrega que " la diferencia sintáctica que existe entre su sistema y el de Poole [Poole88], es que en este último, se usa el cálculo de predicados de primer orden para representar conocimiento, y el suyo usa los programas lógicos extendidos. Para su sistema, que una fórmula tenga una explicación, no implica que valga en una extensión En este sentido el razonamiento default se distingue claramente de la abducción."

Ejemplo 2.5.1 : [Inoue91]
Dado el PLE

$$Q \leftarrow \sim P(a), \ \sim P(b)$$
$$\sim Q \leftarrow \text{TRUE}$$

si se agrega "$\sim P(X) \leftarrow$ `not` $P(X)$" (CWA), entonces según la definición de PLE, no hay conjunto de respuestas. Mientras que lo que se quiere que ocurra es tener dos conjuntos de respuestas consistentes: $\{\sim Q, \sim P(a)\}$ y $\{\sim Q, \sim P(b)\}$, ignorando las cláusulas que causan inconsistencia. De esta forma el sistema puede verse como un sistema de *resolución por inconsistencia.* □

Sin embargo, el conjunto de respuesta del ejemplo anterior debería haber sido { $\sim Q, \sim P(a), \sim P(b)$ }, resolviendo el conflicto entre Q y $\sim Q$.

Definición 2.5.2 : [Inoue91] Sea Π un PLE, por conveniencia lo clasificaremos de la siguiente manera: Π es *consistente* si tiene un conjunto de respuestas consistente. Π es *contradictorio* si tiene un conjunto de respuesta inconsistente. Π es *incoherente* si no tiene un conjunto de respuestas. □

Proposición 2.5.1 : [Inoue91] Un PLE Π es contradictorio sssi el conjunto de cláusulas de Π que no contienen el operador `not` es contradictorio. □

Por lo tanto una contradición no puede evitarse aunque se agregue cláusulas, o se saque alguna con `not`. Considerar todas las cláusulas como defaults, lleva (igual que en el ejemplo anterior) a programas incoherentes. Aquí Inoue aclara " nuestro objetivo principal es por lo tanto, resolver programas incoherentes y no contradictorios."

Definición 2.5.3 : [Inoue91] Sea $\mathcal{K} = (T,H)$ un sistema de conocimiento. $\mathcal{K}$ es *consistente* si existe un conjunto $E \subseteq H$ tal que $T \cup E$ sea consistente. $\mathcal{K}$ es *contradictorio* si para cualquier conjunto $E \subseteq H$, $T \cup E$ es contradictorio. $\mathcal{K}$ es *incoherente* si no es ninguna de las dos cosas anteriores. □

Proposición 2.5.2 : [Inoue91]
1. $\mathcal{K}$ is contradictorio sssi T es contradictorio.
2. Si T es consistente entonces $\mathcal{K}$ es consistente.
3. Si $\mathcal{K}$ es incoherente entonces T es incoherente. □

Cabe destacar, que agregar hipótesis a un programa incoherente puede permitir que el sistema sea consistente. Por ejemplo si $T = \{ P \leftarrow$ `not` $P \}$ y $H = \{P \leftarrow$ TRUE$\}$. Aquí T es incoherente mientras que $\mathcal{K}$ es consistente, ya que tiene el conjunto de respuestas $\{P\}$.

2.5.2 Razonamiento default

Definición 2.5.4 : [Inoue91] Una *extensión base* de $\mathcal{K}$, es un conjunto de respuestas de $T \cup E$ donde E es un conjunto maximal de H, y $T \cup E$ es consistente. □

Para obtener las extensiones base, algunas hipótesis son ignoradas, pero ninguna puede dejar de tomarse en cuenta, a menos que sea necesario.

Ejemplo 2.5.2 : [Inoue91]
Sea $H = \{$ `vuela(X)` $\leftarrow$ `ave(X)` $\}$, y T el conjunto formado por:

$$\sim\texttt{vuela(X)} \leftarrow \texttt{ping(X)}$$
$$\texttt{ave(X)} \leftarrow \texttt{ping(X)}$$
$$\texttt{ave(lucas)} \leftarrow \text{TRUE}$$
$$\texttt{ping(petete)} \leftarrow \text{TRUE}$$

Aunque $T \cup H$ es contradictorio, existe una extensión base de $\mathcal{K}$: { ave(lucas), vuela(lucas), ping(petete), ave(petete), $\sim$vuela(petete) } Nótese que la hipótesis se uso para X=lucas, pero fue ignorada para X=petete. □

Sin embargo, no queda claro que hubiera pasado, si la cláusula "$\sim$`vuela(X)` $\leftarrow$ `ping(X)`" estubiera en H en lugar de T. En ese caso, habría dos extensiones base, una en la que petete vuela, y otra en la que no vuela. El problema es que en este trabajo no hay definido un criterio de selección sobre las extensiones (ni sobre las hipótesis).

2.6 Conclusiones

La programación en lógica ha sido utilizada ampliamente en sistemas de representación de conocimiento. Esto ha llevado a considerar extensiones que permitan una mayor expresividad en el lenguaje. Cada una de estas extensiones ha incorporado ventajas, pero también han traido aparejado problemas de índole conceptual y computacional. Como fue mostrado en este capítulo, Gelfond y Lifschitz en *Logic Programs with Classical Negation* [GelLif90] definen los programas lógicos extendidos (PLE) para permitir la utilización de la negación clásica. Los PLE presentan ventajas importantes a la hora de representar conocimiento. Pero lamentablemente cuando dos literales complementarios (por ej. p y $\sim p$) pueden derivarse de un PLE el *conjunto de respuestas* del programa es igual a todo el lenguaje. Como habitualmente el conocimiento maneja información tentativa (defaults), y la representación de información tentativa lleva a bases de conocimiento inconsistentes, en la mayoría de los casos el programa derivará todo el lenguaje.

Kowalski y Sadri motivados por este trabajo, y por su investigación en representación de conocimiento con restricciones de integridad, extendieron el trabajo de Gelfond y Lifschitz en *Logic Programs with Exceptions* [KowSad90], introduciendo la posibilidad de

representar excepciones dentro de un programa lógico. La idea que proponen es representar las reglas generales con cláusulas de la forma "$V \leftarrow P$"y las excepciones con cláusulas "$\sim V \leftarrow Q$", luego a la hora de calcular la respuesta, las excepciones tienen prioridad sobre las reglas. Lamentablemente la forma en que diferencian una regla de una excepción, es que las primeras tienen un consecuente positivo y las últimas un consecuente negado. Esto representa una restricción muy grande, y trae aparejado confusiones, ya que la cláusula "los mamiferos no vuelan" representaría una excepción a la regla que "los murcielagos vuelan".

Katsumi Inoue, en *Extended Logic Programs with Default Assumptions* [Inoue91] incorpora hipótesis default a los programas lógicos extendidos de Gelfond y Lifschitz. También ataca el problema de los conjuntos de respuestas inconsistentes, pero lo hace separando la información absoluta, de la información tentativa o hipotética. De esta forma un "sistema de conocimiento" es un par (T, H) donde T es un conjunto de hechos que son verdaderos en el dominio en cuestión, y H es un conjunto de hipótesis. La tarea del sistema es hallar un subconjunto E de H tal que el conjunto de respuestas de $T \cup E$ sea consistente. El enfoque de Inoue es bueno, y como se verá en el capítulo siguiente, al igual que los sistemas de argumentación rebatible, mantiene separada en la base de conocimiento la información tentativa de la estricta. No obstante, el sistema presenta dificultades cuando tiene que elegir entre dos metas contradictorias.

Las tres extensiones estudiadas incorporan características deseables para la programación en lógica, pero sin embargo, presentan problemas al momento de derivar conclusiones contradictorias. En el capítulo 4 se definirá la *programación en lógica rebatible*, como una extensión de la programación en lógica convencional. Este nuevo lenguaje capturará aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Permitirá la representación de información incompleta y potencialmente inconsistente, y utilizará los conceptos de la *argumentación rebatible* (ver capítulo 3) a fin de poder decidir entre metas contradictorias. Este nuevo lenguaje solucionará los problemas mencionados para las extensiones de la programación en lógica analizadas en este capítulo.

Capítulo 3

La argumentación rebatible.

El objetivo de este capítulo es introducir los principales conceptos de la *argumentación rebatible*. Este formalismo será utilizado en el capítulo siguiente para desarrollar el mecanismo de inferencia de la programación en lógica rebatible. Aunque se han desarrollado varios sistemas de argumentación rebatible (ver [Pollock87, SimLou92, Nute92, Prakken, Vrees93]), en esta tésis se utilizará el sistema propuesto por Simari y Loui en [SimLou92] y extendido luego en [GCS93b], [SCG94], y [SCG94].

Los sistemas de argumentación rebatible constituyen una formalización del *razonamiento rebatible*, en el cual las conclusiones obtenidas pueden ser rechazadas ante la aparición de nueva evidencia. En estos sistemas se pone especial énfasis en la noción de *argumento*. Un argumento para una conclusión P constituye una pieza de razonamiento tentativa que un agente inteligente está dispuesto a aceptar para explicar P. Si el agente adquiriese luego nueva información, la conclusión P junto con el razonamiento que la produjo podrían quedar invalidados. Las relaciones existentes entre los argumentos permiten capturar características del razonamiento común que resultarían difíciles de expresar en otros formalismos, tales como lógica default [Reiter80], lógica no-monotónica [McDDoy80], lógica autoepistémica [Moore85], o circunscripción [McCar80].

En el sistema de argumentación rebatible que se introducirá a continuación, un literal m será aceptado como una nueva creencia, cuando exista un argumento que sea una *justificación* de m. El proceso de obtención de una justificación para m, involucra la construcción de un argumento $\mathcal{A}$, para m, que no esté derrotado. Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su

vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, llamado *árbol de dialéctica*, donde $\mathcal{A}$ es la raíz, y todo nodo es derrotador de su padre. En el árbol de dialéctica, argumentos y contraargumentos son comparados utilizando un criterio de preferencia.

La comparación de argumentos es un problema abierto, y existen diferentes propuestas que intentan solucionarlo. Una de ellas es el *criterio de especificidad*, el cuál fue descripto por Poole en [Poole85a], y separadamente por Loui en [Loui87]. Algunos sistemas de razonamiento rebatible han utilizado este criterio para comparar sólo reglas rebatibles [Nute92], y otros para comparar argumentos enteros [SimLou92]. El sistema que se describirá a continuación, utiliza esta última propuesta.

3.1 Sistema de argumentación rebatible

El conocimiento de un agente está representado en este sistema, por un par ($\mathcal{K}$,Δ), llamado *Estructura de Lógica Rebatible*, donde $\mathcal{K}$ representa la parte irrebatible del conocimiento, mientras que Δ representa información tentativa. $\mathcal{K}$ es un conjunto consistente finito de fórmulas bien formadas (fbf) de un lenguaje de primer orden $\mathcal{L}$, en el cual se distinguen dos subconjuntos disjuntos: $\mathcal{K}_P$ (*conocimiento particular*), que contiene fbf totalmente instanciadas (básicas), y $\mathcal{K}_G$ (*conocimiento general*). Δ es un conjunto finito de *reglas rebatibles* denotadas "$\beta \succ \alpha$", las cuales deben interpretarse como: "razones para creer en el antecedente β, son buenas razones para creer en el consecuente α". Se denota con $\Delta^{\downarrow}$ al conjunto de todas las instancias básicas de elementos de Δ. En esta presentación se utilizará la notación habitual de la programación en lógica. Es por esto, que las fórmulas que habitualmente se escriben utilizando los antecedentes a la izquierda, como "$\beta \rightarrow \alpha$", o "$\beta \succ \alpha$", se escribirán aquí con el antecedente en la derecha, como "$\alpha \leftarrow \beta$" o "$\alpha \prec\!\!- \beta$"

Ejemplo 3.1.1 : La figura 3.1 muestra un ejemplo de una estructura lógica rebatible ($\mathcal{K}$,Δ). □

Definición 3.1.1 : *Consecuencia rebatible* [SimLou92]
Sea $\Gamma \subseteq (\mathcal{K} \cup \Delta^{\downarrow})$. Se dirá que A es una *consecuencia rebatible* de Γ, abreviado $\Gamma \mid\!\sim A$, si y sólo si existe una secuencia $B_1, \ldots, B_n$ tal que $A = B_n$ y para todo i, o bien $B_i \in \Gamma$, o B_i es una consecuencia directa de otros elementos que le preceden en la secuencia, usando modus ponens o instanciación de una sentencia cuantificada universalmente. Las instancias de reglas rebatibles se consideran como implicaciones materiales al aplicar modus ponens. □

vuela(X) $-\!\prec$ ave(X)
ave(X) $\leftarrow$ gallina(X)
ave(X) $\leftarrow$ paloma(X)
ave(X) $\leftarrow$ pingüino(X)
$\sim$vuela(X) $\leftarrow$ pingüino(X)
pingüino(petete)
gallina(coco)
asustado(coco)
paloma(pepa)
$\sim$vuela(X) $-\!\prec$ gallina(X)
vuela(X) $-\!\prec$ gallina(X) $\wedge$ asustado(X)
anida-arbol(X) $-\!\prec$ vuela(X)
$\sim$anida-arbol(X) $-\!\prec$ gallina(X)

Figura 3.1: Reglas rebatibles y no rebatibles de ($\mathcal{K}$,Δ)

3.1.1 Argumentos

Una vez fijado el lenguaje de representación y la noción de inferencia, es posible definir el concepto de argumento.

Definición 3.1.2 : *Argumento* [SimLou92]
Se dice que $\langle \mathcal{A}, h \rangle_{\mathcal{K}}$, o simplemente $\langle \mathcal{A}, h \rangle$, es una *estructura de argumento* para un literal h en el contexto $\mathcal{K}$, si $\mathcal{A}$ es un subconjunto de $\Delta^{\downarrow}$ que cumple las siguientes condiciones:

1. $\mathcal{K} \cup \mathcal{A} \mid\sim h$ (h es una consecuencia rebatible de $\mathcal{K} \cup \mathcal{A}$),
2. $\mathcal{K} \cup \mathcal{A} \not\mid\sim \bot$ ($\mathcal{A}$ es consistente con $\mathcal{K}$) y
3. No existe un subconjunto $\mathcal{A}'$ de $\mathcal{A}$, tal que, $\mathcal{K} \cup \mathcal{A}' \mid\sim h$ ($\mathcal{A}$ es minimal).

Si $\langle \mathcal{A}, h \rangle$ es un estructura de argumento, entonces $\mathcal{A}$ es un *argumento* para h. □

Ejemplo 3.1.2 : Las siguientes estructuras de argumento pueden construirse utilizando el ($\mathcal{K}$,Δ) del ejemplo 3.1.1: $\langle$ $\mathcal{A}_1$,vuela(coco) $\rangle$, $\langle$ $\mathcal{A}_2$,vuela(coco) $\rangle$, $\langle$ $\mathcal{A}_3$,$\sim$vuela(coco) $\rangle$, $\langle$ $\mathcal{A}_4$,$\sim$vuela(petete) $\rangle$, y $\langle$ $\mathcal{A}_5$,anida-arbol(pepa) $\rangle$, donde:

$\mathcal{A}_1$= { vuela(coco) $-\!\prec$ ave(coco) }
$\mathcal{A}_2$= { vuela(coco) $-\!\prec$ gallina(coco) $\wedge$ asustado(coco) }
$\mathcal{A}_3$= { $\sim$vuela(coco) $-\!\prec$ gallina(coco) }
$\mathcal{A}_4$= { }
$\mathcal{A}_5$= { anida-arbol(pepa) $-\!\prec$ vuela(pepa) , vuela(pepa) $-\!\prec$ ave(pepa) }

Obsérvese además que no existe argumento para "vuela(petete)" porque no se verifica la condición de consistencia de la definición. □

3.1.2 Relaciones entre argumentos

Las estructuras de argumento pueden interactuar unas con otras de diferentes maneras a través de las relaciones de *especificidad, desacuerdo, contra-argumento* y *derrota*, definidas a continuación.

Definición 3.1.3 : *Especificidad* [SimLou92]
Sea $\mathbf{L}$ = $\{l : l$ es un literal instanciado y $\mathcal{K} \cup \Delta^{\downarrow} \mid\!\sim l\ \}$. El argumento $\mathcal{A}$ para h_1 es *estrictamente más específico* que el argumento $\mathcal{B}$ para h_2, si y sólo si

1. para todo conjunto $C \subseteq \mathbf{L}$
   si $\mathcal{K}_G \cup C \cup \mathcal{A} \mid\!\sim h_1$ (C activa $\mathcal{A}$) y $\mathcal{K}_G \cup C \not\mid\!\sim h_1$, (activación no trivial)
   entonces $\mathcal{K}_G \cup C \cup \mathcal{B} \mid\!\sim h_2$. ($C$ activa $\mathcal{B}$)
2. existe un conjunto $C' \subseteq \mathbf{L}$ tal que:
   $\mathcal{K}_G \cup C' \cup \mathcal{B} \mid\!\sim h_2$, ($C'$ activa $\mathcal{B}$), $\mathcal{K}_G \cup C' \not\mid\!\sim h_2$ (activación no trivial)
   y $\mathcal{K}_G \cup C' \cup \mathcal{A} \not\mid\!\sim h_1$. ($C'$ no activa $\mathcal{A}$)

Si un argumento $\mathcal{A}$ es estrictamente más específico que $\mathcal{B}$, se denotará $\mathcal{A} \succ \mathcal{B}$. En el caso que $\mathcal{A} \not\succ \mathcal{B}$ y que $\mathcal{B} \not\succ \mathcal{A}$, se dirá que $\mathcal{A}$ y $\mathcal{B}$ son incomparables, y se notará $\mathcal{A} \not\approx \mathcal{B}$. □

Ejemplo 3.1.3 : Considérense los argumentos del ejemplo 3.1.2. Utilizando el criterio de especificidad, resulta que $\mathcal{A}_3 \succ \mathcal{A}_1$, y $\mathcal{A}_2 \succ \mathcal{A}_3$. □

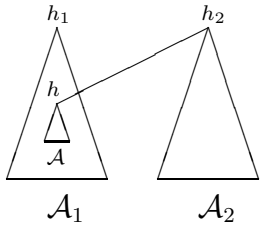
Definición 3.1.4 : *Subargumento* [SimLou92]
Un argumento $\langle \mathcal{B}, q \rangle$ es un *subargumento* de $\langle \mathcal{A}, h \rangle$ si $\mathcal{B} \subseteq \mathcal{A}$. □

Definición 3.1.5 : *Desacuerdo* [SimLou92]
Se dirá que $\langle \mathcal{A}_2, h_2 \rangle$ y $\langle \mathcal{A}, h \rangle$ están en desacuerdo cuando se cumpla que: $\mathcal{K} \cup \{h, h_2\} \vdash \bot$
□

Definición 3.1.6 : *Contraargumento* [SimLou92]
Se dirá que $\langle \mathcal{A}_2, h_2 \rangle$ *contraargumenta a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que $\langle \mathcal{A}, h \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ están en desacuerdo. El argumento $\langle \mathcal{A}, h \rangle$ se llamará subargumento de desacuerdo, y al literal h se lo llamará punto de contraargumentación. □

Definición 3.1.7 : *Derrotador* [SCG94]
Un argumento $\langle \mathcal{A}_2, h_2 \rangle$ *derrota a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que: $\langle \mathcal{A}_2, h_2 \rangle$ contraargumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el literal h y se cumple una de estas dos opciones:
(1) $\langle \mathcal{A}_2, h_2 \rangle$ es estrictamente más específico que $\langle \mathcal{A}, h \rangle$ (derrotador propio), o
(2) $\langle \mathcal{A}_2, h_2 \rangle$ es incomparable con $\langle \mathcal{A}, h \rangle$ (derrotador de bloqueo). □

Con el fin de graficar esquemáticamente algunas situaciones donde dos o más argumentos se relacionan, los argumentos serán representadas con triángulos (en relación con su forma de árbol de derivación). El vértice superior del triángulo será etiquetado con el literal que corresponde a la conclusión del argumento, y debajo del triángulo será indicado el nombre asociado al argumento. Los subargumentos serán representados por triángulos más pequeños dentro de los argumentos que correspondan, y la relación de derrota será indicada por una línea que une al derrotador con el subargumento de desacuerdo del argumento derrotado. A continuación se muestra un esquema gráfico donde $\mathcal{A}_2$ derrota a $\mathcal{A}_1$ en el punto h.



3.1.3 Justificaciones

En este sistema, un literal m será aceptado como una nueva creencia, cuando exista un argumento que sea una *justificación* de m. El proceso de obtención de una justificación para m, involucra la construcción de un argumento $\mathcal{A}$, para m, que no esté derrotado.

Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen contraargumentos que son posibles derrotadores de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, donde $\mathcal{A}$ es la raíz, y cada nodo tiene por hijos a sus derrotadores. La definición de árbol de dialéctica es la que caracteriza este proceso.

Definición 3.1.8 : *Árbol de dialéctica* [SCG94]
Un *árbol de dialéctica* para $\langle \mathcal{A}, h \rangle$, denotado $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, se define recursivamente como sigue:

1. Un nodo que contiene una estructura de argumento $\langle \mathcal{A}, h \rangle$ sin derrotadores (propios o de bloqueo), es un árbol de dialéctica para $\langle \mathcal{A}, h \rangle$, y es también la raíz del árbol.
2. Supóngase que $\langle \mathcal{A}, h \rangle$ es una estructura de argumento con derrotadores (propios o de bloqueo) $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$. El árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, para $\langle \mathcal{A}, h \rangle$ se construye poniendo a $\langle \mathcal{A}, h \rangle$ como nodo raíz, y haciendo que este nodo sea el padre de las raíces de los árboles de dialéctica de $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$, *i.e.*, $\mathcal{T}_{\langle \mathcal{A}_1, h_1 \rangle}$, $\mathcal{T}_{\langle \mathcal{A}_2, h_2 \rangle}$, ..., $\mathcal{T}_{\langle \mathcal{A}_n, h_n \rangle}$.

□

Para toda estructura de argumento existe un árbol de dialéctica, el cuál todo nodo puede rotularse siguiendo la siguiente definición:

Definición 3.1.9 : *Rótulos de un árbol de dialéctica* [SCG94]
Los nodos de un árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ se etiquetan recursivamente como *nodo no-derrotado* (nodo-U) o *nodo derrotado* (nodo-D) de la siguiente forma:

1. Una hoja de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un *nodo-U*.
2. Sea $\langle \mathcal{B}, q \rangle$ un nodo interno de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$.
   a) $\langle \mathcal{B}, q \rangle$ será un *nodo-U* sssi todo hijo de $\langle \mathcal{B}, q \rangle$ es un *nodo-D*.
   b) $\langle \mathcal{B}, q \rangle$ será un *nodo-D* sssi tiene al menos un hijo que es *nodo-U*.

□

Por lo tanto, se definirá el concepto de justificación, utilizando un árbol de dialéctica rotulado.

Definición 3.1.10 : *Justificación* (provisoria)
$\langle \mathcal{A}, h \rangle$ es una *justifición* para h sssi la raíz de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un nodo-U. □

3.2 Control de argumentaciones falaces

El proceso de obtención de justificaciones genera un árbol de argumentos, (árbol de dialéctica), donde cada nodo tiene por hijos a sus derrotadores (propios o de bloqueo). La raíz del árbol de dialéctica es el argumento que sustenta a la conclusión m, y las hojas son argumentos que no tienen derrotadores.

Como se verá a continuación hay situaciones en las cuales pueden formarse ciclos, impidiendo que el proceso de justificación termine. Estos ciclos representan argumentaciones recurrentes donde no puede llegarse a ninguna conclusión. La razón por la cual pueden aparecer este tipo de situaciones, radica en la posibilidad de incurrir en falacias dentro de la argumentación. Se mostrará a continuación como refinar el sistema de argumentación rebatible para evitar este tipo de situación falaces, el desarrollo completo de este tema puede encontrarse en [SCG94].

Una vez construído el árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ para un argumento $\mathcal{A}$, un camino en $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, desde $\mathcal{A}$, hasta una hoja, corresponderá a una diferente línea en la argumentación que se desarrolló. Cada línea estará formada por el argumento $\mathcal{A}$ construído para la consulta efectuada, y alternativamente se intercalarán argumentos en contra de $\mathcal{A}$ (interferencia) y a favor de $\mathcal{A}$ (soporte). A continuación se formalizan estos conceptos.

Definición 3.2.1 : *Linea de argumentación* [SCG94]
Sea $\langle \mathcal{A}_0, h_0 \rangle$ una estructura de argumento, y $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$ su árbol de dialéctica asociado. Todo camino λ en $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$ desde la raíz $\langle \mathcal{A}_0, h_0 \rangle$ hasta una hoja $\langle \mathcal{A}_n, h_n \rangle$, denotado $\lambda = [\ \langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$, constituye una *línea de argumentación* para $\langle \mathcal{A}_0, h_0 \rangle$. □

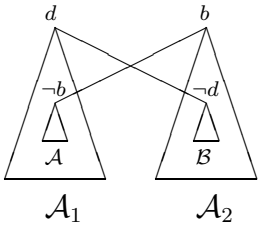
Definición 3.2.2 : *Argumentos de soporte y de interferencia* [SCG94]
Sea $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$ un árbol de dialéctica, y $\lambda = [\ \langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$, una línea de argumentación para $\langle \mathcal{A}_0, h_0 \rangle$. Toda estructura de argumento $\langle \mathcal{A}_i, h_i \rangle$ perteneciente a λ puede rotularse como un *argumento de soporte* o un *argumento de interferencia*:

1. $\langle \mathcal{A}_0, h_0 \rangle$ es un argumento de soporte en λ, y
2. Si $\langle \mathcal{A}_i, h_i \rangle$ es un argumento de soporte (resp. interferencia) en λ, entonces $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ es un argumento de interferencia (resp. soporte) en λ.

Se denota con λ_S al conjunto de argumentos de soporte pertenecientes a λ, y con λ_I al conjunto de argumentos de interferencia de λ. □

3.2.1 Derrotadores recíprocos

El siguiente ejemplo mencionado por Prakken en [Prakken], presenta una situación en la cual se incurre en una falacia al utilizar el mismo argumento derrotado para derrotar a su derrotador.



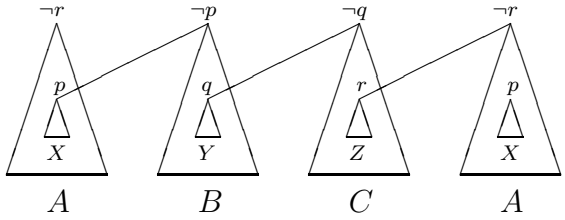
En este ejemplo se tienen dos derrotadores recíprocos, lo cual establece una situación sin sentido. $\langle\ \mathcal{A}_1,d\ \rangle$ derrota a $\langle\ \mathcal{A}_2,b\ \rangle$, porque este último contiene el subargumento $\langle\ \mathcal{B},\neg d\ \rangle$; pero también es el caso que $\langle\ \mathcal{A}_2,b\ \rangle$ es un derrotador para $\langle\ \mathcal{A}_1,d\ \rangle$ atacando al subargumento $\langle\ \mathcal{A},\neg b\ \rangle$. Como consecuencia, aparece un ciclo en el árbol de dialéctica para $\langle\ \mathcal{A}_1,d\ \rangle$, incurriendose en la siguiente línea de argumentación infinita $[\langle\ \mathcal{A}_1,d\ \rangle, \langle\ \mathcal{A}_2,b\ \rangle, \langle\ \mathcal{A}_1,d\ \rangle, \langle\ \mathcal{A}_2,b\ \rangle, \ldots]$ (*i.e.*, $\langle\ \mathcal{A}_1,d\ \rangle$ es derrotado por $\langle\ \mathcal{A}_2,b\ \rangle$, $\langle\ \mathcal{A}_2,b\ \rangle$ es derrotado por $\langle\ \mathcal{A}_1,d\ \rangle$, y esto se repite indefinidamente). En un caso como este, el mecanismo de inferencia entrará en un ciclo infinito. Por lo tanto, hay razones intuitivas y técnicas para eliminar la presencia de derrotadores recíprocos en una línea de argumentación.

La forma de solucionar esto es incluyendo la siguiente condición en la línea de argumentación. Cuando un argumento $\langle \mathcal{A}_2, h_2\rangle$ es construído para derrotar a $\langle \mathcal{A}_1, h_1\rangle$, entonces se debe cumplir que para todo subargumento propio $\langle \mathcal{B}, q\rangle$ de $\langle \mathcal{A}_2, h_2\rangle$, no sea el caso que $\langle \mathcal{B}, q\rangle$ y $\langle \mathcal{A}_1, h_1\rangle$ estén en desacuerdo.

De esta forma, se eliminan los ciclos de longitud unitaria. No obstante, esto no cubre todas las situaciones indeseables, porque existe la posibilidad de que se generen ciclos de longitud arbitraria en una línea de argumentación.

3.2.2 Argumentación contradictoria

El siguiente ejemplo muestra una situación donde un argumento de soporte es introducido nuevamente en la línea de argumentación como un argumento de interferencia.

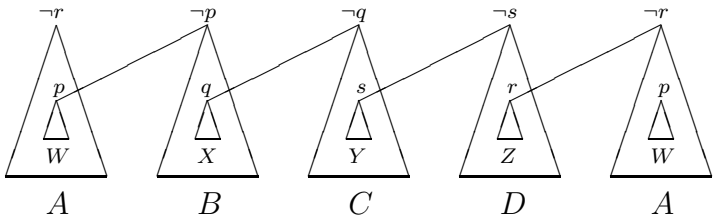


En este ejemplo, $\langle$ B,$\neg p$ $\rangle$ derrota a $\langle$ A,$\neg r$ $\rangle$, y $\langle$ C,$\neg q$ $\rangle$ derrota a $\langle$ B,$\neg p$ $\rangle$ reinstanciado a $\langle$ A,$\neg r$ $\rangle$. Pero $\langle$ C,$\neg q$ $\rangle$ es también derrotado por $\langle$ A,$\neg r$ $\rangle$. De esta forma $\langle$ A,$\neg r$ $\rangle$ se convierte en un argumento de interferencia para él mismo, lo cual no tiene sentido. Es claro que debería existir algún criterio de *concordancia* entre los argumentos de interferenca (resp. soporte) en una línea de argumentación. La coherencia interna es esencial en el proceso dialéctico. Para solucionar este problema, tanto λ_S, como λ_I, deben contener argumentos concordantes. A continuación figura la definición.

Definición 3.2.3 : *Argumentos concordantes* [SimLou92]
Dos argumentos $\mathcal{A}_1$ y $\mathcal{A}_2$ son concordantes si: $\mathcal{K} \cup \mathcal{A}_1 \cup \mathcal{A}_2 \not\sim \bot$.
En general, un conjunto de argumentos $\{\langle \mathcal{A}_i, h_i \rangle\}_{i=1}^{n}$ es concordante si: $\mathcal{K} \cup (\bigcup_{i=1}^{n} \langle \mathcal{A}_i, h_i \rangle) \not\sim \bot$.
□

3.2.3 Circularidad

Como generalización de los derrotadores recíprocos vistos anteriormente puede considerarse, la generación de ciclos de un número par de argumentos.



En este caso hay, aunque hay concordancia, el mecanismo de inferencia genera una cadena de argumentación infinita. Aquí la falacia subyace en la manera en que se construye la línea de la argumentación. Debe asegurarse que cada paso en la línea de argumentación utilice nuevos argumentos, *i.e.*, que la argumentación sea progresiva.

3.2.4 Árbol de dialéctica aceptable

A continuación se mostrará como solucionar los problemas de circularidad presentados, y se definirá definitivamente el concepto de justificación.

Definición 3.2.4 : *Línea de argumentación aceptable* [SCG94]
Sea $\lambda = [\ \langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ \ldots, \ \langle \mathcal{A}_n, h_n \rangle]$, una línea de argumentación para $\langle \mathcal{A}_0, h_0 \rangle$. Se dirá que λ es un línea de argumentación aceptable si se cumplen las siguientes condiciones:

1. Todo derrotador $\langle \mathcal{A}_i, h_i \rangle$ de la línea cumple que para todo subargumento propio $\langle \mathcal{B}, q \rangle$ de $\langle \mathcal{A}_i, h_i \rangle$, no sea el caso que $\langle \mathcal{B}, q \rangle$ y $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$ esten en desacuerdo.

2. Tanto λ_S, como λ_I, son conjuntos de argumentos concordantes.

3. Un argumento $\langle \mathcal{A}_n, h_n \rangle$ no es subargumento de algún otro argumento $\langle \mathcal{A}_i, h_i \rangle$ de la línea $(i < n)$.

□

Definición 3.2.5 : *Árbol de dialéctica aceptable* [SCG94]
Un árbol de dialéctica se dirá aceptable, si todas sus líneas de argumentación son aceptables. □

Definición 3.2.6 : *Justificación* [SCG94]
Un argumento $\langle \mathcal{A}, h \rangle$ es una *justifición* para h sssi la raíz del árbol de dialéctica aceptable $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un nodo-U. □

3.3 Conclusiones

El sistema de argumentación rebatible descripto, permite la representación de información tentativa utilizando reglas rebatibles "$\alpha \prec \beta$", donde creer en el antecedente β es una buena razón (rebatible) para creer en el consecuente α. La base de conocimiento del sistema está formada por el conjunto $\mathcal{K}$ de reglas estrictas, y el conjunto Δ de reglas rebatibles, separando de esta forma la información estricta de la tentativa. Tener dos tipos de cláusulas será una de las características que tendrá el lenguaje que se definirá en el capítulo siguiente.

En este formalismo, un argumento para una conclusión C es un conjunto de reglas rebatibles que junto con $\mathcal{K}$ permiten derivar C. Este conjunto de reglas rebatibles debe ser consistente con $\mathcal{K}$, y mínimo con respecto a la inclusión de conjuntos. De esta forma, un argumento está formado sólo por información tentativa, la cual puede ser refutada con otros argumentos. Por lo tanto, la noción de inferencia dentro de este sistema no significa simplemente la obtención de un argumento, sino que el argumento obtenido no debe estar derrotado, convirtiéndose en una justificación para la conclusión que sustenta. Esto permitirá definir la noción de inferencia dentro de la programación en lógica rebatible, ya que en la mayoría de los casos podrán derivarse (rebatiblemente) metas que serán contradictorias.

La obtención de una justificación involucra la construcción de un *árbol de dialéctica*, un árbol de argumentos donde todo nodo es derrotador de su padre. Las hojas de este árbol son argumentos sin derrotadores, y por lo tanto son nodos no derrotados. Un nodo interno que tenga al menos un hijo no derrotado se convertirá en un nodo derrotado, pero si todos sus hijos están derrotados entonces será un nodo no derrotado. Esto define un

proceso que partiendo de las hojas, rotula a cada nodo como derrotado o no derrotado, hasta llegar a la raíz. De esta forma, el argumento raíz del árbol de dialéctica será una justificación si es un nodo no derrotado.

Cada camino en el árbol de dialéctica, desde la raíz hasta una hoja, representa una linea de argumentación que surge a partir del argumento raíz. El sistema impone ciertas restricciones sobre estas lineas para que la argumentación no sea circular, y evitar también caer en otro tipo de falacias dentro de la argumentación.

Capítulo 4

Un Lenguaje de Programación en Lógica Rebatible.

El objetivo de este capítulo es definir un lenguaje de programación en lógica rebatible, como una extensión de la programación en lógica convencional. Este nuevo lenguaje capturará aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Permitirá además la representación de información incompleta y potencialmente inconsistente, y utilizará los conceptos de la *argumentación rebatible* a fin de poder decidir entre metas contradictorias. Como se verá a lo largo del capítulo, este nuevo lenguaje soluciona los problemas mencionados para las extensiones de la programación en lógica presentadas en el capítulo 2.

Los *programas lógicos rebatibles* (PLR), estarán formados por dos tipos de cláusulas: las *cláusulas de programa extendido*, y las *cláusulas de programa rebatible*. De esta forma, un programa estará formado por dos conjuntos disjuntos de cláusulas: uno que representa conocimiento estricto (seguro), y otro que contiene información rebatible (tentativa). En la sección 4.1, se define la sintaxis de este lenguaje.

En la sección 4.2 se definirá la noción de *inferencia rebatible*, con la cuál se podrá construir un *argumento* para las consultas del programa. Un argumento será definido como un subconjunto particular de un PLR, y una vez construído, se podrán utilizar los conceptos de *justificación* y *derrota* de la argumentación rebatible para obtener la respuesta de la consulta efectuada.

La semántica de un PLR (ver sección 4.5) quedará caracterizada por cuatro conjuntos de repuestas: el conjunto de respuestas *positivas*, formado por aquellos literales para los

cuales existe una justificación; el conjunto de respuestas *negativas*, compuesto por aquellos literales cuyos argumentos han sido derrotados; el conjunto de respuestas *indecisas*, formado por aquellos literales para los cuales no es posible decidir entre sus argumentos a favor y en contra; y por último el conjunto de respuestas *desconocidas* por el programa, que corresponde a aquellos literales que no poseen argumento que los sustente.

Los programas lógicos rebatibles permiten utilizar dos tipos de negación: la *negación por falla finita* (`not`), y la *negación clásica* ($\sim$). Esto permite representar información incompleta, y potencialmente inconsistente. Además, la *hipótesis de mundo cerrado* (closed world assumption o CWA) se podrá representar directamente como cláusulas de programa. Como la cláusula "$\sim$`p(X)` $\leftarrow$ `not p(X)`" establece que, "de no poder probar `p(X)`, entonces es posible derivar $\sim$`p(X)`", entonces ante la falta de información sobre `p(X)` se asumirá $\sim$`p(X)`, con lo cuál se estará usando la CWA para `p(X)`. La posibilidad de incluir la hipótesis de mundo cerrado dentro del mismo lenguaje, permite por un lado aplicar la CWA sólo a los predicados deseados, y además, representar otras formas de CWA como "`p(X)` $\leftarrow$ `not` $\sim$`p(X)`", "`p(X)` $\leftarrow$ `not p(X)`" y "$\sim$`p(X)` $\leftarrow$ `not` $\sim$`p(X)`".

No obstante, como los PLR poseen además cláusulas rebatibles, entonces es posible representar nuevas formas de CWA como por ejemplo "$\sim$`p(X)` $-\!\prec$ `not p(X)`", que debe leerse: *no poder probar "p(X)" es una buena razón (rebatible) para asumir "$\sim$p(X)"*. Este nuevo tipo de cláusulas de CWA rebatibles soluciona problemas que pueden plantearse al utilizar sólo cláusulas estrictas (no rebatibles). Todo lo referente a las clásulas de CWA se encuentra desarrollado en la sección 4.7.

Los PLR incorporan algunos conceptos que no estaban presentes en los sistemas de argumentación rebatible, como la negación por falla, y las presuposiciones. Por lo tanto, se extendió la definición del criterio de especificidad, de tal manera que pueda comparar correctamente argumentos que contengan presuposiciones y el operador de negación por falla. Este desarrollo se encuentra en la sección 4.8.

4.1 El lenguaje de los programas lógicos rebatibles

En el lenguaje de la programación en lógica rebatible, un *literal* "l" es un átomo "a" o un átomo negado "$\sim a$", siguiendo la definición de Lloyd [Lloyd87] (ver capítulo 2). El símbolo "$\sim$" representará la negación clásica, y el símbolo "`not`" la negación por

falla (el significado y uso de estos dos tipos de negación será explicado en detalle en la sección 4.6). El símbolo " — " en cambio, se utilizará para indicar el complemento de un literal con respecto a la negación clásica, *i.e.*, $\overline{a} = \sim a$, y $\overline{\sim a} = a$. Dos literales se dicen complementarios, si uno es el complemento del otro, y la complementariedad está definida siempre sobre la negación clásica "$\sim$".

Definición 4.1.1 : *Cláusula de programa extendido* (CPE) [GelLif90]: Es una cláusula de programa de la forma "$l \leftarrow p_1, \ldots, p_n$", $(n \geq 0)$ donde l es un literal, y cada p_i es un literal o un literal precedido por el símbolo "`not`" de la negación por falla. Si n=0, entonces se denotará "$l \leftarrow$ `true`", y se dirá que l es un *hecho* (donde la constante `true` tiene la interpretación usual). □

Como convención tipográfica, se utilizarán letras minúsculas para denotar literales (l), mayúsculas para conjuntos de literales (L), mayúsculas caligráficas para conjuntos de cláusulas ($\mathcal{L}$), y la convención habitual de Prolog para cláusulas de programa. En algunas ocasiones una cláusula se notará "$c \leftarrow L$" donde L representará la conjunción de los elementos del antecedente.

Ejemplo 4.1.1 : Las siguientes son ejemplos de CPE:

```
ñandú(charo) ← true.
ave(X) ← ñandú(X).
∼vuela(X) ← ñandú(X).
∼vuela(X) ← not vuela(X).
peligroso(X) ← not ∼peligroso(X), desconocido(X).
```

□

Obsérvese que de acuerdo a la sintaxis definida "`∼p ← not ∼q`" es una CPE válida mientras que "`not p ← q`" , "`p ← ∼∼p`", "`p ← not not p`", y "`p ← ∼not q`" no lo son.

La sintaxis de las CPE es la misma que la introducida por Gelfond y Lifschitz en [GelLif90], pero su forma de uso será diferente. El objetivo de las CPE será representar información irrefutable, mientras que las cláusulas que se definirán a continuación representarán información tentativa.

Definición 4.1.2 : *Cláusula de programa rebatible* (CPR) [SimGar95]: Es una cláusula de la forma "$l \prec p_1, \ldots, p_n$" , $(n \geq 0)$ donde l es un literal, y cada p_i es un literal o

un literal precedido por el simbolo "`not`" de la negación por falla. Si n=0 se denotará "l $\prec$ `true`", y se dira que l es una *presuposición*. El símbolo "$\prec$ " se utiliza para distinguir una CPE de una CPR, porque una CPR se utilizará para representar conocimiento rebatible, *i.e.*, información tentativa que puede ser usada en la medida que no sea contradecida. Una cláusula "l $\prec$ A" debe leerse como: "razones para creer en el antecedente A son buenas razones para creer en el consecuente l". □

Ejemplo 4.1.2 : Las siguientes son ejemplos de CPR:

`vuela(X)` $\prec$ `ave(X)`
`~anida_árbol(X)` $\prec$ `~vuela(X)`
`anida_suelo(X)` $\prec$ `not anida_árbol(X)`
`pingüino(petete)` $\prec$ `true`
`vuela(X)` $\prec$ `gallina(X),asustado(X)`
`~ataca-personas(X)` $\prec$ `perro(X)`
`ataca-personas(X)` $\prec$ `perro(X),enojado(X)`

□

Las CPEs se utilizarán para representar conocimiento seguro (o no rebatible) como "`ave(X)` ← `pigüino(X)`", mientras que las CPRs se utilizarán para representar información rebatible como "`vuela(X)` $\prec$ `ave(X)`" (compárese la información representada en los ejemplos 4.1.2 y 4.1.1).

Las cláusulas de programa no poseen un valor de verdad. Deben interpretarse como reglas de inferencia, y no como condicionales. Su semántica informal es " si el antecedente es verdadero, entonces el consecuente es verdadero". Además, todas las variables en las cláusulas se asumen universalmente cuantificadas.

Las cláusulas de programa pueden contener dos tipos de negación. Por un lado, puede utilizarse la negación clásica (denotada con el símbolo "~"), que permite expresar cláusulas como "`~carnivoro(X)` ← `caballo(X)`" (los caballos no son carnívoros), o "`trabajo(X)` $\prec$ `~feriado(X)`" (habitualmente los días que no son feriados, se trabaja). Pero, por otro lado, se puede utilizar la negación por falla "`not`", la cual es útil para casos como "`buscar(X)` $\prec$ `perdido(X),not muerto(X)` (si alguien está perdido, y no se puede probar que esté muerto, entonces hay que buscarlo), o "`~cruzar_la_via` $\prec$ `not ~viene_el_tren`" (no cruzar la vía si no estoy seguro que no viene el tren).

A continuación se definen los conceptos de programa y meta en este lenguaje, y luego se presenta un ejemplo de programa.

Definición 4.1.3 : *Programa lógico rebatible* (PLR)
Un programa lógico rebatible, es un conjunto finito de CPE y CPR. Si $\mathcal{P}$ es un PLR, por conveniencia se distinguirá con $\mathcal{S}$ al conjunto de CPE de $\mathcal{P}$ y con $\mathcal{D}$ al conjunto de CPR de $\mathcal{P}$. □

Definición 4.1.4 : *Meta o consulta*
Una meta o consulta para un programa $\mathcal{P}$, es una cláusula de la forma " $\prec$ $m_1, \ldots, m_k$". Se utiliza el símbolo " $\prec$ " en lugar de " $\leftarrow$ ", porque la derivación que se realizará es rebatible. Los elementos m_i de la consulta se llamarán submetas. □

Ejemplo 4.1.3 :

```
perro(chicho) ← true.
toro(pepe) ← true.
∼carnivoro(X) ← toro(X).
tiene-cuernos(X) ≺ toro(X).
∼peligroso(X) ≺ ∼carnivoro(X).
peligroso(X) ≺ ∼carnivoro(X), tiene-cuernos(X).
peligroso(X) ← not ∼peligroso(X).
```

□

4.2 Derivaciones rebatibles

Una *derivación rebatible* para una meta m será el conjunto instancias de cláusulas de programa (CPE y CPR) que permiten concluir m. La siguiente definición formaliza este concepto:

Definición 4.2.1 : Dado un PLR $\mathcal{P}$, una *derivación rebatible* para una meta m a partir de $\mathcal{P}$, es un conjunto finito de CPE y CPR definido recursivamente de la siguiente forma:

1. Si existe un hecho "$c \leftarrow$ `true`" (o una presuposición "$c \prec$ `true`") en $\mathcal{P}$, tal que m unifica con c, con unificador mas general σ, entonces el conjunto $\{c\sigma \leftarrow$ `true`$\}$ (resp. $\{c\sigma \prec$ `true` $\}$) es una derivación rebatible para m.

2. Si existe una CPE "$c \leftarrow L$" (o una CPR "$c \prec L$") tal que c unifica con m, (con unificador más general σ, y existe un derivación rebatible $\mathcal{F}_i$ para cada uno de los elementos de $L\sigma$, entonces, $\{c\sigma \leftarrow L\sigma\} \cup (\bigcup_i \mathcal{F}_i)$ (resp. $\{c\sigma \prec L\sigma\} \cup (\bigcup_i \mathcal{F}_i)$) es una derivación rebatible para m. A fin de que no se produzcan ciclos, la cláusula "$c\sigma \leftarrow L\sigma$" (resp. "$c\sigma \prec L\sigma$") no debe aparecer en ninguno de los conjuntos $\mathcal{F}_i$.

3. Si m es una submeta que tiene el operador de negación por falla `not`, *i.e.*, m es `not` l, y no existe una *justificación* para el literal l, entonces el conjunto vacío es una derivación rebatible para m.

4. Si no se da ninguno de los casos anteriores, entonces no existe una derivación rebatible para m.

Si existe una derivación rebatible para m a partir de $\mathcal{P}$, se dirá además que $\mathcal{P}$ deriva rebatiblemente a m, o que m se deriva rebatiblamente a partir de $\mathcal{P}$. En ocasiones también se usará la notación $\mathcal{P} \mid\!\sim m$, donde el símbolo "$\mid\!\sim$ " será una abreviatura de "deriva rebatiblemente a". □

Obsérvese que la negación por falla esta definida en función del concepto de justificación, y no en función de la falla en la derivación como ocurre en PROLOG. Esto se debe fundamentalmente a que la inferencia en este lenguaje está definida por justificaciones, y no por derivaciones. En la sección 4.6 se explicará en detalle la semántica de la negación por falla en este lenguaje, y en el ejemplo 4.6.1 se mostrará la importancia de que la falla esté la construcción de una justificación.

Ejemplo 4.2.1 : Utilizando el PLR del ejemplo 4.1.3, es posible hallar una derivación rebatible para las siguientes metas: $\prec$ `perro(chicho)`, $\prec$ `toro(pepe)`, $\prec$ `~carnivoro(pepe)`, $\prec$ `tiene-cuernos(pepe)`, $\prec$ `~peligroso(pepe)`, $\prec$ `peligroso(pepe)`, y $\prec$ `peligroso(chicho)`.
La meta "`peligroso(pepe)`" tiene la siguiente derivación rebatible:

{ `toro(pepe)` $\leftarrow$ `true`
`~carnivoro(pepe)` $\leftarrow$ `toro(pepe)`
`tiene-cuernos(pepe)` $\prec$ `toro(pepe)`
`peligroso(pepe)` $\prec$ `~carnivoro(pepe), tiene-cuernos(pepe)` }

Mientras que la meta "`~peligroso(pepe)`" tiene la siguiente derivación rebatible:

$\{$ toro(pepe) $\leftarrow$ true
$\sim$carnivoro(pepe) $\leftarrow$ toro(pepe)
$\sim$peligroso(pepe) $\prec$ $\sim$carnivoro(pepe) $\}$

Por otro lado, utilizando la regla "peligroso(X) $\leftarrow$ not $\sim$peligroso(X)" es posible derivar rebatiblemente "peligroso(chicho)". □

Definición 4.2.2 : *Consistencia*
Un conjunto de cláusulas de programa $\mathcal{A}$ es consistente si no se puede derivar rebatiblemente un par de literales complementarios a partir de $\mathcal{A}$. Analogamente, un conjunto de cláusulas de programa $\mathcal{A}$ es inconsistente si puede derivarse rebatiblemente un par de literales complementarios a partir de $\mathcal{A}$. □

El ejemplo 4.2.1 muestra que a partir de un PLR es posible obtener derivaciones rebatibles para dos literales complementarios, como lo son " peligroso(pepe)" y "$\sim$peligroso(pepe)". Más adelante se mostrará el tratamiento de las conclusiones contradictorias pero antes, se establecerá la siguiente restricción sobre el conocimiento no rebatible representado en un PLR. *Dado un* PLR $\mathcal{P}$*, el conjunto* $\mathcal{S}$ *de* CPE *de* $\mathcal{P}$ *debe ser consistente, mientras que el conjunto* $\mathcal{D}$ *de* CPR *de* $\mathcal{P}$*, y el propio* $\mathcal{P}$ *(SS* $\cup$ $\mathcal{D}$*) pueden ser inconsistentes.* De esta forma, un PLR podrá contener información potencialmente inconsistente, mientras que su conocimiento no rebatible será consistente. En el ejemplo 4.1.3, $\mathcal{S}$ es consistente, mientras que $\mathcal{D}$, y por lo tanto $\mathcal{P}$, no lo son.

4.3 Expresividad de los PLR

Los PLR extienden la expresividad de la programación en lógica convencional, ya que permiten:

1. la derivación de conclusiones negativas (utilizando la negación clásica),
2. la representación de información tentativa (utilizando CPRs),
3. y la posibilidad de representar información potencialmente inconsistente.

Además extienden los sistemas de argumentación rebatible, ya que les incorporan la negación por falla que permite la representación de información incompleta.

A continuación se mostrará una serie de ejemplos donde se puede apreciar la expresividad de los PLR.

Ejemplo 4.3.1 :

vuela(X) $\prec$ ave(X).
ave(X) $\leftarrow$ pingüino(X).
$\sim$vuela(X) $\leftarrow$ pingüino(X).
pingüino(petete) $\leftarrow$ true.
ave(X) $\leftarrow$ gallina(X).
$\sim$vuela(X) $\prec$ gallina(X).
vuela(X) $\prec$ gallina(X),asustado(X).
anida-arbol(X) $\prec$ vuela(X).
anida-suelo(X) $\prec$ not anida-arbol(X).
gallina(coco) $\prec$ true.
asustado(coco) $\leftarrow$ true.

□

Ejemplo 4.3.2 :

salió(navegante1) $\leftarrow$ true.
salió(navegante2) $\leftarrow$ true.
regresó(navegante2) $\leftarrow$ true.
$\sim$regresó(X) $\leftarrow$ not regresó(X).
perdido(X) $\prec$ salió(X), $\sim$regresó(X).
buscar(X) $\prec$ perdido(X), not muerto(X).
$\sim$buscar(X) $\prec$ clima(muy-malo).
clima(muy-malo) $\leftarrow$ true.

□

Ejemplo 4.3.3 :

$\sim$culpable(X) $\leftarrow$ not culpable(X).
$\sim$preso(X) $\leftarrow$ inocente(X).
inocente(X) $\leftarrow$ $\sim$culpable(X).
preso(X) $\prec$ not culpable(X), alta-sospecha(X).
alta-sospecha(pepe) $\leftarrow$ true.

□

Ejemplo 4.3.4 :

$\sim$pista-libre(X) $-\!\prec$ not pista-libre(X).
pista-ocupada(X) $\leftarrow$ $\sim$pista-libre(X).
$\sim$pista-ocupada(X) $-\!\prec$ not pista-libre(X), orden-torre(X).
aterrizo(X) $-\!\prec$ $\sim$ocupada(X).
$\sim$aterrizo(X) $-\!\prec$ ocupada(X).
orden-torre(sur) $\leftarrow$ true.

□

4.4 Argumentos y justificaciones para las consultas

La noción de derivación rebatible no prohibe que puedan derivarse rebatiblemente dos literales complementarios. Por lo tanto, resulta necesario definir un criterio de inferencia, a fin de que sólo una de las metas complementarias sea aceptada como una nueva creencia. Para ello se definirá un *argumento* como un subconjunto de CPR de un programa lógico rebatible, y una vez hecho esto se podrán utilizar los conceptos de la *argumentación rebatible* introducidos en el capítulo 3, a fin de decidir entre conclusiones contradictorias.

Un *argumento* para una meta m será un subconjunto minimal y consistente de CPRs instanciadas con las cuales es posible derivar rebatiblemente m.

Definición 4.4.1 : *Argumento para un meta* m
Dado un PLR, formado por el conjunto $\mathcal{S}$ de CPE, y el conjunto $\mathcal{D}$ de CPR, un *argumento* $\mathcal{A}$ para una meta m, es un subconjunto de CPR instanciadas de $\mathcal{D}$, tal que:

1. Existe una derivación rebatible de m a partir de SS $\cup$ $\mathcal{A}$ (*i.e.*, SS $\cup$ $\mathcal{A}|\sim$ m),
2. SS $\cup$ $\mathcal{A}$ es consistente, y
3. $\mathcal{A}$ es el menor subconjunto (con respecto a la inclusión de conjuntos) que cumple las dos condiciones anteriores.

Si $\mathcal{A}$ es un argumento para m, también se dirá que $\langle$ $\mathcal{A}$,m $\rangle$ es una *estructura de argumento.*
□

A continuación se repetirán algunas de las definiciones del capítulo 3, sólo a efectos de clarificar el desarrollo de esta sección, y de reformular algunos conceptos para que se adapten al nuevo lenguaje definido.

Definición 4.4.2 : *Subargumento*
Un argumento $\langle \mathcal{B}, q \rangle$ es un *subargumento* de $\langle \mathcal{A}, h \rangle$ si $\mathcal{B} \subseteq \mathcal{A}$. □

Ejemplo 4.4.1 : En el ejemplo 4.3.1 existen derivaciones rebatibles para "`vuela(coco)`", y "`vuela(petete)`", pero sin embargo, sólo "`vuela(coco)`" tiene un argumento, ya que la derivación de "`vuela(petete)`" es inconsistente con $\mathcal{S}$. A continuación se muestra un argumento $\mathcal{A}_1$ para "∼`vuela(coco)`", y dos argumentos $\mathcal{A}_2$ y $\mathcal{A}_3$ para "`vuela(coco)`".

$\mathcal{A}_1 =$ { ∼`vuela(coco)` $-\!\prec$ `gallina(coco)` ; `gallina(coco)` $-\!\prec$ `true` }
$\mathcal{A}_2 =$ { `vuela(coco)` $-\!\prec$ `gallina(coco),asustado(coco)` ; `gallina(coco)` $-\!\prec$ `true` }
$\mathcal{A}_3 =$ { `vuela(coco)` $-\!\prec$ `ave(coco)` ; `gallina(coco)` $-\!\prec$ `true` }

□

Utilizando la información del programa del ejemplo 4.3.1, tanto "`vuela(coco)`", como "∼`vuela(coco)`", pueden derivarse rebatiblemente, y además en forma consistente con la información no rebatible. En este caso, se esperaría que la respuesta a la consulta "`vuela(coco)`" fuera "SI", mientras que a la consulta "∼`vuela(coco)`" sería "NO"

En la programación en lógica convencional hay dos respuesta posibles a una consulta: "SI" cuando se puede derivar con éxito la consulta, y "NO" en caso contrario. Por lo tanto, si se usara el método de inferencia de la programación en lógica tradicional, ante la consulta " $-\!\prec$ `vuela(coco)`" el programa respondería SI, pero ante la consulta " $-\!\prec$ ∼`vuela(coco)`" también respondería SI. Lo cuál no permite utilizar información contradictoria.

En la extensión a la programación en lógica propuesta por Gelfond y Lifschitz en [GelLif90] (ver capítulo 2), hay tres respuestas posibles: SI, NO, y DESCONOCIDO. Pero si se utilizara su método de inferencia, el programa del ejemplo 4.3.1 se volvería contradictorio, y por lo tanto se podría derivar todo el lenguaje.

Como se vió en el capítulo 2, la propuesta de Kowalsky y Sadri [KowSad90] intenta solucionar lo anterior, pero en este ejemplo, la regla "∼`vuela(X)` $\prec$ `gallina(X)`" se volvería una excepción de "`vuela(X)` $\prec$ `gallina(X),asustado(X)`" y por lo tanto, la respuesta a " $\prec$ `vuela(coco)`" sería NO, y " $\prec$ ∼`vuela(coco)`" sería SI. Que es justamente lo contrario a lo que se espera.

Por último, si se hubiera utilizado la propuesta de Inoue [Inoue91] como método de inferencia, habría dos extensiones, pero no se sabría cuál es la respuesta.

Lo que se propone hacer en este lenguaje, es utilizar el mecanismo de inferencia de la argumentación rebatible, el cuál puede decidir entre conclusiones contradictorias utilizando un criterio de comparación. Utilizando este método de inferencia, habrá cuatro respuestas posibles: SI, NO, DESCONOCIDO, e INDECISO. El proceso de obtención de una respuesta está explicado brevemente a continuación, y luego es explicado en detalle.

La consulta de una meta m tendrá éxito (y su respuesta será SI), cuando m pertenezca al *conjunto de respuestas positivas* del lenguaje, esto es, cuando exista un argumento que sea una *justificación* de m. El proceso de obtención de una justificación para m, involucra la construcción de un argumento $\mathcal{A}$, para m, que no esté derrotado. Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, llamado *árbol de dialéctica*, donde $\mathcal{A}$ es la raíz, y cada nodo tiene por hijos a sus derrotadores. En el árbol de dialéctica, argumentos y contraargumentos son comparados utilizando un criterio de preferencia.

Como se verá en detalle a continuación, en el caso del ejemplo 4.3.1 la respuesta para " $\prec$ `vuela(coco)`" será SI, mientras que la respuesta para " $\prec$ ∼`vuela(coco)`" será NO.

Tanto la argumentación rebatible, como la programación en lógica rebatible, pueden definirse independientemente del criterio de preferencia entre argumentos. En este trabajo se utilizará el criterio de *especificidad* para comparar argumentos, dicho criterio estará explicado en detalle en la sección 4.8. A continuación figuran las definiciones de los conceptos de la argumentación rebatible mencionados antes.

Definición 4.4.3 : *Contraargumento* [SCG94]
Sea $\mathcal{S}$ el conjunto de CPE del PLR. Se dirá que $\langle \mathcal{A}_2, h_2 \rangle$ *contraargumenta a* $\langle \mathcal{A}_1, h_1 \rangle$ en

un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que $\mathcal{S} \cup \{h, h_2\}$ es inconsistente.
El argumento $\langle \mathcal{A}, h \rangle$ se llamará subargumento de desacuerdo, y al literal h se lo llamará punto de contraargumentación. □

La definición de contraargumento establece que tiene que haber una inconsistencia entre el argumento $\langle \mathcal{A}_2, h_2 \rangle$ y algún subargumento de $\langle \mathcal{A}_1, h_1 \rangle$. Esto significará que ambos argumentos están implicando información que es contradictoria. Por ejemplo, en el caso de los argumentos $\mathcal{A}_1$ para "∼`vuela(coco)`", y $\mathcal{A}_2$ para "`vuela(coco)`". El argumento $\mathcal{A}_2$ es un contraargumento para $\mathcal{A}_1$, ya que $\mathcal{S} \cup \{$ ∼`vuela(coco)`, `vuela(coco)` $\}$ es inconsistente, con lo cual "∼`vuela(coco)`" es un punto de desacuerdo, y en este caso, el propio $\mathcal{A}_1$ es el subargumento de desacuerdo.

Definición 4.4.4 : *Derrotador* [SCG94]
Un argumento $\langle \mathcal{A}_2, h_2 \rangle$ *derrota a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h si y sólo si existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que: $\langle \mathcal{A}_2, h_2 \rangle$ contraargumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el literal h y se cumple una de estas dos opciones:
(1) $\langle \mathcal{A}_2, h_2 \rangle$ es estrictamente más específico que $\langle \mathcal{A}, h \rangle$ (derrotador propio), o
(2) $\langle \mathcal{A}_2, h_2 \rangle$ no puede compararse con $\langle \mathcal{A}, h \rangle$ (derrotador de bloqueo). □

Los PLR incorporan algunos conceptos que no estaban presentes en los sistemas de argumentación rebatible, como la negación por falla, y las presuposiciones, por lo tanto, se extendió la definición del criterio de especificidad, de tal manera que pueda comparar correctamente argumentos que contengan presuposiciones y el operador de negación por falla. Este desarrollo se encuentra en la sección 4.8.

Siguiendo con ejemplo anterior, $\mathcal{A}_2$ es estrictamente más específico que $\mathcal{A}_1$, y por lo tanto $\mathcal{A}_2$ es un derrotador propio de $\mathcal{A}_1$.

A continuación se mostrará cómo es el proceso por el cuál se decide que un argumento es una justificación para una meta m. Como se explicó antes, un argumento $\mathcal{A}$ será una *justificación* de m, cuando $\mathcal{A}$ no esté derrotado. Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, donde $\mathcal{A}$ es la raíz, y cada nodo tiene por hijos a sus derrotadores. La definición de árbol de dialéctica es la que caracteriza este proceso.

Definición 4.4.5 : Un *árbol de dialéctica* para $\langle \mathcal{A}, h \rangle$, denotado $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, se define recursivamente como sigue:

1. Un nodo que contiene una estructura de argumento $\langle \mathcal{A}, h \rangle$ sin derrotadores (propios o de bloqueo), es un árbol de dialéctica para $\langle \mathcal{A}, h \rangle$, y es también la raíz del árbol.
2. Supóngase que $\langle \mathcal{A}, h \rangle$ es una estructura de argumento con derrotadores (propios o de bloqueo) $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$. El árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, para $\langle \mathcal{A}, h \rangle$ se construye poniendo a $\langle \mathcal{A}, h \rangle$ como nodo raíz, y haciendo que este nodo sea el padre de las raíces de los árboles de dialéctica de $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$, *i.e.*, $\mathcal{T}_{\langle \mathcal{A}_1, h_1 \rangle}$, $\mathcal{T}_{\langle \mathcal{A}_2, h_2 \rangle}$, ..., $\mathcal{T}_{\langle \mathcal{A}_n, h_n \rangle}$.

□

Para todo argumento existe un árbol de dialéctica, el cuál puede rotularse utilizando la siguiente definición:

Definición 4.4.6 : Los nodos de un árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ se etiquetan recursivamente como *nodo no-derrotado* (nodo-U) o *nodo derrotado* (nodo-D) como sigue:

1. Una hoja de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un *nodo-U*.
2. Sea $\langle \mathcal{B}, q \rangle$ un nodo interno de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$.
   a) $\langle \mathcal{B}, q \rangle$ será un *nodo-U* sssi todo hijo de $\langle \mathcal{B}, q \rangle$ es un *nodo-D*.
   b) $\langle \mathcal{B}, q \rangle$ será un *nodo-D* sssi tiene al menos un hijo que es *nodo-U*.

□

Si en un árbol de dialéctica, la etiqueta de un nodo (que no sea la raíz) es *nodo-D*, entonces estará derrotado, y no podrá utilizarselo para derrotar a su padre. Pero lo importante es que si la etiqueta del nodo raíz (que es el argumento de la meta m), es *nodo-U*, entonces no estará derrotado, y por lo tanto se convertirá en una justificación para m.

Definición 4.4.7 : Un argumento $\langle \mathcal{A}, h \rangle$ es una *justifición* para h sssi la raíz de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un nodo-U. □

Ejemplo 4.4.2 : Al efectuar la consulta " $\prec\!\!\!-$ `~vuela(coco)`" al PLR del ejemplo 4.3.1, se puede construir el argumento $\mathcal{A}_1$. Pero el argumento $\mathcal{A}_2$ es un derrotador para $\mathcal{A}_1$, y como no existe otro argumento que derrote a $\mathcal{A}_2$, entonces $\mathcal{A}_2$, es un nodo-U en el árbol de dialéctica. Por lo tanto $\mathcal{A}_1$ no es una justificación.

Si se efectúa la consulta " $\prec\!\!\!-$ `vuela(coco)`" al mismo PLR, por el orden en que están dispuestas las cláusulas, primero se obtiene el argumento $\mathcal{A}_3$ (ver ejemplo 4.4.1). En este caso el argumento $\mathcal{A}_1$ es un derrotador de $\mathcal{A}_3$, pero como $\mathcal{A}_2$ es su vez un derrotor de $\mathcal{A}_1$, entonces $\mathcal{A}_1$ es un nodo-D y por lo tanto $\mathcal{A}_3$, es una justificación para "`vuela(coco)`". □

4.5 Semántica

La semántica de un PLR quedará caracterizada por cuatro conjuntos de repuestas: el conjunto de respuestas *positivas*, formado por aquellos literales para los cuales existe una justificación; el conjunto de respuestas *negativas*, compuesto por aquellos literales cuyos argumentos han sido derrotados; el conjunto de respuestas *indecisas*, formado por aquellos literales para los cuales no es posible decidir entre sus argumentos a favor y en contra; y por último el conjunto de respuestas *desconocidas* por el programa, que corresponde a aquellos literales que no poseen argumento que los sustente.

Dado un PLR, es posible identificar un conjunto finito de literales que tendrán una justificación. Estos literales corresponden a los que un razonador respondería afirmativamente ante una consulta.

Definición 4.5.1 : *Conjunto de respuestas positivas*
Sea $\mathcal{P}$ un PLR. El conjunto de respuestas positivas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, existe un argumento $\mathcal{A}$ que es una justificación para h. □

Aplicando la definición anterior, el conjunto de respuestas positivas del ejemplo 4.1.3 es { `toro(pepe)`, ∼`carnivoro(pepe)`, `tiene-cuernos(pepe)`, `peligroso(pepe)`, `perro(chicho)`, `peligroso(chicho)` }. Por otro lado, en el mismo ejemplo no es posible inferir "∼`peligroso(pepe)`", porque existe un argumento más específico para "`peligroso(pepe)`" (*i.e.*, un derrotador propio). En este caso, la respuesta para la consulta " $\multimap$ ∼`peligroso(pepe)`" debería ser negativa.

De esta forma, el conjunto de respuestas negativas de un PLR queda caracterizado de la siguiente manera:

Definición 4.5.2 : *Conjunto de respuestas negativas*
Sea $\mathcal{P}$ un PLR. El conjunto de respuestas negativas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, se cumple que para todo argumento $\mathcal{A}$ de h, existe un derrotador propio de $\mathcal{A}$, rotulado como nodo-U. □

Además de las respuestas positivas y negativas, existen consultas ante las cuales un razonador puede manifestarse indeciso, como en el caso de la consulta " $\multimap$ `buscar(navegante1)`" en el ejemplo 4.3.2. El razonador estará indefinido ya que aunque puede construir un argumento para "`buscar(navegante1)`", también tiene un argumento para "∼`buscar(navegante1)`" y no puede establecer que un argumento sea mejor que el otro. Este tipo de respuestas queda caracterizado por la siguiente definición.

Definición 4.5.3 : *Conjunto de repuestas indecisas*
Sea $\mathcal{P}$ un PLR. El conjunto de respuestas indecisas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, se cumple que para todo argumento $\mathcal{A}$ de h, $\mathcal{A}$ no tiene derrotadores propios, pero si tiene al menos un derrotador de bloqueo rotulado como nodo-U, (es decir, posee derrotadores que interfieren la justificación pero no puede probarse que sean mejores que $\mathcal{A}$). □

Por último existe un conjunto de respuestas, que por no tener ningún tipo de información al respecto, son totalmente desconocidas. Por ejemplo que número saldrá en la loteria mañana.

Definición 4.5.4 : *Conjunto de respuestas desconocidas*
Sea $\mathcal{P}$ un PLR. El conjunto de respuestas desconocidas de $\mathcal{P}$ es un conjunto de literales L, tal que para todo $h \in L$, h no pertenece a los conjuntos de respuestas anteriores. □

Observación 4.5.1 : Dado un PLR $\mathcal{P}$, los conjuntos de respuestas positivas y negativas de $\mathcal{P}$, son conjuntos consistentes (*i.e.*, no contienen un par de literales complementarios). En cambio, el conjunto de respuestas indecisas, es normalmente un conjunto inconsistente.

Observación 4.5.2 : Dado un PLR $\mathcal{P}$, los conjuntos de respuestas positivas, negativas, e indecisas son conjuntos finitos. En cambio, el conjunto de respuestas desconocidas es un conjunto infinito.

Una vez definido los posibles conjuntos de respuestas que puede generar un PLR, se establecerá como debe comportarse un intérprete de este lenguaje ante una consulta.

Definición 4.5.5 : Dado un PLR $\mathcal{P}$ y una consulta m, un intérprete de programas lógicos rebatibles, responderá:

- SI, en el caso que m pertenezca al conjunto de respuestas positivas.
- NO, en el caso que m pertenezca al conjunto de respuestas negativas.
- INDECISO, en el caso que m pertenezca al conjunto de respuestas indecisas.
- DESCONOCIDO, en el caso que m pertenezca al conjunto de respuestas desconocidas.

Si la consulta está precedida por `not`, entonces como se verá a continuación, la respuesta podrá ser SI, o DESCONOCIDO. □

4.6 La negación en los PLR

La semántica de la negación en la programación en lógica tradicional es la *negación por falla finita*, representada habitualmente por el símbolo "`not`". La consulta de una meta "`not` p" tiene éxito cuando existe una falla finita de la derivación del predicado p. Este tipo de semántica puede ser útil para algunos predicados en algunos contextos, como en el caso de la cláusula "`buscar(X)` ← `not muerto(X),perdido(X)`", (esto es, si '`X`' está perdido y no puedo probar que esté muerto, sigo buscándolo). No obstante existen numerosas ocasiones donde la utilización de la negación por falla lleva a resultados no deseables. Por ejemplo, si se quiere representar la regla "aterrizar cuando la pista no está ocupada" con la cláusula "`aterrizar` ← `not pista-ocupada`", la vida de los pasajeros correrá mucho peligro cuando no se posea información sobre la pista. Por lo tanto, parecería conveniente que la negación por falla sea utilizada sólo en algunos casos y la negación clásica en otros.

Los PLR permiten utilizar dos tipos de negación: la negación clásica denotada con el símbolo "∼", y la negación por falla finita denotada con el símbolo "`not`". El objetivo de incluir la negación clásica es poder representar información potencialmente inconsistente, ya que tanto una meta de la forma "∼`a`", como una de la forma "`a`" pueden ser derivadas rebatiblemente a partir de un PLR. Aunque no se permiten anidamientos del símbolo "∼", se asume que ∼∼`a` es `a`. Utilizando la negación clásica es posible escribir "`aterrizar` ← ∼`pista-ocupada`", que representa el significado deseado, esto es, "aterrizar si puedo probar que la pista no está ocupada".

La negación por falla, en cambio, se incluye para poder trabajar con información incompleta. Por ejemplo, al escribir la cláusula "`inocente(X)` $-\!\prec$ `not culpable(X)`", se está indicando que "no poder probar la culpabilidad de `X`, es una buena razón para asumir que `X` es inocente". Es por ello que la utilización de la negación por falla se restringe sólo al cuerpo de una cláusula.

La diferencia fundamental entre las dos negaciones está en su significado. El significado buscado para "∼`a`" es "*existe una justificación para* ∼`a`", mientras que el de "`not a`" es "*no existe una justificación para* `a`". Es importante destacar que aunque las metas precedidas por "`not`" puedan probarse rebatiblemente, estas nunca forman parte de los conjuntos de respuestas, (como "`not` ∼`peligroso(pepe)`" en el ejemplo 4.1.3).

La consulta de una meta "∼`a`" tendrá éxito cuando "∼`a`" figure en el conjunto de respuestas positivas del PLR (*i.e.*, tenga una justificación), como ocurre con

"∼carnivoro(pepe)" en el ejemplo 4.1.3. Pero sin embargo, una submeta "not l" tendrá éxito, cuando "l" no figure en el conjunto de respuestas positivas, *i.e.*, cuando "l" no tenga éxito. En el ejemplo 4.1.3 "∼peligroso(pepe)" no figura en el conjunto de respuestas positivas del PLR, y por lo tanto puede probarse rebatiblemente "not ∼peligroso(pepe)".

Es importante destacar que la negación por falla de los PLR es diferente a la negación por falla de PROLOG. Aquí not p tiene éxito cuando no es posible justificar p, mientras que en PROLOG, not p tiene éxito cuando no es posible derivar p. La razón fundamental es que la inferencia en este lenguaje está caracterizada por la justificacón, y no por la derivación rebatible. En el ejemplo 4.6.1 puede verse claramente esta situación. Como existe una derivación rebatible para muerto(popeye), si se utilizara al not como falla de la derivación, entonces buscar(popeye) no daría éxito. Sin embargo, no existe una justificación para muerto(popeye), ya que el argumento para muerto(popeye) tiene un derrotador de bloqueo en el punto ∼toma_agua(popeye). Por lo tanto utilizando el criterio establecido para el not de los PLR, la consulta $\prec$ buscar(popeye) tendrá éxito. Como la negación por falla en los PLR, es diferente a la de PROLOG, entonces se utilizará el símbolo "naf" para la negación por falla de PROLOG.

Ejemplo 4.6.1 :

```
buscar(X) ≺ perdido_mar(X), not muerto(X).
muerto(X) ≺ ∼toma_agua(X).
∼toma_agua(X) ≺ perdido_mar(X).
toma_agua(X) ≺ junta_lluvia(X).
perdido_mar(popeye) ≺ true.
junta_lluvia(popeye) ≺ true.
```

□

Una consulta de la forma not p puede tener éxito o fallar, pero nunca tendrá una respuesta indecisa, por lo tanto una consulta $\prec$ not p tendrá sólo dos respuestas posibles: "SI", en el caso que p no figure en el conjunto de respuestas positivas del programa, y "DESCONOCIDO", en el caso que p pertenezca al conjunto de respuestas positivas.

Proposición 4.6.1 : *La negación en este lenguaje cumple con el criterio de coherencia* [AlfPer93], *esto es, si la meta "∼m" tiene éxito, entonces "not m" también lo tendrá.* Demostración: si "∼m" tiene éxito, entonces "∼m" pertenece al conjunto de respuestas positivas, es decir existe una justificación para "∼m", luego, no existirá una justificación para "m", y por lo tanto "not m" tendrá éxito. □

Es fácil ver que la recíproca no es cierta, es decir, que "`not m`" tenga éxito no implica que el literal "∼`m`" lo tenga. En el ejemplo 4.1.3 aunque pueda derivarse "`not toro(chicho)`", no ocurre lo mismo con "∼`toro(chicho)`".

Es importante destacar que tanto "`not`" como "∼" son importantes, y como se verá a continuación, no pueden definirse uno en función del otro. Una cláusula como "`p` ← `not q`" indica que al no poder justificarse `q`, entonces puedo derivar `p`. Mientras que utilizando "`p` ← ∼`q`", se podrá derivar `p` cuando se pueda derivar ∼`q`. No obstante, en un PLR puede escribirse una presuposición "∼`q` –≺ `true`" para indicar que "hay razones (rebatibles) para creer en ∼`q`". De esta manera se podría pensar que la cláusula "`p` ← `not q`" puede reemplazarse por el par { `p` ← ∼`q` ; ∼`q` –≺ `true` }, obteniéndose un resultado equivalente. Los siguientes PLRs $\mathcal{P}_1$ y $\mathcal{P}_2$ muestran ambas situaciones:

$\mathcal{P}_1$	$\mathcal{P}_2$
`p` ← `not q.`	`p` ← ∼`q.`
	∼`q` –≺ `true.`

Si $\mathcal{P}_1$ y $\mathcal{P}_2$ fueran equivalentes, entonces para todo PLR que posea una meta con `not`, se podría obtener un PLR equivalente sin la utilización del operador de negación por falla, simplemente haciendo los reemplazos correspondientes. Aunque a primera vista parece que el resultado fuera equivalente, hay varias diferencias. En primer lugar, si se usan las cláusulas de $\mathcal{P}_2$ como reemplazo de $\mathcal{P}_1$, se pierde el significado buscado con el uso del operador "`not`", ya que ahora `p` podrá derivarse cuando exista una justificación de ∼`q`.

Pero además, para que $\mathcal{P}_1$ y $\mathcal{P}_2$ sean equivalentes, deben tener los mismos conjuntos de respuestas. Sin embargo, el conjunto de respuestas positivas de $\mathcal{P}_1$ es {`p`}, mientras que el de $\mathcal{P}_2$ es {`p, ∼q`}.

Por otro lado, al agregar "`q` –≺ `true`" a ambos PLR, se acentúan las diferencias, ya que el conjunto de respuestas positivas para $\mathcal{P}_1 \cup$ { `q` –≺ `true` } es {`q`}, mientras que el de $\mathcal{P}_2 \cup$ { `q` –≺ `true` } es $\emptyset$. Por otro lado resulta imposible agregar "∼`p` ← `true`" a $\mathcal{P}_1$ porque el conjunto de CPE se vuelve inconsistente, mientras que sí es posible agregarlo a $\mathcal{P}_2$ y el conjunto de respuestas positivas para $\mathcal{P}_2 \cup$ { ∼`p` ← `true` } es {∼`p` , ∼`q`}. Por último, el conjunto de respuestas positivas para $\mathcal{P}_1 \cup$ { ∼`p` –≺ `true` } es {`p`}, mientras que el de $\mathcal{P}_2 \cup$ { ∼`p` –≺ `true` } es { ∼`q`}, ya que los literales `p` y ∼`p` pasan a pertenecer al conjunto de respuestas indecisas.

Otra diferencia importante entre "`not`" y "$\sim$" es que la consistencia está definida sobre "$\sim$"y no sobre "`not`". Por ejemplo, para que un conjunto de CPR sea un argumento (ver definición 4.4.1) se establece una condición de consistencia, que impide que dentro de un argumento existan dos literales complementarios con respecto a la negación clásica. Sin embargo con la negación por falla no ocurre lo mismo. Supóngase que se tiene el siguiente PLR: $\mathcal{P}$= { `a` $\prec$ `b,c` ; `b` $\prec$ `p` ; `c` $\prec$ `not p` ; `p` $\prec$ `e` ; $\sim$`p` $\prec$ `d` ; `d` $\leftarrow$ `true` ; `e` $\leftarrow$ `true` }. Aquí es posible construir el argumento $\mathcal{A}$= { `a` $\prec$ `b,c` ; `b` $\prec$ `p` ; `p` $\prec$ `e` ; `c` $\prec$ `not p` } para el literal "`a`". Obsérvese que en el mismo argumento se usaron como submetas a "`p`" y "`not p`". Esto es posible porque el literal "`p`" puede probarse rebatiblemente, pero no existe un justificación para "`p`", ya que es bloqueada por el argumento "{$\sim$`p` $\prec$ `d` }". La siguiente proposición muestra que aunque el argumento anterior pueda construirse, nunca podrá ser una justificación.

Proposición 4.6.2 : *Un argumento $\mathcal{A}$ donde figuren las submetas "l" y "*`not` *l", nunca será una justificación.*
Demostración: (a) si "l" está justificado, entonces "`not` l" no podrá probarse, invalidando al argumento. (b) si "`not` l" está probado, entonces no existe una justificación para "l", esto es, existe un derrotador aceptable para todo argumento de l, y por lo tanto existirá un derrotador aceptable para $\mathcal{A}$. □

4.7 La CWA como cláusulas de programa

En un programa lógico tradicional, la respuesta a una consulta es 'SI' o 'NO', ya que automáticamente se aplica la *hipótesis de mundo cerrado* (*closed world assumption* o CWA), asumiendo como falso a todo átomo instanciado que no pueda derivarse del programa. Esto resulta demasiado restrictivo y puede llevar a situaciones no deseables.

Como se vió anteriormente, en la programación en lógica rebatible, hay cuatro respuestas posibles para una consulta: SI, NO, INDECISO, y DESCONOCIDO. La respuesta a una consulta que no puede derivarse del programa es 'DESCONOCIDO', en lugar de 'NO', y por lo tanto no se aplica automáticamente CWA a ningún predicado. No obstante, al disponer de dos tipos de negación, los PLR permiten representar la *hipótesis de mundo cerrado* directamente como cláusulas de programa, y además crear nuevas formas de CWA.

La hipótesis de mundo cerrado de un predicado particular p en un PLR, puede lograrse incluyendo en el PLR la cláusula "$\sim$`p(X)` $\leftarrow$ `not p(X)`" con lo cual se está indicando

que: *si no se puede justificar* `p(X)`*, entonces puede derivarse* ∼`p`; que es justamente lo que establece la hipótesis de mundo cerrado. De esta forma, es posible representar dentro del mismo lenguaje, algo que hasta ahora era una condición metalingüística. Por ejemplo en un PLR con la cláusula "∼`culpable(X)` ← `not culpable(X)`", si no se puede justificar que `X` sea culpable, se puede asumir que no lo es. Como las cláusulas de CWA se representan dentro del propio lenguaje, pueden representarse ahora otras formas de CWA:

`p(X)` ← `not` ∼`p(X)`	`peligroso(X)` ← `not` ∼`peligroso(X)`
`p(X)` ← `not p(X)`	`inocente(X)` ← `not inocente(X)`
∼`p(X)` ← `not` ∼`p(X)`	∼`culpable(X)` ← `not` ∼`culpable(X)`

Estas formas de CWA ya habían sido introducidas por Gelfond y Lifschitz en [GelLif90], donde también se aprovechaba la facilidad de disponer de dos tipos de negación. No obstante, hasta el momento, todo el desarrollo sobre cláusulas de CWA se ha realizado utilizando cláusulas de programa extendido (*i.e.*, CPE). Como se verá a continuación esto puede traer algunos problemas.

Considérese por ejemplo el siguiente PLR:

$\mathcal{P}_3$: `pista-ocupada(X)` ← `not pista-ocupada(X)`
∼`pista-ocupada(norte)` ← `true`

En la propuesta de Gelfond y Lifschitz, un programa como $\mathcal{P}_3$ es "contradictorio", y por lo tanto a partir de él se puede inferir todo el lenguaje. Por otro lado, $\mathcal{P}_3$ tampoco puede ser un programa lógico rebatible, porque la definición de PLR establece que el conjunto de CPE debe ser consistente.

Afortunadamente, el lenguaje de programación en lógica rebatible dispone de dos tipos de cláusulas: las CPE, y las CPR. La diferencia entre utilizar una CPE o una CPR es importante, las primeras representan conocimiento seguro (estricto), mientras que las CPR representan información tentativa (rebatible). Los PLR, permitirán entonces, obtener un nuevo tipo de cláusula de CWA: las *cláusulas de CWA rebatibles*, escribiendo por ejemplo "∼`p(X)` $-\!\prec$ `not p(X)`", esto es, " *no poder justificar p(X) es una buena razón (rebatible) para derivar* ∼*p(X)*". Al utilizar este nuevo tipo de CWA, se pueden eliminar los problemas de $\mathcal{P}_3$, escribiendo:

$\mathcal{P}_3'$: `pista-ocupada(X)` $\prec$ `not pista-ocupada(X).`
$\sim$`pista-ocupada(norte)` $\leftarrow$ `true.`

En $\mathcal{P}_3'$ ahora sólo es posible inferir el literal "$\sim$pista-ocupada(norte)", desapareciendo el problema de la inconsistencia. Obsérvese que aunque existe la derivación rebatible { `pista-ocupada(norte)` $\prec$ `not pista-ocupada(norte)` } para el literal "`pista-ocupada(norte)`", ésta no es consistente con el conjunto de CPE, y por lo tanto no es un argumento.

En general para cualquier predicado P toda cláusula de la forma "$P \leftarrow$ `not` P" producirá un problema similar al del programa $\mathcal{P}_3$, ya que permite que el predicado P pueda derivarse rebatiblemente cuando no se pueda justificarlo. Esto hará que la inclusión de cualquier instancia de $\sim P$, produzca una inconsistencia. El mismo problema aparece con la cláusula "$\sim P \leftarrow$ `not` $\sim P$". Por lo tanto siempre será conveniente representar este tipo de reglas con cláusulas rebatibles.

Otra situación problemática se presenta al utilizar una cláusula de CWA como "$\sim P \leftarrow$ `not` P". Considérese por ejemplo el siguiente PLR:

$\sim$`culpable(X)` $\leftarrow$ `not culpable(X).` (*)
$\sim$`preso(X)` $\leftarrow$ `inocente(X).`
`inocente(X)` $\leftarrow$ $\sim$`culpable(X).`
`preso(X)` $\prec$ `not culpable(X), alta-sospecha(X).`
`alta-sospecha(pepe)` $\leftarrow$ `true.`

En el PLR anterior aunque esté presente el hecho "`alta-sospecha(pepe)`", la meta "$\sim$`preso(pepe)`" tiene una justificación, gracias a la cláusula (*). Contrario a las espectativas de que exista una justificación para "`preso(pepe)`", el conjunto { `preso(pepe)` $\prec$ `not culpable(pepe), alta-sospecha(pepe)` } no es un argumento ya que es inconsistente con el conjunto de CPE.

Sin embargo, si la cláusula (*) es reemplazada por la cláusula rebatible r = "$\sim$`culpable(X)` $\prec$ `not culpable(X)` ", entonces es posible construir un argumento $\mathcal{A}$ para "$\sim$`preso(pepe)`" y otro argumento $\mathcal{B}$ para "`preso(pepe)`".

$\mathcal{A}$= { $\sim$`culpable(pepe)` $\prec$ `not culpable(pepe)` }
$\mathcal{B}$= { `preso(pepe)` $\prec$ `not culpable(pepe), alta-sospecha(pepe)` }

Utilizando el criterio de especificidad, $\mathcal{B}$ es un derrotador propio de $\mathcal{A}$. Por lo tanto utilizando la CPR r, $\mathcal{B}$ se convierte en una justificación para "`preso(pepe)`", obteniéndose el resultado esperado.

Una de las ventajas más importantes de la programación en lógica rebatible es su adaptabilidad al cambio. Al manejar adecuadamente la inconsistencia, permite que se agregue información en forma dinámica, sin presentar problemas. Por ejemplo, en el PLR $\mathcal{P}_4$, puede obtenerse tanto una justificicación para "`pista-ocupada(norte)`", como para "`pista-ocupada(sur)`".

$\mathcal{P}_4$:
`pista-ocupada(X)` $\prec$ `not pista-ocupada(X).`
$\sim$`pista-ocupada(X)` $\leftarrow$ `pista-libre(X).`

Sin embargo, si se agregan dos nuevos hechos (ver $\mathcal{P}_4'$), entonces ya no es posible justificar los literales anteriores. Esto se debe a que los argumentos que se habían construído antes, con el agregado de los nuevos hechos, no cumplen con la condición de consistencia de la definición 4.4.1. En el nuevo PLR $\mathcal{P}_4'$ ahora existe una justificación para " $\sim$pista-ocupada(norte)", "pista-libre(sur)" y "$\sim$pista-ocupada(sur)".

$\mathcal{P}_4'$:
`pista-ocupada(X)` $\prec$ `not pista-ocupada(X).`
$\sim$`pista-ocupada(X)` $\leftarrow$ `pista-libre(X).`
$\sim$`pista-ocupada(norte)` $\leftarrow$ `true.`
`pista-libre(sur)` $\leftarrow$ `true.`

Observación sobre los ciclos: Al utilizar una regla como "$\sim$`a` $\leftarrow$ `not a`" el mecanismo de derivación no incurre en un ciclo, ya que la definición de derivación rebatible lo prevee explícitamente. No obstante, al encontrar una submeta "`not` l" en una cláusula C se necesita saber si "l" tiene o no una justificación. A fin de no incurrir en un ciclo entre justificaciones, al intentar justificar "l", no debe permitirse que se utilice nuevamente la cláusula C que disparó el proceso. Por ejemplo en el PLR { `a` $\prec$ `not b` ; `b` $\prec$ `not a` }, no existe una justificación para "`a`", ni para "`b`".

4.8 El criterio de especificidad en los PLR

Cuando dos metas complementarias pueden ser derivadas rebatiblemente de un PLR, resulta necesario definir un criterio de inferencia, a fin de que sólo una de las metas tenga éxito. Para ello se utiliza un criterio de comparación entre los argumentos que sustentan dichas metas. La comparación de argumentos es un problema abierto, y existen diferentes propuestas que intentan solucionarlo. Una de ellas es el *criterio de especificidad*, el cuál fue descripto por Poole en [Poole85a], y separadamente por Loui en [Loui87]. Algunos sistemas de razonamiento rebatible han utilizado este criterio para comparar sólo reglas rebatibles (ver Nute [Nute92]), y otros para comparar argumentos enteros (ver Simari [SimLou92]).

El objetivo de esta sección es definir el criterio de especificidad para los PLR. Se tomará como punto de partida la definición de especificidad introducida en [SimLou92], y se construirá una definición equivalente a ésta pero que sólo utiliza la información de los argumentos. De esta manera, el cálculo de la especificidad será mucho más eficiente desde el punto de vista del tiempo de ejecución. Luego se extenderá la nueva definición para comparar correctamente argumentos que contengan presuposiciones y el operador de negación por falla.

La definición 4.8.1, muestra la relación de especificidad entre argumentos del sistema de argumentación rebatible definido en [SimLou92]. En dicho sistema, $\Delta^{\downarrow}$ representa el conjunto de reglas rebatibles instanciadas, y $\mathcal{K}$ al conjunto de reglas no rebatibles ($\mathcal{K}$= $\mathcal{K}_G \cup \mathcal{K}_P$), distinguiendo con $\mathcal{K}_P$ al conocimiento particular (subconjunto de hechos instanciados) y con $\mathcal{K}_G$ al conocimiento general (reglas sin instanciar). El símbolo "$\mid\!\sim$ " se utiliza para denotar la derivación rebatible.

Definición 4.8.1 : Especificidad [SimLou92]
Sea $\mathbf{L} = \{l : l$ es un literal instanciado y $\mathcal{K} \cup \Delta^{\downarrow} \mid\!\sim l \}$. El argumento $\mathcal{A}$ para h_1 es *estrictamente más específico* que el argumento $\mathcal{B}$ para h_2, si y solo si

1. para todo conjunto $C \subseteq \mathbf{L}$
   si $\mathcal{K}_G \cup C \cup \mathcal{A} \mid\!\sim h_1$ (C activa $\mathcal{A}$) y $\mathcal{K}_G \cup C \not\vdash h_1$, (activación no trivial)
   entonces $\mathcal{K}_G \cup C \cup \mathcal{B} \mid\!\sim h_2$. ($C$ activa $\mathcal{B}$)
2. existe un conjunto $C' \subseteq \mathbf{L}$ tal que:
   $\mathcal{K}_G \cup C' \cup \mathcal{B} \mid\!\sim h_2$, ($C'$ activa $\mathcal{B}$), $\mathcal{K}_G \cup C' \not\vdash h_2$ (activación no trivial)
   y $\mathcal{K}_G \cup C' \cup \mathcal{A} \not\mid\!\sim h_1$. ($C'$ no activa $\mathcal{A}$)

Si un argumento $\mathcal{A}$ es estrictamente más específico que $\mathcal{B}$, se denotará $\mathcal{A} \succ \mathcal{B}$. En el caso que $\mathcal{A} \not\vdash \mathcal{B}$ y que $\mathcal{B} \not\vdash \mathcal{A}$, se dirá que $\mathcal{A}$ y $\mathcal{B}$ son incomparables, y se notará $\mathcal{A} \not\succ\!\prec \mathcal{B}$. □

Ejemplo 4.8.1 : Considérense los argumentos $\mathcal{A}_1 = \{\texttt{a}_1 \prec \texttt{b,c}\}$, $\mathcal{A}_2 = \{\texttt{a}_2 \prec \texttt{b}\}$, $\mathcal{A}_3 = \{\texttt{a}_3 \prec \texttt{d}\}$, $\mathcal{A}_4 = \{\texttt{a}_4 \prec \texttt{c} \ ; \ \texttt{c} \prec \texttt{d}\}$. Utilizando el criterio de especificidad, resulta: $\mathcal{A}_1 \succ \mathcal{A}_2$, $\mathcal{A}_3 \succ \mathcal{A}_4$, $\mathcal{A}_1 \not\succ\!\prec \mathcal{A}_3$, $\mathcal{A}_2 \not\succ\!\prec \mathcal{A}_3$, $\mathcal{A}_1 \succ \mathcal{A}_4$, y $\mathcal{A}_2 \not\succ\!\prec \mathcal{A}_4$. □

Observación 4.8.1 : A partir del PLR formado por $\mathcal{S}$= {a ← h ; h ← true } y $\mathcal{D}$= {b ≺ h }, se puede obtener el argumento $\mathcal{A}$= {} para el literal "a", y el argumento $\mathcal{B}$={ b ≺ h} para el literal "b". Nótese que un argumento vacío, corresponde a una derivación estricta, esto es, no utiliza ninguna CPR, y por lo tanto no hay nada en él que pueda ser refutado. Contrario a las espectativas de que el argumento $\mathcal{A}$ sea más específico que $\mathcal{B}$, siguiendo la definición 4.8.1 ambos argumentos son incomparables. Por lo tanto, a fin de que la especificidad se comporte adecuadamente con argumentos vacíos, un argumento vacío será preferido a uno no vacío, y en el caso que ambos sean vacíos se considerarán incomparables. No obstante, en la argumentación rebatible el criterio de especificidad se utiliza para comparar argumentos en desacuerdo, esto es, sus conclusiones son inconsistentes con el conjunto $\mathcal{S}$. En este caso, el problema de los argumentos vacíos no aparece, ya que la condición de consistencia de la definición de argumento impedirá la formación del argumento no vacío. Por ejemplo en el PLR $\mathcal{P}$= {a ← h ; h ← true ; ∼a ≺ h } es imposible construir un argumento para ∼a.

Al aplicar la definición 4.8.1 hay que considerar todos los subconjuntos de **L**. El conjunto **L** contiene a todos los literales que se pueden derivar rebatiblemente del programa, y si tiene n elementos, entonces habrá 2^n conjuntos para considerar. El principal problema, es que al aplicar la definición, se están considerando gran cantidad de conjuntos de literales que no están relacionados con los argumentos en comparación, siendo irrelevantes para la misma.

Una definición equivalente a la 4.8.1, pero con un menor costo computacional, puede obtenerse si se focaliza únicamente sobre los literales de los argumentos. A continuación se introducirán algunos conceptos a fin de obtener la nueva definición de especificidad. En lo que resta de la sección, se asumirá que las cláusulas de programa no poseen presuposiciones ni submetas con el operador "not". La incorporación de estos dos nuevos elementos se analizará en las dos secciones siguientes. Para adaptar la definición de especificidad a los PLR, se debe reemplazar a $\mathcal{K}$ por el conjunto de CPE $\mathcal{S}$, y a $\Delta^{\downarrow}$ por el conjunto de CPR instanciadas.

Definición 4.8.2 : *Argumento completado.*
Sea $\mathcal{A}$ un argumento para el literal instanciado h, el argumento completado de $\mathcal{A}$, denotado $\mathcal{A}^c$, es un conjunto de CPE y CPR formado por las CPR de $\mathcal{A}$, más el subconjunto de CPE de $\mathcal{S}$ que no son hechos, y que fueron utilizadas para obtener la derivación de h a partir de SS $\cup$ $\mathcal{A}$. □

Ejemplo 4.8.2 : En el PLR formado por $\mathcal{S}$={ h ← a ; b ← d ; c ← d ; d ← true ; c ← true}, y $\mathcal{D}$= {a $\prec$ b,c }, $\mathcal{A}$= {a $\prec$ b,c} es un argumento para el literal h. El argumento completado para $\mathcal{A}$ es $\mathcal{A}^c$={ h ← a ; a $\prec$ b,c ; b ← d }. □

Observación 4.8.2 : Dado un argumento $\mathcal{A}$ para h, no existe un único argumento completado $\mathcal{A}^c$, ya que puede haber diferentes reglas en $\mathcal{S}$ con las cuales construir $\mathcal{A}$. No obstante, la diferencia entre dos argumentos completados $\mathcal{A}^c$ y $\mathcal{A}^{c\prime}$, son sólo CPE. Por ejemplo $\mathcal{A}^{c\prime}$={ h ← a ; a $\prec$ b,c ; b ← d ; c ← d } es otra forma de completar el argumento $\mathcal{A}$ del ejemplo anterior.

Definición 4.8.3 : *Conjunto de literales básicos de $\mathcal{A}^c$.*
Sea $\mathcal{A}^c$ un argumento completado, el conjunto de literales básicos de $\mathcal{A}^c$ denotado $Lit(\mathcal{A}^c)$, es el conjunto de literales que aparecen en los antecedentes y consecuentes de toda cláusula de $\mathcal{A}^c$. □

Definición 4.8.4 : *Conjunto de activación.*
Sea $\mathcal{A}^c$ un argumento completado, y sea $Lit(\mathcal{A}^c)$ el conjunto de literales correspondiente. Un subconjunto $U \subseteq Lit(\mathcal{A}^c)$ es un *conjunto de activación* de $\mathcal{A}^c$, si U junto con $\mathcal{A}^c$ derivan rebatiblemente a h (*i.e.*, $U \cup \mathcal{A}^c \mid\sim h$), y además U es minimal con respecto a la inclusión de conjuntos (*i.e.*, no existe $U' \subseteq U$ tal que $U' \cup \mathcal{A}^c \mid\sim h$). Se denotará con Act-sets($\mathcal{A}^c$), al conjunto de todos los conjuntos de activación de $\mathcal{A}^c$. □

Los conjuntos de activación serán los encargados de reemplazar a los subconjuntos C de **L** que se utilizan en la definición 4.8.1. La diferencia entre considerar sólo los conjuntos de activación en lugar de todos los subconjuntos de **L**, es apreciable en el siguiente ejemplo.

Ejemplo 4.8.3 : Sea $\mathcal{P}$ el PLR formado por: $\mathcal{S}$= { a ← b,c ; c ← f,g ; z ← h ; x ← h ; y ← h ; h ← true ; i ← true ; j ← true ; k ← true ; g ← true } y, $\mathcal{D}$= { b $\prec$ d,e ; d $\prec$ h,i ; f $\prec$ j,k ; t $\prec$ x ; u $\prec$ y }.
Sea $\mathcal{A}^c$ el argumento completado {a ← b,c ; b $\prec$ d,e ; d $\prec$ h,i ; c ← f,g ; f $\prec$ j,k}

para el literal "a"
En este caso **L**= { a,b,c,d,e,f,g,h,i,j,k,u,t,x,y,z }, por lo tanto hay 2^{16} subconjuntos para considerar, además $Lit(\mathcal{A}^c)$= {a,b,c,d,e,f,g,h,i,j,k}, lo cuál sólo bajaría el número de subconjuntos a 2.048. Sin embargo, solamente hay 10 conjuntos de activación para considerar: {a}, {b,c}, {d,e,c}, {h,i,e,c}, {h,i,e,f,g}, {h,i,e,j,k,g}, {d,e,f,g}, {d,e,j,k,g}, {b,f,g}, {b,j,k,g}. □

Obsérvese que dado un argumento completado $\mathcal{A}^c$ para una meta h, la definicion 4.8.4, no excluye la posibilidad de que un conjunto de activación U derive trivialmente la meta h (*i.e.*, $U \cup \mathcal{K}_G \vdash h$). En el ejemplo 4.8.3 { b,c }, {a}, y {b,f,g} son conjuntos de activación que trivialmente derivan la conclusión h. Esta clase de conjuntos de activación no son tenidos en cuenta por la definición de especificidad, ya que la condición "$U \cup \mathcal{K}_G \nvdash h$", lo prohibe. Por lo tanto, sería más util, eliminar dichos conjuntos durante la generación de los conjuntos de activación. Para esto es que se define un conjunto de activación no trivial.

Definición 4.8.5 : *Conjunto de activación no trivial.*
Dado un argumento completado $\mathcal{A}^c$ para la meta h, se dirá que U es un *conjunto de activación no trivial* de $\mathcal{A}^c$, si U es un conjunto de activación de $\mathcal{A}^c$, y $U \cup \mathcal{K}_G \nvdash h$. Se denotará con NTAct-sets($\mathcal{A}^c$) al conjunto de todos los conjuntos de activación no triviales de $\mathcal{A}^c$. □

El algoritmo de la figura 4.1, indica como calcular todos los conjuntos de activación no triviales de un argumento completado. Para determinar si un conjunto de activación es trivial o no, se observa si fue usada o no una CPR. Nótese que dado un argumento $\mathcal{A}$ para una meta h, el conjunto {h} es un conjunto de activación trivial de $\mathcal{A}$.

Como puede verse, el algoritmo calcula todos los conjuntos de activación de una manera eficiente, simplemente recorriendo el argumento completado. Aplicando el algoritmo al PLR del ejemplo 4.8.3, se obtiene NTAct-sets($\mathcal{A}^c$)= { {d,e,c}, {h,i,e,c}, {h,i,e,f,g}, {h,i,e,j,k,g}, {d,e,f,g}, {d,e,j,k,g}, {b,j,k,g} }. El siguiente paso será entonces definir el criterio de especificidad para que sólo tenga que considerar los conjuntos de activación no triviales.

Definición 4.8.6 : *Especificidad*
Dado el argumento no vacío $\mathcal{A}$ para la meta h_1, y el argumento no vacío $\mathcal{B}$ para h_2, y $\mathcal{A}^c$, $\mathcal{B}^c$ sus respectivos argumentos completados. Se dirá que $\mathcal{A}$ es *estrictamente más*

Algoritmo: *Conjuntos de activación*
Entrada: El argumento completado $\mathcal{A}^c$ para h.
Salida: NTAct-sets($\mathcal{A}^c$) y Act-sets($\mathcal{A}^c$).

1. Se inicializa una cola C con el par ({h},trivial).
2. Act-sets($\mathcal{A}^c$) y NTAct-sets($\mathcal{A}^c$) se inicializan en vacío.
3. REPETIR HASTA QUE C esté vacía
   (a) Sacar de C un par (*conj*,*tipo*),
   (b) Para cada literal $l_i \in$ *conj* que sea consecuente de una cláusula r de $\mathcal{A}^c$, reemplazar en *conj* a l_i por los literales del antecedente de r, con lo cuál se obtiene un nuevo conjunto de activación c_i.
   El *tipo* de c_i será *trivial* sólo si el *tipo* de *conj* es *trivial* y r es un CPE. En caso contrario el *tipo* de c_i será *no trivial*.
   (c) Todos los nuevos conjuntos de activación c_i que no hallan sido previamenente expandidos, son agregados a C.
   (d) El conjunto *conj* se agrega a Act-sets($\mathcal{A}^c$).
   Si el tipo de *conj* es *no trivial* entonces *conj* se agrega a NTAct-sets($\mathcal{A}^c$).
4. RETORNAR NTAct-sets($\mathcal{A}^c$) y Act-sets($\mathcal{A}^c$).

Figura 4.1: Algoritmo para construir los conjuntos de activación

específico que $\mathcal{B}$ si y sólo si,
(1) para todo conjunto $U \in$ NTAct-sets($\mathcal{A}^c$), se cumple que $U \cup \mathcal{K}_G \cup \mathcal{B}^c \mid\sim h_2$, y
(2) existe un conjunto $U' \in$ NTAct-sets($\mathcal{B}^c$), se cumple que $U' \cup \mathcal{A}^c \not\mid\sim h_1$.
En el caso que alguno de los argumentos sea vacío se procederá como indica la observación 4.8.1 □

La definición 4.8.6 será equivalente a la 4.8.1 sólo cuando halla un único completamiento para cada argumento (ver observación 4.8.2). Sin embargo se puede construir una definición equivalente de la siguiente forma: supóngase que en lugar de un único $\mathcal{A}^c$, existe un conjunto $\{\mathcal{A}^c{}_i\}$ de argumentos completados para $\mathcal{A}$. Se puede definir los conjuntos Act-sets($\mathcal{A}$)=$\bigcup_{i=1}^{n}$ Act-sets($\mathcal{A}^c{}_i$) y NTAct-sets($\mathcal{A}$)=$\bigcup_{i=1}^{n}$ NTAct-sets($\mathcal{A}^c{}_i$), y entonces reformular la nueva definición de especificidad, reemplazando NTAct-sets($\mathcal{A}^c$) por NTAct-sets($\mathcal{A}$), y NTAct-sets($\mathcal{B}^c$) por NTAct-sets($\mathcal{B}$).

4.8.1 Comparación de argumentos que continen presuposiciones

En la programación en lógica rebatible, un *hecho* es una CPE denotada "h ← true", mientras que una *presuposición* es una CPR con antecedente vacío, la cuál se denota "p $-\!\prec$ true". Los hechos representan información altamente segura ya que son cláusulas no rebatibles incondicionales, mientras que las presuposiciones representan información tentativa, introducida por el programador, para ser usada por el PLR ante la ausencia de mejor información. La definición de especificidad estaba desarrollada para argumentos donde no había presuposiciones. En esta sección se extenderá la nueva defínción de especificidad para que permita comparar correctamente argumentos con presuposiciones.

Durante el desarrollo de la extensión de la especificidad, se analizaron diferentes alternativas, las cuales fueron confrontadas con un conjunto de ejemplos para ver sus resultados. La figura 4.2 muestra este conjunto de ejemplos, donde se tienen diferentes casos de pares de argumentos para ser comparados utilizando el criterio de especificidad. Cada caso intenta capturar una situación diferente en la utilización de presuposiciones. En todos los casos se asume que están presentes los siguientes hechos: "h ← true" y "a ← true".

La siguiente definición será utilizada en las alternativas que se analizarán a continuación.

Definición 4.8.7 : *Argumento basado en hechos, o basado en presuposiciones*
Un argumento completado estará *basado en hechos*, si fue utilizado al menos un hecho en su derivación rebatible, o posee una submeta "not p". De lo contrario, si se utilizaron sólo presuposiciones, se dirá que no está basado en hechos, o que está *basado en presuposiciones.* □

Lo que sigue son las diferentes alternativas consideradas para extender el criterio de especificidad, a fin de que considere argumentos que tienen presuposiciones. En la figura 4.3 se muestra el resultado esperado para cada uno de los ejemplos de la figura 4.2, y se muestra además el resultado que se obtiene la comparar los argumentos $\mathcal{A}$ y $\mathcal{B}$ de todos los casos, con cada una de las alternativas que se analizaron. Por ejemplo, si en la columna (b) fila (1) figura $\succ$, significa que siguiendo la alternativa 1: $\mathcal{A}_b \succ \mathcal{B}_b$.

Alternativa 1: *Utilizar la definición de especificidad tal cuál está planteada.* En este caso, como las presuposiciones forman parte del argumento completado, estas se activarán con cualquier conjunto de activación. Como puede verse en los resultados expuestos en

$\mathcal{A}_a = \{$ $c_a \prec h \}$ $\mathcal{B}_a = \{$ $\sim c_a \prec true \}$	$\mathcal{A}_b = \{$ $c_b \prec a,h,s$ $s \prec true \}$ $\mathcal{B}_b = \{$ $\sim c_b \prec a\}$	$\mathcal{A}_c = \{$ $c_c \prec s,h$ $s \prec true \}$ $\mathcal{B}_c = \{$ $\sim c_c \prec h \}$
$\mathcal{A}_d = \{$ $c_d \prec s,h$ $s \prec true \}$ $\mathcal{B}_d = \{$ $\sim c_d \prec s$ $s \prec true \}$	$\mathcal{A}_e = \{$ $c_e \prec s,r$ $s \prec true$ $r \prec true \}$ $\mathcal{B}_e = \{$ $\sim c_e \prec r$ $r \prec true \}$	$\mathcal{A}_f = \{$ $c_f \prec h \}$ $\mathcal{B}_f = \{$ $\sim c_f \prec s,r$ $s \prec true$ $r \prec true \}$
$\mathcal{A}_g = \{$ $c_g \prec true \}$ $\mathcal{B}_g = \{$ $\sim c_g \prec s$ $s \prec true \}$	$\mathcal{A}_h = \{$ $c_h \prec s$ $s \prec true \}$ $\mathcal{B}_h = \{$ $\sim c_h \prec q$ $q \prec s$ $s \prec true \}$	$\mathcal{A}_i = \{$ $c_i \prec p$ $p \prec h \}$ $\mathcal{B}_i = \{$ $\sim c_i \prec true \}$
$\mathcal{A}_j = \{$ $c_j \prec h\}$ $\mathcal{B}_j = \{$ $\sim c_j \prec s$ $s \prec true \}$	$\mathcal{A}_k = \{$ $c_k \prec p$ $p \prec h \}$ $\mathcal{B}_k = \{$ $\sim c_k \prec s$ $s \prec true\}$	$\mathcal{A}_l = \{$ $c_l \prec true \}$ $\mathcal{B}_l = \{$ $\sim c_l \prec true \}$
$\mathcal{A}_m = \{$ $c_m \prec h\}$ $\mathcal{B}_m = \{$ $\sim c_m \prec p,q$ $p \prec s \;\; q \prec r$ $s \prec true$ $r \prec true \}$	$\mathcal{A}_n = \{$ $c_n \prec h\}$ $\mathcal{B}_n = \{$ $\sim c_n \prec p$ $p \prec s$ $s \prec true \}$	$\mathcal{A}_o = \{$ $c_o \prec true \}$ $\mathcal{A}_o = \{$ $\sim c_o \prec s,r$ $s \prec true$ $r \prec true \}$

Figura 4.2: Ejemplos de argumentos con presuposiciones

la figura 4.3, en los casos e, g, y h, donde los dos argumentos están basados sólo en presuposiciones, la relación de especificidad no se comporta de acuerdo a lo esperado. Lo mismo pasa en casos (como el c) donde la diferencia entre dos argumentos es un conjunto de presuposiciones. Este comportamiento es debido a que las presuposiciones están incluídas en el argumento completado, y los conjuntos de activación no tienen el efecto deseado. Obsérvese que con los hechos no pasa lo mismo porque no forman parte del argumento completado, por lo tanto, la solución será excluir las presuposiciones del argumento completado, al momento de realizar la comparación por especificidad.

Alternativa 2: *No considerar las presuposiciones del argumento completado y usar la*

Resultado	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
esperado	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$
(1)	$\succ$	$\succ$	$\not\approx$	$\succ$	$\not\approx$	$\succ$	$\not\approx$	$\not\approx$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\not\approx$
(2)	$\prec$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\not\approx$	$\succ$	$\prec$	$\not\approx$	$\not\approx$	$\not\approx$	$\not\approx$	$\not\approx$	$\not\approx$
(3)	$\prec$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\prec$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$
(4)	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$

Figura 4.3: Resultados de aplicar las diferentes alternativas

definición de especificidad tal cuál está planteada. En esta propuesta, las presuposiciones no pueden activarse con cualquier conjunto de activación, ya que no están en el argumento completado. De esta manera, los casos e, g y h dan el resultado esperado. Pero lamentablemente, en casos como f, j, k, m, n y o, donde antes se lograba el resultado esperado, ahora no es así, debido a que desde el punto de vista de los conjuntos de activación las presuposiciones se convierten en hechos. Sin embargo, es posible remediar fácilmente esta situación, si en esos casos se prefieren los argumentos que están basados en hechos sobre los que están basados en presuposiciones.

Alternativa 3: *Utilizar la alternativa 2, y en los casos que sean incomparables preferir aquellos argumentos que estén basados en hechos por sobre aquellos que estén basados en presuposiciones* Esto mejora muchos de los casos de la alternativa 2, salvo los casos como a o i donde uno de los argumentos fue construído sólo con presuposiciones. En estos casos el criterio establecido en la observación 4.8.1 indicaba preferir el argumento vacío, pero en ese punto todavía no habían sido consideradas las presuposiciones.

Alternativa 4: *Utilizar la alternativa 3, pero con el siguiente criterio para los argumentos vacíos: (a) los argumentos vacíos basados en hechos, son preferidos a los no vacíos, y a los que sólo contienen presuposiciones; y (b) los argumentos que sólo contienen presuposiciones serán preferidos a los que contienen alguna* CPR *que no es una presuposición.* Esta alternativa corresponde a un refinamiento de las alternativas 2 y 3, y da los resultados esperados. Por lo tanto será la elegida para comparar argumentos en los programas lógicos rebatibles.

Definición 4.8.8 : *Extensión de la relación de especificidad.*
Se utilizará la definición de especificidad tal cuál está planteada, pero las presuposiciones (al igual que los hechos) no formarán parte del argumento completado al momento de cal-

cular los conjuntos de activación. En los casos en que los argumentos sean incomparables se preferirá aquellos que estén basados en hechos por sobre aquellos que estén basados en presuposiciones. Por último, los argumentos vacíos basados en hechos, son preferidos a los no vacíos, y a los que sólo contienen presuposiciones; además, los argumentos que sólo contienen presuposiciones serán preferidos a los que contienen alguna CPR que no es una presuposición.

los argumentos vacíos basados en hechos, serán preferidos a los no vacíos, y a los vacíos basados en presuposiciones; y los argumentos vacíos basados en presuposiciones serán preferidos sólo a los no vacíos basados en presuposiciones. □

4.8.2 La especificidad y la negación por falla

Al igual que con las presuposiciones, la relación de especificidad no fue definida para manejar situaciones donde esté presente la negación por falla. Al incluir el operador "`not`" en el lenguaje, hay que definir un criterio para que puedan compararse por especificidad argumentos que contengan cláusulas con "`not`". En esta sección se analizarán diferentes alternativas y se desarrollará una extensión de la relación de especificidad. Considérese el siguiente ejemplo:

Ejemplo 4.8.4 : Dado el PLR:

`peligroso(X)` $\prec$ `not perro-bueno(X)`

$\sim$`peligroso(X)` $\prec$ `not perro-bueno(X), not mordió-alguien(X)`

$\sim$`peligroso(X)` $\prec$ $\sim$`perro-callejero(X)`

$\sim$`perro-callejero(negro)` $\leftarrow$ `true`

se pueden construir los argumentos

$\mathcal{A}$={ `peligroso(negro)` $\prec$ `not perro-bueno(negro)` }

$\mathcal{B}$={$\sim$`peligroso(negro)` $\prec$ `not perro-bueno(negro), not mordio-alguien(negro)`}

$\mathcal{C}$={ $\sim$`peligroso(negro)` $\prec$ $\sim$`perro-callejero(negro)` }

□

a)	$\mathcal{A}_a$ =\{	$\mathtt{c}_a \prec \mathtt{a}$ \}	b)	$\mathcal{A}_b$ =\{	$\mathtt{c}_b \prec \mathtt{a, not\ p}$ \}
	$\mathcal{B}_a$ =\{	$\sim\mathtt{c}_a \prec \mathtt{not\ p}$ \}		$\mathcal{B}_b$ =\{	$\sim\mathtt{c}_b \prec \mathtt{not\ p}$ \}
		$\mathcal{A}_a \not\approx \mathcal{B}_a$			$\mathcal{A}_b \succ \mathcal{B}_b$
c)	$\mathcal{A}_c$ =\{	$\mathtt{c}_c \prec \mathtt{a, not\ p}$ \}	d)	$\mathcal{A}_d$ =\{	$\mathtt{c}_d \prec \mathtt{not\ p, not\ q}$ \}
	$\mathcal{B}_c$ =\{	$\sim\mathtt{c}_c \prec \mathtt{a}$ \}		$\mathcal{B}_d$ =\{	$\sim\mathtt{c}_d \prec \mathtt{not\ q}$ \}
		$\mathcal{A}_c \succ \mathcal{B}_c$			$\mathcal{A}_d \succ \mathcal{B}_d$
e)	$\mathcal{A}_e$ =\{	$\mathtt{c}_e \prec \mathtt{not\ p, not\ q}$\}	f)	$\mathcal{A}_f$ =\{	$\mathtt{c}_f \prec \mathtt{not\ p}$ \}
	$\mathcal{B}_e$ =\{	$\sim\mathtt{c}_e \prec \mathtt{a}$ \}		$\mathcal{B}_f$ =\{	$\sim\mathtt{c}_f \prec \mathtt{true}$\}
		$\mathcal{A}_e \not\approx \mathcal{B}_e$			$\mathcal{A}_f \succ \mathcal{B}_f$
g)	$\mathcal{A}_g$ =\{	$\mathtt{c}_g \prec \mathtt{a,h, not\ p}$ \}	h)	$\mathcal{A}_h$ =\{	$\mathtt{c}_h \prec \sim\mathtt{p}$ \}
	$\mathcal{B}_g$ =\{	$\sim\mathtt{c}_g \prec \mathtt{a}$ \}		$\mathcal{B}_h$ =\{	$\sim\mathtt{c}_h \prec \mathtt{not\ p}$ \}
		$\mathcal{A}_g \succ \mathcal{B}_g$			$\mathcal{A}_h \not\approx \mathcal{B}_h$

Figura 4.4: Ejemplos de argumentos con el operador "not"

.

En el ejemplo anterior, uno esperaría que el argumento $\mathcal{B}$ sea más específico que $\mathcal{A}$ ($\mathcal{B} \succ \mathcal{A}$), ya que $\mathcal{B}$ está utilizando mayor información para llegar a la conclusión. En el caso de $\mathcal{C}$ y $\mathcal{A}$ ambos utilizan diferente información, por lo que se esperaría que ninguno fuera más específico que el otro ($\mathcal{A} \not\approx \mathcal{C}$). Como se verá a continuación existen diferentes formas de considerar submetas de la forma "not p" al momento de comparar por especificidad.

Alternativa 1: El criterio más simple y directo, sería ignorar todas las submetas "not p", esto es, no incluirlas en los conjuntos de activación. Siguiendo esta propuesta, los argumentos $\mathcal{A}$ y $\mathcal{B}$ del ejemplo 4.8.4 no tendrían conjuntos de activación no triviales, y por lo tanto resultarían incomparables ($\mathcal{A} \not\approx \mathcal{B}$). El argumento $\mathcal{C}$ tendría un solo conjunto de activación: { ∼pero-callejero(negro) }, y por lo tanto $\mathcal{C}$ sería estrictamente más específico que $\mathcal{A}$. Como puede verse, los resultados de utilizar esta propuesta no son los esperados.

Alternativa 2: Como una submeta "not p" tiene éxito cuando no existe un justificación para p, las apariciones de una submeta "not p" pueden considerarse como información muy débil. Otro criterio sería entonces preferir aquellos argumentos que tengan menos submetas con "not". Sin embargo, utilizando este criterio, $\mathcal{C} \succ \mathcal{A}$ y $\mathcal{B} \succ \mathcal{A}$, lo cuál tampoco es el resultado esperado.

Alternativa 3: De lo analizado hasta el momento puede verse, que resulta importante que las submetas con el operador "`not`" sean consideradas de igual manera que un literal probado rebatiblemente. Por lo tanto, se debe considerar a las submetas "`not` p" de un argumento como un literal más dentro del conjunto de activación. Para implementar este criterio, únicamente hay que modificar la definición 4.8.3.

Definición 4.8.9 : (4.8.3 modificada) *Conjunto de literales básicos de* $\mathcal{A}^c$.

Sea $\mathcal{A}^c$ un argumento completado, el conjunto de literales básicos de $\mathcal{A}^c$ denotado $Lit(\mathcal{A}^c)$, es el conjunto de literales que aparecen en los antecedentes y consecuentes de toda cláusula de $\mathcal{A}^c$, más las submetas de la forma "`not` p" (p un literal instanciado), que aparecen en las cláusulas de $\mathcal{A}^c$. □

De esta forma, en el ejemplo 4.8.4, el conjunto de activación no trivial de $\mathcal{A}$ es { `not perro-bueno(negro)` }, el de $\mathcal{B}$ { `not perro-bueno(negro), not mordio-alguien(negro)` }, y el de $\mathcal{C}$ es { ∼`perro-callejero(negro)` }. Utilizando ahora la definición de especificidad con estos conjuntos de activación, $\mathcal{B} \succ \mathcal{A}$, y $\mathcal{C} \not\succ \mathcal{A}$.

Esta alternativa es la que mejor responde a las espectativas propuestas, ya que prefiere aquellos argumentos que están basados en información más específica. Por lo tanto este será el criterio elegido para extender la definición de especificidad. En la figura 4.4 se muestran diferentes ejemplos, con el resultado de aplicar el criterio elegido. En todos los casos se asume que están presente los siguientes hechos: "`h` ← `true`" y "`a` ← `true`".

Observación 4.8.3 : Dentro de un argumento, los literales que tienen negación por falla sólo están como hojas del árbol de derivación rebatible. Esto permite eliminar del análisis la comparación por directitud [Loui87], ya que no es posible una cadena de reglas con "`not`".

4.9 Conclusiones

En este capítulo se definió el lenguaje de la programación en lógica rebatible, como una extensión de la programación en lógica convencional. Este nuevo lenguaje captura aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Permite la representación de información incompleta y potencialmente

inconsistente, y utiliza los conceptos de la argumentación rebatible a fin de poder decidir entre metas contradictorias. Como se vió a lo largo del capítulo, este nuevo lenguaje incorpora ciertas características de las extensiones de la programación en lógica presentadas en el capítulo 2, y soluciona los problemas que fueron mencionados.

La utilización de dos tipos de cláusulas y dos tipos de negación, le incorpora al lenguaje una mayor expresividad para representar conocimiento. Se mostró en este capítulo que ambos tipos de negación son necesarios, y se analizó en forma detallada sus diferencias y la forma en que pueden interactuar. Se extendió el criterio de especificidad para tratar con argumentos que incluyen negación por falla y presuposiciones, y se desarrolló un algoritmo para el cálculo de la especificidad que sólo utiliza la información de los argumentos involucrados.

Por lo tanto el primer objetivo de esta tesis está cumplido. Resta ahora construir una máquina abstracta para la implementación de este nuevo lenguaje. Para ello, en el capítulo siguiente se desarrollarán algoritmos para obtener argumentos y derrotadores, y luego se definirá la nueva máquina abstracta.

Capítulo 5

Algoritmos para construcción de argumentos y derrotadores

En el capítulo anterior se definió el lenguaje de la programación en lógica rebatible, que utiliza los conceptos de la argumentación rebatible para definir el criterio de inferencia dentro del lenguaje. El objetivo de lo que resta de esta tesis es construir una máquina abstracta para la implementación de este nuevo lenguaje. En el capítulo siguiente se describirá la máquina abstracta de Warren (WAM) la cuál se utilizará como punto de partida para construir una nueva máquina abstracta para la programación en lógica rebatible.

El objetivo de este capítulo es mostrar los algoritmos que fueron diseñados para implementar cada uno de los conceptos involucrados en la obtención de una justificación. La implementación de los algoritmos se realizó en su totalidad utilizando encadenamiento hacia atrás (backward chaining) de las cláusulas de programa. Se priorizó en todos los casos, la obtención de un algoritmo eficiente, que sólo considere la información que está presente en los argumentos involucrados. De esta forma, el costo de obtener una justificación estará relacionado únicamente a los argumentos que forman el árbol de dialéctica, y será independiente del tamaño del programa. Estos algoritmos serán utilizados para la implementación de las instrucciones de la nueva máquina abstracta.

En la sección 5.1 se muestra que la consistencia de un argumento puede verificarse durante la construcción del mismo, utilizando encadenamiento hacia atrás, y sin la necesidad de considerar reglas contrapositivas. Se presenta además un algoritmo para la verificación de la minimalidad, que garantiza la eficiencia en la construcción del argumento.

La implementación de la obtención de contraargumentos se explicará en la sección 5.2, y también se realizará utilizando únicamente la información del argumento completado. A partir del argumento completado, se obtendrá el conjunto de los posibles puntos de contraargumentación, y todos los contraargumentos puedrán obtenerse por encadenamiento hacia atrás, a partir de estos puntos.

Para obtener una justificación, se debe construir un árbol de dialéctica aceptable, y por lo tanto, se deberá controlar que toda línea de argumentación carezca de derrotadores recíprocos, argumentación circular, y argumentos que no sean concordantes. En el capítulo 7 se mostrará como serán construídos los árboles de dialéctica, utilizando los algoritmos que se presentan aquí. La verificación de ciclos y concordancia se realizará simultaneamente a la construcción del árbol de dialéctica, lo cual permitirá podar aquellas ramas que no son aceptables.

5.1 Construcción de argumentos

Como se vió en el capítulo anterior, la noción de argumento es un concepto central dentro de los PLR. En esta sección se describirá un conjunto de algoritmos que permite la construcción de un argumento partiendo de la consulta efectuada, y utilizando sólo encadenamiento hacia atrás de las cláusulas de programa.

Todo programa lógico rebatible (PLR), está formado por un conjunto finito $\mathcal{S}$ de cláusulas de programa extendido (CPE), y un conjunto finito $\mathcal{D}$ de cláusulas de programa rebatible (CPR). El conjunto $\mathcal{S}$ es siempre consistente. Un argumento es un subconjunto de un PLR, con la siguiente definición:

Definición 5.1.1 : *Argumento para un meta* `m`
Dado un PLR, formado por el conjunto $\mathcal{S}$ de CPE, y el conjunto $\mathcal{D}$ de CPR, un *argumento* $\mathcal{A}$ para una meta `m`, es un subconjunto de CPR instanciadas de $\mathcal{D}$, tal que:

1. Existe una derivación rebatible de `m` a partir de SS $\cup$ $\mathcal{A}$ (*i.e.*, SS $\cup$ $\mathcal{A}|\sim$ `m`),
2. SS $\cup$ $\mathcal{A}$ es consistente, y
3. $\mathcal{A}$ es el menor subconjunto (con respecto a la inclusión de conjuntos) que cumple las dos condiciones anteriores.

Si $\mathcal{A}$ es un argumento para $\mathtt{m}$, también se dirá que $\langle\ \mathcal{A}$,$\mathtt{m}\ \rangle$ es una *estructura de argumento.* □

Por lo tanto, la implementación de la construcción de un argumento debe considerar lo siguiente: (1) la obtención de una derivación rebatible para la meta $\mathtt{m}$, (2) la verificación de la consistencia de las CPR utilizadas, y (3) la minimalidad. En lo que sigue se desarrollará cada uno de estos temas,

La implementación de la noción de derivación rebatible se realizará siguiendo la definición 4.2.1 introducida en el capítulo anterior. El proceso comenzará a partir de la consulta efectuada, y se realizará una búsqueda en profundidad (depth-first) utilizando el mecanismo de encadenamiento hacia atrás de las cláusulas del programa. La verificación de la consistencia de la derivación se realizará en forma conjunta con la construcción de la derivación, y será explicada en detalle a continuación.

5.1.1 Consistencia de un argumento

La definición de argumento establece que SS $\cup\ \mathcal{A}$ debe ser consistente, donde $\mathcal{A}$ es el conjunto de CPR que se quiere tomar como argumento. En esta sección se analizará cómo implementar la verificación de consistencia del conjunto SS $\cup\ \mathcal{A}$. Recuérdese que el conjunto de CPE $\mathcal{S}$ es siempre consistente, y que la consistencia se definía de la siguiente forma:

Definición 5.1.2 : *Consistencia de SS* $\cup\ \mathcal{A}$
Sea $\mathcal{S}$ un conjunto de CPE (consistente), y $\mathcal{A}$ un conjunto de CPR instanciadas. El conjunto SS $\cup\ \mathcal{A}$ será *consistente* cuando no sea posible obtener una derivación rebatible de un par de literales complementarios a partir de SS $\cup\ \mathcal{A}$. Analogamente, SS $\cup\ \mathcal{A}$ será *inconsistente* cuando pueda obtenerse una derivación rebatible de un par de literales complementarios a partir de SS $\cup\ \mathcal{A}$. □

En algunas oportunidades se deberá verificar la consistencia de $\mathcal{S}$ unido a un conjunto de literales $L = \{l_1, \ldots, l_n\}$. Para ello, se asumirá a cada literal l_i como un hecho $l_i \leftarrow \mathtt{true}$, y de esa forma se podrá utilizar la definición anterior para verificar la consistencia de $\mathcal{S} \cup L$. Por ejemplo, aunque el conjunto $\mathcal{S}$ sea consistente, el conjunto $\mathcal{S} \cup \{h, \sim h\}$ no lo es.

Definición 5.1.3 : Se llamará $Co(\mathcal{A})$ al conjunto de literales instanciados que son consecuentes de las CPRs de un conjunto de cláusulas $\mathcal{A}$. □

La siguiente proposición ayudará a determinar cuando un conjunto de CPRs puede asumirse consistentemente con $\mathcal{S}$.

Proposición 5.1.1 : *Sea $\mathcal{S}$ el conjunto de* CPEs *de un* PLR*, y $\mathcal{A}$ un argumento para una meta m. El conjunto SS $\cup$ $\mathcal{A}$ será consistente si y sólo si el conjunto $\mathcal{S} \cup Co(\mathcal{A})$ es consistente. .*

Demostración: Si existe una derivación rebatible para la meta m a partir del conjunto SS $\cup$ $\mathcal{A}$, entonces existe un derivación rebatible para todos los consecuentes de toda CPR de $\mathcal{A}$, a partir de SS $\cup$ $\mathcal{A}$. Por lo tanto, todo literal de $Co(\mathcal{A})$ se puede derivar a partir de SS $\cup$ $\mathcal{A}$. Si el conjunto SS $\cup$ $\mathcal{A}$ es consistente, entonces no permite probar rebatiblemente un par de literales complementarios. Supóngase que el conjunto $\mathcal{S} \cup Co(\mathcal{A})$ fuera inconsistente, entonces permitiría probar un par de literales complementarios, pero si esto fuera así, como todo literal de $Co(\mathcal{A})$ se prueba a partir de SS $\cup$ $\mathcal{A}$, entonces SS $\cup$ $\mathcal{A}$ permitiría derivar un par de literales complementarios. Por lo tanto si el conjunto SS $\cup$ $\mathcal{A}$ es consistente, $\mathcal{S} \cup Co(\mathcal{A})$ también lo es. La recíproca es directa. □

Observación 5.1.1 : Verificar que $\mathcal{A}$ es consistente con $\mathcal{S}$ podría realizarse de la siguiente forma: obtener el conjunto $Co(\mathcal{A})$ de $\mathcal{A}$, y luego verificar que no exista ningún literal p tal que a partir de $\mathcal{S} \cup Co(\mathcal{A})$ pueda obtenerse una derivación de p y $\overline{p}$. Pero esto presenta dos problemas:

1. Si se descubriera que $\mathcal{S} \cup Co(\mathcal{A})$ es inconsistente, entonces se debería volver a calcular todo el argumento, lo cuál implica haber malgastado recursos en la construcción de un argumento que resulta inconsistente.

2. El método de computación que se utiliza para obtener derivaciones a partir de un PLR es de encadenamiento hacia atrás. Por lo tanto mostrar que no es posible obtener una derivación para un par de literales arbitrarios p y $\overline{p}$ a partir de $\mathcal{S} \cup Co(\mathcal{A})$, implica un proceso con un alto costo computacional: o bién se debería utilizar un procedimiento de encademiento hacia adelante, o bién se debería intentar probar p y $\overline{p}$ para todo literal p de $\mathcal{S} \cup Co(\mathcal{A})$.

El objetivo de esta sección será encontrar un método para verificar la consistencia de un argumento que se lleve a cabo simultáneamente con la construcción del argumento, y que utilice únicamente encadenamiento hacia atrás. Una solución al primer problema de

la observación 5.1.1 es prohibir el uso de una CPR que luego produzca una inconsistencia. Para esto, cada vez que se utiliza una CPR "$c \prec L$" en la derivación rebatible de una meta h, se debe verificar que SS$\cup\{c\}$ sea consistente, es decir, si es posible asumir c consistentemente con $\mathcal{S}$.

Definición 5.1.4 : CPR *aceptada*
Dado el conjunto $\mathcal{S}$ (que es consistente), una CPR instanciada (grounded) "$c \prec L$" se dirá *aceptada* y por lo tanto podrá ser utilizada en la construcción de un argumento $\mathcal{A}$, sólo si $\mathcal{S} \cup Co(\mathcal{A}') \cup \{c\}$ es consistente. Donde $\mathcal{A}'$ es el conjunto de CPR aceptadas previamente en la construcción de un argumento $\mathcal{A}$, y por lo tanto $\mathcal{S} \cup Co(\mathcal{A}')$ es consistente. □

El siguiente ejemplo muestra porqué, en la definición 5.1.4, c debe ser un literal instanciado.

Ejemplo 5.1.1 : En el siguiente PLR, al tratar de resolver la consulta "`vuela(X)`", si se intenta ver si la CPR sin instanciar "`vuela(X)` $\prec$ `ave(X)`" puede ser aceptada, se encuentra que $\mathcal{S} \cup \{$ `vuela(X)` $\}$ es inconsistente al unificar `X` con `charo`.

$$\sim\texttt{vuela(X)} \leftarrow \texttt{ñandú(X)}$$
$$\texttt{vuela(X)} \prec \texttt{ave(X)}$$
$$\texttt{ñandú(charo)} \leftarrow \texttt{true}$$
$$\texttt{ave(petete)} \leftarrow \texttt{true}$$

Sin embargo, puede verse claramente que existe un argumento para `vuela(petete)`, ya que $\mathcal{S} \cup \{$ `vuela(petete)` $\}$ es consistente. Esto muestra que la consistencia de una submeta debe hacerse sobre un literal instanciado. □

Definición 5.1.5 : *Hecho temporario*
Si $\mathcal{A}'$ es el conjunto de CPRs aceptadas en la construcción de un argumento $\mathcal{A}$, entonces, $\mathcal{S} \cup Co(\mathcal{A}')$ es consistente. Cada elemento h de $Co(\mathcal{A}')$ se llamará *hecho temporario*, ya que, a los efectos de la verificación de la consistencia durante la construcción del argumento, una vez que h fue aceptado, se comportará igual que un hecho del PLR. Será temporario porque sólo existirá como un hecho durante la construcción del argumento. □

La definición 5.1.4 soluciona uno de los problemas planteados en la observación 5.1.1, ya que ahora es posible desechar las CPR que producen inconsistencia durante la construcción del argumento, y de esta forma evitar realizar pasos de más. Si una CPR produce una inconsistencia, y no puede ser utilizada, entonces el mecanismo de backtracking de los PLR buscará otra forma de construir el argumento.

Ejemplo 5.1.2 : Dado el PLR formado por:

$\sim$e $\leftarrow$ c.	a $\prec$ b,c.
j $\leftarrow$ true.	b $\prec$ e.
d $\leftarrow$ true.	b $\prec$ p.
p $\leftarrow$ true.	e $\prec$ j.
	c $\prec$ d.

Al intentar construir un argumento para la meta a, se puede obtener la derivación rebatible: { a $\prec$ b,c ; b $\prec$ e ; e $\prec$ j ; c $\prec$ d }. Sin embargo, la regla "c $\prec$ d" no puede utilizarse como parte del argumento, ya que se produce una inconsistencia. Como hay otra forma de probar rebatiblemente a b, entonces el mecanismo de backtracking generará la derivación { a $\prec$ b,c ; b $\prec$ p ; c $\prec$ d }, que sí es consistente. □

Resta entonces solucionar el segundo problema de la observación 5.1.1. Se necesita que la verificación de consistencia de un literal pueda hacerse por encadenamiento hacia atrás partiendo de una meta conocida. El lema 5.1.1 permitirá que esto sea posible.

Cuando se quiere verificar que el conjunto SS$\cup\{h\}$ es consistente se debe verificar que SS$\cup\{h\}$ no derive un par de literales complementarios. Si el par de literales fuera h y $\overline{h}$, entonces sería sencillo de implementar, pero esto no siempre es así. Por ejemplo, si el conjunto $\mathcal{S}$ es { $\sim$p $\leftarrow$ q ; p $\leftarrow$ h ; q $\leftarrow$ true }, SS$\cup\{h\}$ es inconsiste, y el par de literales es p y $\sim$p. Si uno pudiera asumir la contrapositiva de las CPE, en el ejemplo anterior se podría haber generado la siguiente derivación para $\sim$h: { $\sim$h $\leftarrow$ $\sim$p ; $\sim$p $\leftarrow$ q ; q $\leftarrow$ true }.

Como se indicó en el capítulo anterior, las CPE no son contrapositivas. Esto es, si se tiene la CPE "a $\leftarrow$ b" no se puede asumir que exista la cláusula "$\sim$b $\leftarrow$ $\sim$a". Esto presenta una limitación para la implementación de la verificación de consistencia. Sin embargo, aunque la contrapositiva no pueda utilizarse en las derivaciones rebatibles, se usarán CPE invertidas sólo para verificar la existencia del par de literales complementarios, y no para producir una derivación. La siguiente definición introduce el concepto de CPE *invertida*, el cuál será utilizado en la demostración del lema 5.1.1.

Definición 5.1.6 : CPE *invertidas*
Invertir una CPE "$p \leftarrow a_1, \ldots, a_i, \ldots, a_n$" significa reemplazar el consecuente con el complemento de un antecedente, y dicho antecedente con el complemento del consecuente,

obteniendo por ejemplo la siguiente CPE invertida: "$\overline{a_i}$ @– $a_1, \ldots, \overline{p}, \ldots, a_n$". Se utilizará el símbolo " @– " para distinguir una CPE invertida de una CPE común. Esto también se conoce como estrategia de Loveland [Poole85b, Love78]. □

Observación importante: Las CPE invertidas no son cláusulas de programa, y por lo tanto no pueden formar parte de ningún argumento. Estas cláusulas serán utilizadas solamente para acelarar la búsqueda de un par de literales complementarios durante la verificación de la consistencia. Aunque el proceso de invertir una CPE se asemeje a la contraposición de la implicación no debe pensarse de ningún modo que las CPE sean contrapositivas. Las siguientes restricciones garantizan que su utilización sea la correcta.

Definición 5.1.7 : *Restricciones de uso de una* CPE *invertida*

1. Las CPE invertidas sólo pueden utilizarse al principio de una derivación, ya que no formarán parte de ella.
2. Una vez que una cláusula de programa es usada, en la derivación, ya no se podrán utilizar más CPE invertidas.
3. Si la CPE "$p \leftarrow a_1, \ldots, a_i, \ldots, a_n$" es invertida para ser usada como "$\overline{a_i}$ @– $a_1, \ldots, \overline{p}, \ldots, a_n$". entonces los literales $a_1, \ldots, a_n$ deberán derivarse únicamente utilizando CPE o hechos temporarios.

□

El siguiente lema muestra que la verificación de la consistencia puede hacerse con encadenamiento hacia atrás partiendo de una meta conocida. En su demostración se verán claramente las razones de las restricciones sobre las CPE invertidas.

Lema 5.1.1 : *Dado un conjunto consistente de* CPE $\mathcal{S}$ *y un literal* h. *El conjunto* $SS \cup \{h\}$ *es inconsistente si y solo si existe una derivación de* $\overline{h}$ *a partir de* $SS \cup \{h\}$, *utilizando (si fuera necesario, y con las restricciones dadas) alguna* CPE *invertida.*

Demostración:
($\Rightarrow$)
Por hipótesis el conjunto SS$\cup\{h\}$ es inconsistente, entonces existe un literal p tal que tanto p como $\overline{p}$ pueden derivarse a partir de SS$\cup\{h\}$. A continuación se mostrará que si lo anterior ocurre, entonces $\overline{h}$ se deriva a partir de SS$\cup\{h\}$.

Caso particular: si $p = h$ entonces trivialmente se demuestra que existe una derivación para $\overline{h}$ a partir de SS$\cup\{h\}$.
Caso general: supongamos que $p \neq h$, como $\mathcal{S}$ es consistente entonces es imposible que p y $\overline{p}$ se deriven de $\mathcal{S}$, y por lo tanto tiene que darse uno de estos tres casos:
(a) de $\mathcal{S}$ no se deriva $\overline{p}$ y no se deriva p.
(b) de $\mathcal{S}$ se deriva $\overline{p}$ y no se deriva p.
(c) de $\mathcal{S}$ se deriva p y no se deriva $\overline{p}$.

Caso (a): de $\mathcal{S}$ no se deriva $\overline{p}$ y no se deriva p.
Como de $\mathcal{S}$ no se deriva p, pero de SS$\cup\{h\}$ se deriva p, entonces h es imprescindible para derivar p. Puede ocurrir en particular que simplemente halla una regla de la forma "$p \leftarrow h, D$" donde D es un conjunto de literales que pueden derivarse a partir de SS$\cup\{h\}$. Invirtiendo la regla anterior queda "$\overline{h} \multimap \overline{p}, D$". Como por hipótesis, $\overline{p}$ se deriva a partir de SS$\cup\{h\}$, entonces utilizando la regla invertida, puede derivarse $\overline{h}$ a partir de SS$\cup\{h\}$.

Pero en el caso general no existirá una sola regla, sino que existirá una secuencia de reglas como la siguiente:

$$
\begin{aligned}
&c_1 \leftarrow h, B_0\\
&c_2 \leftarrow c_1, B_1\\
&\ldots\\
&c_n \leftarrow c_{n-1}, B_{n-1}\\
&p \leftarrow c_n, B_n
\end{aligned}
$$

(donde los B_i son conjuntos de literales que se derivan a partir de SS$\cup\{h\}$).

Invirtiendo apropiadamente las CPE queda la secuencia:

$$
\begin{aligned}
&\overline{h} \multimap \overline{c_1}, B_0\\
&\overline{c_1} \multimap \overline{c_2}, B_1\\
&\overline{c_{n-1}} \multimap \overline{c_n}, B_{n-1}\\
&\overline{c_n} \multimap \overline{p}, B_1
\end{aligned}
$$

como de SS$\cup\{h\}$ se deriva $\overline{p}$, entonces existe una derivación de $\overline{h}$ a partir de SS$\cup\{h\}$ como resultado de invertir algunas reglas en $\mathcal{S}$.

Caso (b): de $\mathcal{S}$ se deriva $\overline{p}$ y no se deriva p.
(La demostración es igual al caso a)

Caso (c): de $\mathcal{S}$ se deriva p y no se deriva $\overline{p}$.
La demostración es análoga al caso (b) pero intercambiando p por $\overline{p}$ y viceversa.

($\Leftarrow$)
Por hipótesis es posible derivar $\overline{h}$ a partir SS$\cup\{h\}$ invirtiendo algunas de las CPE de $\mathcal{S}$, y se desea derivar que el conjunto SS$\cup\{h\}$ es inconsistente, esto es, existe un literal p tal que p y $\overline{p}$ pueden derivarse rebatiblemente a partir de SS$\cup\{h\}$, sin utilizar CPE invertidas.

Caso particular: si $\overline{h}$ puede derivarse de SS$\cup\{h\}$ directamente (sin utilizar ninguna CPE invertida), entonces ya se ha encontrado un p con las características deseadas.
Caso general: supóngase que $\overline{h}$ no puede derivarse directamente de SS$\cup\{h\}$, pero invirtiendo algunas de las CPE de $\mathcal{S}$, es posible derivar $\overline{h}$ a partir de SS$\cup\{h\}$. Entonces debe existir una secuencia de CPE invertidas como la siguiente:

$$\begin{aligned}
&\overline{h} \ \ \text{@—}\ \overline{c_1}, D_1\\
&\overline{c_1} \ \ \text{@—}\ \overline{c_2}, D_2\\
&\ldots\\
&\overline{c_{n-1}} \ \ \text{@—}\ \overline{c_n}, D_n
\end{aligned}$$

donde los conjuntos D_i, y c_n pueden derivarse directamente a partir de SS$\cup\{h\}$, *i.e.*, sin utilizar CPE invertidas. Usando la mismas CPE pero sin invertir, puedo derivar c_n a partir de SS$\cup\{h\}$.

$$\begin{aligned}
&c_1 \leftarrow h, D_1\\
&c_2 \leftarrow c_1, D_2\\
&\ldots\\
&c_n \leftarrow c_{n-1}, D_n
\end{aligned}$$

Por lo tanto se ha encontrado un par de literales complementarios (c_n y $\overline{c_n}$) que pueden ser probados a partir de SS$\cup\{h\}$, y sin utilizar CPE invertidas. La importancia de esto último es que implica que invertir las reglas es solo un artificio para la demostración y para acelerar el proceso de verifición de consistencia, y no necesariamente implica que las CPE deban ser contrapositivas.

□

El lema 5.1.1 permite entonces obtener un procedimiento de encadenamiento hacia atrás para verificar si una CPR puede utilizarse consistentemente con el conjunto $\mathcal{S}$. El siguiente teorema indicará cuando una CPR puede utilizarse durante la construcción de un argumento.

Teorema 5.1.1 : *Una* CPR *"$h \prec L$"* puede ser utilizada *en la construcción de $\mathcal{A}$, si y sólo si no es posible derivar $\overline{h}$ a partir de $\mathcal{S} \cup\ Co(\mathcal{A}')\ \cup \{h\}$.*
Demostración: una CPR "$h \prec L$" puede ser utilizada en la construcción de $\mathcal{A}$ (definción 5.1.4), si y solo si $\mathcal{S} \cup\ Co(\mathcal{A}')\ \cup \{h\}$ es consistente, lo cuál por el lema 5.1.1 es cierto si y sólo si no es posible derivar $\overline{h}$ a partir de $\mathcal{S} \cup\ Co(\mathcal{A}')\ \cup \{h\}$. □

De esta forma, la verificación de la consistencia de una CPR se realizará en el momento en que el concecuente de una CPR se encuentre probado, y de no ser consistente, el mecanismo de backtracking podrá buscar otra forma de construir el argumento. Obsérvese que la verificación de la consistencia se realiza de "abajo hacia arriba" en el árbol de derivación, mientras que el mecanismo de derivación rebatible es de "arriba hacia abajo"

Ejemplo 5.1.3 : Considérese el siguiente PLR:

`~e(X) ← c(X).`	`a(X) -< b(X),c(X).`
`j(1) ← true.`	`b(X) -< e(X).`
`d(1) ← true.`	`b(X) -< p(X).`
`p(1) ← true.`	`e(X) -< j(X).`
`~a(2) ← true.`	`c(X) -< d(X).`

Al intentar construir un argumento para la meta "`a(X)`" se llega por encadenamiento hacia atrás hasta el hecho "`j(1)`", instanciándose `X` en 1. (Obsérvese que si se hubiera verificado la consistencia del literal "a(X)" antes de instanciarlo, el hecho "~a(2)" hubiera detenido la construcción del argumento antes de tiempo).

El literal "`j(1)`" es un hecho, por lo tanto no hay que verificar su consistencia. Pero el literal "`e(1)`" pertenece al conjunto $Co(\mathcal{A}')$ del argumento en construcción ($\mathcal{A}'$), y por lo tanto debe verificarse la consistencia de $\mathcal{S} \cup \{\texttt{e(1)}\}$, esto es, si $\mathcal{S} \cup \{\texttt{e(1)}\} \mathrel{|\!\sim} \sim\texttt{e(1)}$, lo cual no es cierto. Luego se verifica que $\mathcal{S} \cup \{\ \texttt{e(1), b(1)}\ \} \not\mathrel{|\!\sim} \sim\texttt{b(1)}$, y por último cuando se intenta verificar la consistencia de $\mathcal{S} \cup \{\ \texttt{e(1), b(1), c(1)}\ \}$, se descubre que $\mathcal{S} \cup \{\ \texttt{e(1), b(1), c(1)}\ \} \mathrel{|\!\sim} \sim\texttt{c(1)}$, lo cuál indica que hay una inconsistencia. Ante esto, el mecanismo de backtracking generará otra derivación para "`a(1)`" donde no hay inconsistencia. □

5.1.2 Minimalidad

Otra condición que impone la definición 5.1.1 de argumento, es la minimalidad. El siguiente ejemplo muestra que no es fácil obtener un algoritmo para verificar la minimalidad de un argumento, basándose en el órden de las cláusulas del programa.

Ejemplo 5.1.4 : Considérese el siguiente PLR:

$$\begin{array}{ll} \texttt{b} \leftarrow \texttt{f}. & \texttt{a} \prec \texttt{b}. \\ \texttt{f} \leftarrow \texttt{e}. & \texttt{b} \prec \texttt{c}. \\ \texttt{b} \leftarrow \texttt{h}. & \texttt{c} \prec \texttt{d}. \\ \texttt{h} \leftarrow \texttt{c}. & \texttt{e} \prec \texttt{c}. \\ \texttt{d} \leftarrow \texttt{true}. & \end{array}$$

El proceso de derivación rebatible utilizará algún orden predefinido en la elección de las cláusulas de programa. Supóngase que se eligen de arriba hacia abajo.

1. Si las CPE son elegidas primero, entonces de la derivación rebatible de "a", se obtiene el siguiente conjunto de CPR:

$$\mathcal{A}_1 = \{\ \texttt{a} \prec \texttt{b}\ ;\ \texttt{e} \prec \texttt{c}\ ;\ \texttt{c} \prec \texttt{d}\ \}$$

2. Si en cambio las CPR son elegidas primero, se obtiene el siguiente conjunto:

$$\mathcal{A}_2 = \{\ \texttt{a} \prec \texttt{b}\ ;\ \texttt{b} \prec \texttt{c}\ ;\ \texttt{c} \prec \texttt{d}\ \}$$

Sin embargo, ninguno de los conjuntos anteriores puede ser un argumento porque el siguiente es un subconjunto propio de ambos que también permite derivar rebatiblemente la meta "a"

$$\mathcal{A} = \{\ \texttt{a} \prec \texttt{b}\ ;\ \texttt{c} \prec \texttt{d}\ \}$$

Si las reglas son elegidas de abajo hacia arriba, se puede alterar el orden de las cláusulas para obtener un resultado igual al anterior. □

Este ejemplo demuestra que no se puede aprovechar el orden de las cláusulas para construir una derivación minimal con respecto a las CPR. Un algoritmo directo pero ineficiente para verificar la minimalidad es calcular todos los posibles argumentos y luego eliminar aquellos conjuntos de CPR que sean superconjuntos de otros. No obstante, el siguiente algoritmo recursivo, obtiene un argumento $\mathcal{A}$ para $\mathtt{m}$ de una forma más razonable:

1. Se construye una derivación rebatible y consistente $\mathcal{A}$ para $\mathtt{m}$.
2. se intenta construir otra derivación rebatible y consistente $\mathcal{A}'$ pero utilizando únicamente las CPR de $\mathcal{A}$. Si esto es posible, y $\mathcal{A}' \subset \mathcal{A}$, entonces se desecha el argumento anterior, y se verifica la minimalidad de $\mathcal{A}'$, con este mismo algoritmo.

5.2 Derrotadores de un argumento

Como se indicó antes, la consulta de una meta m tendrá éxito, si m pertenece al *conjunto de respuestas positivas* del lenguaje, esto es, existe un argumento que es una *justificación* de m. El proceso de obtención de una justificación para m, involucra la construcción de un argumento $\mathcal{A}$, para m, que no esté derrotado. Para verificar si un argumento $\mathcal{A}$ está derrotado, se construyen *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Como los derrotadores son argumentos, se debe verificar que no estén a su vez derrotados, y así siguiendo. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, llamado *árbol de dialéctica*, donde $\mathcal{A}$ es la raíz, y cada nodo tiene por hijos a sus derrotadores.

Se mostrará en esta sección como construir a partir de un argumento $\mathcal{A}$, el conjunto de todos sus derrotadores, utilizando únicamente la información provista por el argumento, y por encadenamiento hacia atrás. A continuación se reescriben algunas definiciones que fueron introducidas en los capítulos anteriores, y que serán utilizadas a lo largo de la sección.

Definición 5.2.1 : *Desacuerdo*
Sea $\mathcal{S}$ el conjunto de CPE de un PLR. Dos argumentos $\langle \mathcal{A}, h\rangle$ y $\langle \mathcal{A}_2, h_2\rangle$ estarán en *desacuerdo* si y sólo si el conjunto $\mathcal{S} \cup \{h\} \cup \{h_2\}$ es inconsistente. □

Definición 5.2.2 : *Contraargumento*
Se dirá que $\langle \mathcal{A}_2, h_2 \rangle$ *contraargumenta a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que $\mathcal{S} \cup \{h, h_2\}$ es inconsistente.
El argumento $\langle \mathcal{A}, h \rangle$ se llamará subargumento de desacuerdo, y al literal h se lo llamará punto de contraargumentación. □

Definición 5.2.3 : *Derrotador*
Un argumento $\langle \mathcal{A}_2, h_2 \rangle$ *derrota a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que: $\langle \mathcal{A}_2, h_2 \rangle$ contraargumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el literal h y se cumple una de estas dos opciones:
(1) $\langle \mathcal{A}_2, h_2 \rangle$ es estrictamente más específico que $\langle \mathcal{A}, h \rangle$ (derrotador propio), o
(2) $\langle \mathcal{A}_2, h_2 \rangle$ no puede compararse con $\langle \mathcal{A}, h \rangle$ (derrotador de bloqueo). □

A continuación se mostrará como verificar si dos argumentos están en desacuerdo, y cómo construir contraargumentos y derrotadores.

5.2.1 Argumentos en desacuerdo

Dos argumentos $\langle \mathcal{A}, h \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ están en desacuerdo cuando los literales h y h_2 producen una inconsistencia en $\mathcal{S}$ (cabe recordar que $\mathcal{S}$ es un conjunto que debe ser consistente). La forma mas simple es que h sea el complemento de h_2, pero sin que h y h_2 sean literales complementarios, puede ocurrir que el conjunto SS$\cup\{h, h_2\}$ se vuelva inconsistente. Por ejemplo con $\mathcal{S}$= { $p \leftarrow h$; $\sim p \leftarrow h_2$ }

Observación 5.2.1 : Para que $\langle \mathcal{A}, h \rangle$ sea un argumento, SS $\cup$ $\mathcal{A}$ debe ser consistente, por lo tanto en particular el conjunto SS$\cup\{h\}$ debe ser consistente.

En el caso de dos argumentos $\langle \mathcal{A}, h \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ que están en desacuerdo, el conjunto $\mathcal{S}\cup\{h\} \cup \{h_2\}$ es inconsistente, mientras que los conjuntos $\mathcal{S}\cup\{h\}$ y $\mathcal{S}\cup\{h_2\}$ son consistentes (ver observación 5.2.1). Como puede verse en la siguente proposición, esta particularidad permitirá la implementación de la verificación de la propiedad de desacuerdo fácilmente.

Proposición 5.2.1 : *Dados dos argumentos* $\langle \mathcal{A}, h \rangle$ *y* $\langle \mathcal{A}_2, h_2 \rangle$ *verificar si están en desacuerdo, es equivalente a mostrar que* $\overline{h_2}$ *se deriva a partir de* $\mathcal{S}\cup\{h\}$ *(utilizando si fuera necesario alguna* CPE *invertida)* .

Demostración: Dos argumentos $\langle \mathcal{A}, h \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ están en desacuerdo si y sólo si el conjunto $\mathcal{S} \cup \{h\} \cup \{h_2\}$ es inconsistente. Por la observación anterior, el conjunto $\mathcal{S} \cup \{h\}$ es consistente, por lo tanto por el lema 5.1.1 verificar que $\mathcal{S} \cup \{h\} \cup \{h_2\}$ es inconsistente, es equivalente a mostrar que $\overline{h_2}$ se deriva a partir de $\mathcal{S} \cup \{h\}$ (utilizando si fuera necesario alguna CPE invertida). □

La implementación de la relación de desacuerdo entre argumentos se realizará entonces utilizando lo desarrollado para verificar la consistencia de un conjunto de cláusulas.

5.2.2 Contraargumentos

En la búsqueda de una justificación para una meta h_1, la computación comienza con la construcción de un argumento $\langle \mathcal{A}_1, h_1 \rangle$. Una vez construido $\langle \mathcal{A}_1, h_1 \rangle$, se debe verificar si existen contraargumentos, es decir, se deben buscar todos aquellos argumentos $\langle \mathcal{A}_2, h_2 \rangle$ que están en desacuerdo con algún subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$. Para encontrar todos los contraargumentos hay que (a) encontrar los puntos de contrargumentación, y (b) encontrar un argumento que esté en desacuerdo en ese punto. Un algoritmo posible (aunque poco eficiente) sería el siguiente:

1. calcular todos los subargumentos de $\langle \mathcal{A}_1, h_1 \rangle$, lo cual daría todos los posibles puntos de contraargumentación.
2. para cada uno de estos subargumentos calcular todos los argumentos que estén en desacuerdo.

Sin embargo, a lo largo de esta sección se mostrará que es posible desarrollar un algoritmo que obtiene todos los contraargumentos de $\mathcal{A}_1$ utilizando únicamente la información de $\mathcal{A}_1$, y por encadenamiento hacia atrás de las cláusulas de programa.

Búsqueda de puntos de contraargumentación

Supóngase que se ha construido el argumento $\langle \mathcal{A}_1, h_1 \rangle$, y se desea buscar sus contraargumentos. Para buscar posibles puntos de contraargumentación es necesario buscar dentro del argumento $\langle \mathcal{A}_1, h_1 \rangle$.

Se reescribirán a continuación algunas definiciones introducidas en el capítulo anterior y que serán utilizadas aquí.

Definición 5.2.4 : *Argumento completado.*
Sea $\mathcal{A}$ un argumento para el literal instanciado h, el argumento completado de $\mathcal{A}$, denotado $\mathcal{A}^c$, es un conjunto de CPE y CPR formado por las CPR de $\mathcal{A}$, más el subconjunto de CPE de $\mathcal{S}$ que no son hechos, y que fueron utilizadas para obtener la derivación de h a partir de SS $\cup$ $\mathcal{A}$. □

Definición 5.2.5 : Dado un argumento completado $\mathcal{A}^c$, se llamará $Coc(\mathcal{A}^c)$ al conjunto de literales instanciados que son consecuentes de las de las cláusulas de $\mathcal{A}^c$. □

Definición 5.2.6 : *Subargumento*
Un argumento $\langle \mathcal{A}, h \rangle$ es un *subargumento* de $\langle \mathcal{B}, k \rangle$, si $\mathcal{A} \subseteq \mathcal{B}$, (independientemente de los literales h y k). □

La noción de subargumento sólo considera las CPR independientemente de los literales que estén sustentando los argumentos. El siguiente ejemplo muestra esta particularidad.

Ejemplo 5.2.1 :

d ← true.	a —≺ b,c.
e ← true.	b —≺ d.
q ← a.	c —≺ e.
p ← b.	
r ← c.	
s ← b,c.	

Utilizando el PLR anterior, puede construirse para el literal a, el argumento $\mathcal{B}$ = { a —≺ b,c ; b —≺ d ; c —≺ e}, el cuál posee los siguientes subargumentos:

$\mathcal{A}_1$= ⟨ {b —≺ d } ,p ⟩
$\mathcal{A}_2$= ⟨ {a —≺ b,c ; b —≺ d ; c —≺ e},q ⟩
$\mathcal{A}_3$= ⟨ {c —≺ e} ,r ⟩
$\mathcal{A}_4$= ⟨ {b —≺ d ; c —≺ e} ,s ⟩

Los argumentos anteriores pueden ser completados de la siguente forma:

$\mathcal{A}_1{}^c$= ⟨ {p ← b ; b $\prec$ d ; d ← true ; e ← true } ,p ⟩
$\mathcal{A}_2{}^c$= ⟨ {q ← a ; a $\prec$ b,c ; b $\prec$ d ; c $\prec$ e ; d ← true ; e ← true} ,q ⟩
$\mathcal{A}_3{}^c$= ⟨ {r ← c ; c $\prec$ e ; d ← true ; e ← true } ,r ⟩
$\mathcal{A}_4{}^c$= ⟨ {s ← b,c ; b $\prec$ d ; c $\prec$ e ; d ← true ; e ← true} ,s ⟩

□

Obsérvese que para todo literal $h \in Coc(\mathcal{B}^c)$ existe un subargumento $\langle \mathcal{A}, h\rangle \subseteq$ ⟨ $\mathcal{B}$,a ⟩. Pero como muestra el ejemplo 5.2.1 en realidad pueden existir subargumentos de $\mathcal{B}$, que no sean argumentos para literales de $Coc(\mathcal{B}^c)$. Esto es, que exista $p \notin Coc(\mathcal{B}^c)$ tal que $\mathcal{A} \subseteq$ ⟨ $\mathcal{B}$,a ⟩ y $\mathcal{A}$ sea un argumento para p. Ahora bien, el subargumento $\mathcal{A}$ posee únicamente CPRs de $\mathcal{B}$, por lo tanto los puntos de contrarargumentación que no estén en $Coc(\mathcal{B}^c)$ deberán estar en $\mathcal{S}$.

Dado $\langle \mathcal{A}_1, h_1\rangle$, y un subargumento $\langle \mathcal{A}, h\rangle$, existirán tres casos representativos de puntos de contraargumentación h:

(a) $h \in Coc(\mathcal{A}_1^c)$, y se posee un argumento para $\overline{h}$ (ver ejemplo 5.2.2).

(b) $h \in Coc(\mathcal{A}_1^c)$, pero se posee un argumento para $p \neq \overline{h}$ tal que $\mathcal{S} \cup \{h\} \vdash \overline{p}$ (ver ejemplo 5.2.3).

(c) $h \notin Coc(\mathcal{A}_1^c)$, pero se posee un argumento en desacuerdo con $\langle \mathcal{A}, h\rangle$ (ver ejemplo 5.2.4).

Ejemplo 5.2.2 : Dado el PLR:

b ← true. a $\prec$ b.
c ← true. ∼a $\prec$ c.
d ← true. ∼a $\prec$ d.

El argumento $\mathcal{A}_1$= { a $\prec$ b }, tiene dos contraargumentos para el mismo punto de contraargumentación "a": $\mathcal{A}_2$= {∼a $\prec$ c }, y $\mathcal{A}_3$= {∼a $\prec$ d}. □

Ejemplo 5.2.3 : Dado el PLR:

∼k ← m. h $\prec$ i.
m ← i. i $\prec$ j.
j ← true. k $\prec$ l.
l ← true.

El argumento $\mathcal{A}_1$= { h $-\prec$ i ; i $-\prec$ j }, tiene el contraargumento $\mathcal{A}_2$= { k $-\prec$ l } para el punto de contraargumentación "i", con argumento de desacuerdo $\mathcal{A}$= {i $-\prec$ j }. □

Ejemplo 5.2.4 : Dado el PLR:

$\sim$c $\leftarrow$ q,r. e $-\prec$ q,r.
a $\leftarrow$ true. q $-\prec$ a.
b $\leftarrow$ true. r $-\prec$ b.
c $\leftarrow$ p. p $-\prec$ a,b.

El argumento $\mathcal{A}_1$= { e $-\prec$ q,r ; q $-\prec$ a ; r $-\prec$ b } tiene el contraargumento $\mathcal{A}_2$= { p $-\prec$ a,b }, y el punto de contraargumentación es "$\sim$c" (que no pertenece a $Coc(\mathcal{A}^c)$), con subargumento de desacuerdo $\mathcal{A}$= { q $-\prec$ a ; r $-\prec$ b }. □

El ejemplo 5.2.4 muestra un caso donde el punto de contraargumentación está afuera del conjunto $Coc(\mathcal{A}_1^c)$, lo cual parecería indicar que el conjunto $Coc(\mathcal{A}_1^c)$ no es suficiente para calcular los posibles puntos de contraargumentación. Sin embargo, se mostrará a continuación que toda la información para buscar puntos de desacuerdo está en el conjunto $Coc(\mathcal{A}_1^c)$.

Cabe recordar que para verificar que un argumento $\langle \mathcal{A}_2, h_2 \rangle$ está en desacuerdo con $\langle \mathcal{A}, h \rangle$ hay que verificar que SS$\cup\{h, h_2\}$ es inconsistente. Para lo cuál posiblemente se utilice alguna CPE invertida. Ahora bien, la CPE "$\sim$c $\leftarrow$ q,r" del ejemplo anterior puede ser invertida a "$\sim$q $-@$ c,r"; y además existe una derivación rebatible para "r", ya que"r" es parte del argumento $\mathcal{A}_1$. Por lo tanto, existe una derivación $\mathcal{A}_3$= { $\sim$q $-@$ c,r ; c $\leftarrow$ p ; p $-\prec$ a,b ; r $-\prec$ b } para el literal "$\sim$q" tal que está en desacuerdo con el subargumento { q $-\prec$ a } de $\mathcal{A}_1$. Con la diferencia que ahora el literal del punto de desacuerdo es "q" el cuál pertenece a $Coc(\mathcal{A}_1^c)$.

El conjunto $\mathcal{A}_3$ en realidad no es un argumento, ya que se uso una CPE invertida para su construcción. Pero, el subargumento $\mathcal{A}_2$= { p $-\prec$ a,b } de $\mathcal{A}_3$ si lo es. Por lo tanto, usar una CPE invertida, con las condiciones establecidas para la verificación de la consistencia (ver definición 5.1.7) permitirá encontrar los contraargumentos para literales fuera de $Coc(\mathcal{A}^c)$.

Proposición 5.2.2 : *Sea $\mathcal{S}$ el conjunto de* CPEs *de un* PLR*, y $\langle \mathcal{A}_1, h_1 \rangle$ un argumento. Existe un contraargumento $\langle \mathcal{A}_2, \overline{p} \rangle$, en un punto de contraargumentación $p \notin Coc(\mathcal{A}_1^c)$,*

si y sólo si para algún $h \in Coc(\mathcal{A}_1^c)$, es posible encontrar una derivación para $\overline{h}$ a partir de $\mathcal{S} \cup \mathcal{A}_2 \cup \mathcal{A}_1$, utilizando algunas CPE *invertidas con las condiciones establecidas antes.*

Demostración: ($\Rightarrow$)
Por hipótesis existe un contraargumento $\langle\ \mathcal{A}_2, \overline{p}\ \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$, en un punto de contraargumentación $p \notin Coc(\mathcal{A}_1^c)$. Por lo tanto, $\mathcal{A}_2$ es un argumento para $\overline{p}$, y existe un subargumento de desacuerdo $\mathcal{A}$ de $\mathcal{A}_1$ ($\mathcal{A} \subseteq \mathcal{A}_1$), talque $\mathcal{A}$ es un argumento para p.
Se quiere probar que para algún $h \in Coc(\mathcal{A}_1^c)$, es posible encontrar una derivación para $\overline{h}$ a partir de $\mathcal{S} \cup \mathcal{A}_2 \cup \mathcal{A}_1$, utilizando algunas CPE invertidas con las condiciones establecidas antes.

Caso particular: Como $p \notin Coc(\mathcal{A}_1^c)$, pero $\mathcal{A}$ es un argumento para p, entonces existe una CPE "$p \leftarrow h, D$" que pertenece al subargumento de desacuerdo (completado) $\mathcal{A}^c$, pero no pertenece a $\mathcal{A}_1^c$. Donde D es un conjunto (posiblemente vacío) de literales para los cuales existe una derivación rebatible a partir de SS $\cup\ \mathcal{A}_1$, y como caso particular $h \in Coc(\mathcal{A}_1^c)$. Observesé que si p pudiera derivarse directamente de S, entonces el argumento $\mathcal{A}_2$ no hubiera sido construído por ser inconsistente con $\mathcal{S}$. Además si en D hubiera literales que se derivan con CPRs que no pertenezcan a SS $\cup\ \mathcal{A}_1$, entonces $\mathcal{A}$ no sería subargumento de $\mathcal{A}_1$.
De la CPE anterior se puede obtener la CPE invertida "$\overline{h}\ \text{@—}\ \overline{p}, D$". Como $\mathcal{A}_2$ es un argumento para $\overline{p}$, entonces se tiene una derivación para $\overline{h}$, a partir de $\mathcal{S} \cup \mathcal{A}_2 \cup \mathcal{A}_1$, donde el literal $h \in Coc(\mathcal{A}_1^c)$.

Caso general: en general no existirá una sola regla como en el caso anterior, sino que habrá una secuencia de reglas

$$p \leftarrow D_1, c_1$$
$$c_1 \leftarrow D_2, c_2$$
$$\dots$$
$$c_n \leftarrow D_n, h$$

donde: $h \in Coc(\mathcal{A}^c)$, y los D_i son conjuntos (posiblemente vacíos) de literales para los cuales existe una derivación rebatible a partir de SS $\cup\ \mathcal{A}_1$,

Invirtiendo apropiadamente las CPE anteriores, queda la secuencia:

$$\overline{h}\ \text{@—}\ D_n, \overline{c_n}$$

$$\ldots$$
$$\overline{c_2} \ @\!\!-\ D_2, \overline{c_1}$$
$$\overline{c_1} \ @\!\!-\ D_1, \overline{p}$$

Como $\mathcal{A}_2$ es un argumento para $\overline{p}$, entonces se tiene una derivación para $\overline{h}$, a partir de $\mathcal{S} \cup \mathcal{A}_2 \cup \mathcal{A}_1$, donde el literal $h \in Coc(\mathcal{A}_1^c)$.

($\Leftarrow$)
Por hipótesis $\mathcal{A}_1$ es un argumento para h_1, y para algún $h \in Coc(\mathcal{A}_1^c)$, es posible encontrar una derivación para $\overline{h}$ a partir de $\mathcal{S} \cup \mathcal{A}_2 \cup \mathcal{A}_1$, utilizando algunas CPE invertidas con las condiciones establecidas antes. Se quiere probar que existe un un contraargumento $\langle\ \mathcal{A}_2, \overline{p}\ \rangle$, en un punto de contraargumentación $p \notin Coc(\mathcal{A}_1^c)$.

Si se sigue la secuencia de CPE invertidas utilizadas para derivar $\overline{h}$:

$$\overline{h} \ @\!\!-\ B_n, \overline{c_n}$$
$$\ldots$$
$$\overline{c_2} \ @\!\!-\ B_2, \overline{c_1}$$
$$\overline{c_1} \ @\!\!-\ B_1, \overline{p}$$

entonces puede verse claramente que el conjunto $\mathcal{A}_2$ de CPRs que se utilizó para derivar $\overline{p}$ es un argumento no vacio, ya que de lo contrario el argumento $\mathcal{A}_1$ no existiría por ser inconsistente. Además $\mathcal{A}_2$ está en desacuerdo con el subargumento de $\mathcal{A}_1$ que sustenta el literal h. Por lo tanto existe un contraargumento para $\mathcal{A}_1$.

□

5.2.3 Obtención de contraargumentos a partir de $Coc(\mathcal{A}^c)$

Como se explicó antes, existen tres casos a considerar en la búsqueda de contraargumentos para $\mathcal{A}_1$. La implementación del caso (a) es sumamente sencilla, simplemente consiste en buscar un argumento para el complemento de los literales de $Coc(\mathcal{A}_1^c)$. El caso (b) es un caso particular del (c), y en cuanto al caso (c) la respuesta a como implementarlo está en la siguiente observación:

Obsérvese que en la proposición 5.2.2 el argumento $\mathcal{A}_2$ para $\overline{p}$, es justamente un argumento que contraargumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el punto p fuera de $Coc(\mathcal{A}_1^c)$. Esto es justamente lo que permitirá calcular los contraargumentos de $\langle \mathcal{A}_1, h_1 \rangle$, partiendo unicamente

de literales que pertenezcan a $Coc(\mathcal{A}_1^c)$. A continuación se mostrará como obtener el contraargumento buscado.

ALGORITMO **5.2.1** *"Búsqueda de contraargumentos fuera de $Coc(\mathcal{A}_1^c)$"*

1. Sea $R = h \leftarrow a_1, \ldots, a_n$ tal que algún $a_i \in Coc(\mathcal{A}_1^c)$. Utilizar la CPE invertida $\overline{a_i} \multimap \overline{h},\ a_1, \ldots, a_n$
2. Probar $\{a_1, \ldots, a_n\} - \{a_i\}$ con SS $\cup\ \mathcal{A}_1$.
3. Probar $\overline{h}$ con todo el PLR.
4. Si existe un argumento $\mathcal{A}_2$ para $\overline{h}$, entonces es el argumento que se está buscando. Sino, utilizar otra CPE invertida "$\overline{h} \multimap a_1, \ldots, a_n$", y este mismo algoritmo para continuar la derivación.

Obsérvese finalmente que las CPE son invertidas únicamente para hallar el contraargumento, pero no son parte del mismo. **Por lo tanto no puede considerarse que las CPE sean usadas como contrapositivas.**

El ejemplo 5.2.5 muestra la razón de la restricción de probar los antecedentes de las CPE invertidas con SS $\cup\ \mathcal{A}_1$.

Ejemplo 5.2.5 : Dado el PLR:

h ← b,c.	b ⇀ a.
a ← true.	∼h ⇀ e.
d ← true.	c ⇀ d.
e ← true.	

Se puede construir el argumento $\mathcal{A}_1$= { b ⇀ a }, y aunque no exista contraargumento para $\mathcal{A}_1$, utilizando la CPE invertida ∼b ⊸ ∼h,c es posible construir la derivación { ∼b ⊸ ∼h,c ; ∼h ⇀ e ; c ⇀ d }. En este caso no es que la proposición 5.2.2 falle, sino que la derivación anterior, no respeta una de las restricciones para el uso de CPE invertidas, ya que el literal "c" se prueba con una CPR que no pertenece a $\mathcal{A}$. □

5.3 Conclusiones

A fin de obtener una justificación para una meta M, se debe construir un árbol de dialéctica formado por argumentos y derrotadores. En este capítulo se desarrollaron algoritmos que permiten la construcción de argumentos y derrotadores, utilizando únicamente encadenamiento hacia atrás de cláusulas de programa. Esto permitirá la extensión de la máquina abstracta de Warren (presentada en el capítulo siguiente) para obtener argumentos y derrotadores.

En todos los algoritmos presentados se utiliza sólo la información de los argumentos involucrados, esto permitirá que la obtención de justificaciones se realice de una manera eficiente e independiente del tamaño del programa. Para esto, las nociones de hecho temporario y de CPE invertida fueron introducidas.

En este capítulo se prueba además que el uso de CPE invertidas para la aceleración de la búsqueda de inconsistencias, da resultados equivalentes a los que se hubieran obtenido sin el uso de ellas. Lo cual demuestra que su uso no modificará las respuestas del sistema.

Capítulo 6

La Máquina Abstracta de Warren

En 1977 David H. D. Warren, diseñó una *máquina abstracta* para la ejecución de Prolog, que consiste de una arquitectura de memoria y un conjunto de instrucciones [Warren77, Warren83]. Este diseño se denominó posteriormente Maquina Abstracta de Warren o WAM (Warren's Abstract Machine) y se ha convertido en el estándar *de facto* para la implementación de compiladores para el lenguaje Prolog [Aït91].

La WAM fue descripta por primera vez en el trabajo de D. H. D. Warren *An Abstract Prolog Instruction Set* [Warren83], donde se presenta la WAM en una forma minimalista y bastante críptica. Poco fue escrito desde esa fecha hasta la publicación en 1991, del libro *Warren's abstract machine, a tutorial reconstruction*, de Hassan Aït-Kaci [Aït91]. Este libro describe la WAM de una manera muy clara, introduciendo los conceptos de una manera gradual, hasta llegar a los detalles de la implementación y sus optimizaciones. El trabajo de Hassan Aït-Kaci fue tomado en esta tesis, como base para la confección de este capítulo, y también para la realización de un implementación de la WAM. No obstante, las secciones 6.3 y 6.6 son un desarrollo nuevo que se realizó para esta tesis.

Este capítulo está dedicado integramente a la presentar la Maquina Abstracta de Warren. Se presentará su arquitectura, el funcionamiento de sus instrucciones, y la forma en que se implementan. También se incluye un desarrollo propio de la implementación de un compilador para programas PROLOG, que genera como resultado código en instrucciones WAM. Por último, se mostrará cómo implementar a la WAM, como una máquina virtual.

6.1 Concepción básica de la WAM

En el lenguaje de la programación en lógica (ver capítulo 2), un *término* es una *variable*, o una *estructura* de la forma $f(t_1, \ldots, t_k)$ $(k \geq 0)$ donde cada t_i es un término. Si $k = 0$, f se llama *constante*. Un *predicado atómico* es una estructura de la forma $p(t_1, \ldots, t_k)$, $k \geq 0$, donde p es el nombre del predicado y cada t_i es un término, que se llamará *subtérmino* o *parámetro* [1] del predicado p. Se llama *aridad* del predicado p a la cantidad de subtérminos que posee p, y se utiliza la notación p/k para indicar que el predicado p tiene aridad k. Para diferenciar las variables de los demás elementos de un programa, éstas se denotan con un identificador que comienza con una letra mayúscula.

Un *cláusula de programa* [Lloyd87] es una cláusula de la forma " P `:-` $Q_1, \ldots, Q_n$" donde el consecuente "P" es un predicado atómico, y el antecedente "$Q_1, \ldots, Q_n$" representa una conjunción de predicados atómicos. Todas las variables de una cláusula de programa están implicitamente clausuradas universalmente. Las cláusulas de programa no son implicaciones de la lógica clásica; deben tomarse como reglas de inferencia, y no poseen un valor de verdad. Su semántica informal [Lloyd87] es: *si el antecedente es verdadero, entonces el consecuente es verdadero.*

Un *programa* es un conjunto finito de cláusulas de programa. Dada una cláusula "P `:-` $Q_1, \ldots, Q_n$" se distingue al átomo P como *cabeza* de la cláusula, y a la conjunción $Q_1, \ldots, Q_n$ como su *cuerpo*. Siguiendo esta terminología, al símbolo " `:-` " habitualmente se lo llama *cuello* de la cláusula. De esta manera es posible distinguir los siguientes tipos de cláusulas:

- Reglas: cláusulas con cabeza y cuerpo, "P `:-` $Q_1, \ldots$ Q_n" $(n \geq 0)$.
- Hechos: son cláusulas sin cuerpo "P `:-` " que establecen una aserción incondicional. Habitualmente se los denota sin utilizar el cuello (*i.e.* "P").
- Consultas: cláusulas sin cabeza " `:-` $Q_1, \ldots, Q_n$" que denotan un conjunto (conjunción) de n *metas* a resolver.

Para el estudio de la WAM es interesante considerar únicamente los siguientes tipos:

- un hecho "P" que estará probado incondicionalmente.
- una consulta " `:-` Q" que es simplemente una meta a resolver.

[1] En lugar de parámetro, habitualmente se utiliza la palabra argumento, pero en el contexto de esta tésis el término argumento tiene otro significado y es por esto que se optó por usar la palabra parámetro.

Ya que, en una regla "P `:-` $Q_1, \ldots, Q_n$", puede considerarse que la cabeza "P" estará probada, si se satisfacen todas las consultas "Q_i" del cuerpo. Por lo tanto, puede pensarse que las reglas están formadas por los dos elementos anteriores, esto es, consultas "Q_i" que al resolverse implican un nuevo hecho "P". Esta forma de ver las reglas permite que todo el desarrollo de la WAM se haga sólo sobre consultas y hechos.

A continuación se provee un ejemplo simple pero lo sufientemente expresivo, para mostrar que significa traducir un programa lógico a código WAM, y los principios sobre los cuáles trabaja la máquina abstracta. El programa lógico del ejemplo 6.1.1 utiliza sólo predicados proposicionales, esto es, no utiliza predicados con subtérminos o variables. Por lo tanto, la ejecución del programa involucrará únicamente llamadas de una cláusula a otra, permitiendo mostrar con claridad como se desarrolla el flujo de control dentro de la WAM.

Ejemplo 6.1.1 :

```
q.
r.
p :- q,r.
:- p.
```

□

En el ejemplo 6.1.1, siguiendo la semántica operacional de Prolog, la consulta "`:- p`" tendrá éxito si es posible probar "`p`" a partir del programa lógico. Utilizando la regla "`p :- q,r`", el predicado "`p`" estará probado si es posible probar las consultas "`q`" y "'`r`". Pero tanto "`q`" como "`r`" son hechos, y por lo tanto se prueban incondicionalmente. Por lo tanto la consulta "`:- q,r`" tiene exito, y por ende "`p`" queda probado, lo cual satisface la consulta "`:- p`".

La WAM es una máquina abstracta (o máquina virtual) que define una arquitectura (procesador, memoria, registros, etc) y un conjunto de instrucciones que se ejecuta sobre el procesador virtual. Dado un programa lógico, éste es traducido a un conjunto de instrucciones de la WAM, que se ejecutan sobre la arquitectura de la máquina virtual.

A continuación se mostrará la traducción del programa del ejemplo 6.1.1 a código WAM (cómo aún no fue explicado nada acerca de la WAM, el código que se encuentra a continuación fue simplificado, y no es exactamente el que se obtendrá finalmente). Mas

adelante se describirá en detalle la arquitectura de la WAM, sus instrucciones, y cómo se realiza el proceso de traducción.

La cláusula (hecho) "`q`", se traduce en la siguiente secuencia de intrucciones WAM:

```
q/0 :  just_me
       proceed
```

donde cada linea corresponde a una instrucción WAM, y "`q/0:`" es una etiqueta que se utiliza para indicar el comienzo del código de la cláusula "`q`" de aridad 0. Mas adelante se explicará en detalle cada intrucción, pero por el momento conviene aclarar que la instrucción `just_me` indica que la secuencia de código que la precede (hasta la instrucción `proceed`), es la única forma de probar "`q`". La instrucción `proceed` indica el fin del código de la cláusula, y devuelve el control a la sentencia siguiente a la que llamó a "`q/0`". Análogamente, el hecho "`r`", se traduce en la siguiente secuencia de intrucciones WAM:

```
r/0 :  just_me
       proceed
```

La secuencia de intrucciones WAM para la regla "`p :- q,r`", es la siguiente:

```
p/0 :  just_me
       call q/0
       call r/0
       proceed
```

El código de una regla, es similar al de un hecho, con la diferencia que se utiliza la instrucción `call q/0` para transferir el control a la instrucción que esté indicada por la etiqueta "`q/0`", esto es, la primer cláusula o hecho con cabeza `q` y de aridad 0.

Por último el código de la consulta "`:- p`" es el siguiente:

```
?  :  call p/0
      proceed
```

```
q/0 :  just_me
       proceed
r/0 :  just_me
       proceed
p/0 :  just_me
       call q/0
       call r/0
       proceed
  ?:   call p/0
       proceed
```

Figura 6.1: Código WAM del ejemplo 6.1.1

La etiqueta "`?:`" se utiliza para identificar el comienzo del código de la consulta, que será el lugar donde comienza la ejecución del programa. La figura 6.1 muestra el código WAM completo del programa del ejemplo 6.1.1.

La WAM funciona básicamente de la siguiente manera: en la memoria está almacenado el código WAM del programa lógico, y el de la consulta. La máquina abstracta ejecuta las instrucciones generadas comenzando en la instrucción rotulada con "`?:`". La ejecución se desarrolla secuencialmente excepto cuando se encuentran instrucciones de salto (como `call`), las cuáles modifican el flujo del control del programa lógico.

En el ejemplo anterior, la ejecución comienza donde lo indica la etiqueta "?:", *i.e.*, en la instrucción `call p/0`, que produce un salto hasta la instrucción con etiqueta `p/0`, la instrucción `just_me` no tiene ninguna acción asociada salvo indicar que esta es la única forma de probar el predicado. Luego de `just_me` se ejecuta `call q/0`, y se efectua un nuevo salto a la instrucción `just_me` con etiqueta `q/0`. A continuación, la instrucción `proceed` retorna el control a la instrucción siguiente a la llamada de `q/0`, esto es a `call r/0`. Después de ejecutar el código del hecho `r/0`, se retorna hasta devolver el control a la instrucción siguiente a `call p/0`. Como el código de la consulta `:- p` se ejecuta en su totalidad, entonces se dice que la consulta tiene *éxito*. Pero también podría existir un fallo en alguna parte del código, entonces la consulta se abortará y no tendrá éxito. En este caso se dirá que la consulta *falla* (ver ejemplo 6.1.2).

Ejemplo 6.1.2 : Dado el siguiente programa lógico:

```
p :- q.
s.
:- p.
```

el código WAM será:

```
p/0 :  just_me
       call q/0
       proceed
s/0 :  just_me
       proceed
?   :  call p/0
       proceed
```

Al ejecutarlo, como no existe una cláusula con cabeza `q`, la consulta "`:- p`" *falla*. □

Obsérvese que el modelo de ejecución de la WAM consiste simplemente en una consulta que llama y un hecho (o cabeza de una regla) que espera ser llamado. Como las reglas a su vez tienen consultas, esto produce el encadenamiento hacia atrás (backward chaining) de reglas. En las próximas secciones se verá como se trabaja con variables, constantes y estructuras, pero básicamente lo que se hará es: generar código para el flujo de control del programa y para realizar unificaciones.

Para ejecutar el programa, la WAM posee un registro llamado P que almacena la dirección de la instrucción en curso (program counter). Después de ejecutar cada instrucción, el registro P es incrementado a fin de disponer de la dirección de la siguiente instrucción, salvo cuando se ejecuta la instrucción `call`, que modifica al registro P asignándole la dirección del código del predicado a ejecutar. Además de P, existe un registro CP (continuation pointer) que almacena el punto de retorno de una llamada, esto es, la dirección siguiente a la instrucción `call`.

Asociada a la ejecución de un programa PROLOG hay una pila llamada STACK, que entre otras cosas es la encargada de mantener los distintos puntos de retorno cada vez que se llama a una nueva cláusula. Por lo tanto, la instrucción `call` debe calcular el punto

de retorno (CP) y éste debe ser almacenado en el STACK. La instrucción `proceed` es la encargada de reestablecer el P con el valor de CP que figure en ese momento en el tope de la pila.

Una particularidad de los programas lógicos, es que puede haber varias cláusulas que definan un mismo predicado, como puede verse en el ejemplo 6.1.3, donde el predicado `p` está definido por tres cláusulas. En este caso, `p` estará probado, si es posible tener éxito a través de al menos una de las cláusulas.

Ejemplo 6.1.3 :

```
p :- q.
p :- r.
p :- s.
s.
:- p.
```

□

```
p/0 :  try_me_else E1
       call q/0
       proceed
 E1:   retry_me_else E2
       call r/0
       proceed
 E2 :  trust_me
       call s/0
       proceed
s/0 :  just_me
       proceed
 ?  :  call p/0
       proceed
```

Figura 6.2: Código WAM del ejemplo 6.1.3

Esta característica requiere de un mecanismo de *backtracking*, el cuál está implementado en la WAM utilizando las instrucciones `try_me_else`, `retry_me_else`, y `trust_me`

(la implementación del mecanimo de backtracking está desarrollado en detalle en la sección 6.4). En el ejemplo 6.1.3 la consulta tiene éxito gracias a la tercera cláusula que define a `p`. El código WAM de este ejemplo está en la figura 6.2.

La instrucción `try_me_else E1` indica que de no poder probar el predicado `p` con esta parte del código, queda una posibilidad en la dirección etiquetada con `E1`. Por su parte `retry_me_else E2` indica que ya hubo intentos antes pero aun queda esperanza en ella o en la etiqueta `E2`. Por último `trust_me` indica que es la última posibilidad para probar este predicado.

A continuación se encuentra una implementación simplificada de las instrucciones `call` y `proceed`, más adelante se mostrará la implementación completa. También figura una implementación muy esquemática de la operación "Backtrack".

En la sección siguiente se verá en detalle como está organizada la arquitectura de la WAM, y luego como traducir y ejecutar un programa con variables, constantes y estructuras.

```
PROCEDURE call(Predicado) {provisoria}
 IF definido(Predicado)        { Si esta definido el predicado }
 THEN
     CP:=P+1                      { calculo el punto de retorno }
     Apilar(STACK,CP)             { guardo el punto de retorno }
     P:=Direccion(Predicado)      { salto a la direccion del predicado }
 ELSE
     Backtrack;               { Sino hago backtraking }
END call;

PROCEDURE proceed  {provisoria}
     CP:=Tope(STACK);         { Obtengo el valor del punto de retorno }
     Desapilar(STACK);
     P:=CP;                   { vuelvo al punto de retorno }
END proceed;

PROCEDURE Backtrack {provisoria}
IF OtraDefinicion(predicado)             {Si existe otra definicion}
 THEN P:=SiguienteClausula(predicado)        {salto a la proxima definicion}
 ELSE ABORTAR;                           {Sino termina el programa}
```

```
END Backtrack
```

6.2 Arquitectura de la WAM

En la sección anterior se introdujo, a través de un ejemplo con cláusulas proposicionales, cuáles son las ideas básicas sobre las cuales trabaja la WAM. Pero sólo se mostró cómo se implementa el flujo de control de un programa lógico. En esta sección se mostrará cómo está organizada la arquitectura de la máquina abstracta. Se explicará cómo se representa un término como "`p(f(X,Y),g(a,f(b)))`" y cómo se implementa la unificación, a fin de poder mostrar cómo funciona la ejecución de un programa que tiene cláusulas con variables y términos en su predicados.

La WAM está formada por los siguientes elementos:

a) Un conjunto de instrucciones (junto con la implementación de cada instrucción).

b) Una estructura de memoria de la máquina abstracta, que permite: almacenar el código WAM del programa, representar términos, almacenar los identificadores, realizar la unificación, almacenar los registros de activación durante la ejecución, etc.

c) Un conjunto de registros para tratamiento de las variables y subtérminos.

6.2.1 Conjunto de instrucciones

La WAM tiene definido un conjunto de intrucciones a las cuales se traduce todo programa lógico. En general, cada símbolo de Prolog corresponde a una instrucción WAM. En la sección anterior se vieron algunas instrucciones WAM de control, y más adelante se detallarán todas las instrucciones. Cada instrucción consiste de un código de operación (opcode) y uno o más operandos. Como cada instrucción WAM define una acción que debe realizar la máquina abstracta, cada instrucción tiene un código asociado que ejecutará el procesador virtual. En el cap¡tulo anterior puede verse el código, en una notación estilo Pascal, que define a las instrucciones `call`, y `proceed`. Si la WAM es implementada como una máquina virtual, entonces habrá un procedimiento que implemente las acciones de cada instrucción WAM. Si por el contrario, la WAM es implementada directamente sobre una máquina real, entonces habrá código de bajo nivel, que implemente cada instrucción WAM.

Las instrucciones de la WAM pueden agruparse según su función, en: instrucciones de control, instrucciones para consultas, e instrucciones para hechos.

a) Instrucciones de control: `call`, `proceed`, `allocate`, `deallocate`, `just_me`, `try_me_else`, `retry_me_else`, `trust_me`, `cut`, `get_level`, `neck_cut`, `save_query_var`, `show_answer`, y `end_query`.

Estas instrucciones, como se vió en la sección 6.1, son las encargadas de llevar el flujo de control del programa. La instrucción `call` es la encargada de buscar la definición de la cláusula que unifique con la consulta; `proceed` simplemente devuelve el control a la instrucción siguiente de la llamada; `allocate` y `deallocate` son las encargadas de apilar y desapilar los registros de activación que contendrán información que debe preservarse mientras se resuelven las consultas del cuerpo de la cláusula (ver sección 6.2.7). Las instrucciones `just_me`, `try_me_else`, `retry_me_else`, `trust_me`, `cut`, `get_level` y `neck_cut` son las encargadas de manejar el mecanismo de backtracking y se verán con más detalle en la sección 6.4. La instrucción `end_query` indica el fin de una consulta y es utilizada por el intérprete, junto con `save_query_var` y `show_answer` para dar la respuesta de las consultas (ver sección 6.6).

b) Instrucciones para consultas: `put_variable`, `put_temp_var`, `put_value`, `put_structure`, `put_constant`, `put_list`, `set_variable`, `set_value`, `set_constant`, y `set_void`.

La ejecución de un programa está direccionada por consultas, son ellas las encargadas de cargar los registros o la memoria con valores nuevos (put), o fijar su valor con algún valor anterior (set). Existe una instrucción especializada para cada tipo de elemento: variables, constantes, estructuras, o listas. En la sección 6.3 se explicará en detalle cuando se utiliza cada una de estas instrucciones.

c) Instrucciones para hechos: `get_value`, `get_list`, `get_variable`, `get_structure`, `get_constant`, `unify_variable`, `unify_value`, `unify_constant`, y `unify_void`.

Los hechos (o las cabezas de reglas) son los encargados de tomar los datos que las consultas han dejado en algún lugar de la memoria. Para ello se tienen dos tipos de instrucciones: las que toman datos (get), y las que intentan unificar dos elementos (unify). Al igual que en el caso de las consultas, existe una instrucción especializada para cada tipo de elemento: variables, constantes, listas, o estructuras (ver sección 6.3).

6.2.2 Arquitectura de la memoria

La memoria está formada por:

1. Un área que contiene el código del programa ya compilado.
2. Una pila para el manejo de estructuras llamada "HEAP" o pila local.
3. Una pila para los registros de activación llamada "STACK" o pila global.
4. Una pila para guardar variables durante el backtracking llamada "TRAIL" (rastro).
5. Una pila para realizar la unificación de términos llamada "PDL" (push-down list).

El HEAP contiene todas las estructuras creadas por unificación y por invocación de procedimientos. El TRAIL contiene referencias a variables que han sido ligadas durante la unificación y que deben ser desligadas al hacer backtracking. El STACK contiene dos clases de objetos: los *entornos* (environments), y los *puntos de elección* (choice points), ambos serán explicados en detalle más adelante.

La organización de la memoria de la máquina abstracta puede verse en la figura 6.3, donde se muestra cada área de la memoria y la dirección hacia donde crece. Es importante que el HEAP y el STACK queden de esa forma ya que la estrategia de ligar siempre la variable con la dirección más baja, al hacer una ligadura variable-variable, es suficiente para prevenir referencias pendientes. En la figura 6.3 también están indicados los registros que tienen punteros a cada una de las áreas de memoria mencionadas.

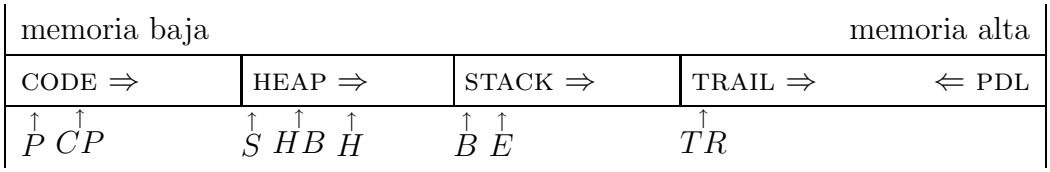


Figura 6.3: Disposición de la memoria y registros.

El estado de una computación en Prolog, está definido por el conjunto de registros que contienen punteros a las áreas de memoria mencionadas. En la tabla 6.1 se detalla cada uno de ellos.

Además de los registros anteriores, existe un conjunto de registros llamados *registros de términos* y *registros de parámetros*. Los registros de términos se notarán X_j y se utilizarán

Registro	Contenido
P	Puntero a la instruccón en curso, en el área de código.
CP	Puntero a la instrucción de retorno de un call (continuation pointer).
H	Puntero al tope del HEAP.
E	Puntero al último entorno del STACK.
B	Puntero al último punto de elección del STACK.
HB	Valor de H del último B.
TR	Puntero al tope del TRAIL.
S	Puntero a una estructura del HEAP.

Tabla 6.1: Principales registros de la WAM.

durante la ejecución del programa lógico para almacenar temporariamente apuntadores a términos en el HEAP. Los registros de parámetros se notarán A_i y se utilizarán para implementar el pasaje de parámetros entre una consulta que invoca a otra. En descripciones anteriores de la WAM, se llamaba a los registros A_i *registros de argumentos*, se le cambió el nombre aquí, porque la palabra argumento es utilizada en esta tésis para indicar un concepto central, y distinto al que se usaba en los reportes de la WAM.

Todo identificador del programa lógico, (esto es, símbolos de constantes, símbolos de los functores, nombres de predicados, etc.) se almacenará en una estructura separada del HEAP, llamada *tabla de identificadores* o IDENT. Aunque esta tabla no está presente en el diseño original de la WAM, resulta adecuada a efectos de no romper la uniformidad de la estructura del HEAP.

6.2.3 Principios de la WAM

A continuación se enuncian ciertos principios sobre los cuales esta basada la WAM. Algunas decisiones de diseño que se verán en las siguientes secciones, tienen su fundamento en estos principios. El lector deberá referirse a ellos cuando lo crea necesario.

- **Principio 1:** El espacio en el HEAP debe usarse lo menos posible, ya que los términos construidos en el HEAP tienden a ser relativamente persistentes.
- **Principio 2:** Los registros deben ser asignados de forma tal, de evitar movimientos de datos innecesarios y minimizar el tamaño del código.

- **Principio 3:** Situaciones particulares que aparecen frecuentemente, aunque puedan ser manejadas correctamente por instrucciones de proposito general, deben ser tratadas por instrucciones especiales, si esto ahorra espacio o tiempo.

6.2.4 Representación de estructuras en el HEAP

Una de las tareas más frecuentes en la ejecución de un programa lógico, es la unificación de dos términos. Para que dos términos puedan unificarse deben estar almacenados en alguna estructura que permita leer y escribir sobre ellos. La estrucutura de memoria que se utiliza para almacenar todos los términos del programa, es el HEAP. A continuación se mostrará como representar términos en el HEAP.

Recuérdese que un término (de primer orden) tiene la siguiente definición recursiva. Un término es:

1. o bien una *variable* (denotada con una Mayúscula inicial),
2. o bien una *constante* (denotada con una minúscula inicial),
3. o bien una *estructura* de la forma $f(t_1, \ldots, t_n)$ donde f se llama símbolo de functor (denotado igual que una constante) y los t_i son términos (también se los suele llamar *subtérminos*).

El número de subtérminos para un functor dado, es fijo y se llama su *aridad*. El mismo símbolo de un functor puede utilizarse con diferentes aridades, pero serán considerados functores diferentes. Es por esto que se usará la notación f/n para referirse a un functor de símbolo f y aridad n. Una constante c puede verse como un caso especial de estructura de aridad 0, y por lo tanto puede denotarse como el functor $c/0$.

Una variable puede tener ligado (bound) un término, o estar sin ligar (unbound). Por lo tanto una variable será representada en el HEAP como una dirección al término a la que esté ligada, o como una referencia a si misma en el caso que esté sin ligar.

Una constante es simplemente un identificador. Por lo tanto estará representada en el HEAP por su identificador.

Por último, una estructura de la forma $f(t_1, \ldots, t_n)$ estará representada como la secuencia $f/n, @_1, @_2, \ldots, @_n$ donde $@_i$ es la dirección de memoria de los subtérminos de f/n.

Como se vió anteriormente, el HEAP es una zona de memoria de la WAM que se maneja en forma de pila. A fin de almacenar diferentes clases de elementos (variables, constantes, functores) cada elemento o celda del HEAP tendrá una etiqueta que indicará su contenido, por lo tanto el HEAP tendrá la siguiente estructura de datos, indicada en notación estilo Pascal (ver también figura 6.4):

```
Heap = ARRAY [Direccion] of CeldaMemoria;

CeldaMemoria = RECORD
                  clase:ClaseCeldaMemoria;
                  direccion:Direccion;
               END;
```

donde en principio se tendrán las siguientes clases de elementos:

```
ClaseCeldaMemoria = (referencia,estructura,constante,functor);
```

La clase de la celda de memoria se utiliza para darle un significado al segundo elemento del par. Esto es, si la celda contine (referencia,d), entonces será la representación de una variable, y d inidicará la dirección de memoria donde está ubicado el término ligado a la variable. Si la celda es (constante,c), será una celda que contenga una constante, cuyo identificador se encuentra en la posición c de la tabla de identificadores. Por su parte (functor,f), representará un functor que está ubicado en posición f de la tabla de identificadores; y (estructura,d), indicará que en la dirección d del HEAP comienza una estructura.

	clase	dirección
1	...	...
2	...	...
3	...	...
⋮	⋮	⋮

Figura 6.4: Estructura del HEAP

Ejemplo 6.2.1 : Si la constante "a" (cuyo identificador está en la dirección i de la tabla de identificadores) está alojada en la dirección h del HEAP; la variable X sin ligar está alojada en la dirección d del HEAP; y la variable Y ligada a la constante "a" está alojada en la dirección $d+1$; entonces el HEAP contendrá lo que indica la figura 6.5. □

	clase	dirección
	⋮	⋮
h	constante	i
	⋮	⋮
d	referencia	d
$d+1$	referencia	h
	⋮	⋮

Figura 6.5: HEAP con dos variables y una constante

Observación 6.2.1 : El identificador de una constante o functor, es importante para la unificación y debe ser guardado en la memoria de la máquina abstracta. Pero el identificador de las variables, no es necesario y se pierde definitivamente una vez que el programa es compilado. Una vez que una estructura está almacenada en el HEAP las variables se identifican por su dirección de memoria y no por el identificador que tenían en el programa. Esto permite entre otras cosas, utilizar el mismo identificador de variable en diferentes cláusulas, sin que esto implique que sea la misma variable (en otras palabras, simplifica notoriamente la implementación de la clausura de las variables sobre la cláusula). Es por esto también, que un intérprete Prolog no puede mostrar el identificador que tenía una variable en el programa, una vez que su código fue compilado.

La definición recursiva de una estructura, lleva naturalmente a una representación recursiva de las estructuras dentro del HEAP. Una estructura con functor f/n estará formada por $n+1$ celdas de memoria contiguas, más lo necesario para representar los n sub-términos. De esta forma, para alojar en la dirección d del HEAP la estructura $f(t_1, \ldots, t_n)$, se almacenará en la dirección d la posición de la tabla de identificadores donde está alojado el símbolo del functor, y en las n celdas siguientes, la dirección a la representación de los n sub-términos de f/n, que podrán ser variables, constantes, o estructuras. En la sección siguiente se verá en detalle como obtener a partir de una estructura cualquiera, la representación en el HEAP. Se verá a continuación un ejemplo de un término representado en el HEAP.

Ejemplo 6.2.2 : La figura 6.6 muestra la representación del término `p(Z,h(Z,W),f(W))` a partir de la dirección 106 del HEAP.

Como puede verse en el HEAP de la figura 6.6, el functor `p/3` comienza en la dirección 106, donde el par (functor, 3) indica que se trata de un functor, cuyo identificador está

	Clase	Dirección
	⋮	⋮
101	functor	1
102	referencia	102
103	referencia	103
104	functor	2
105	referencia	103
106	functor	3
107	referencia	102
108	estructura	101
109	estructura	104
	⋮	⋮

	IDENT
1	h/2
2	f/1
3	p/3

Figura 6.6: HEAP con la estructura `p(Z,h(Z,W),f(W))`

en la posición 3 de la tabla de identificadores. El functor `p` es de aridad 3, por lo tanto, las tres celdas siguientes (107,108 y 109) contienen las direcciones de los subtérminos de p. La celda 107 contiene la dirección de la variable `Z` que se encuentra en la posición 102, y las celdas 108 y 109 contienen las direcciones de los subtérminos `h` y `f`. El subtérmino `h(Z,W)` se encuentra alojado a partir de la posición 101, seguido por sus dos parámetros: las variables `Z` y `W`. Como ambas variables no han sido ligadas, contienen una referencia a ellas mismas. El functor `f/1` esta alojado en la posición 104 del HEAP y su parámetro es una referencia a la variable `W` alojada en la posición 103. Obsérvese finalmente que como toda aparición de `Z` corresponde a la misma variable, todas las apariciones hacen una referencia a la celda 102 que es donde `Z` está alojada. Lo mismo ocurre con la variable `W`. □

Hay que destacar que en el trabajo de Hassan Aït-Kaci [Aït91] la representación de una estructura difiere un poco de la anterior. Un functor de f/n está representado en $n+2$ celdas, la primera indica la dirección del HEAP donde se encuentra el símbolo del functor, la segunda contiene el símbolo del functor, y luego se encuentran las n celdas con las n direcciones de sus sub-términos. En la representación usada aquí, existe una tabla especial para almacenar los identificadores, y por lo tanto los identificadores no son almacenados en el HEAP.

6.2.5 Registros de términos y registros de parámetros

El manejo de las variables y la unificación, son temas centrales dentro de la definición de la máquina abstracta, ya que:

1. La computación dentro de un programa lógico se produce fundamentalmente a través de la unificación de las variables de una consulta con las estructuras que son subtérminos de un hecho, y

2. Todas las apariciones de una variable dentro de una cláusula toman el mismo valor (por clausura universal implícita de las cláusulas).

Para esto, es necesario establecer un mecanismo de comunicación entre una consulta que intenta resolverse y el hecho que espera ser llamado. Este mecanismo es parecido al pasaje de parámetros de los lenguajes imperativos.

La WAM posee un conjunto de registros dedicados exclusivamente para estas operaciones. Estos registros tienen la misma estructura que una celda de memoria.

(a) Los *registros de términos*, que se notan X_j (con $j = 1, 2, \ldots$) y se utilizan durante la ejecución del programa lógico para almacenar temporariamente apuntadores a términos en el HEAP; y

(b) Los *registros de parámetros*, que se notan A_i (con $i = 1, 2, \ldots$) y se utilizan para implementar el pasaje de parámetros entre una consulta que invoca a una cláusula.

Los registros X_j son los que "conservan" el valor de las variables durante la ejecución de una cláusula. En una cláusula cada variable tiene un único registro asociado. En cambio, los registros de parámetro A_i, sólo sirven para pasar la infomación de una consulta a un hecho y por lo tanto hay un registro por cada parámetro en la consulta. Las estructuras tienen un registro X asociado para permitir referirse a ellas en el HEAP, ya que, como se vio en el ejemplo, los subtérminos están separados de los términos que los contienen. Las constantes en cambio no necesitan tener un registro X ya que su representación es una sola celda del HEAP

Ejemplo 6.2.3 :
En la consulta " $\leftarrow$ `p(V)`", hay un registro de término X_1 para la variable `V` y un registro

de parámetro A_1 para el único parámetro `V`.
Sin embargo, en la consulta "$\leftarrow$ `p(U,V,f(U,V,W),U,a)`" hay cuatro registros de términos: X_1 para la variable `U`, X_2 para la `V`, X_3 para el functor `f` y X_4 para la `W`. Y cinco registros de parámetro: A_1 para la `U`, A_2 para la `V`, A_3 para la estructura `f(U,V,W)`, A_4 para la tercera aparición de `U`, y A_5 para la constante `a`. □

Se verá a continuación un ejemplo sencillo donde se utiliza un registro de parámetro con el objetivo de unificar el parámetro de la consulta con el del hecho.

Ejemplo 6.2.4 :

Programa:		Código WAM:	Registros:
`:- r(a).`	`?:`	`put_constant 2` A_1	A_1=(constante,2)
		`call r/1`	
`r(a).`	`r/1:`	`just_me`	
		`get_constant 2` A_1	
		`proceed`	

□

A fin de entender el funcionamiento del código anterior, recúerdese que el modelo de ejecución de la WAM consiste en una consulta que llama y un hecho (o cabeza de regla) que espera ser llamado. La consulta cargará registros y celdas de memoria antes de llamar al predicado, y el hecho tratará de unificar lo cargado por la consulta con la información que él posee.

En el ejemplo 6.2.4, al ejecutarse la instrucción "`put_constant 2` A_1" se carga el registro de parámetro A_1 con el valor (constante,2). Esto indica que el primer parámetro de la consulta es una constante cuyo identificador ("`a`") está en la posición 2 de la tabla de identificadores. Luego, al ejecutarse el código del hecho, la instruccón "`get_constant 2` A_1", indica que debe unificarse el contenido del registro A_1 con la constante "`a`". Lo cuál en este ejemplo tiene éxito porque justamente el registro A_1 tiene alojada a la constante "`a`".

Obsérvese que en este caso sólo se utilizó el registro A_1, y que no hubo necesidad de utilizar el HEAP (ver el principio 1 en la sección 6.2.3). En la figura 6.7 se muestra el código de las instrucciones `put_constant` y `get_constant`, donde además figura el código de una operación auxiliar llamada `deref`. La operación `deref` toma una dirección de

memoria y en el caso de que haya un encadenamiento de referencias, devuelve la última dirección del encadenamiento, esto es, siempre devuelve una dirección a un objeto que no es una referencia. Excepto cuando la dirección sea la de una variable sin ligar, en cuyo caso `deref` devuelve la dirección de una referencia. En el caso del ejemplo 6.2.4 el uso de la operación `deref` devuelve la misma dirección enviada, ya que se trata de la dirección de un objeto (constante), pero como se verá a continuación, en otros casos puede seguir una cadena de referencias.

```
PROCEDURE put_const(Aj,pos_ident)
       Aj:=(Constante,pos_ident)
       P:=Inc(P);
END put_const;

PROCEDURE get_const(Aj,pos_ident)
       dir:=deref(Aj);
       Case Memoria[dir] of
          (referencia,v): Memoria[dir]:=(constante,pos_ident);
          (constante,v): Falla := pos_ident <> v;
          Else Falla:=true;
       End case;
       P:=Inc(P);
END get_const;

PROCEDURE deref(dir:TDireccion):TDireccion;
       (Clase,Valor):=Memoria[dir];
       IF (clase = referencia) AND (valor <> dir)
       THEN RETURN deref(valor)
       ELSE RETURN dir
END deref;
```

Figura 6.7: Instrucciones WAM para constantes

Ejemplo 6.2.5 : Si se tiene una configuración del HEAP y registros, como indica la figura 6.8, deref(101) devolverá 102, deref(A_1) devolverá A_1, deref(X_1) devolverá 102, y deref(104) devolverá 104. □

101	referencia	103
102	constante	1
103	referencia	102
104	referencia	104
⋮	⋮	⋮

A_1=(constante,2)
A_2=(referencia,101)
X_1=(referencia,102)

Figura 6.8: HEAP y registros.

6.2.6 Variables permanentes y temporarias

Aquellas variables que aparecen en más de una consulta en el cuerpo de una cláusula, deben ser protegidas de los efectos colaterales de las instrucciones put. En la cláusula "`p(X,Y) :- q(X,Z) , r(Z,Y).`" por ejemplo, si las variables `Y` y `Z` pudieran ser accedidas sólo desde los registros de parámetros, entonces no habría garantía que los valores que tenían al principio de la ejecución de la cláusula, estén ahí al finalizar la ejecución del cuerpo de `p`. Por lo tanto es necesario que las variables de este tipo (permanente) sean protegidas en el entorno asociado a la activación de esta cláusula, y por lo tanto serán almacenadas en el STACK.

Siguiendo la definición de Warren, se distinguirán dos tipos de variables en una cláusula, las *permanentes* (equivalentes a las variables locales de un procedimiento en un lenguaje imperativo), y las *temporarias* (globales). Para determinar si una variable es permanente o temporaria, Hassan Aït-Kaci [Aït91] propone la siguiente definición: "Una variable será permanente si aparece en más de una consulta en el cuerpo de una cláusula, (considerando a la cabeza de una cláusula como parte de la primera consulta del cuerpo). Si no es permanente, entonces será temporaria."

La instrucción `put_variable` debe comportarse diferente ante una variable temporaria que ante una permanente. Es por ese motivo, que hay dos instrucciones: la anterior, para variables permanentes, y `put_temp_var` para las temporarias.

De la misma forma que se nota con A_i, y X_j las direcciones de memoria de los registros, y a efectos de denotar de alguna manera las direcciones de las variables permanentes, se denotará con Y_n la dirección de la i-ésima variable del entorno en curso. Por ejemplo Y_2 corresponderá a la segunda variable permanente del registro en curso. También en algunos casos se usará la notación V_n para indicar que se puede tratar de X_n o de Y_n.

La figura 6.9 muestra el código de ambas instrucciones. Obsérvese que en el caso de las variables permanentes, el parámetro de la instrucción es el desplazamiento dentro del entorno, y por lo tanto se puede obtener la dirección absoluta de la misma. El ejemplo 6.2.6 muestra un programa que usa una variable temporaria.

```
PROCEDURE put_variable (Yn,Ai);
    Dir:=DireccionAbsoluta(Yn);
    STACK[Dir] := (referencia,Dir);
    Ai := STACK[Dir];
END put_variable

PROCEDURE put_temp_var (Xn, Ai);
    HEAP[H] := (referencia,H);
    Xn := HEAP[H];
    Ai := HEAP[H];
    H := H+1;
END put_temp_var
```

Figura 6.9: Implementación de instrucciones para variables

Ejemplo 6.2.6 :

Programa:		Código WAM:	Registros:
← r(X).	?:	put_temp_var X_1 A_1	X_1=(referencia,h1)
		call r/1	A_1=(referencia,h1)
r(a).	r/1:	just_me	
		get_constant 2 A_1	
		proceed	

En este ejemplo se utiliza el HEAP y los registros X_1 y A_1. Al ejecutarse el código de la consulta, la variable sin ligar `X` queda alojada en la posición h_1 del HEAP, como el par (referencia,h_1). Tanto el registro A_1, como X_1, quedan referenciando a h_1. Al ejecutarse el código del hecho, se unifica la variable `X`, con la constante `a`, colocando en HEAP[h_1] dicha constante, esto es, el par (constante,2). □

6.2.7 Ejecución de un programa y uso del STACK

Como se indicó antes, asociada a la ejecución de un programa, hay una pila llamada STACK, que tiene los registros de activación de los predicados. Estos registros son apilados y desapilados a medida que se llama y se retorna de los predicados, y contienen la información del entorno del predicado que se está ejecutando. Los registros de entorno son almacenados en el STACK, ya que al retornar de una llamada, se debe disponer de la información que se poseía antes de haberla efectuado. El registro global E apunta al último registro de entorno.

En la figura 6.10 se muestra la información que posee un registro de entorno.

E:	contiene el E anterior,
E+1:	contiene CP (Continuation Pointer)
E+2:	contiene la cantidad N de var. permanentes
E+3:	Primer variable permanente
⋮	⋮
E+2+N:	N-ésima variable permanente

Figura 6.10: Estructura y contenido del registro de entorno

Las instrucciones encargadas de cargar y descargar los registros de entornos del STACK son `allocate` y `deallocate`. La instrucción `allocate N` guarda el valor del CP y tiene un parámetro que indica el espacio que debe reservar para las variables permanentes que se alojarán en el registro de entorno. Este valor es almacenado en el registro, a fin de conocer durante la ejecución del programa cuanto espacio ocupa realmente cada registro de entorno. La instrucción `deallocate` simplemente reestablece los valores del registro CP, y del registro E, con lo cual se desapila un entorno. A continuación figura el código de ambas. Además, ahora es posible mostrar una versión cercana a la final del código de `call` y `proceed`. Observar que `allocate` y `deallocate` son las encargadas de guardar y recuperar el punto de retorno (CP).

```
PROCEDURE call(Predicado) {provisoria}
 IF definido(Predicado)        { Si esta definido el predicado }
 THEN   CP:=P+1                      { calculo el punto de retorno }
        P:=Direccion(Predicado)      { salto a la direccion del predicado }
 ELSE Backtrack;               { Sino hago backtraking }
END call;

PROCEDURE proceed
  P:=CP; {la ejecucion sigue donde indica CP que fue recuperado por allocate}
END proceed
```

Figura 6.11: Instrucciones WAM de control

Ejemplo 6.2.7 :

```
a(X) :- b(X),c(X).
b(X) :- e(X).
e(X) :- g(X).
c(1).
g(1).
```

Al efecturar la consulta `a(X)` sobre el programa anterior, se apila el entorno de `a(X)`, luego el entorno de `b(X)`, el de `e(X)`, y por último el de `g(X)`. El STACK se encuentra en la siguiente situación:

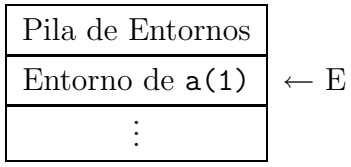
Pila de Entornos	
Entorno de `a(X)`	
Entorno de `b(X)`	
Entorno de `e(X)`	
Entorno de `g(X)`	← E
⋮	

La prueba de `g(X)` tiene éxito con `X=1`, entonces al retornar se desapilan los entornos hasta llegar al de `a(X)`. Pero en este punto, ya se ha unificado `X=1`.

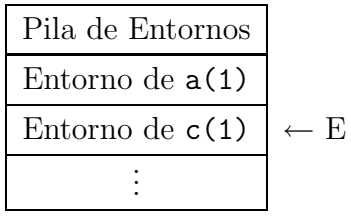
```
PROCEDURE allocate(N)
 IF (E = StackVacio)             { El valor del nuevo E depende del viejo E}
  THEN NuevoE := PrimeroStack
  ELSE NuevoE:= E + STACK[E+2] + 3;  { E+2 contiene la cantidad de variables }
 STACK[NuevoE]:=E;               { Guardo el E anterior }
 STACK[NuevoE+1]:=CP;            { el punto de retorno  }
 STACK[NuevoE+2]:=N;             { y la cantidad de variables permanentes }
 E:=NuevoE;                      { Por ultimo modifico E }
END allocate

PROCEDURE deallocate
 CP:=STACK[E+1];    { Obtengo el punto de retorno guardado }
 E:=STACK[E];       { y reestablesco el antiguo valor de E }
END deallocate
```

Figura 6.12: Instrucciones WAM de control



Ahora se llama a `c(1)` y se apila el entorno correspondiente:



Como la consulta `c(1)` tiene éxito, entonces se retorna y la consulta original `a(X)` resulta exitosa con la unificación `X=1`. □

A continuación se muestra un programa y su correspondiente código WAM, donde puede apreciarse un buen número de instrucciones WAM. A la derecha de las instrucciones se

colocó a modo de comentario los elementos de la cláusula, a fin de que pueda observarse que elemento dio origen a cada instrucción. Si se observa en detalle puede verse claramente que las consultas colocan datos en lugares de memoria con instrucciones "`put`" y "`set`", mientras que los hechos los obtienen con instrucciones "`get`" y "`unify`". Debe recordarse que las reglas se interpretan como un hecho, seguido de varias consultas. En la sección siguiente se explicará como generar el código WAM, y como funciona cada una de estas instrucciones en la máquina abstracta.

Ejemplo 6.2.8 : El programa

```
p(X,X,f(X,Y)) :- r(g(X,Y),X,Y).
r(g(a,b),a,b).
:- p(a,X,E).
```

tiene el siguiente código WAM:

p/0:	just_me	p	r/0:	just_me	r
	allocate	(		allocate	(
	get_variable X1 A_1	X,		get_structure 3 A_1	g(
	get_value X1 A2	X,		unify_constant 5	a,
	get_structure 2 A3	f(		unify_constant 6	b),
	unify_value X1	X,		get_constant 5 A2	a,
	unify_variable X3	Y)		get_constant 6 A3	b
	put_structure 3 A_1	$\leftarrow$ r(g		deallocate	)
	set_value X1	(X,		proceed	.
	set_value X3	Y),	?:	allocate	$\leftarrow$ p(
	put_value X1 A3	Y		put_constant 5 A_1	a,
	call r/0	)		put_variable A2	X,
	deallocate	.		put_variable A3	E)
	proceed			call p/3	.
				end_query	

A la derecha del código WAM se muestra el símbolo que produjo cada instrucción. □

6.3 Compilación y generación de código

En lo que sigue se detallará cómo generar código WAM, a partir de un programa Prolog. Se presentará un conjunto de técnicas y algoritmos de compilación desarrollados durante la elaboración de esta tésis.

Como pudo verse a través de los ejemplos presentados, las instrucciones WAM generadas para una consulta difieren de las generadas para un hecho. Esto se debe a que, siguiendo el principio enunciado anteriormente, una consulta cargará registros y celdas de memoria, y luego invocará a una cláusula C que intentará unificar su cabeza con lo cargado en memoria. De tener éxito, se invocarán las consultas del cuerpo de C (si las hubiera), y así sucesivamente.

Cada cláusula de un programa puede compilarse por separado, si cierta información es guardada en una tabla de símbolos que perdure hasta que se haya compilado todo el programa. A fin de generar el código WAM de un programa, se realizó un analizador sintáctico recursivo descendente de una pasada, que recorre las cláusulas de izquierda a derecha, y almacena la información procesada en dos tablas de símbolos: Una *tabla de símbolos temporaria* (figura 6.13) cuyo tiempo de vida es la compilación de una cláusula, y otra que perdura incluso durante la ejecución, que se llamará *tabla de símbolos permanente* o *tabla de identificadores* (figura 6.14).

Número de átomo
Identificador
Categoría
Número de registro X
Número de registro A
Aridad
Instrucción WAM
Parametros de la instrucción WAM

Figura 6.13: Estructura de la tabla de símbolos temporaria

Como se vió en los ejemplos anteriores, casi todos los elementos de una cláusula tienen asociado una instrucción WAM. Por lo tanto, las instrucciones se van generando y guardando en la tabla temporaria, a medida que se va compilando la cláusula. Una vez que se ha completado la compilación de una cláusula, se carga en memoria el código WAM, la

tabla temporaria es vaciada, y se continúa con la compilación de la siguiente cláusula del programa.

La tabla de símbolos temporaria posee la siguiente información:

1. Número de átomo dentro de la cláusula: esta información es necesaria para determinar cuando una variable es temporaria o permanente.
2. Identificador del elemento encontrado: puede ser el identificador de la consulta, predicado, functor, variable, constante, etc.
3. Categoría del elemento encontrado: variable, constante, estructura, predicado, etc..
4. Número de registro de término X que se le asigna al elemento encontrado
5. Número de registro de parámetro A, asignado según el orden de los parámetros encontrados.
6. Aridad: sólo en el caso de los predicados y estructuras.
7. La instrucción WAM asociada al elemento: obtenida según los algoritmos que se verán a continuación.
8. Parámetros de la instrucción WAM.

La tabla de símbolos permanente o tabla de identificadores, se utiliza para almacenar los datos globales del programa. Por cada cláusula se almacena: el identificador del predicado de la cabeza, su aridad, la posición donde comienza el código de dicha cláusula, e información para el backtraking.

Identificador
Aridad
Clase de identificador (functor, constante, etc.)
Dirección del código de la primer cláusula
Dirección del código de la última cláusula

Figura 6.14: Estructura de la tabla de símbolos permanente

La compilación de una cláusula está dividida en la compilación de la cabeza (un hecho), y luego, si existen, la compilación de cada una de las consultas del cuerpo. Al detectar

el nombre del predicado de la cláusula, este se guarda en la tabla temporaria, se generan las instrucciones `just_me` y `allocate` y luego se siguen compilando (recursivamente) los parámetros del predicado (ver ejemplo 6.3.1). Al terminar de compilar los parámetros, se conoce la aridad del predicado, la cuál es guardada en la tabla de símbolos. La compilación de cada una de las consultas del cuerpo de la cláusula es similiar a la de la cabeza: se guarda el identificador de la consulta en la tabla temporaria, se genera el código de los parámetros, y luego con el identificador de la consulta que está en la tabla, se genera la instrucción `call`. Una vez que se ha terminado de compilar el cuerpo de la cláusula, se generan las instrucciones `deallocate` y `proceed`.

Ejemplo 6.3.1 : Al finalizar la compilación de la cláusula `p(X,Y) :- q(f(X),a),r(g(X,Y))` se tiene la siguiente información en las tablas de símbolos temporaria y permanente.

Átomo	Ident	Ari	Categoría	A_i	X_j	Instrucción	Parámetros
1	p	2	functor			just_me	
						allocate	2
1	X		variable	A_1	Y_1	get_variable	A_1Y_1
1	Y		variable	A_2	Y_2	get_variable	A_2Y_2
1	f	1	functor	A_1	X_1	put_structure	A_1
1	X		variable		Y_1	set_value	Y_1
1	a	0	constante	A_2		put_constant	A_2
1	q	2	functor			call	q/2
2	g	2	functor	A_1	X_1	put_structure	A_1
2	X		variable		Y_1	set_value	Y_1
2	Y		variable		Y_2	set_value	Y_2
2	r	1	functor			call	r/1
						deallocate	
						proceed	

Ident	Aridad	Clase	Dir-pri	Dir-ult
p	2	predicado	1	1
f	1	functor		
a	0	constante		
q	2	functor		
g	2	functor		
r	1	functor		

□

6.3.1 Generación de código para variables

Al encontrar una variable, la instrucción WAM que debe generarse depende de varios factores: si está en una consulta, o en un hecho, si es la primera aparición, si está dentro de una estructura, etc.. Hay ocho instrucciones especializadas que corresponden al código generado para una variable, estas son: `put_variable`, `put_value`, `set_variable`, `set_value`, `get_variable`, `get_value`, `unify_variable`, `unify_value`. En el momento de encontrar una variable, el compilador genera la instrucción correcta siguiendo el algoritmo indicado en el cuadro de la tabla 6.2:

Dentro de:	1ra. aparición	en estructura	Instrucción
consulta	si	no	`put_variable Vn Ai`
consulta	si	si	`set_variable Vn`
consulta	no	no	`put_value Vn Ai`
consulta	no	si	`set_value Vn`
hecho	si	no	`get_variable Vn Ai`
hecho	si	si	`unify_variable Vn`
hecho	no	no	`get_value Vn Ai`
hecho	no	si	`unify_value Vn`

Tabla 6.2: Instrucciones especializadas para variables

6.3.2 Generación de código para constantes

El tratamiento de las constantes es bastante más sencillo, hay cuatro instruciones especializadas: `put_const`, `set_const`, `get_const`, `unify_const`. La tabla 6.3 muestra como obtiene el compilador la instrucción adecuada.

Dentro de:	en estructura	Instrucción
consulta	no	`put_constant c Ai`
consulta	si	`set_constant c`
hecho	no	`get_constant c Ai`
hecho	si	`unify_constant c`

Tabla 6.3: Instrucciones especializadas para constantes

A continuación se muestra el código generado para un programa que sólo contiene variables y constantes.

Ejemplo 6.3.2 :

```
p(X,X) :- r(X,X).
r(a,Y).
 :- p(U,V).
```

```
p/2:  just_me              | p
      allocate             | (
      get_variable X1 A1   | X,
      get_value X1 A2      | X)←
      put_value X1 A1      | r(X
      put_value X1 A2      | ,X
      call r/0             | )
      deallocate           | .
      proceed              |
```

```
r/2:  just_me              | r
      allocate             | (
      get_constant a A1    | a,
      get_variable X1 A2   | Y
      deallocate           | )
      proceed              | .
  ?:  allocate             | ← p(
      put_variable X1 A1   | U,
      put_variable X2 A2   | V
      call p/0             | ).
      deallocate           |
      end_query            |
```

□

La generación de código para estructuras es un poco más complicada. En lo que sigue se expondrá como traducir términos a instrucciones WAM, y se utilizará basicamente el enfoque de Hassan Aït-Kaci [Aït91].

6.3.3 Compilación de términos que pertenecen a una consulta

El código de una consulta además de invocar a las cláusulas correspondientes, como ya se vió en la sección 6.1, debe preparar los registros y la memoria para que la ejecución del código de la cabeza de una cláusula encuentre todas las estructuras que tiene que unificar. En las secciones anteriores se mostró como generar código para variables y constantes, en esta sección se mostrará como generar código para estructuras que pueden estar anidadas. Se verá en detalle la utilización de las instrucciones `put_structure`, `set_variable`, `set_value`, y `set_const`.

A continuación se verá como traducir un término T que pertenece a una consulta a una secuencia de instrucciones WAM. La ejecución de esta secuencia deberá construir la representación del término T en el HEAP.

Como se vió en la sección anterior una estructura con functor f/n estará formada por $n+1$ celdas contiguas de memoria, más lo necesario para representar los n sub-términos. De esta forma, para alojar en la dirección d del HEAP la estructura $f(t_1, \ldots, t_n)$, en la dirección d estará la dirección de donde está alojado el símbolo del functor, y en las n celdas siguientes, la dirección de la representación de los n subtérminos de f/n, *que tienen que heber sido colocados en el* HEAP *antes*. Esto es, al generar la representación en el HEAP de una estructura con subestructuras, primero hay que almacenar las subestructuras, y luego la estructura podrá referenciar a los subtérminos.

A fin de almacenar temporariamente apuntadores a términos y subtérminos durante la compilación, se utilizarán registros de términos. Se asociará un registro a cada elemento del término (functor, variable o constante). Por ejemplo, en el término `p(Z,h(Z,W),f(W))`, los registros son asignados de la siguiente forma:

X_1= `p(`X_2`,`X_3`,`X_4`)`
X_2= `Z`
X_3= `h(`X_2`,`X_5`)`
X_4= `f(`X_5`)`
X_5= `W`

De esta forma, un término es visto como una secuencia de ecuaciones X_i = Variable, o $X_i = f(X_{p_1}, \ldots, X_{p_n})$ donde los X_{pj} son registros distintos a X_i. Esto se llamará *forma plana* del término.

Al ejecutarse el código de un término que pertenece a una consulta, se construye en el HEAP la representación del término. Pero antes de construir el término, deberá haberse construído con anterioridad los subtérminos, para poder hacer una referencia a ellos. Por lo tanto, es necesario ordenar la secuencia de registros de la forma plana de tal forma que *ningún registro aparezca en la parte derecha, si no fue usado previamente en la parte izquierda.* Esta forma será llamada **forma plana de términos que pertenecen a una consulta (FPC)**.

La FPC del término del ejemplo anterior es:

X_2= `Z`
X_5= `W`
X_3= `h(`X_2`,`X_5`)`
X_4= `f(`X_5`)`
X_1= `p(`X_2`,`X_3`,`X_4`)`

A partir de esta representación, es posible "olvidar" los nombres externos de las variables, y referirse a ellas por su registro X asociado. De esta forma la FPC queda sólo con:

X_3= `h(`X_2`,`X_5`)`
X_4= `f(`X_5`)`
X_1= `p(`X_2`,`X_3`,`X_4`)`

donde X_2 es la variable `Z`, y X_5 la variable `W`.

Una vez obtenida la FPC de un término, ya se está muy cerca de obtener el código WAM correspondiente. Obsérvese que en la FPC existen tres clases de items a procesar:

1. Un registro asociado a un functor, como es el caso de X_3. Esto implicará en ejecución, apilar en el heap el functor y copiarlo al registro correspondiente. Por lo tanto la instrucción a generar es `put_structure`.

2. Un registro no encontrado previamente, como X_2, o X_5. En este caso se tratará necesariamente de una variable, y por lo tanto las instrucciones a generar serán `set_variable`, o `set_value`, según lo visto antes para variables.

3. Un registro ya encontrado antes, que en ejecución corresponderá a copiar en el HEAP lo que ya existe en el registro. La instrucción a generar es `set_value`.

En la figura 6.15 se encuentra la secuencia de instrucciones que se desprende de la FPC de `p(Z,h(Z,W),f(W))`, se muestra además el elemento que generó cada instrucción y los nombres externos de las variables.

`put_structure h/2` X_3	`h`	`h`
`set_variable` X_2	(X_2,	(Z,
`set_variable` X_5	X_5)	W)
`put_structure f/1` X_4	`f`	`f`
`set_value` X_5	(X_5)	(W)
`put_structure p/3` X_1	`p`	`p`
`set_value` X_2	(X_2,	(Z,
`set_value` X_3	X_3,	...,
`set_value` X_4	X_4)	...)

Figura 6.15: Código WAM de `p(Z,h(Z,W),f(W))` si pertenece a una consulta

La figura 6.16 muestra la representación en el HEAP generada a partir de la ejecución (en forma secuencial) de las instrucciones de la figura 6.15. En ella se asume que el HEAP se comienza a utilizar en la dirección 101.

	(clase)	(dirección)
	⋮	⋮
101	functor	1
102	referencia	102
103	referencia	103
104	functor	4
105	referencia	103
106	functor	5
107	referencia	102
108	estructura	101
109	estructura	104
	⋮	⋮

	Identificadores
1	h/2
4	f/1
5	p/3

Figura 6.16: HEAP con `p(Z,h(Z,W),f(W))` que pertenece a una consulta

A continuación se verá entonces como compilar un término que está en la cabeza de una regla, o en un hecho.

6.3.4 Compilación de términos que pertenecen a hechos

La compilación de un término que pertenece a un hecho será muy similar a la de las consultas. Sin embargo, previamente a la ejecución del código WAM de un hecho, necesariamente debió ejecutarse el código de una consulta. Por lo tanto, la consulta previamente ejecutada puede haber dejado:

1. una estructura sin variables, que el código del hecho deberá intentar unificar simplemente **leyendo** del HEAP;
2. variables sin ligar, que el código del hecho deberá unificar **escribiendo** en el HEAP la nueva estructura.

Esto implica que al ejecutarse un hecho se podrán tener dos modos: uno de lectura (read) y otro de escritura (write).

Igual que antes, un término de un hecho deberá traducirse a una secuencia de instrucciones WAM. En este caso se mostrará en detalle el uso de las siguiente instrucciones: `get_structure`, `unify_variable`, `unify_value`, y `unify_const`.

Los términos de consultas necesitaban ordenarse en FPC, a fin de que los subtérminos sean apilados en el HEAP primero. En el caso de los términos que pertenecen a hechos, la situación es distinta, porque la información a unificar ya está disponible en el HEAP. Por lo tanto la **forma plana de un término que pertenece a un hecho** (FPH) estará ordenada de afuera hacia adentro del término (top-down).

Por ejemplo la FPH del siguiente término `p(f(X),h(Y,f(a)),Y)` es:

$X_1 = \mathrm{p}(X_2,X_3,X_4)$
$X_2 = \mathrm{f}(X_5)$
$X_3 = \mathrm{h}(X_4,X_6)$
$X_6 = \mathrm{f}(\mathrm{a})$

Al igual que en un término de consulta, la forma plana nos permite obtener directamente las instrucciones WAM:

1. `get_estructure f/n` X_j, en el caso de encontrar un registro asociado a un functor.
2. `unify_variable` X_j, en el caso de una variable que aparece por primera vez.

3. `unify_value` X_j, en el caso de un registro ya encontrado antes, o una variable que aparece por segunda vez

4. `unify_constant c`, en el caso de una constante. [2]

Siguiendo este criterio, el código WAM correspondiente al término anterior se muestra en la figura 6.17:

`get_structure p/3` X_1	`p`
`unify_variable` X_2	`(`X_2
`unify_variable` X_3	X_3
`unify_variable` X_4	`Y)`
`get_structure f/1` X_2	`f`
`unify_variable` X_5	`(X)`
`get_structure h/2` X_3	`h`
`unify_value` X_4	`(Y,`
`unify_variable` X_6	X_6`)`
`get_structure f/1` X_6	`f`
`unify_constant a`	`(a)`

Figura 6.17: Código WAM de `p(f(X),h(Y,f(a)),Y)` si pertenece a un hecho

Si se ejecuta el código del término de la consulta de la figura 6.15 (`p(Z,h(Z,W),f(W))`) e inmediatamente a continuación se ejecuta el código del término del hecho de la figura 6.17, entonces el HEAP quedará modificado como indica la figura 6.18.

Si se observa el código de la consulta, la variable `Z` quedó alojada en la dirección del HEAP 102, mientras que la variable `W` quedó en la dirección 103. Siguiendo las referencias del HEAP puede verse que después de la ejecución del código anterior, se obtuvo el resultado esperado. Esto es, `Z = f(f(a))` y `W=f(a)`.

[2] En el libro de Aït-Kaci, primero se define a las constantes como una estructura de aridad 0, y luego como una optimización se introducen instrucciones especiales para constantes. El enfoque aquí será tratar de introducir todos los conceptos tal cuál quedarán definitivos.

	(clase)	(dirección)
	⋮	⋮
101	functor	4
102	referencia	111
103	estructura	114
104	functor	3
105	referencia	103
106	functor	2
107	referencia	102
108	estructura	101
109	estructura	104
110	estructura	111
111	functor	3
112	referencia	103
113	estructura	114
114	functor	3
115	constante	5
	⋮	⋮

	Identificadores
2	p/3
3	f/1
4	h/2
5	a/0

Figura 6.18: HEAP después de ejecutar `p(f(X),h(Y,f(a)),Y)` sobre la figura 6.16

6.3.5 Algoritmo de compilación de términos

Las variables y constantes se compilan siguiendo lo desarrollado en las secciones 6.3.1 y 6.3.2 En el caso de las estructuras, el algoritmo para su compilación será básicamente el siguiente: traduzco los subtérminos, generando la forma plana correspondiente (según sea hecho o consulta) y luego al finalizar como conozco la aridad, y ya procesé los subtérminos, genero el código para el término en sí. La tabla de símbolos temporaria es utilizada para absorber las diferencias entre los hechos y las consultas. Cada vez que se compila un subtérmino, la tabla de símbolos es modificada para que respete el orden de la forma plana. De esta forma, un recorrido secuencial de la tabla de símbolos, será suficiente para generar el código WAM. A continuación se encuentra el algoritmo para compilar un término.

```
ALGORITMO Compilar termino
Si es una variable o constante, utilizo los algoritmos anteriores
Si es una estructura se realiza lo siguiente:
      * Si esta en un hecho generar 'get_structure', sino 'put_structure'
      * Aridad:=0;
      * REPETIR HASTA QUE se acaben los parametros del functor
           a. Aridad:=Aridad + 1;
           b. Llamo recursivamente a Compilar termino con cada subtermino
           c. Acomodo la tabla de simbolos para que quede la forma plana:
               c.1 SI es una consulta y el subtermino procesado es un functor:
                   ENTONCES generar la forma plana de terminos de consulta
                   i) muevo todas las celdas que de la tabla de simbolos que
                      corresponden al subfunctor a una posicion imediatamente
                      antes que el functor que se esta procesando.
                   ii) creo una nueva celda apuntando al subfunctor.
               c.2 SI es un hecho, y he procesado alguna subestructura
                   ENTONCES debo generar la forma plana de terminos en hechos
                   SI el parametro fue un subfunctor
                   ENTONCES inserto una celda referenciando al registro del
                          subfunctor debajo de la celda del anterior parametro
                   EN CASO CONTRARIO (variable o constante) muevo la celda del
                         parametro debajo de la celda  del anterior parametro.
        FIN REPETIR
      * Actualizo en la tabla de simbolos la aridad del functor
FIN ALGORITMO Compilar Termino
```

6.3.6 Variables anónimas

Las variables anónimas denotadas "_" y aquellas apariciones únicas de una variable que no es el parámetro de un predicado, pueden tratarse como casos especiales a fin de optimizar el tiempo en una consulta, y utilizar menos espacio en el HEAP.

Considérese el programa del ejemplo 6.3.3, una variable sin nombre puede estar en una consulta, en un hecho, y además ser un parámetro o estar dentro de una estructura anidada. La figura 6.19 muestra el código WAM del programa del ejemplo 6.3.3, puede verse que los diferentes casos de apariciones de variables sin nombre, tienen un tratamiento diferente.

Ejemplo 6.3.3 :

```
v(a,_,f(_))
 :- v(_,g(_),X)
```

□

v/4:	`just_me`	`v(`
	`allocate 0`	
	`get_constant a` A_1	`a,_`
	`get_structure f` A_3	`f(`
	`unify_void`	`_)`
	`deallocate`	
	`proceed`	`).`
?:	`allocate 0`	`v(`
	`put_variable_temp` A_1	`_`
	`put_structure g` A_2	`g(`
	`set_void`	`_),`
	`put_temp_var` X_2 A_3	`X`
	`call v/4`	`)`
	`deallocate`	`.`
	`end_query`	

Figura 6.19: Código WAM del ejemplo 6.3.3

La generación de instrucciones sigue el siguiente criterio: Si la variable se encuentra en un hecho (o cabeza de cláusula) y dentro de una estructura, entonces se genera la instrucción `unify_void`, y en el caso que no esté dentro de una estructura, no se genera ninguna instrucción. Si la variable está en una consulta y no está anidada en un estructura, entonces se genera la instrucción `put_temp_var`; en caso contrario se utiliza la instrucción `set_void`.

6.4 Backtracking

Cuando un predicado P tiene más de una definición (*i.e.*, existe un conjunto de cláusulas p_i para definir a P), entonces el conjunto de cláusulas que definen a P es tomado como una disyunción. Esto es, una consulta P tendrá éxito, si al menos una de las cláusulas que definen a P, tiene éxito. Si al intentar probar una de las definiciones de P, esta falla, entonces se debe intentar probar con otra, y asi sucesivamente. A este mecanismo se lo denomina *backtraking*.

En el conjunto de instrucciones de la máquina abstracta, hay tres instruciones especializadas en manejar el backtraking. Éstas son: `try_me_else, retry_me_else` y `trust_me.`

Ejemplo 6.4.1 : El programa que está a continuación posee tres cláusulas que definen al predicado `p/1`. El código WAM de la primera cláusula comenzará con la instrucción `try_me_else L1` que debe interpretarse como "pruebe ésta cláusula y sinó pruebe con la que está en la dirección $L1$". El código de la segunda cláusula comenzarán con `retry_me_else L`, y el de la última con `trust_me`.

```
p(a).
p(b).
p(c).
```

```
p/1:  try_me_else L1
      allocate 0
      get_const a A1
      deallocate
      proceed
L1:   retry_me_else L2
      allocate 0
      get_const b A1
      deallocate
      proceed
L2:   trust_me
      allocate 0
      get_const c A1
      deallocate
      proceed
```

□

6.4.1 Compilación

Las cláusulas que definen un predicado pueden estar intercaladas con otras cláusulas, e incluso estar en diferentes archivos. Por lo tanto cada vez que encuentre una nueva definición de un predicado, el compilador deberá modificar el código compilado para colocar la instrucción de backtracking que corresponda. Para esto se utiliza información almacenada en la tabla de símbolos permanente (tabla de identificadores). Cada vez que se compila una cláusula que es la primera que define a un predicado, su código es almacenado en la memoria encabezado por la instrucción `just_me`, para indicar que el predicado tiene una sola definición. Esta instrucción no estaba incluída en el conjunto de instrucciones original de la WAM, fue incluída sólo con el propósito anterior.

El proceso de compilación utiliza la siguiente estrategia para generar la instrucción que corresponda al encabezado de cada cláusula. En la tabla de símbolos temporaria, toda cláusula tiene como primera instrucción `just_me`. Al generar el código WAM, si es la primera aparición de una cláusula para definir el predicado, entonces se deja `just_me`, y se actualiza la información de la tabla de símbolos permanente. Pero si ya había una definición anterior de este predicado (esto es, ya figuraba en la tabla de símbolos permanente como predicado definido), entonces se debe corregir su código. Pueden ocurrir dos cosas: (1) que sea la segunda cláusula en aparecer, entonces se reemplaza la instrucción `just_me` por `try_me_else`, o (2) que ya existan dos o más cláusulas que lo definen, y entonces se reemplaza en la última, `trust_me` por `retry_me_else`. En ambos casos la primera instrucción del código de la nueva cláusula es `trust_me`.

Como ya se vió en la sección anterior, en la tabla de símbolos permanente se posee junto con el identificador del predicado, la dirección del comienzo de la última cláusula que define al predicado. Estas dos direcciones son necesarias para ubicar el lugar del código donde se realizarán los cambios que se explicaron en el parrafo anterior.

6.4.2 Puntos de elección

Considérese el siguiente ejemplo:

Ejemplo 6.4.2 :

```
a(X) :- b(X),c(X).
b(X) :- e(X).
e(X) :- f(X).
e(X) :- g(X).
c(1).
f(2).
f(3).
g(1).
```

□

En el programa del ejemplo 6.4.2, hay cláusulas que pueden probarse de diferentes maneras, como `e/1` y `f/1`. Cada vez que se elige una de estas cláusulas *se genera un punto de elección, al cual se debe retornar cuando se desee hacer backtracking.* Por lo tanto, además de los registros de entornos que se almacenan en el STACK, se debe tener una pila de puntos de elección (ver figura 6.20).

Siguiendo el ejemplo 6.4.2, al efectuar la consulta `a(X)`, se apila el entorno de `a(X)`, luego el entorno de `b(X)`, y al llegar a `e(X)` hay que "elegir" una de las dos formas de probarla, y por lo tanto hay un punto de elección. Una vez elegida la cláusula a utilizar, se genera un punto de elección, y se apila entonces el entorno de `e(X)`. Supóngase que se eligió la primer cláusula, entonces hay que probar `f(X)`, lo cual nos genera otro punto de elección. Por lo tanto se necesita también apilar los puntos de elección, lo cuál equivaldría a tener la siguiente situación:

Pila de Entornos	
Entorno de `a(X)`	
Entorno de `b(X)`	
Entorno de `e(X)`	
Entorno de `f(X)`	← E
⋮	

Pila de Puntos de Elección	
Elección de `e(X)`	
Elección de `f(X)`	← B
⋮	

La prueba de `f(X)` tiene éxito con `X=2`, entonces se desapilarían los entornos hasta llegar al de `a(X)`:

Pila de Entornos	
Entorno de `a(X)`	
Entorno de `c(X)`	← E
⋮	

Pila de Puntos de Elección	
Elección de `e(X)`	
Elección de `f(X)`	← B
⋮	

Luego se llama a `c(2)` que falla, y por lo tanto habría que ir a efectuar una nueva elección de `f(X)`. Pero como puede verse en la pila de entornos, los entornos de `b(X)` y `e(X)` se perdieron, ya que fueron sobreescritos por el de `c(X)`.

Por lo tanto **cada punto de elección debe proteger a los entornos**. Para lograr esto, tanto los puntos de elección como los entornos, son almacenados en la misma pila. De esta forma los puntos de elección sirven de protección a los entornos que están bajo ellos, permitiendo que en caso de necesidad sean recuperados y no puedan ser sobreescritos por nadie. En el caso que no sea necesario utilizar el punto de elección, este se desapila liberando además la memoria de los entornos que estaban protegidos bajo él. Por lo tanto la pila "AND" de entornos y la pila "OR" de puntos de elección se mantendrán en la misma pila: el STACK. De esta forma el STACK contiene (en una especie de simbiosis) dos tipos de registros con información diferente; y posee por lo tanto dos punteros uno al tope de cada tipo de registro.

Tanto los entornos como los puntos de elección están en la misma pila pero con tamaños diferentes, y obviamente con diferente información. El registro global E apuntará al último registro de entornos, y el registro global B al último registro de backtraking (punto de elección).

En la figura 6.20 se muestra el contenido de un punto de elección. A medida que se desarrollen los temas, se irá explicando el significado de cada elemento que está almacenado en ellos.

B:	cantidad N de parámetros del predicado
B+1:	parámetro 1
⋮	⋮
B+N:	parámetro N
B+N+1:	CE (continuation environment)
B+N+2:	CP (continuation pointer)
B+N+3:	B (valor anterior de B)
B+N+4:	BP (próxima clausula)
B+N+5:	TR (puntero al TRAIL)
B+N+6:	H (puntero al HEAP)
B+N+7:	B0 (registro del cut)

Figura 6.20: Estructura y contenido del punto de elección

Durante la ejecución del ejemplo 6.4.2 el STACK se comporpotará de la siguiente secuencia:

Entorno de `a(X)`	
Entorno de `b(X)`	
Punto de elección de `e(X)`	
Entorno de `e(X)`	
Punto de elección de `f(X)`	← B
Entorno de `f(X)`	← E
⋮	

Entorno de `a(X)`	(protegido)
Entorno de `b(X)`	(protegido)
Punto de elección de `e(X)`	(protegido)
Entorno de `e(X)`	(protegido)
Punto de elección de `f(X)`	← B
Entorno de `c(X)`	← E
⋮	

6.4.3 Manejo del STACK

Las instruciones especializadas para manejar los puntos de elección son: `try_me_else`, `retry_me_else` y `trust_me`. A continuación figura el código de todas ellas, y de la operación `BackTrack`. Además ahora es posible mostrar una versión cercana a la final del código de `call` y `allocate`.

La operación `BackTrack` es invocada desde muchas instrucciones WAM, al producirse una falla. Su objetivo ante la falla, es ver si existe un punto de elección en el STACK a fin de probar con otro camino en el árbol de derivación. Si existe un punto de elección apilado, entonces modfica el registro P, y la ejecución continuará en la cláusula para la que hay todavía otra opción.

```
PROCEDURE call(Predicado) {provisoria}
 IF definido(Predicado)        { Si esta definido el predicado }
 THEN   CP:=P+1                       { calculo el punto de retorno }
        Aridad:=Aridad(Predicado)     { guardo la aridad  para try_me_else}
```

```
        P:=Direccion(Predicado)      { salto a la direccion del predicado }
 ELSE Backtrack;                { Sino hago backtraking }
END call;

PROCEDURE allocate(N)
 IF (E = StackVacio) AND (B = StackVacio)         { El valor del nuevo E }
 THEN NuevoE := PrimeroStack                      { depende de la configuracion }
 ELSE IF E > B THEN NuevoE:= E + STACK[E+2] + 3;  { del STACK de acuerdo a los }
               ELSE NuevoE:= B + STACK[B] + 8;    { valores de E y B. }
 STACK[NuevoE]:=E;               { Guardo el E anterior }
 STACK[NuevoE+1]:=CP;            { el punto de retorno  }
 STACK[NuevoE+2]:=N;             { y la cantidad de variables permanentes }
 E:=NuevoE;                      { Por ultimo modifivo E }
END allocate

PROCEDURE try_me_else L;
 IF (E = StackVacio) AND (B = StackVacio)
    THEN NuevoB:=PrimeroStack
    ELSE IF E > B THEN NuevoB:= E + LongitudRegistroE
                  ELSE NuevoB:= B + LongitudRegistroB;
 N:=Aridad; {Aridad es una variable global cargada en call}
 STACK[NuevoB]:=N;
 FOR i:=1 TO N DO STACK[NuevoB+i]:=A[i];
 STACK[NuevoB+N+1]:=E;
 STACK[NuevoB+N+2]:=CP;
 STACK[NuevoB+N+3]:=B;
 STACK[NuevoB+N+4]:=L;
 STACK[NuevoB+N+5]:=TR;
 STACK[NuevoB+N+6]:=H;
 STACK[NuevoB+N+7]:=B0;
 B:=NuevoB;
 HB:=H;
END try_me_else;

PROCEDURE retry_me_else L;
  {Uso el mismo espacio del STACK pero con distinta informaci\'on}
```

```
 N:=STACK[B];                          {obtengo la aridad}
 FOR i:=1 TO N DO A[i]:=STACK[B+i];  {reestablesco los registros Ai}
 E:=STACK[B+N+1];                      {reestablesco E}
 CP:=STACK[B+N+2];                     {reestablesco CP}
 STACK[B+N+4]:=L;                      {guardo la nueva etiqueta L}
 Unwind_trail(STACK[B+N+5],TR)
 TR:=STACK[B+N+5];                     {reestablesco TR}
 H:=STACK[B+N+6];                      {reestablesco H}
 HB:=H;                                {fijo HB}
END retry_me_else

PROCEDURE trust_me
  {Uso el mismo espacio del STACK pero con distinta informaci\'on
  Como ya no habra mas backtracking, entonces de actualiza B con el guardado}
BEGIN
 N:=STACK[B];                          {obtengo la aridad}
 FOR i:=1 TO N DO A[i]:=STACK[B+i];  {reestablesco los registros Ai}
 E:=STACK[B+N+1];                      {reestablesco E}
 CP:=STACK[B+N+2];                     {reestablesco CP}
 STACK[B+N+4]:=L;                      {guardo la nueva etiqueta L}
 Unwind_trail(STACK[B+N+5],TR)
 TR:=STACK[B+N+5];                     {reestablesco TR}
 B:=STACK[B+N+3];                      {reestablesco B}
 HB:=STACK[B+N+6];                     {reestablesco HB desde el nuevo B}
END trust_me

PROCEDURE BackTrack
  FALLA := FALSE
  IF B = STACK_VACIO
  THEN FALLA := TRUE   {si no hay mas puntos de eleccion entonces Falla}
  ELSE B0 := STACK[B+STACK[B]+7]; {recupero el B0}
       P  := STACK[B+STACK[B]+4]; {obtengo la direccion de la otra clausula}
END BackTrack
```

6.4.4 Trail

Cuando una cláusula es elegida entre todas las que definen un predicado para ver si satisface una consulta, su ejecución crea efectos secundarios **ligando las variables** que residen en el STACK y en el HEAP. Estos efectos deben deshacerse cuando se tiene que elegir otra cláusula durante el backtracking. Por lo tanto, debe guardarse la información de cuales variables deben volver a quedar sin ligar ("unbound") al hacer backtracking. Para realizar esta tarea, se utiliza un área de memoria llamada TRAIL (rastro), donde se almacenan las direcciones de memoria de aquellas variables que deben ser vueltas a "unbound". Obsérvese que el TRAIL también trabajará como una pila, y que se necesita un registro global (TR) que apunte al tope.

Es importante remarcar que no todas las ligaduras deben ser almacendas en el TRAIL. Sólo las ligaduras "condicionales", es decir una ligadura que afecta a una variable existente de antes de la creación del punto de elección (choice point) actual. Para determinar esto, se usará un nuevo registro global HB, que contiene el valor de H al momento de producirse el último punto de elección (en realidad puede prescindirse de él, porque el valor de H es guardado en el STACK, pero se lo mantendrá por cuestiones didácticas). Por lo tanto, sólo las variables ligadas del HEAP (o STACK) cuya dirección sea menor a HB (o B) necesitan ser guardadas en el TRAIL.

La operación `trail(d)` donde `d` es una dirección de memoria, indica que la variable en `d` debe volver a unbound en caso de backtracking. La llamada `trail(d)` se produce desde la operación `bind`. Luego la operación `unwind_trail(d1,d2)` llamada desde `retry_me_else` o desde `trust_me`, es la encargada de poner en "unbound" todas las variables cuya dirección esté entre TRAIL`[d1]` y TRAIL`[d2]`. De esta forma, al hacer backtracking, se deshacen las instanciaciones realizadas por otro punto de elección. La figura 6.21 muestra estas dos operaciones.

6.4.5 Cut "!"

En el reporte original de la WAM no hay ningún tratamiento para el cut. Lo que sigue es el desarrollo de Hassan Aït-Kaci que, según sus propias palabras, está extrapolado de lo esbozado en [Warren88], y las sugerencias de Dean Rosenzweig.

El operador de corte (cut) denotado con el símbolo "!", es un operador extra lógico, cuya semántica operacional es simplemente tener éxito con un efecto colateral sobre el

```
PROCEDURE trail (Dir:Direccion)
    IF (Dir < HB ) OR ((H < Dir) AND ( Dir < B ))
    THEN  trail[TR] := Dir
          TR := TR+1
END trail

PROCEDURE unwind_trail (D1,D2:Direccion)
    FOR i:=D1 TO D2-1 DO MEMORIA[ TRAIL[I] ] := (referencia,TRAIL[I])
END unwind_trail
```

Figura 6.21: Operaciones para manejo del trail

backtraking: una vez que fue ejecutado, hace que se "olvide" toda otra alternativa potencial del predicado P en donde se encuentra, así como también de los otros predicados que preceden al cut en el cuerpo del predicado P . En términos de la arquitectura WAM, el efecto es logrado descartando todos los puntos de elección que fueron creados después del punto de elección que estaba a la derecha antes de la llamada (call) del predicado que contiene el cut.

En realidad pueden considerarse dos tipos de cut: *shallow-cut* y *deep-cut*. Un shallow-cut está ubicado en el cuello de la cláusula, esto es inmediatamente despues del símbolo "`:-`", como en el siguiente ejemplo, "`p(X) :- !,r(X)`" (este tipo de cut también es llamado *neck-cut*). Un deep-cut se produce cuando está en otro lugar que no sea inmediatamente después de `:-`, como por ejemplo en "`p(X) :- r(X),!,q(X)`".

Ejemplo 6.4.3 : El siguiente programa muestra los efectos de la diferente ubicación del cut.

Programa:	Corrida:
p0(X) :- !.	?- p0(X).
p0(0).	X = _0038 ->;
p1(X) :- !,r(X).	no
p1(1).	?- p1(X).
p2(X) :- r(X),!.	X = a ->;
p2(2).	X = b ->;
p3(X,Y) :- r(X),!,q(Y).	X = c ->;
r(a).	X = d ->;
r(b).	no
r(c).	?- p2(X).
r(d).	X = a ->;
q(a).	no
q(b).	?- p3(X,Y).
	X = a
	Y = a ->;
	X = a
	Y = b ->;
	no
	?

□

El valor del punto de elección apropiado para regresar en el momento de un cut se mantiene en un registro B0 (llamado registro del cut). Es un B al cual regresar para acortar el backtracking.

Ejecutar un shallow-cut entonces se convierte simplemente en hacer B:=B0, y acomodar adecuadamente el TRAIL. El ejemplo 6.4.4 muestra un programa con un shallow-cut, y el código WAM generado.

Ejemplo 6.4.4 :

```
p :- !,r.
p :- q.
```

```
p/0:  try_me_else L
      allocate 0
      neck_cut
      call r/0
      deallocate
      proceed
L:    trust_me
      allocate 0
      call q/0
      deallocate
      proceed
```

□

La figura 6.22 muestra el código de las operaciones WAM `neck_cut` y `tidy_trail`, y la figura 6.23 la modificación de la instrucción `call`.

```
PROCEDURE Neck_cut
      IF B > B0
      THEN   B:=B0;
             IF B > StackVacio THEN tidy_trail;
END Neck_cut

PROCEDURE Tidy_trail
  i:=STACK[ B + STACK[B] + 5]; {Obtiene la posicion del TR anterior}
  WHILE i < TR DO
    IF (TRAIL[i] < HB) OR ((H < TRAIL[i]) AND (TRAIL[i] < B))
    THEN i:=i+1
    ELSE   TRAIL[i]:=TRAIL[TR-1];
           TR:=TR-1;
  END {while}
END Tidy_trail;
```

Figura 6.22: Definición de las instrucciones para el shallow-cut

```
PROCEDURE call(Predicado)
 IF definido(Predicado)        { Si esta definido el predicado }
 THEN   CP:=P+1                      { calculo el punto de retorno }
        Aridad:=Aridad(predicado)    { guardo la aridad para try_me_else}
        B0:=B;                       { guardo el B en B0 para un neck cut}
        P:=Direccion(Predicado)      { salto a la direccion del predicado }
 ELSE Backtrack;               { Sino hago backtraking }
END call;
```

Figura 6.23: Definición de la instrucción `call`

En el caso de un shallow-cut, el registro B0 es fijado por la instrucción `call` al llamar a la cláusula que contiene el cut. Como la instrucción `neck_cut` es ejecutada antes que cualquier otro `call`, entonces B0 funciona como una variable global que mantiene el mensaje desde el `call` hasta el `neck_cut`.

Pero en el caso de un deep-cut, antes de que se ejecute la instrucción del cut se llama a otros predicados, y cada `call` sobreescribe el registro B0 perdiéndose la información anterior. Es por esto que debería guardarse B0 en el STACK como un elemento más del entorno de la cláusula (igual que CP por ejemplo). Sin embargo sólo habrá que guardarlo para aquellas cláusulas que tienen un deep cut, y no para todas. Por lo tanto, en lugar de agrandar el entorno para que contenga a B0, se lo guardará como una "variable permanente". De esta forma, estará guardado en el STACK, y sólo ocupará lugar cuando haya un cut.

La situación sería análoga a tener un predicado `!(Y)`, donde `Y` es una variable permanente distinta a cualquier otra de la cláusula. Esta variable se aloja en el entorno en curso como cualquier otra variable permanente real, esto es, su desplazamiento es mayor al de las variables de las metas anteriores al "!" y menor al de las variables de las metas que siguen.

En el caso de un deep-cut se necesitan entonces dos intrucciones:
(1) `get_level Yn`, que debe ir inmediatamente despues del `allocate` de la cláusula que tiene el cut, a fin de poner en `Yn` el valor que en ese momento tenga B0. Donde Yn es una variable permanente que se genera en forma virtual (en realidad no existe) a fin de

poder usar su ubicación en el STACK para almacenar B0.
(2) `cut Yn`, que va en el lugar donde se encontró al cut (!) cuyo efecto es descartar todos los puntos de elección que estén después de lo indicado por `Yn`. Además actualiza el trail, modificando los vínculos incondicionales que haya hasta ese punto.

Por ejemplo, la cláusula `a :- b,c,!,d` se traduce a:

```
a/0:  just_me 0
      allocate 1
      get_level Y1
      call b/0
      call c/0
      cut Y1
      call d/0
      deallocate
      proceed
```

A continuación figura el código de las instrucciones `get_level` y `cut`:

```
PROCEDURE get_level Yn
BEGIN
   STACK[E+n+1]:=B0;
END get_level
```

```
PROCEDURE cut Yn
BEGIN
   IF B>STACK[E+n+1] THEN
   BEGIN
         B:=STACK[E+n+1];
         IF B > StackVacio THEN tidy_trail;
   END
END cut
```

Figura 6.24: Definición de las instrucciones para el deep-cut

6.5 Listas

Las listas son un tipo de dato muy utilizado en la programación en lógica. Una lista tiene la siguiente definición recursiva: una lista es vacía, o es la concatenación de un elemento con una lista. En Prolog una lista vacía se nota "`[ ]`", y una lista no vacía con "`[elemento|<lista>]`". Por ejemplo la lista que tiene los elementos `a,b,c` se nota `[a | [b | [c | [ ] ]]]`. Prolog también soporta una notación abreviada, que se indica de la siguente forma: `[a,b,c]`. También es posible escribir cualquier combinación de ambas notaciones, como por ejemplo `[a,b|[c]]`.

Los elementos de una lista pueden ser constantes, variables, estructuras, y también listas. Por ejemplo la siguiente lista contiene elementos de todos estos tipos:

```
[a, f(a) , f(g(b),X), [1,2,3] , f([],[a,b]) ] .
```

6.5.1 Representación

La lista vacía "`[ ]`" es una constante, y por lo tanto se la representará como tal. Siguiendo la definición recursiva, una lista no vacía puede pensarse como un functor que tiene dos subtérminos, un elemento y una lista. Un functor de aridad 2 se representaba en el HEAP utilizando 3 celdas, la primera para el nombre del functor, y las dos siguientes para los dos subtérminos del functor. En el caso de las listas no hay un nombre de functor, y por lo tanto es posible ahorrar una celda en la representación.

De esta forma, una lista se representará en el HEAP con dos celdas consecutivas. La primera para el primer elemento, y la segunda para indicar la posición del HEAP donde se representa el resto de la lista. En el caso de los elementos que no son listas, una celda del HEAP tendrá la forma que ya fue descripta. Para las listas se agrega una nueva etiqueta `lista` que indica que la celda del HEAP contiene la dirección del comienzo de una lista.

Ejemplo 6.5.1 : A continuación se muestra la representación en el HEAP de la lista `[a,b,c]` (equivalente a `[a|[b|[c|[]]]]`) que se encuentra a partir de la dirección 102. Si el registro A_1 estuviera ligado a la lista anterior, este tendría el siguiente contenido A_1=`(lista,102)`.

	(clase)	(dirección)
⋮	⋮	⋮
102	constante	a
103	lista	104
104	constante	b
105	lista	106
106	constante	c
107	constante	nil
⋮	⋮	⋮

□

"Debido a que las listas son un tipo de dato privilegiado en la programación en lógica, vale la pena introducir instrucciones especiales para su manejo" [Aït91]. Las consultas generarán la instrucción `put_list` y los hechos `get_list`. Además, otras funciones como "unify" manejarán datos de tipo lista. La figura 6.25 muestra las dos intrucciones para listas.

```
PROCEDURE put_list Ai
    Ai := (lista,H)
END put_list

PROCEDURE get_list Ai
    Dir:=defer(Ai)
    CASE MEMORIA[Dir] OF
    (referencia,_): HEAP[H]:=(lista,H+1);
                    bind(Dir,H); H:=H+1; Modo:=write;
         (lista,d): S := d; Modo:=read;
              ELSE: FALLA:=TRUE
    END CASE
    IF FALLA THEN BackTrack
END get_list
```

Figura 6.25: Instrucciones WAM para manejo de listas

Ejemplo 6.5.2 : Las figuras 6.26 y 6.27 muestran el código WAM del siguiente programa:

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
app([],Lista,Lista).
app([X|Resto],Lista,[X|RestoApp]) :- app(Resto,Lista,RestoApp).
```

□

member/2:	try_me_else M1	member
	allocate	(
	get_var X_1A_1	X,
	get_list A_2	[
	unify_value X_1	X|
	unify_void	_]
	deallocate	)
	proceed	.
M1:	trust_me	member
	allocate	(
	get_var X_1A_1	X,
	get_list A_2	[
	unify_void	_|
	unify_var X_3	Y])
	put_value X_1A_1	:-member(X,
	put_value X_3A_2	Y
	call member/2	
	deallocate	)
	proceed	.

Figura 6.26: Código de `member/2`

6.5.2 Compilación de listas

En el caso de una lista vacía, el compilador genera el código correspondiente a la constante`[]`. La compilación de una lista que está en notación recursiva, es muy parecida a

app/3	try_me_else A1	app
	allocate	(
	get_const $X_3 A_1$	[],
	get_var $X_1 A_2$	Lista,
	get_value $X_1 A_3$	Lista
	deallocate	)
	proceed	.
A1:	trust_me	app
	allocate	(
	get_list A_1	[
	unify_var X_2	X|
	unify_var X_3	Resto],
	get_var $X_4 A_2$	Lista,
	get_list A_3	[
	unify_value X_2	X|
	unify_var X_6	RestoApp]
	put_value $X_3 A_1$	:-app(Resto,
	put_value $X_4 A_2$	Lista,
	put_value $X_6 A_3$	RestoApp
	call app/3	
	deallocate	)
	proceed	.

Figura 6.27: Código de app/3

la de una estructura, ya que una lista puede pensarse como un functor de aridad 2, donde el primer subtérmino es un elemento y el segundo una lista. En el caso de una lista en notación abreviada es un poco diferente.

Debido a que hay dos notaciones para listas, y que además pueden estar mezcladas, la idea es que dos listas iguales, independientemente de su notación, generen la misma secuencia de instrucciones. Para esto, el compilador se encargará de absorver las diferencias y generar el mismo código.

El algoritmo que se muestra a continuación realiza el reconocimiento sintáctico y la generación de código WAM, para una lista con la notación recursiva. Obsérvese que sim-

plemente se genera código para `put_list` o `get_list`, y luego se analiza sintácticamente la estructura.

Algoritmo: CompilarLista
1. Generar la instrucción `get_list` o `put_list`
2. Compilar el primer elemento de la lista
3. GenerarFormaPlana
4. Saltear la barra
5. Compilar el resto de la lista utilizando este mismo algoritmo
6. GenerarFormaPlana

Dos listas iguales, independientemente de su notación, deben generar la misma secuencia de instrucciones. Para esto, el compilador siempre generará código como si se hubiera encontrado una lista en notación recursiva. En el caso de la notación abreviada que recorrerá los elementos de la lista, pero generará una secuencia de instrucciones idéntica a la de la notación recursiva.

El primer problema aparece con el fin de la lista. En la notación recursiva, siempre hay una lista vacía al final. En el caso de la notación abreviada, la lista vacía del final está implícita, y por lo tanto hay que incluirla explícitamente. Como la notación puede ser combinada, entonces puede ocurrir que la lista vacía esté o no al final. Para resolver esto, se realiza lo siguiente: si no se encontro una barra, y no hubo una lista vacía inmediatamente antes, entonces hay que generar código para lista vacía explícitamente. En caso contrario, el código se genera al encotrar la lista vacía.

El algoritmo completo para compilar una lista se encuentra a continuación. Como la lista vacía se considera como una constante, el algoritmo no necesita considerar el caso de lista vacías.

Algoritmo: ProcesarLista
1. Generar la instrucción `get_list` o `put_list`
2. Compilar el primer elemento de la lista
3. Mientras haya una coma: compilar un elemento de la lista
4. Si hay una barra: compilar el resto de la lista con este mismo algoritmo
5. Si no hubo lista vacía: incluir lista vacía al final.

6.6 Implementación de la máquina virtual

El conjunto de instrucciones WAM y la arquitectura de memoria descripta, define una máquina virtual sobre la cual se pueden ejecutar programas lógicos en código WAM, y realizar consultas al programa. Por lo tanto, es posible hacer una implementación de la máquina virtual sobre una máquina real, y ejecutar los programas lógicos en ese entorno. Para esto se debe compilar el programa lógico a código WAM, y luego ejecutarlo sobre dicha implementación de la máquina virtual.

Otra alternativa es implementar la WAM directamente sobre una máquina real, *i.e.*, definir e implementar las instrucciones para un procesador de una computadora particular; los programas correrían entonces como un programa más sobre esa plataforma. Esto último permitiría una mayor velocidad en la ejecución de los programas, pero los limitaría al procesador para el cuál fueron implementadas las instrucciones WAM. También es posible construir un nuevo procesador que permita ejecutar directamente el conjunto de instrucciones.

El objetivo de esta sección es simplemente mostrar cómo se puede implementar una máquina virtual para ejecutar los programas lógicos.

6.6.1 Respuestas a una consulta

Una vez compilado un programa y almacenado en la memoria, la máquina virtual deberá responder a las consultas que se le hagan.

Ejemplo 6.6.1 :

```
p(g(a),f(a))
lista([1,2,3])
num(1)
num(2)
num(3)
```

□

Por ejemplo, utilizando el programa del ejemplo 6.6.1 las consultas

```
:- lista([1,2,3])
:- p(g(a),f(a))
```

tendrán éxito, mientras que las consultas

```
:- lista([1,2])
:- p(g,f)
```

fallarán.

Sin embargo, las consultas

```
:- lista(L)
:- p(X,Y)
```

tendrán éxito, ya que las variables de la consulta unifican con elementos de los predicados del programa. En estos casos, no sólo se quiere saber si la consulta tiene éxito, sino también cual es la sustitución de variables que lo permitió. En el caso anterior sería: `L=[1,2,3]` y `X=g(a) Y=f(a)` .

A continuación se muestra una serie de consultas y respuestas que devolvería la máquina virtual, habiendo compilado el programa del ejemplo 6.6.1. Obsérvese que cuando hay más de una respuesta posible, como en el caso de "`:- num(X)`", se permite consultar por todas ellas.

```
? p(g(X),f(X)).
  X = a
  SI.
? lista([X|Resto]).
  X = 1
  Resto = [2,3]
  SI.
? lista([4|X]).
  NO.
? lista([1,X,_]).
  X = 2
  SI.
? num(N).
```

```
  N = 1 ?‘Otra? ;
  N = 2 ?‘Otra? ;
  N = 3
  SI.
? lista([X|Resto]),num(X).
  X = 1
  Resto = [2,3]
  SI.
?
```

Para que la máquina virtual se comporte de la manera antes indicada, esto es, que responda con las sustituciones de las variables de la consulta, se extendió la WAM con dos nuevas instrucciones: `save_query_var` que guarda la posición de la variable para mostrar la respuesta al terminar la consulta, y `show_answer` que muestra la respuesta correspondiente. El ejemplo 6.6.2 muestra el código WAM de la consulta "`:- p(X,Y)`"

Ejemplo 6.6.2 :

`allocate 2`	`p(`
`save_query_var X` Y_1	`X`
`put_var_perm` A_1 Y_1	`,`
`save_query_var Y` Y_2	`Y`
`put_var_perm` A_2 Y_2	`).`
`call p/2`	
`show_answer X`	
`show_answer Y`	
`deallocate`	
`end_query`	

□

Como se explicó en la observación 6.2.1 al compilar un programa, o una consulta, las variables pasan a ser simplemente una dirección de memoria y su identificador desaparece como tal. Sin embargo, el identificador de las variables de una consulta debe recordarse a fin de mostrar la respuesta encontrada.

Al generar el código WAM de una consulta, cuando se detecta la primera aparición de una variable `X`, se guarda el identificador `X` en la tabla de identificadores, y se genera la instrucción "`save_query_var P` Y_n" donde Y_n es el lugar dentro del registro de entorno de la consulta, donde se ubicará la variable `X`, y `P` la posición del `X` en la tabla de identificadores. Cuando se termina de generar el código de la consulta, y antes de la instrucción `deallocate`, se genera una instrucción "`show_answer P`" para cada variable encontrada en la consulta.

La instrucción "`save_query_var Var` Y_n" posee dos parámetros, el primero (`Var`) es la posición en la tabla de identificadores de la variable, y el segundo (Y_n) es la ubicación de dicha variable en el registro de entorno de la consulta. Al ejecutarse la instrucción `save_query_var Var` Y_n, se calcula la dirección de memoria de Y_n, y se almacena dicho valor en la posición `Var` de la tabla de identificadores. De esta forma, la instrucción `show_answer Var`, tendrá la dirección del comienzo de la respuesta a mostrar. La figura 6.28 muestra el código de las nuevas instrucciones.

La instrucción `show_answer Var` se ejecuta después que terminó la consulta en forma exitosa. El parámetro `Var` indica la posición de la tabla de identificadores donde se encuentran: el identificador de la variable y la posición de memoria donde comienza la estructura a mostrar. Si la estructura a mostrar es una variable, muestra la dirección de memoria de la misma precedida del símbolo "_". Si es una constante, muestra el identificador de la misma. Los procedimientos auxiliares `mostrar_estructura` y `mostrar_lista_plana` se encargan de recorrer el HEAP y el STACK para reconstruir e imprimir la respuesta, en el caso de estructuras complejas o listas.

6.6.2 Intérprete

El intérprete de la WAM es básicamente un procesador virtual que tiene acceso a toda la estructura de memoria explicada antes. Cada vez que se quiere ejecutar un programa, se lo compila y es colocado en la memoria de la máquina virtual. Cuando se realiza una consulta, el intérprete compila la consulta y coloca su código en la memoria, a continuación del código WAM del programa. Si el código de la consulta fue cargado a partir de una dirección de comienzo llamada "SA", entonces el registro P es cargado con el valor de SA y se llama al procesador virtual para que ejecute la consulta.

```
PROCEDURE save_query_var (Var, Yn)
  PosMemoria:=DireccionAbsoluta(Yn);
  TablaIdentificadores[Var].direccion:=PosMemoria;
END save_query_var

PROCEDURE show_answer (Var)
  WRITE(TablaIdentificadores[Var].Identificador);
  WRITE(' = ');
  PosMemoria:=TablaIdentificadores[Var];
  PosMememoria:=Deref(PosMemoria);
  <Clase,Valor>:= MEMORIA[PosMemoria];
  CASE Clase OF
  constante:  WRITE(TablaIdentificadores[valor].identificador);
  referencia: WRITE('_',valor);
  estructura: Mostrar_estructura(Valor);
  lista:      BEGIN WRITE('['); Mostrar_lista_plana(Valor); WRITE(']'); END;
  END; {CASE}
  WRITELN;
END show_answer;
```

Figura 6.28: Nuevas instrucciones

El conjunto de instrucciones del procesador virtual es justamente el conjunto de instrucciones de la WAM, por lo tanto una vez que el control es tranferido a la primer instrucción de la consulta, comienza la ejecución de la misma. El procesador virtual tiene todas las definiciones de las instrucciones WAM y se encarga de ejecutar la instrucción que esté en la dirección que indica el registro P, hasta que se llegue a la instrucción `end_query` o hasta que un registro llamado `falla` quede en TRUE.

Si el registro `falla` quedó en TRUE, entonces la consulta no fue exitosa y la respuesta es NO. Si el registro `falla` quedó en FALSE, entonces la consulta tuvo éxito y el código del programa ya ha mostrado la unificación de las variables de la consulta, y sólo resta mostrar la respuesta que en este caso es SI.

Para una consulta puede haber más de una unificación posible para sus variables,

```
REPEAT
     InstruccionWAM:=CODE[P];
     P:=P+1;
     Ejecutar(InstruccionWAM);

     IF (InstruccionWAM = end_query) AND NOT Falla AND HayPuntosDeEleccion
     THEN { se interroga al usuario }
          WRITELN('Otra?')
          IF CaracterLeido = ';' THEN  BackTrack;
UNTIL Falla OR (InstruccionWAM = end_query);

IF NOT Falla
   THEN WRITELN(' SI.')
   ELSE WRITELN(' NO.')
```

Figura 6.29: Esquema del procesador virtual

esto es, puede haber más de una respuesta positiva. Por lo tanto, cuando la consulta es sin instanciar, y han quedado en el STACK puntos de elección sin resolver, el interprete interroga al usuario para ver si quiere otra respuesta, y de ser así, llama voluntariamente a la operación `backtrack` a fin de que se evalue otro camino posible en el árbol de derivación. La figura 6.29 muestra un esquema de la implementación del procesador virtual.

Una vez finalizada la consulta, todas las áreas de memoria y registros, son reinicializadas, a excepción del área de código donde sólo se desecha el área de memoria a partir de la cual se cargó la última consulta (*i.e.*, a partir de la dirección SA). Por lo tanto, lo único que perdura es el código del programa.

6.7 Conclusiones

En este capítulo se describió en detalle a la Máquina Abstracta de Warren, la cual se utilizará como punto de partida para la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. La ventaja de tener una máquina abstracta (MA) es que dado un programa P en un lenguaje $\mathcal{L}$, el programa P puede ser traducido a instrucciones de la MA, y luego ejecutar sobre la arquitectura de la máquina virtual dichas instrucciones. La MA puede implementarse como una máquina virtual, o directamente sobre una máquina real, o diseñarse un nuevo procesador con la arquitectura de la MA diseñada. En el caso de la WAM, estas tres alternativas han sido desarrolladas [VanRoy90].

También en este capítulo se desarrollaron técnicas de compilación para programas lógicos. Dichas técnicas permitieron la implementación de un compilador para PROLOG, el cuál será extendido en el capítulo siguiente para los programas lógicos rebatibles.

En el capítulo siguiente se definirá la nueva máquina abstracta que se llamará JAM (Justification Abstract Machine). La arquitectura de la JAM estará formada al igual que en la WAM por un conjunto de instrucciones, una estructura de memoria, y un conjunto de registros, que permitirán construir argumentos, contraargumentos y generar el árbol de dialéctica necesario para obtener una justificación. También se mostrará como compilar un programa lógico rebatible.

Capítulo 7

Una Máquina Abstracta para la Programación en Lógica Rebatible.

Uno de los objetivos perseguido en esta tesis, es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. En este capítulo se mostrará el desarrollo de una nueva máquina abstracta llamada JAM (Justification Abstract Machine). La JAM está diseñada como una extensión de la WAM, por lo tanto, todo lo desarrollado para la WAM se utiliza en esta nueva máquina. La JAM incorpora un conjunto de instrucciones, y agrega nuevas áreas de memoria para mantener argumentos, conjuntos de activación y el árbol de dialéctica.

7.1 Arquitectura de la JAM

La arquitectura de la JAM está formada al igual que en la WAM por un conjunto de intrucciones, una estructura de memoria, y un conjunto de registros. A continuación se enumerarán los nuevos elementos con los cuales se construyó la JAM, y en las secciones posteriores se explicarán en detalle cada uno de ellos.

7.1.1 Organización de la memoria

Además de las estructuras de memoria de la WAM, la JAM posee:

1. Una pila llamada LINEA que mantiene la linea de argumentación en curso (*i.e.*, una rama del árbol de dialéctica)
2. Una tabla de Hechos Temporarios (THT) donde estarán representados los argumentos.
3. Un nuevo HEAP llamado T-HEAP, para almacenar los hechos temporarios.
4. Una cola que almacena los conjuntos de activación de los argumentos.

7.1.2 Registros especiales

Al igual que en la WAM, el estado en curso de una computación en la JAM estará caracterizada por un conjunto de registros de la máquina. Además de los registros de la WAM, se agregan los indicados en la figura 7.1

IC	(Instruction Counter) apunta al tope del área de código.
`prueba_consistencia`	Indica si se está verificando consistencia.
`uso_cpe_invertidas`	Indica si se pueden utilizar CPE invertidas.
`derivación_rebatible`	Indica cuando una derivación deja de ser fuerte.
T	Apunta el tope del T-HEAP.
F	Apunta al tope de la THT.
`tope_linea`	Apunta al tope de LINEA.
`solo_cpe`	Indica que sólo se pueden usar CPE en las pruebas.
`padre`	Posee la posición del padre del hecho temporario.
`hubo_argumento`	Indica si hubo al menos un argumento.
`derrotador_propio`	Indica si el derrotador hallado es propio o no.
`hubo_bloqueo`	Indica si hubo un derrotador aceptable de bloqueo.
`hubo_inconsistencia`	Indica si no existe argumento por una inconsistencia.

Figura 7.1: Registros propios de la JAM

7.1.3 Conjunto de instrucciones propias de la JAM

La siguiente tabla muestra las instrucciones propias de la nueva máquina abstracta, indicando brevemente el significado de cada una. En las secciones siguientes se explicará en detalle, el uso y la implementación de cada una de estas instrucciones.

`prepare_call` A_1	prepara el código para usar el predicado `call/1`.
`fail_and_backtrack`	provoca una falla y hace backtracking.
`prove_consistency_else E`	marca el inicio de la verificación de consistencia.
`end_consistency_proof`	marca el fin de la verificación de consistencia.
`forbid_inv`	prohibe el uso de CPE inveritidas.
`allow_inv`	permite usar CPE invertidas.
`defeasible_just_me`	idem a just_me pero para CPR.
`defeasible_try_me_else L`	idem a try_me_else pero para CPR.
`defeasible_retry_me_else L`	idem a retry_me_else pero para CPR.
`defeasible_trust_me`	idem a trust_me pero para CPR.
`just_me_checking_consistency`	idem a just_me pero para CPE invertidas.
`try_me_checking_consistency_else L`	idem a try_me_else pero para CPE invertidas.
`retry_me_checking_consistency_else L`	idem a retry_me_else pero para CPE invertidas.
`trust_me_checking_consistency`	idem a trust_me pero para CPE invertidas.
`set_defeasible`	pone el registro `derivación_rebatible` en TRUE.
`save_subterms p/n Tc`	prepara para armar un hecho temporario. `p/n`
`save_fact p/n`	construye un hecho temporario `p/n`.
`unify_subterms p`	unifica términos de `p` del T-HEAP, con los A_i.
`prepare_to_defeat K`	prepara para hallar derrotadores de nivel `K`.
`load_subterms N,T`	carga los registros de parámetros.
`set_disagree`	almacena el punto de desacuerdo.
`not_circular`	verifica circularidad en la argumentación.
`not_less_specific`	verifica que sea derrotador.
`proper_defeat`	verifica que sea derrotador propio.
`blocking_defeat`	falla si hubo derrotadores de bloqueo.
`only_strong`	pone el registro `solo_cpe` en TRUE.
`stop_only_strong`	pone el registro `solo_cpe` en FALSE.

Instrucciones propias de la JAM.

Lo que resta del capítulo está organizado de la siguiente forma: Primero se introducirán algunos predicados y operadores predifinidos a la máquina abstracta. En la sección 7.3 se mostrarán las instrucciones JAM, y las nuevas áreas de memoria que permiten obtener un argumento para una consulta. Luego en la sección 7.4 se introducirán las instrucciones que permiten la generación del árbol de dialéctica para un argumento. En la sección 7.5 se mostrará como la JAM, implementa la comparación de argumentos utilizando el criterio

de especificidad. La sección 7.6 está dedicada a la implementación de la negación por falla en los PLR. Por último en la sección 7.7 se muestra como se obtiene la respuesta de una consulta en un PLR.

7.2 Predicados y operadores predefinidos

En Prolog, existen algunos predicados y operadores predefinidos, que no están incluidos en el reporte original de la WAM, y tampoco en el trabajo de Hassan Aït-Kaci. Tal es el caso del operador de negación por falla finita, el predicado `call/1`, y los predicados `true/0` y `fail/0`. Antes de comenzar con el desarrollo de la nueva máquina abstracta, se mostrará como extender la WAM para que incorpore los elementos citados como predefinidos en el lenguaje. El objetivo de esta extensión es doble, por un lado enriquecer a la WAM con estos predicados, y por otro lado simplificar el desarrollo de la JAM.

7.2.1 El predicado `true/0`

Los PLR permiten el uso de *presuposiciones* (*i.e.*, CPRs sin cuerpo). Para distiguir presuposiciones de hechos, se utiliza la sintaxis: "`p -< true`". El predicado `true/0` siempre da éxito, por lo tanto, es equivalente a tener el hecho "`true`" en el programa .

Una implementación posible sería incluir en todo PLR, a `true/0` como un hecho más. Sin embargo, esto quedaría en manos del programador, y su olvido significaría no poder utilizar ninguna presuposición. Es por esta razón, que se prefirió tratar a `true/0` como un predicado predefinido, incluyendo en todo programa el siguiente código WAM:

```
true/0:  proceed
```

El código anterior simplemente sigue la semántica del predicado `true/0`, esto es, tener éxito siempre. De esta forma, `true/0` estará presente en todo programa, y el programador no necesitará incluirlo como un hecho más.

Como el código del predicado `true` es nulo, otra alternativa sería que el compilador directamente no genere ninguna instrucción al encontrarse con una llamada al predicado `true/0`.

7.2.2 El predicado `fail/0`

El predicado `fail/0`, en contraposición con `true/0`, falla siempre. Es equivalente a un hecho que no está presente en un programa. Por la misma razón que antes, se prefirió incluirlo como un predicado predefinido para el programa, utilizando el siguiente código:

```
fail/0:  neck_cut
         fail_and_backtrack
```

Donde la nueva instrucción `fail_and_backtrack`, lo único que hace es fallar y llamar a backtracking, y la instrucción `neck_cut` previene de que se redefina `fail/0` y pueda dar éxito.

También podría hacerse que el compilador genere directamente la instrucción `fail_and_backtrack` en el código de la cláusula que llama a `fail/0`.

7.2.3 El predicado `call/1`

El predicado extra lógico `call/1` recibe por parámetro un término `P` que corresponde a un predicado, y efectúa la consulta `P`. Como se verá en las secciones siguientes, este predicado será de suma utilidad para la implementación de la JAM. Una llamada a `call(+P)` puede hacerse sólo si `P` es: una constante (predicado de aridad 0), un término `p/n`, o una variable que esta instanciada en alguno de los dos elementos anteriores. En el caso de la variable, si no está unificada a un predicado como en los dos primeros casos, `call(X)` fallará.

Obsérvese que una consulta o submeta "`q(p(f(X)))`" genera el siguiente código WAM:

```
put_structure f/1 X1
set_variable Y1
put_structure p/1 A1
set_value X1
call q/1
```

Esto es, antes de efecutar la llamada `call q/1`, ya se ha cargado en el HEAP la estructura del subtérmino `p(f(X))`, y el registro A_1 tiene la dirección del HEAP donde comienza dicha estructura.

Si en lugar de `q(p(f(X)))` se hace la consulta `call(p(f(X)))`, el código generado debe ser el mismo, con la diferencia que en lugar de llamar a `q/1` se llama a `call/1`.

```
put_structure f/1 X1
set_value Y1
put_structure p/1 A1
set_value X1
call call/1
```

Al ejecutar una submeta `call(P)`, se carga primero la estructura `P` en el HEAP, y luego se llama al predicado `call/1`. Por lo tanto el registro A_1 tiene la dirección de comienzo de la estructura `P` en el HEAP.

Las consideraciones anteriores muestran claramente como implementar un predicado predefinido `call/1`: utilizando el registro A_1 obtener el nombre del predicado a llamar y su aridad, cargar los registros de argumento utilizando la información del HEAP, y luego ejecutar una instrucción `call` al predicado correspondiente.

Por lo tanto, el código del predicado `call/1` debería ser:

```
call/1:  allocate 0
         prepare_call A1
         call ??
         deallocate
         proceed
```

Pero usando la instruccion `execute`, que no modifica el registro CP (continuation pointer), puede optimizarse el código a:

```
call/1:  prepare_call A1
         execute ??
```

El código del predicado predefinido `call/1` estará siempre en memoria, y debe cargarse antes que cualquier programa. El parámetro de la instrucción `execute` estará indefinido hasta el momento en que se ejecute. La instrucción `prepare_call` será la encargada de colocar el parámetro correcto para `execute`, y haber cargado todos los registros de argumentos que utilizará el predicado a llamar.

La instrucción `prepare_call` utiliza la dirección que tiene en el registro A_1 para obtener el nombre del predicado a llamar y su aridad. Si el predicado a llamar no está definido,

entonces falla. Una vez obtenida la aridad se cargan los registros de argumento, y luego se modifica el código del programa para colocar el parámetro correcto a la instrucción `execute` (que es la siguiente a `prepare_call`). La figura 7.2 muestra la implementación de esta nueva instrucción.

```
PROCEDURE prepare_call (A1)
     D:=deref(A1)
     CASE D OF
     <estructura,PosHeap>:  <fun,PosTablaIdent>:=HEAP[PosHeap];
                            IF Definido(predicado(posTablaIdent))
                            THEN Aridad:=ObtenerAridad(PosTablaIdent);
                                 FOR i:=1 TO Aridad DO Ai:=HEAP[PosHeap+i];
                                 CODE[PC+2]:=PosTablaIdent;
                            ELSE FALLA:=TRUE;
     <constante,PosTablaIdent>:IF Definido(predicado(posTablaIdent))
                               THEN CODE[PC+2]:=PosTablaIdent;
                               ELSE FALLA:=TRUE;
     ELSE: FALLA:=TRUE;
     END; {case}  IF FALLA THEN Backtrack;
END prepare_call;

PROCEDURE execute (Predicado);
 IF definido(Predicado)        { Si esta definido el predicado }
 THEN   Aridad:=Aridad(predicado);    { guardo la aridad para try_me_else}
        B0:=B;                        { guardo el B en B0 para un neck cut}
        P:=Direccion(Predicado);      { salto a la direccion del predicado }
 ELSE Backtrack;               { Sino hago backtraking }
END execute;
```

Figura 7.2: Implementación de las instrucciones `prepare_call` y `execute`

7.2.4 El operador de negación por falla de Prolog

El operador `not` de negación por falla de Prolog será de suma utilidad en el desarrollo de la nueva máquina abstracta, y es por esto que se lo incluirá como un operador predefinido. Como los PLR tienen definido su propio `not`, que tiene una semántica diferente a la negación por falla de Prolog, en esta tésis se usará "`naf`" para referirse al operador de negación por falla de Prolog, y "`not`" para el correspondiente a los PLR.

Obsérvese en primer lugar que es posible definir al operador `naf` como un predicado dentro de un programa, si se dispone de los predicados extra lógicos `call/1`, `!` (cut) y `fail/0`. El ejemplo 7.2.1 muestra una forma de definir el `naf`.

Ejemplo 7.2.1 :

```
naf(X) :- call(X),!,fail.
naf(_).
```

□

Si se compila el programa del ejemplo 7.2.1, se obtiene el siguiente código:

```
naf/1:  try_me_else L
        allocate 2
        get_level Y2
        get_variable V1 A1
        put_value V1 A1
        call call/1
        cut Y2
        call fail/0
        deallocate
        proceed
   L:   trust_me
        allocate 0
        deallocate
        proceed
```

Aunque el predicado anterior responde adecuadamente a lo que se espera del predicado `naf`, utilizarlo de esta forma trae aparejado algunos problemas: el programador debe incluir dicho predicado en todo programa, y además no se puede utilizar al `naf` como un operador sin paréntesis. Por estas razones es que se implementará al `naf` como un predicado predefinido, incorporando el código al igual que el predicado `call/1`, al principio del programa. Esto permitirá además algunas optimizaciones en el código.

Se tomará como base el código anterior. La llamada a `fail/0` puede obviarse si se utiliza directamente la nueva instrucción `fail_and_backtrack`. Obsérvese que las instrucciones siguentes a `fail_and_backtrack` nunca se ejecutarán, por lo tanto no es necesario que existan. El código predefinido para el predicado `naf/1` será entonces:

```
naf/1:  try_me_else L
        allocate 2
        get_level Y2
        get_variable V1 A1
        put_value V1 A1
        call call/1
        cut Y2
        fail_and_backtrack
   L:   trust_me
        proceed
```

Observación importante: Lo que fue desarrollado en esta sección es para el operador de negación por falla de Prolog, la negación por falla en un PLR está explicada en la sección 7.6.

Para que todo programa pueda utilizar estos predicados y operadores predefinidos, su código será incluído al principio del programa compilado. Esto es, el compilador generará en primer lugar el código para `true/0`, `fail/0`, `call/1` y `naf/1`, y lo colocará al principio del código del programa que se está compilando. De esta forma, todo programa contendrá en su código los predicados predefinidos.

El operador `naf/1`, será tratado como un operador infijo sin paréntesis para su operando. Es decir, en lugar de "`naf(P)`" deberá usarse "`naf P`". El compilador será el encargado de generar el código de la llamada a `naf` correctamente.

Por último, cabe destacar, que todos estos predicados, además de estar disponibles para cualquier programa, serán utilizados para implementar las nuevas instrucciones de la JAM.

7.3 Construcción de argumentos

En un PLR, una consulta q tendrá éxito, si es posible construir un argumento $\mathcal{A}$ para q tal que $\mathcal{A}$ sea una justificación. Para construir un argumento para q, se realizará una derivación rebatible de q, utilizando el mismo mecanismo que utiliza la WAM para una consulta de Prolog. Pero sin embargo, la existencia de un argumento para una consulta, no depende sólo de que exista una derivación rebatible. La derivación rebatible obtenida debe ser además consistente.

Para lograr que la JAM sea capaz de obtener un argumento para una consulta, se introducirán nuevas instrucciones y áreas de memoria, a fin de que el código generado para un PLR obtenga derivaciones rebatibles consistentes (argumentos). Para lograr esto, se utilizará el enfoque y los algoritmos desarrollado en el capítulo 5. Siguiendo el criterio establecido en la definicion 5.1.4, el código compilado de un PLR, verificará que la conclusión de cada cláusula usada sea consistente con el conjunto de CPE, y la derivación rebatible construida hasta el momento.

Por lo tanto, en la JAM, las derivaciones rebatibles obtenidas serán consistentes, y constituirán un argumento para la meta consultada. A continuación se mostrará como se consigue lograr lo anterior. Obsérvese, que las consultas serán denotadas con " $\prec\!\!-$ ", ya que producirán una derivación rebatible.

Ejemplo 7.3.1 : El siguiente PLR será utilizado para ejemplificar la implementación de la verificación de la consistencia en la JAM.

```
huye-volando(X) -< vuela(X),asustado(X).
vuela(X) -< ave(X).
ave(X) <- pingüino(X).
∼vuela(X) <- pingüino(X).
pingüino(petete) -< true.
ave(X) <- gallina(X).
gallina(coco) -< true.
asustado(coco).
```

Obsérvese que existen derivaciones rebatibles para las consultas " $\prec$ `vuela(coco)`", y " $\prec$ `vuela(petete)`", pero sin embargo, sólo `vuela(coco)` tiene un argumento, ya que `vuela(petete)` es inconsistente (ver ejemplo 7.3.2). □

7.3.1 Verificación de consistencia

En esta sección se mostrará como la JAM verifica la consistencia de una derivación rebatible. Para esto se utilizará el esquema y los algoritmos desarrollados en el capítulo 5. En la sección siguiente se desarrollará la implementación de los *hechos temporarios*, sin los cuales la verificación de la consistencia no sería posible.

Para que una derivación rebatible sea consistente, cada vez que se termina de probar rebatiblemente el cuerpo de una cláusula de programa, se debe verificar que la cabeza `C` de la cláusula sea consistente con lo derivado hasta el momento. Según el teorema de la consistencia (5.1.1), esto es equivalente a probar que no sea el caso que exista una *derivación fuerte* del complemento de `C`. Se llamará derivación fuerte (en contraposición a una derivación rebatible), a una derivación que utilice únicamente las CPE del PLR, los hechos temporarios que se tenga en ese momento, y alguna CPE invertida.

Observación 7.3.1 : A continuación se enumeran algunas consideraciones sobre el proceso de verificación de consistencia:

1. Cuando la máquina abstracta esté realizando una derivación fuerte, no se debe verificar la consistencia de las cláusulas usadas. Por lo tanto, durante la verificación de la consistencia de una submeta, no se verifica la consistencia de las CPE utilizadas.

2. Las CPE utilizadas en la derivación fuerte para la verificación de la consistencia no deben generar hechos temporarios, ya que no forman parte del argumento en construcción.

3. Las presuposiciones son CPR que necesitan de la verificación de la consistencia

4. las submetas "`not q`" (*i.e.*, con negación por falla en los PLR) no interactuan con la consistencia porque nunca pueden ser cabeza de una cláusula.

5. las conclusiones basadas sólo en CPEs no necesitan de verificación de consistencia, ya que toda la información que se utilizó para su argumento, es de por sí consistente.

Código JAM para las CPE invertidas

Cómo se vió en el capítulo 5, en los PLR las CPE no son contrapositivas, entonces las CPE invertidas no son cláusulas del programa. Su utilización está limitada a la aceleración de la búsqueda de inconsistencias. Es por esto, que las CPE invertidas sólo pueden utilizarse con ciertas restricciones: Sólo al principio de la búsqueda de inconsistencia, y una vez que se utilizó una cláusula que no es invertida, debe continuarse la derivación con CPE normales y hechos temporarios. Además, a excepción de la consulta que era la cabeza en la CPE sin invertir, el cuerpo de una CPE invertida debe probarse utilizando únicamente CPE y hechos temporarios (no pueden utilizarse CPR, o CPE invertidas).

Las CPE invertidas deben estar en el programa compilado, antes de que se efectúe una consulta. Para resolver esto, al compilar un PLR, se generarán todas las CPE invertidas a partir de las CPE del programa, y luego se compilará el programa normalmente. Para que el compilador distinga una CPE invertida de las demás cláusulas, se utilizará el nuevo símbolo " `@-` ". Por ejemplo de la CPE `~p <- q1,not q2,~q3` se pueden obtener dos CPE invertidas:

```
~q1 @- p, not q2,q3.
q3 @- q1,not q2,p.
```

Obsérvese que las consultas `not P` no pueden formar parte de la cabeza de un predicado, y por lo tanto no pueden generar una CPE invertida.

Las CPE invertidas pueden ir en cualquier parte de un programa, porque no importa en que orden son encontradas. Por lo tanto se las ubicará al final del PLR a fin de usar primero las cláusulas normales, y como último recurso las CPE invertidas. El código JAM de una CPE invertida será como el de una CPE común, salvo la instrucción de encabezado (más adelante se explicarán en detalle estas instrucciones). A continuación se muestran las CPE invertidas que se generan en el PLR del ejemplo 7.3.1.

```
~ pingüino(X) @- ~ ave(X).
~ pingüino(X) @- vuela(X).
~ gallina(X) @- ~ ave(X).
```

Código JAM para la verificación de la consistencia

Durante la compilación se debe generar el código necesario para que se realice la verificación de la consistencia durante la construcción de un argumento. Por ejemplo, una vez probado el cuerpo de una cláusula como `p(X) :- q1(X),q2(X)` se debe verificar la consistencia de la cabeza `p(X)` (con la instanciación que haya sufrido `X`). Por lo tanto, la máquina abstracta necesita dos modos de prueba, uno que genera derivaciones rebatibles (para la construcción del argumento), y otro que genera derivaciones fuertes (para la verificación de la consistencia).

Probar la consistencia de una cláusula con cabeza `p/n`, es equivalente a probar la no existencia de una derivación fuerte de una consulta `p/n`, lo cual es equivalente a tener éxito en una consulta `naf p`, (que sólo utilice las CPE del PLR, los hechos temporarios que se tenga en ese momento, y alguna CPE invertida). Recordar que `naf` es el operador de negación por falla de PROLOG.

Por lo tanto, el compilador generará al final de cada cláusula una consulta adicional que al compilarse generará el código necesario para la verificación de la consistencia. Esto es, se agregará el complemento de la cabeza de la cláusula, como una consulta adicional al final del cuerpo, precedida con el símbolo "`#`". Por ejemplo, la cláusula "`p(X) :- q1(X),q2(X)`" se transformará en "`p(X) :- q1(X),q2(X) # ∼ p(X)`"; y la cláusula "`∼r(X) :- q(X)`", se transformará en "`∼r(X) :- q(X) # r(X)`". El símbolo "`#`" debe interpretarse como "*y no puede probarse en forma fuerte*", esto es, "`# ∼vuela(charo)`" debe leerse como "y no puede probarse en forma fuerte `∼vuela(charo)`". El siguente ejemplo muestra el programa del ejemplo 7.3.1 con lo agregado por el compilador antes de generar el código JAM. Es importante destacar que esto nunca es visto por el programador, y es realizado por un preprocesador, justo antes de compilar el programa.

```
huye-volando(X) -< vuela(X),asustado(X) # ∼ huye-volando(X).
vuela(X) -< ave(X) # ∼ vuela(X).
ave(X) <- pingüino(X) #∼ ave(X).
∼vuela(X) <- pingüino(X) # vuela(X).
pingüino(petete) -< true # pingüino(petete) .
ave(X) <- gallina(X) # ∼ ave(X).
gallina(coco) -< true # ∼ gallina(coco).
asustado(coco).
∼ pingüino(X) @- ∼ ave(X).
∼ pingüino(X) @- vuela(X).
∼ gallina(X) @- ∼ ave(X).
```

Dos nuevas instrucciones `prove_consistency_else` y `end_consistency_proof` serán las encargadas de marcar el inicio y el fin de una prueba de consistencia, De esta forma, toda secuencia de código que esté entre estas dos instrucciones se ejecutará en modo "*derivación fuerte*", *i.e.*, sólo podrán usarse CPE, hechos temporarios, y, quizás, alguna CPE invertidas.

El símbolo `#` le indicará al compilador que lo que sigue es la verificación de consistencia de la cláusula que está compilando, y producirá que se genere la instrucción `prove_consistency_else`. Luego se compilará al submeta que sigue al `#` como una submeta estandar del PLR, y al terminar con la compilación de la submeta, se agregará la instrucción `end_consistency_proof`.

La Figura 7.3 muestra el código WAM de la cláusula "`ave(X) <- pingüino(X)`", y a su derecha el código JAM con la verificación de la consistencia incluída.

Un registro global llamado `prueba_consistencia` indicará que se está realizando la verificación de consistencia, y por lo tanto impedirá que se utilicen CPR en las derivaciones. La instrucción `prove_consistency_else` pondrá el registro `prueba_consistencia` en TRUE, mientras que `end_consistency_proof` lo volverá a FALSE. Además, otro registro global llamado `uso_cpe_invertidas` indicará que se pueden utilizar CPE invertidas. Al igual que antes, la instrucción `prove_consistency_else` pondrá el registro `uso_cpe_invertidas` en TRUE, mientras que `end_consistency_proof` lo volverá a FALSE. Ambos registros serán guardados en los puntos de elección del STACK a fin de actualizarlos durante el backtracking. En la figura 7.5 se muestra la implementación de estas dos instrucciones.

	Código WAM		Código JAM
ave/1:	try_me_else L	ave/1:	try_me_else L
	allocate 1		allocate 1
	get_variable V1 A1		forbid_inv
	put_value V1 A1		get_variable V1 A1
	call pingüino/1		put_value V1 A1
	deallocate		call pingüino/1
	proceed		prove_consistency_else E
			put_structure ∼ ave/1 A1
			set_value V1
			call naf/1
			end_consistency_proof
		E:	deallocate
			proceed

Figura 7.3: Código JAM para la cláusula `ave(X) <- pingüino(X)`

A primera vista se podría pensar en simplificar y usar un sólo registro en lugar de los dos anteriores, pero sin embargo, el uso de las CPE invertidas tiene una complicación adicional. Al principio de la prueba de consistencia se podrán usar CPE invertidas, pero una vez que se utilice alguna CPE, o hecho temporario, ya no se podrán utilizar más. Por lo tanto, las CPE y los hechos temporarios, deben prohibir el uso posterior de una CPE invertida. Una nueva instrucción `forbid_inv` (ver figura 7.5) presente en las CPE, pondrá el registro `uso_cpe_invertidas` en FALSE. Los hechos temporarios no necesitan usar `forbid_inv` porque se prueban sin llamar a otras cláusulas, y por ende no se podrá usar nuevamente otra CPE invertida. Pero además (ver lema 5.1.1) el resto de una CPE invertida debe probarse sólo con CPE, o hechos temporarios, por lo tanto la misma CPE invertida, debe usar `forbid_inv`, luego de ejecutar la primer consulta de su cuerpo.

Como se vió antes, mientras se está haciendo una prueba de consistencia, no se debe probar la consistencia de ninguna cláusula. Además si el argumento ha utilizado hasta el momento sólo CPE, no es necesario verificar la consistencia ya que el conjunto de CPE es consistente. Es por esto, que la instrucción `prove_consistency_else` tiene un paramentro `E` que indica la dirección siguiente a `end_consistency_proof`, a fin de poder saltar allí, cuando no es necesario verificar la consistencia. De esta forma, una vez que comience la

prueba de consistencia de un literal, no se realizará la verificación de consistencia de otra cláusula mientras `prueba_consistencia` esté en TRUE. Esto es, hasta que se encuentre la instrucción `end_consistency_proof`, que indica que la prueba de consistencia finalizó con éxito, o bien haya habido backtracking a un punto donde el registro `prueba_consistencia` estaba en FALSE.

Para que efectivamente las CPR fallen cuando el registro `prueba_consistencia` está en TRUE, se crearon cuatro nuevas instrucciones para encabezar las CPR:

```
defeasible_just_me
defeasible_try_me_else
defeasible_retry_me_else
defeasible_trust_me
```

El código de las CPE mantendrá la primer instrucción igual que en la WAM, mientras que en las CPR se utilizarán las cuatro anteriores. De esta forma se podrá diferenciar en tiempo de ejecución, que tipo de cláusula se está utilizando. El uso y la forma de generar estas nuevas instrucciones será el mismo que se utilizaba para las cuatro instrucciones en la WAM.

En cuanto a la implementación de estas cuatro instrucciones, será practicamente la misma que sus cuatro equivalentes en la WAM salvo que se les agrega al final del su implementación la siguiente condición:

```
IF prueba_consistencia THEN BackTrack;
```

La figura 7.4 muestra dos de estas instrucciones. De esta forma, si no se está verificando la consistencia, se comportarán igual que su correspondiente instrucción de la WAM. Pero cuando se esté verificando la consistencia, primero acomodarán el punto de elección y luego fallarán, impidiendo que se use la CPR, y permitiendo que el mecanismo de backtracking busque una CPE o hecho temporario que unifique.

Para que las CPE invertidas se utilicen sólo cuando el registro `uso_cpe_invertidas` esté en TRUE, usando la misma estrategia que con las CPR, se crearon cuatro nuevas instrucciones para encabezar las CPE invertidas:

```
PROCEDURE defeasible_just_me
     IF Prueba_consistencia THEN BackTrack;
END defeasible_just_me;

PROCEDURE defeasible_trust_me
     ...Codigo de trust_me...
     IF Prueba_consistencia THEN BackTrack;
END trust_me;

PROCEDURE just_me_checking_consistency
     IF NOT Uso_cpe_invertidas THEN BackTrack;
END just_me_checking_consistency;

PROCEDURE trust_me_checking_consistency
     ...Codigo de trust_me...
     IF NOT Uso_cpe_invertidas THEN BackTrack;
END trust_me_checking_consistency;
```

Figura 7.4: Implementación de instrucciones para encabezar CPR, y CPE invertidas

```
just_me_checking_consistency
try_me_checking_consistency_else
retry_me_checking_consistency_else
trust_me_checking_consistency
```

Igual que antes, el código de las cuatro instrucciones es el mismo que el que tiene su equivalente en la WAM salvo que se les agrega al final del su implementación la siguiente condición:

```
IF NOT uso_cpe_invertidas THEN BackTrack;
```

la cuál no interrumpe el flujo de control del PLR, pero impide que se utilicen CPE invertidas cuando no deben usarse. La figura 7.4 muestra dos de las nuevas instrucciones.

A continuación figura el código de la CPE invertida $\sim$ `pingüino(X) @- vuela(X)`

```
~pingüino/1:  trust_me_checking_consistency
              allocate 1
              get_variable V1 A1
              put_value V1 A1
              call vuela/1
              forbid_inv
              deallocate
              proceed
```

Una de las consideraciones de la observación 7.3.1 decía: *las conclusiones basadas solo en* CPE*s no necesitan de verificación de consistencia.* Para lograr esta optimización se creó un registro global `derivación_rebatible`, que indica cuando una derivación deja de ser fuerte y se transforma en rebatible; y una nueva instrucción `set_defeasible` que lo único que hace es poner el registro `derivación_rebatible` en TRUE (ver figura 7.5). Dicho registro debe guardarse en los puntos de elección del STACK, ya que debe actualizarse durante el backtracking.

Al compilar una CPR, se genera el código de la nueva instrucción `set_defeasible`, inmediatamente antes de generar el código para la verificación de consistencia de dicha CPR. Al comenzar la ejecución de cada consulta, el registro `derivación_rebatible` es puesto en FALSE. De esta forma mientras no sea usada ninguna CPR en la prueba de una consulta, el registro `derivación_rebatible` permanecerá en FALSE, indicando que no es necesaria la verificación de la consistencia de las reglas usadas en la derivación. Si una CPR es usada, la instrucción `set_defeasible` pondrá el registro en TRUE, y a partir de allí, se verificará la consistencia de toda submeta. Salvo que el backtracking lleve la derivación a una situación en la cuál el registro `derivación_rebatible` estaba en FALSE, en cuyo caso el mecanismo de backtracking actualiza el registro y se sigue sin verificar la consistencia.

A continuación se muestra el código JAM de las cláusulas `vuela(X) -< ave(X)` y `pingüino(petete) -< true` Ambas son CPR, y por lo tanto poseen la instrucción `set_defeasible`.

```
PROCEDURE set_defeasible;
      Derivacion_rebatible:=TRUE;
END set_defeasible;

PROCEDURE forbid_inv;
      Uso_cpe_invertidas:=FALSE;
END forbid_inv;

PROCEDURE prove_consistency_else (E);
     IF (NOT Derivacion_rebatible) OR Prueba_consistencia
     THEN  PC:=E {salto a donde indica el parametro}
     ELSE  Prueba_consistencia:=TRUE;
           Uso_invertidas:=TRUE;
END prove_consistency_else;

PROCEDURE end_consistency_proof;
     Prueba_consistencia:=FALSE;
     Uso_invertidas:=FALSE;
END end_consistency_proof;
```

Figura 7.5: Implementación de las instrucciones para verificación de consistencia

```
vuela/1:  defeasible_just_me
          allocate 1
          forbid_inv
          get_variable V1 A1
          put_value V1 A1
          call ave/1
          set_defeasible
          prove_consistency_else L
          put_structure ∼ vuela/1 A1
          set_value V1
          call naf/1
          end_consistency_proof
L:        deallocate
          proceed
```

```
pingüino/1:  defeasible_just_me
             allocate 0
             forbid_inv
             get_constant petete/0 A1
             call true/0
             set_defeasible
             prove_consistency_else L
             put_structure ∼ pingüino/1 A1
             set_value V1
             call naf/1
             end_consistency_proof
         L:  deallocate
             proceed
```

7.3.2 Hechos Temporarios

Durante la construcción una derivación rebatible se deben ir guardando los hechos temporarios (ver ejemplo 7.3.2). Esto es, cada vez que una cláusula utilizada en la derivación rebatible es aceptada (no produce inconsistencia), entonces la cabeza de la cláusula, con la instanciación que tienen sus variables en ese momento, deben guardarse como un hecho más en el programa. Este hecho tiene su vida limitada a la derivación en curso, y es por esto que se denominará temporario. Los hechos temporarios generados se utilizarán para la verificación de la consistencia de las demás cláusulas de la derivación. El desarrollo completo de este tema pueden encontrarse en el capítulo 5.

Los únicos elementos de un programa que no generan hechos temporarios son los hechos (CPE sin cuerpo), y los hechos temporarios ya generados. Las demás cláusulas deben generar un hecho temporario si son aceptadas como parte del argumento en construcción.

Ejemplo 7.3.2 : Al realizar la consulta " $\rightharpoonup$ `huye-volando(coco)`" al PLR del ejemplo 7.3.1 se genera la derivación rebatible que muestra la figura 7.6 a la izquierda. En el transcurso de la derivación, se generan varios hechos temporarios, en el siguiente orden: "`gallina(coco)`", "`ave(coco)`", "`vuela(coco)`", y "`huye-volando(coco)`" (nótese que "`asustado(coco)`" no es un hecho temporario, ya que es un verdadero hecho). Al momento de verificar la consistencia de la conclusión "`vuela(coco)`", se tienen los hechos

temporarios: "gallina(coco)", y "ave(coco)", y como puede verse, la conclusión es consistente.

Sin embargo, al realizar la consulta " $\rightharpoonup$ huye-volando(petete)" cuando se llega a la verificación de la consistencia de "vuela(petete)", se tienen los hechos temporarios "pinguino(petete)", y "ave(petete)". Utilizando la CPE ∼vuela(X) <- pingüino(X) se deriva en forma fuerte "∼vuela(petete)" y por lo tanto, la derivación rebatible que se estaba construyendo es inconsistente. Al fallar la verificación de consistencia, y no haber otra forma de probar "huye-volando(petete)", entonces la consulta falla. □

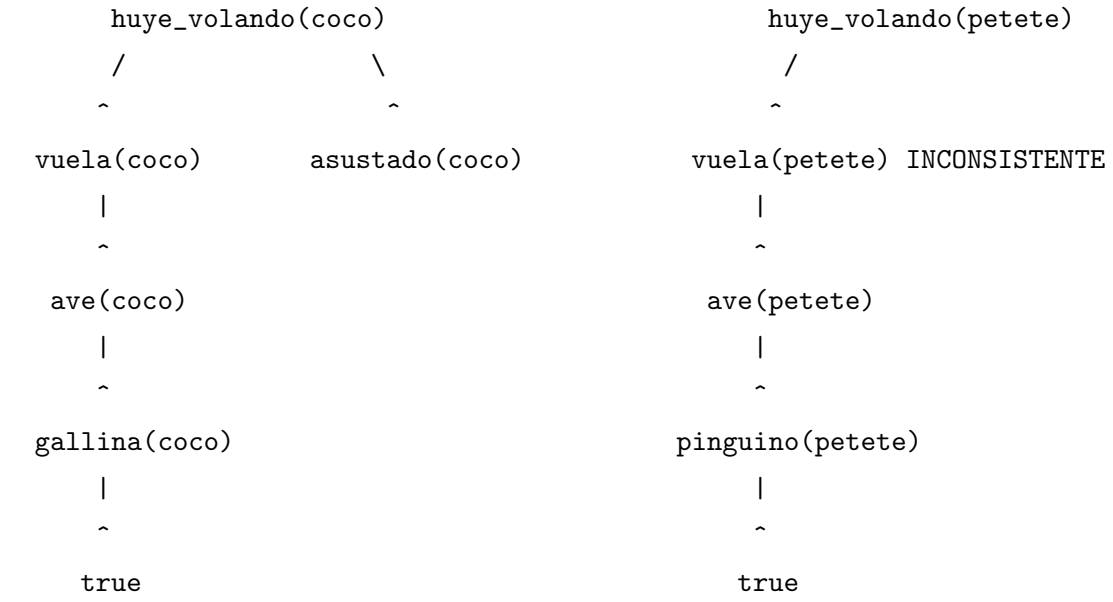


Figura 7.6: Árboles de derivación

Como se verá en la sección siguiente, los hechos temporarios son utilizados a lo largo de la construcción de una justificación para realizar varias tareas: verificar consistencia de un argumento, construir los conjuntos de activación, buscar contrarargumentos, y verificar la concordancia y circularidad de las lineas de argumentación. Por lo tanto es importante detenerse a estudiar en detalle como es que se obtienen, se almacenan, e interactuan con el backtracking.

Mientras un hecho temporario está activo, éste debe comportarse como un hecho más del programa, y por lo tanto, debe existir una secuencia de código JAM que lo represente.

De esta forma, cuando se intente probar una submeta, los hechos temporarios estarán en el código del programa como una cláusula más, permitiendo el normal desenvolvimiento del flujo de control del programa, y fundamentalmente no interfiriendo con el backtracking. Por lo tanto, una vez que un hecho temporario es creado, debe generarse el código JAM correspondiente, y agregarse al área de código del programa.

Para generar los hechos temporarios se incluyeron en la arquitectura, dos nuevas áreas de memoria: una tabla de hechos temporarios (THT), que mantiene la información de los hechos temporarios de cada argumento; y un "Heap Temporario " (llamado T-HEAP) donde se almacenarán los subtérminos de los hechos temporarios, en realidad referencias a las estructuras creadas durante la unificación y que corresponden a los hechos temporarios. Esto es, si "`p(f(a),b,[1,2])`" es un hecho temporario, la THT tendrá la información del predicado `p/3`, y en el T-HEAP habrá tres celdas de memoria consecutivas (una para cada subtérmino de `p`) con las referencias a las estructuras "`f(a)`", "`b`", y "`[1,2]`" que estarán almacenadas en el HEAP.

El T-HEAP es una estructura de memoria igual al HEAP común, pero debe estar en memoria superior al STACK a fin de evitar problemas con las ligaduras entre variables. Es importante destacar que no hay referencias desde el HEAP o el STACK, al T-HEAP, esto permite la eliminación de los elementos del T-HEAP sin generar referencias pendientes.

Tanto la THT como el T-HEAP funcionan como una pila que mantiene el recorrido en profundidad del árbol de derivación rebatible (esto facilitará la eliminación de elementos al hacer backtraking). Un registro global llamado T indicará el tope del T-HEAP (en realidad, la primer celda disponible). Otro registro global llamado F apuntará al tope de la THT. Ambos registros serán guardados en los puntos de elección, lo cuál permitirá que el backtracking deseche los hechos temporarios que no pueden usarse más.

La THT (ver Figura 7.7) mantiene la siguiente información:

1. Posición del identificador del hecho temporario en la Tabla de Identificadores.
2. Posición de los subtérminos del hecho temporario en el T-HEAP.
3. Dirección de la primera instrucción del código del hecho temporario.
4. Posición del hecho temporario padre en árbol de derivación del argumento.
5. Tipo de cláusula que generó el hecho temporario: CPE o CPR.

Puntero a Tabla de Identificadores
Puntero al T-HEAP
Puntero al área de codigo
Puntero al hecho temporario padre en la THT
Tipo de cláusula

Figura 7.7: Estructura de la Tabla de Hechos Temporarios (THT)

La información de la THT se utiliza para la verificación de la consistencia, la construcción del argumento, la construcción de contraargumentos, y la comparación por especificidad, por lo tanto algunos de sus elementos serán explicados más adelante.

Dos nuevas instrucciones: `save_subterms` y `save_fact`, serán las encargadas de la generación de los hechos temporarios. Por lo tanto, todo cláusula de PLR para un predicado `p/n`, que deba generar un hecho temporario al construirse un argumento, debe tener el siguiente código JAM:

```
p/n:  ...
      allocate
      save_subterms p/n Tc
      ⋮
      save_fact p/n
      deallocate
      proceed
```

Esto es, se deben agregar las instrucciones `save_subterms` y `save_fact`, al principio y al final del código de `p/n`. Como los hechos, las CPE invertidas, y por su puesto los hechos temporarios ya generados antes, no deben generar hechos temporarios, entonces no tendrán en su código JAM estas dos nuevas instrucciones. Las instrucciones `save_subterms` y `save_fact`, almacenarán los hechos temporarios en las nuevas estructuras de memoria, y controlarán además, que no se generen hechos temporarios durante la verificación de la consistencia.

A continuación se muestra el código JAM de las cláusulas `ave(X) <- pingüino(X)`, `vuela(X) -< ave(X)` y `pingüino(petete) -< true` con el agregado de las nuevas instrucciones.

```
ave/1:  try_me_else L
        allocate 1
        save_subterms ave/1 1
        forbid_inv
        get_variable V1 A1
        put_value V1 A1
        call pingüino/1
        prove_consistency_else E
        put_structure ∼ ave/1 A1
        set_value V1
        call naf/1 @6
        end_consistency_proof
    E:  save_fact ave/1
        deallocate
        proceed

vuela/1:  defeasible_just_me
          allocate 1
          save_subterms vuela/1 0
          forbid_inv
          get_variable V1 A1
          put_value V1 A1
          call ave/1
          set_defeasible
          prove_consistency_else E1
          put_structure ∼ vuela/1 A1
          set_value V1
          call naf/1
          end_consistency_proof
      E1:  save_fact vuela/1
           deallocate
           proceed
```

```
pingüino/1:  defeasible_just_me
             allocate 0
             forbid_inv
             get_constant petete/0 A1
             call true/0
             set_defeasible
             prove_consistency_else E2
             put_structure ∼ pingüino/1 A1
             set_value V1
             call naf/1
             end_consistency_proof
E2:          save_fact pingüino/1
             deallocate
             proceed
```

Cuando se ejecuta la instrucción `save_subterms p/n Tc`, todavía no se sabe si `p/n` será un hecho temporario o no, por lo tanto sólo se reserva en la THT un lugar para el potencial hecho temporario. Esta instrucción tiene dos parámetros: el predicado `p/n` a guardar, y el tipo de cláusula que representa (1=cpe, 0=cpr). Al ejecutarse realiza lo siguiente:

1. Almacena en la THT el lugar del T-HEAP donde quedarán los subtérminos.
2. Almacena en el registro de entorno del STACK, la dirección de la THT donde quedó el potencial hecho temporario, este valor será utilizado luego por la instrucción `save_fact p/n` para saber exactamente cuál es la entrada de la THT que debe actualizar.
3. Carga en el T-HEAP el contenido de los registros de parámetros del predicado `p/n`.

La instrucción `save_subterms` sólo agrega un elemento a la THT, indicando que se trata de un potencial hecho temporario, que todavía no ha sido probado. Pero si se ejecuta la instrucción `save_fact p/n` es porque dicho hecho temporario ya ha sido probado y debe agregarse (temporariamente) al PLR. Por lo tanto, la instrucción `save_fact p/n` generará el código JAM para el hecho temporario que se acaba de crear, y modificará la THT y la tabla de identificadores para indicar la existencia de un nuevo hecho (temporario).

```
PROCEDURE save_subterms (PosIdent,EsCpe)
  IF NOT Prueba_consistencia
  THEN Aridad:=ObtenerAridadTablaIdentificadores(PosIdent);
       PosF:=AgregarNuevoHechoEnTabladeHechosTemporarios(PosIdent,ValorDeT);
       GuardarEnMemoria(E+DesplazamientoEnEntornoDePosF,numero,PosF);
       FOR i:=1 TO n DO
          TEMP[T]:=A[i];
          T:=T+1;
       END for;
  GuardarEsCpeHechoTemporario(PosF,EsCpe);
  padre:=ObtenerdeEntornoPadreF;
  GuardarPadreHechoTemporario (PosF,padre);
  PadreF:=PosF;
  END if;
END save_subterms;
```

Figura 7.8: Implementación de la instrucción `save_subterms`

Obsérvese que cuando la instrucción `save_fact p/n` se ejecuta, es imposible que el predicado `p/n` no esté definido, ya que hubiera sido imposible generar el hecho temporario. Por lo tanto, al ejecutar "`save_fact p/n`" el predicado "`p/n`" ya estará presente en la tabla de identificadores y habrá código generado y cargado en el área de código. Como la instrucción `save_fact p/n` debe generar código para el nuevo hecho temporario, debe también modificar la tabla de identificadores, y el código de la definición anterior del predicado `p/n`, como si se tratara realmente de una nueva cláusula que se agrega a la definición del predicado `p/n`.

El hecho temporario se podría agregar al final de la definición de `p/n`, sin embargo, se agregará al principio, con lo cuál en lo que resta de la construcción del argumento, serán utilizados primero los hechos temporarios. Al agregar una nueva cláusula para `p/n`, la primera instrucción del código de la primera cláusula que existía antes para `p/n`, debería cambiar, a fin de indicar que es ahora la segunda cláusula. La siguiente tabla indica los reemplazos que deberían hacerse:

Instrucción presente:	Cambiar por:
just_me	trust_me
try_me_else L	retry_me_else L
defeasible_just_me	defeasible_trust_me
defeasible_try_me_else L	defeasible_retry_me_else L

Sin embargo, cuando se termine la construcción del argumento, el código del programa debe volver al estado que estaba antes de generar los hechos temporarios. Para evitar este proceso, no se modificará el código de dicha primer instrucción (como lo indicaba la tabla anterior), sino que se va a generar el siguiente código que al ejecutarse tendrá el mismo efecto que si se hubiera cambiado la instrucción.

```
Q:  nueva instrucción
    jump P+2
```

Donde `P` es la dirección de la instrucción que se hubiera cambiado, y *nueva instrucción* es la que se obtiene de la tabla anterior.

Por ejemplo, en el caso del código de la cláusula `ave(X) <- pingüino(X)` de la Figura 7.3, se generará el siguente código:

```
Q:  retry_me_else L
    jump (ave/1)+2
```

Estas dos instrucciones constituyen un truco para no modificar el programa original. La nueva instrucción `jump Dirección` lo único que hace es modificar el flujo de control del programa, asignándole al registro P (program counter) el valor `Dirección`. Por lo tanto, al ejecutarse el código a partir de la direccion `Q`, se produce el mismo efecto que si se hubiera cambiado verdaderamente la instrucción `try_me_else L` por `retry_me_else L`. Con la ventaja que el código de la cláusula `ave(X) <- pingüino(X)` no fue modificado, y por lo tanto al terminar la consulta se podrán eliminar facilmente los hechos temporarios, sin tener que reestablecer el código JAM del PLR.

Para un mismo predicado `p/n` podrían generarse varios hechos temporarios con diferentes instanciaciones de sus variables, y por lo tanto la instrucción `save_fact` debe poder distinguir si se trata del primer hecho temporario `p/n` que está generando o no. Para

esto, se agregó a la tabla de identificadores dos nuevas entradas para indicar el primer y el último hecho temporario generado. Al tener dos nuevas entradas en la tabla de identificadores no se modifica la información original de la primer y última cláusula que definen a `p/n`. Lo cuál permite que no deba actualizarse esta información al terminar la consulta y desecharse los hechos temporarios.

Con el fin de que la instrucción `call` ejecute primero los hechos temporarios, la operación "`ObtenerDireccionTablaIdentificadores(pos)`" devolverá como primer dirección del predicado, la del hecho temporario (si existiera alguno).

La instrucción `save_fact p/n` realizará entonces lo siguiente:

1. Obtiene del registro de entorno, el lugar de la THT donde va el hecho temporario.
2. Genera el código JAM del nuevo hecho temporario.
3. Actualiza la tabla de identificadores, para indicar que hay un nuevo hecho temporario.
4. Actualiza la THT con la dirección de código del nuevo hecho temporario.

El código JAM para el nuevo hecho temporario se genera de la siguiente manera. Si es el primer hecho temporario para el predicado `p/n`, se genera el siguiente código:

```
 Q:  nueva instrucción
     jump P+2
C1:  try_me_else Q
     unify_subterms HechoTemp
     proceed
```

y la dirección `C1` se convierte en la posición del primer hecho temporario. De esta forma, durante la ejecución del programa, si se llama al predicado `p/n`, la ejecución comenzará en la dirección `C1`. La instrucción `try_me_else Q` dejará el punto de elección preparado para que si falla el hecho temporario, la ejecución continúe en la dirección `Q`. Si el hecho temporario falla, se ejecuta la instrucción de la dirección `Q` y se salta a la primer cláusula real que existe en el PLR del predicado `p/n`.

En cambio, si ya habia otros hechos temporarios para este predicado, entonces se genera el siguiente código JAM:

```
C:  try_me_else Q
    unify_subterms HechoTemp
    proceed
```

Como al agregar este código hay un hecho temporario más, entonces `C` pasa a ser la dirección del último hecho temporario. En la tabla de identificadores, se tenía almacenado otra etiqueta `Cn` que es la dirección del último hecho temporario que había en el programa para este predicado (podría ser `C1` si se tratara del segundo hecho temporario). Por lo tanto, se modifica el parámetro de la instrucción de la dirección `Cn`, reemplazando `Q` por `C`, ya que ahora no será la instrucción del último hecho temporario. De esta forma, el mecanismo de backtracking intentará utilizar el primer hecho temporario, luego el segundo, y asi hasta agotar todos ellos; siguiendo luego con las cláusulas que ya existían en el PLR.

Obsérvese que el código de la etiqueta `Q` se genera sólo una vez, junto con el primer hecho temporario. En el resto de las oportunidades sólo se genera el código del nuevo hecho temporario y se actualiza la tabla de identificadores. Además `C1` sigue siendo siempre la dirección del primer hecho temporario para `p/n`. De esta forma, en el momento que se produzca una llamada `call p/n`, la JAM probará primero los hechos temporarios, y en el caso de ser necesario llegará a la instrucción de la dirección `Q` y por lo tanto "saltará" a las cláusulas originales del programa. La Figura 7.9 muestra la implementación de la instrucción `save_fact`.

El código de los hechos temporarios es extremadamente simple, pues como son hechos, lo único que deben hacer es intentar unificarse con la consulta que llama. Además, como los hechos temporarios se formaron a partir de metas ya probadas, sus estructuras internas ya fueron escritas en el HEAP, y están referenciadas por los elementos del T-HEAP. Por lo tanto, un hecho temporario consta únicamente de tres instrucciones, la primera maneja el backtracking, la segunda unifica los subtérminos del predicado, y la tercera simplemente retorna el control en caso de una unificación exitosa. La instrucción `unify_subterms` es la encargada de realizar el trabajo de unificación. Tiene un parámetro que indica la posición en la THT del hecho temporario que se está ejecutando, con el cuál obtendrá la información de los subtérminos.

Como se pudo ver en lo desarrollado hasta el momento, el código de los hechos temporarios crece monotónicamente, esto es, cuando hay backtracking no se deshace el código.

```
PROCEDURE Ejecutar_save_fact (PosIdent);
BEGIN
IF NOT Prueba_consistencia THEN
BEGIN PosHecho:=STACK[E+LugarEnEDePosF];
      CodigoAsociado:=ObtenerDireccionCodigoDeTablaHechosTemp(PosHecho);
      IF CodigoAsociado = 0 THEN
      BEGIN {No hay codigo y tengo que generarlo}
         Primero:=ObtenerPrimerHechoTempTablaIdentificadores(PosIdent);
         Ultimo:=ObtenerUltimoHechoTempTablaIdentificadores(PosIdent);
         IF  Primero = 0 THEN {es el primer hecho temporario}
         BEGIN
            {obtengo la direccion del predicado PosIdent}
            DireccionPredicado:=ObtenerDireccionTablaIdentificadores(PosIdent);
            Q:=RegistroIC; {guardo la direccion de Q}
            L1:=Codigo(DireccionPredicado+1); {obtengo L1}
            CASE Codigo(DireccionPredicado) OF {genero Q: NuevaI L1}
              just_me: AlmacenarInstruccion(trust_me);
              try_me_else: AlmacenarInstruccion(retry_me_else L1);
              defeasible_just_me: AlmacenarInstruccion(defeasible_trust_me);
              defeasible_try_me_else: AlmacenarInstruccion(defeasible_retry_me_else L1);
            END; {CASE}
            AlmacenarInstruccion(jump DireccionPredicado+2);
            C1:=RegistroIC;
            AlmacenarInstruccion(try_me_else Q);
            AlmacenarInstruccion(unify_subterms PosHecho);
            AlmacenarInstruccion(proceed));
            ActualizarPrimerHechoTempTablaIdentificadores(PosIdent,C1);
            ActualizarUltimoHechoTempTablaIdentificadores(PosIdent,C1);
            GuardarDireccionCodigoHechoTemporario(PosHecho,C1);
         END
         ELSE {NO es el primer hecho temporario} BEGIN
            C:=RegistroIC;
            AlmacenarInstruccion(retry_me_else Codigo[Ultimo+1]);
            AlmacenarInstruccion(unify_subterms PosHecho);
            AlmacenarInstruccion(proceed);
            Codigo[Ultimo+1]:=C;
            GuardarDireccionCodigoHechoTemporario(PosHecho,C);
            ActualizarUltimoHechoTempTablaIdentificadores(PosIdent,C);
         END;
      END;  {SINO se encontro un lugar para actualizar, no se hace nada}
END; {if not prueba_consistencia}
END; {..Fin de "Ejecutar_save_fact"...}
```

Figura 7.9: Implementación de la instrucción `save_fact`

No obstante, como el registro F está guardado en el punto de elección, la THT mantiene sólo los hechos temporarios que verdaderamente corresponden al punto actual de la computación. Por lo tanto, si hubo backtracking y fueron desechados hechos temporarios, estos ya no estarán presentes en la THT, pero si estará su código en el área del programa, y su correspondiente entrada en la tabla de identificadores. Por este motivo, la instrucción `unify_subterms` verifica que el código que se está ejecutando sea de un hecho temporario activo, es decir, de un hecho temporario que corresponde al argumento en curso. Obsérvese que la primera instrucción del código JAM del hecho temporario debe ejecutarse igual, porque es la instrucción que tiene la información del backtracking, y la

```
PROCEDURE unify_subterms (HechoTemp)
    IF HechoTemporarioDisponible(HechoTemp)
    THEN  i:=1;
          Aridad:=ObtenerAridad(HechoTemp);
          PosT:=ObtenerPosicionAHeap(HechoTemp);
          WHILE NOT Falla AND (i<=Aridad)
          DO  unify(A[i],PosT+i-1);
              i:=i+1;
          END while;
    ELSE Falla:=TRUE;
    END if;
    IF Falla THEN backtrack;
END; unify_subterms
```

Figura 7.10: Implemetación de la instrucción `unify_subterms`

encargada de buscar otra forma de probar el hecho ante una falla. De esta forma aunque en el área de código del programa existe código de hechos temporarios inexistentes, su código falla durante la ejecución de la instrucción `unify_subterms`. Podría darse el caso que se haya puesto en la THT otro hecho con el mismo predicado y aridad. Por este motivo, también se verifica que coincida la dirección de código real, con la almacenada en la tabla. En la figura 7.10 se encuentra la implementación de la instruccón `unify_subtems`. Como la a operación "`HechoTemporarioDisponible`" está relacionada con la verificación de la concordancia de un linea de argumentación, su implementación se mostrará más adelante.

Con el fin de eliminar el código de hechos temporarios desechados, se podría guardar el Registro IC (Instruction Counter) en los puntos de elección y de esta forma deshacer el código. Sin embargo esto no garantiza que la tabla de identificadores quede actualizada. Es decir, la tabla de identificadores apuntaría a un hecho inexistente. Por lo tanto, tambíen habría que deshacer lo modificado en la tabla de identificadores. Pero lo mas problemático es que habría que modificar el código de los demás hechos temporarios que quedan aún activos. Es decir, hay que volver a atrás todos los cambios sobre instruciones del tipo "`try_me_else`". Todo este proceso malgastaría un tiempo de ejecución.

En esta implementación se optó por dejar el código generado siempre en memoria y que la instrucción `unify_subterms` falle cuando se trate de un hecho temporario desechado (inactivo). De todas formas, como se verá en las secciones siguientes, el tiempo de vida de los hechos temporarios finaliza cuando se resuelve la consulta, y es ahí donde son eliminados de una forma muy sencilla: Una operación invocada al realizar una nueva consulta, recorre la tabla de identificadores para volver a colocar todas las direcciones de hechos temporarios como nulas. En cada consulta del intérprete, al eliminar el código de la consulta, elimina también el código generado para los hechos temporarios.

Ejemplo 7.3.3 : Una vez que se termina de construir el argumento para `huye-volando(coco)`, se tiene la siguiente información en la THT:

Identificador	Pos. T-HEAP	Pos. código	Tipo clau.	Padre
1. `huye-volando`	53	390	CPR	0
2. `vuela`	54	381	CPR	1
3. `ave`	55	372	CPE	2
4. `gallina`	56	363	CPR	3

y los siguientes hechos temporarios en el área de código:

```
359:  def_trust_me
      jump 253
363:  try_me_else 359
      unify_subterms 4
      proceed
368:  retry_me_else 231
      jump 133
372:  try_me_else 368
      unify_subterms 3
      proceed
```

```
377:  def_trust_me
      jump 102
381:  try_me_else 377
      unify_subterms 2
      proceed
386:  def_trust_me
      jump 66
390:  try_me_else 386
      unify_subterms 1
      proceed
```

□

7.4 Justificación de una consulta

Como ya se mostró en el capítulo 6, cuando se realiza una consulta en Prolog, primero se compila la consulta, y luego se intenta construir una derivación para la meta de la consulta. En los PLR es un poco más complejo, ya que primero se debe construir una derivación rebatible consistente (argumento), y de tener éxito se debe construir el árbol de dialéctica, para el argumento hallado.

El árbol de dialéctica es un árbol formado por argumentos, y los argumentos son árboles de derivación rebatible. En la WAM una falla en la prueba de una meta producía backtracking y se buscaba otra forma de probar la meta. Lo mismo ocurre en la JAM al construir un argumento, una falla en la derivación rebatible, invocará al mecanismo de backtracking para buscar otra alternativa. Pero cuando se está construyendo el árbol de dialéctica una falla en la construcción de un argumento $\mathcal{B}$ para un meta `m`, puede significar buscar otra forma de probar `m`, o puede involucrar backtracking sobre el árbol de dialéctica, para intentar construir otro derrotador, o abandonar esa línea de argumentación.

La dificultad inicial que se presenta, es que hay dos niveles de falla, una a nivel de los árboles de derivación rebatible, y otra al nivel del árbol de dialéctica. Sin embargo, ambas están estrechamente relacionadas y, como se mostrará a continuación, es posible unificarlas y manejar todo con el mismo mecanismo de backtracking desarrollado hasta el momento.

El proceso de generación del árbol de dialéctica se podría especificar en términos de Prolog, de la siguente manera:

```
justificar(Q) :- argumento(Q,A), naf derrotado(A)
derrotado(A) :- hallar_derrotador(A,D), naf derrotado(D)
```

Es decir, existe una justificación `A` para la consulta `Q`, si es posible construir un argumento `A`, y no es posible probar que `A` esté derrotado. Por su parte, un argumento `A` estará derrotado, si existe un derrotador `D` para `A`, tal que `D` no esté derrotado. Como "`naf derrotado(A)`" es una llamada *utilizando la negación por falla de Prolog*, tendrá éxito si no hay derrotadores, y fallará en el caso que los haya. De esta forma, si hay derrotadores para el argumento de la consulta `Q`, la consulta `Q` fallará, y si no hay derrotadores, la consulta `Q` tendrá éxito. Es importante tener en cuenta que se está utilizando la negación

por falla de Prolog, y no la negación por falla de un PLR, ya que esta última tiene una definición diferente (ver capítulo 4).

El esquema de justificación anterior, da una pista muy interesante sobre cómo diseñar la máquina abstracta, para que decida si el argumento hallado para una consulta es o no una justificación. Primero se intenta construir un argumento $\mathcal{A}$ para una consulta `q/n`. Si la construcción del argumento tuvo éxito, entonces se debe probar que no existan derrotadores de nivel 1. Esto es, se debe verificar que no sea posible probar el predicado `defeat1/0`, que se definirá a continuación.

Como se vió en el capítulo 5, para buscar un contraargumento de $\mathcal{A}$, sólo hace falta buscar en los elementos de $Coc(\mathcal{A}^c)$, esto es, en los hechos temporarios del argumento $\mathcal{A}$ (ver algoritmo 5.2.1). Por lo tanto, si el argumento $\mathcal{A}$ tiene N hechos temporarios, el predicado `defeat1/0` estará definido por un conjunto de N cláusulas. Por cada `h` de $Coc(\mathcal{A}^c)$, habrá una cláusula que define a `defeat1/0` que tendrá en su cuerpo la meta $\overline{h}$ (para construir un contraargumento) y luego la meta `naf defeat2/0` para comprobar que el contraargumento hallado no esté derrotado

De esta forma, cada una de las cláusulas de `defeat1/0` intentará hallar un derrotador para algún punto de contraargumentación del argumento $\mathcal{A}$. Si una de las cláusulas de `defeat1/0` falla, entonces el mecanismo usual de backtracking de la máquina abstracta, se encargará de buscar otra forma de probar `defeat1/0` con lo cuál estará intentando construir otro contraargumento.

Si el predicado `defeat1/0` puede construir un derrotador, entonces llamará a `defeat2/0`, el cual puede llamar a `defeat3/0`, y asi sucesivamente. Obsérvese que intentar probar `defeatK/0` significa estar construyendo el derrotador número `K` de la linea de argumentación, o lo que es lo mismo, estar en el nivel `K` del árbol de dialéctica. Por lo tanto, estas llamadas serán las que construyan una línea de argumentación, y el mecanismo de backtracking será el encargado de producir un recorrido en profundidad (depth-first) del árbol de dialéctica. Obsérvese entonces que el mismo mecanismo de backtracking que buscará otro derrotador, u otra linea de argumentación en caso de una falla, es el mismo mecanismo que buscará otra CPE o CPR, para intentar probar una meta.

Antes de construir un argumento para una consulta, no se puede saber cuales serán sus posibles puntos de contraargumentación, y por lo tanto no se puede saber cuales serán las cláusulas que definirán al predicado `defeat1/0`, (y menos al predicado `defeat2/0`).

Por lo tanto los predicados `defeatK/0` se generarán en tiempo de ejecución, y a medida que se los necesite.

De esta forma se construirán las lineas de argumentación (generando implícitamente el árbol de dialéctica) hasta que el argumento para la consulta quede derrotado, o no sea posible derrotarlo. Cabe recordar que la respuesta del intérprete dependerá de que tipo de derrotador se haya encontrado para el argumento de la consulta. Si el derrotador del argumento que sustenta la consulta, es de bloqueo, entonces la respuesta será INDECISO, pero si el derrotador es propio, la respuesta será NO. Por lo tanto, se debe tener una consideración especial con los derrotadores del primer nivel del árbol de dialéctica. Aunque se halla encontrado un derrotador de bloqueo aceptable para el argumento de la consulta, igual se debe seguir buscando un derrotador propio. En el peor de los casos, se construirán todos los derrotadores de bloqueo posibles.

Una nueva estructura de memoria llamada LINEA será la encargada de representar la linea de argumentación en curso. Como la THT trabaja como una pila, entonces se podrá tener en ella, todos los argumentos de la linea de argumentación en curso. Por lo tanto, la LINEA solamente contendrá el inicio en la THT de cada argumento de la linea, y el inicio en la THT de los subargumentos de desacuerdo de cada argumento. La LINEA mantendrá una rama del árbol de dialéctica, y por ende funcionará como una pila. Un registro global `tope_linea` contendrá el tope de LINEA. Dicho registro será guardado en los puntos de elección a fin de actualizarlo durante el backtracking.

7.4.1 Nuevas instrucciones para hallar una justificación

Para que la JAM busque una justificación para una consulta `q/n` se generará el siguiente código JAM:

```
?:  allocate                    ; comienzo de la consulta
    ⋮                           ; código JAM de la consulta
    call q/n                    ; intenta construir el argumento
    prepare_to_defeat 1         ; prepara para buscar derrotadores
    put_constant defeat1 A1     ; intenta probar que ...
    call naf                    ; ... no existen derrotadores
    show_answer ...             ; muestra las respuestas
    deallocate                  ;
    end_query                   ; fin consulta
```

La instrucción `prepare_to_defeat 1` es la encargada de generar en tiempo de ejecución, el código JAM del predicado `defeat1/0` que será el encargado de buscar los derrotadores del argumento hallado para la consulta original `q/n`. Luego de generar el código de `defeat1/0`, se realiza la llamada `naf defeat1`, que como utiliza la negación por falla de Prolog, tendrá éxito si no hay derrotadores, y fallará en el caso que los haya. De esta forma, si hay derrotadores para el argumento de la consulta `q/n`, la consulta `q/n` fallará, y si no hay derrotadores, la consulta `q/n` tendrá éxito. Pero cómo en realidad un derrotador será aceptable si no está a su vez derrotado, y por lo tanto el éxito o falla del predicado `defeat1/0` dependerá del árbol de dialéctica que se forme a partir de él.

La instrucción `prepare_to_defeat` tiene como parámetro el número de nivel `K` en la linea de argumentación, de esta forma en cualquier momento que se la ejecute, podrá preparar el código JAM necesario para buscar el próximo derrotador de nivel `K` en la linea de argumentación en curso.

Una vez que se ha construído un argumento $\mathcal{A}$ de cualquier nivel en la linea de argumentación, su conjunto $Coc(\mathcal{A}^c)$ estará guardado como hechos temporarios en la tabla de hechos temporarios THT. Como ya se mostró en el capítulo 5, es posible buscar todos los puntos de contraargumentación a partir de $Coc(\mathcal{A}^c)$. Por lo tanto para generar los contraargumentos de un argumento dado, se debe ir tomando uno a uno los elementos de $Coc(\mathcal{A}^c)$, y tratar de construir un argumento para el complemento de dicho elemento. Si h es un elemento de $Coc(\mathcal{A}^c)$, se debe construir un argumento para $\overline{h}$. Para buscar un contraargumento es posible utilizar todo el PLR más las CPE invertidas. Sin embargo, para probar los antecedentes de una CPE invertida, se debe usar solamente las CPE y los elementos de $Coc(\mathcal{A}^c)$ (ver algoritmo 5.2.1).

Supóngase que $\overline{h_i}$ son los complementos de los elementos de $Coc(\mathcal{A}^c)$, para los cuales existe al menos una regla en el PLR, que tenga por cabeza un literal que unifique con $\overline{h_i}$. La figura 7.11 muestra el código JAM generado en ejecución por la instrucción `prepare_to_defeat K`.

De esta forma, cuando se ejecute `defeatK/0` realizará lo siguiente:

1. Intentará construir un contraargumento para uno de los posibles puntos de contraargumentación, para lo cuál habilita el uso de CPE invertidas utilizando la instrucción `allow_inv`.

defeatK/0:	try_me_else L2	; intenta con $\overline{h_1}$ ...
	allocate	; ... si falla sigue con L2
	allow_inv	; permite CPE invertidas
	load_subterms N,T	; carga los registros de parámetros
	call $\overline{h_1}$	; intenta construir contraargumento
	set_disagree Hecho	; guarda el punto de desacuerdo
	not_circular	; verifica la circularidad en la linea
	not_less_specific	; verifica que sea derrotador
	prepare_to_defeat K+1	; se prepara para un nuevo nivel
	put_constant defeatK+1 A_1	;
	call naf	; continúa la linea de argumentación
	proper_defeat	; solamente en el caso K=1 ...
	deallocate	; ... verifica que sea derrotador propio
	proceed	;
L2:	retry_me_else L3	; intenta con $\overline{h_2}$ ...
	⋮	; ... si falla sigue con L3
	proceed	;
L3:	retry_me_else L4	;
⋮	⋮	;
LN:	trust_me	; intenta con $\overline{h_n}$ ...
	⋮	; ... si falla no hay derrotador
	proceed	;

Figura 7.11: Código JAM generado por `prepare_to_defeat K`

2. La instrucción `load_subterms` tiene como parámetro la aridad `N` de la meta $\overline{h_i}$, y carga los primeros `N` registros de parámetro A_i con lo que está almacenado en el T-HEAP a partir de la posición `PosT` (ver figura 7.12).

3. La instrucción `call` $\overline{h_i}$ efectúa la consulta que intentará construir un argumento para $\overline{h_i}$.

4. Una vez que obtuvo el contraargumento, guarda la posición del subargumento de desacuerdo en la linea de argumentación utilizando la instrucción `set_disagree` (ver figura 7.12).

5. Verifica con la instrucción `not_circular`, que el contraargumento generado no esté presente en la linea, produciendo circularidad en la argumentación.

6. Verifica que el contraargumento generado sea un derrotador, ya sea de bloqueo o propio. Para esto la instrucción `not_less_specific` verifica que el contraargumento hallado no sea menos específico que el subargumento de desacuerdo.

7. Llama a la instrucción `prepare_to_defeat K+1` a fin de que se prepare el código necesario para buscar un derrotador en un nuevo nivel del árbol de dialéctica, y seguir de esta forma avanzando en la linea de argumentación.

8. Una vez armado el código para la búsqueda de derrotadores de nivel `K+1` llama a `naf defeatK+1/0`

9. Si el derrotador hallado para el nivel `K=1` no está derrotado, se debe verificar que tipo de derrotador es, ya que si es de bloqueo, igual se debe seguir buscando otro derrotador propio. Si en cambio es un derrotador propio, entonces ya no hay mas nada que buscar, pues la consulta no tiene justificación. La instrucción `proper_defeat` es la encargada de realizar esto, pero sólo se genera en el caso de K=1.

Obsérvese que si cualquiera de las acciones anteriores falla, el mecanismo de backtracking, tratará de buscar otra forma de probar las cosas. Lo atractivo de esta estrategia es usar siempre el mismo mecanismo de backtracking, mezclando los dos niveles de árboles: el de dialéctica y el de argumentación.

La instrucción `allow_inv` pondrá el registro `uso_cpe_invertidas` en TRUE, permitiendo que al buscar un contraargumento se utilicen CPE invertidas. Pero se deben utilizar con las mismas consideraciones que al verificar la consistencia. Esto es, una vez que se ha utilizado una cláusula del programa, ya no pueden utilizarse nuevamente CPE invertidas. Además, a partir de la segunda consulta de la CPE invetida, debe utilizarse solo CPE y hechos temporarios.

Al probar el cuerpo de una CPE invertida, durante la verificación de la consistencia, el registro `uso_cpe_invertidas` prohibía el uso de CPE invertidas, y el registro `prueba_consistencia` prohibía las CPR. Pero al buscar un contraargumento, el registro `prueba_consistencia` debe estar en FALSE, y por lo tanto se necesitará de un registro más general que `uso_cpe_invertidas` para probar adecuadamente el cuerpo de una CPE

invertida. Se utilizará entonces, un nuevo registro global llamado `solo_cpe` que prohibe el uso de invertidas, pero además prohibe el uso de CPR. Dos nuevas instrucciónes manejan el estado de `solo_cpe`: `only_strong` y `stop_only_strong`. La primera lo pone en TRUE, y la segunda en FALSE. Ambas aparecen sólo después de la primera consulta en el cuerpo de una CPE invertida, `only_strong` después del primer `call` y `stop_only_strong` antes del `proceed`.

Por lo tanto se debe agregar el nuevo registro al código de las cuatro instrucciónes para las CPR cuya última condición será ahora:

```
IF (prueba_consistencia OR solo_cpe) THEN BackTrack;
```

y en las cuatro correspondientes a las CPE invetidas:

```
IF (NOT uso_cpe_invertidas OR solo_cpe) THEN BackTrack;
```

La instrucción `not_circular`, es la encargada de verificar si el último argumento hallado $\mathcal{A}$ no fue usado antes en la linea de argumentación. Para esto, se toma cada argumento $\mathcal{B}$ de linea (excepto el último) y se verifica que no sea el caso que $\mathcal{B} \subseteq \mathcal{A}$. Al usar la inclusión, se asegura que no fue usado antes como argumento ni como subargumento.

En la sección siguiente se mostrará como implementar el criterio de especificidad para comparar dos argumentos de la linea, y cuál es la implementación de las instrucciones `not_less_specific` y `proper_defeat`.

7.4.2 Concordancia

Para verificar la consistencia de una cláusula, se utilizaban sólo los hechos temporarios del argumento que se estaba construyendo. Pero la verificación de concordancia es una generalización de la verificación de la consistencia. Por lo tanto, para lograr que los argumentos de soporte (resp. interferencia) sean concordantes, al verificar la consistencia de una cláusula, se utilizarán todos lo hechos temporarios de todos los argumentos de soporte (resp. interferencia) de la linea de argumentación.

La dificultad radica en cómo indicar que un hecho temporario puede ser utilizado en la verificación. La implementación se realizó utilizando la información almacenada en la linea

```
PROCEDURE load_subterms(N,PosT);
 FOR i:=1 TO N
     DO A[i]:=T-HEAP[PosT+i-1];
END load_subterms;

PROCEDURE set_disagree (PosF);
 LINEA[TopeLinea-2].Desacuerdo:=PosF;
END set_disagree;

PROCEDURE not_circular;
 iniA:=LINEA[TopeLinea-1].PosTHT;
 finA:=F-1;
 Arg:=1;      Ultimo:=TopeLinea-1;
 WHILE (Arg < Ultimo) AND NOT Falla DO
  IniB:=LINEA[Arg].PosTHT;
  finB:=(LINEA[Arg].PosTHT+1) - 1;
  Falla:=Contenido(iniA,finA,iniB,finB);
  Arg:=Arg+1;
 END while;
 IF Falla THEN BackTrack;
END not_circular;
```

Figura 7.12: Implementación de instrucciones usadas por `defeatK/0`

de argumentación, simplemente haciendo fallar el código de la instrucción `unify_subterms` para aquellos hechos temporarios que no son del mismo tipo de argumento del que se está construyendo. Esto es, si se está construyendo un argumento de soporte, se podrán utilizar para la verificación de la consistencia todos los hechos temporarios que provienen de un argumento de soporte. La operarción `HechoTemporarioDisponible`, llamada desde `unify_subterms` es la que establece si un hecho temporario puede usarse o no. La figura 7.13 muestra la implementación de esta función.

Observación importante: Cuando una linea de argumentación es descartada y se busca otra, (*i.e.*, hay backtracking), habría que deshacer el código generado para todos los

```
FUNCTION HechoTemporarioDisponible(PosF):BOOLEAN ;
    i:=1;
    Pos1THT:= LINEA[TopeLinea-i].PunteroATHT;
    Pos2THT:= F;
    Disponible:= (Pos1THT <= PosF) AND (PosF < Pos2THT);
    i:=i+2;
    WHILE Prueba_Consistencia AND (i < TopeLinea) AND NOT Disponible  DO
          Pos1THT:= LINEA[TopeLinea-i].PunteroATHT;
          Pos2THT:= LINEA[TopeLinea-(i-1)].PunteroATHT;
          Disponible:= (Pos1THT <= PosF) AND (PosF < Pos2THT);
          i:=i+2;
    END; {WHILE}
    RETURN Disponible;
END HechoTemporarioDisponible;
```

Figura 7.13: Implementación de la función `HechoTemporarioDisponible`

predicados `defeatK/0` que se volverán a calcular. Sin embargo esto no es sencillo porque intercalado con el código de los predicados `defeatK`, hay código de hechos temporarios que deben ser utilizados para la concordancia. Por lo tanto no se recuperará el espacio de memoria ocupado por el código de estos predicados, hasta que la consulta halla terminado.

7.5 Comparación de argumentos por especificidad

Existen dos instrucciones JAM que utilizan el criterio de especificidad, estas son: `not_less_especific` y `proper_defeat`. La primera verifica que un contraargumento hallado no sea menos específico que el subargumento de desacuerdo, esto es, que sea un derrotador, propio o de bloqueo, del argumento del nivel anterior en la linea de argumentación. La segunda, sólo se ejecuta cuando se ha encontrado un derrotador aceptable para el argumento que es raíz del árbol de dialéctica, y se utiliza para saber si el derrotador hallado es un derrotador propio o no. La figura 7.14 muestra la implementación de ambas instrucciones.

```
PROCEDURE not_less_especific
    IF MasEspecifico(SubArgumentoDesacuerdo,Contraargumento)
       THEN FALLA := TRUE
    IF Falla THEN  BackTrack;
END not_less_especific

PROCEDURE proper_defeat
    Derrotador_propio:=MasEspecifico(Contraargumento,SubArgumentoDesacuerdo)
    IF NOT Derrotador_propio
       THEN FALLA := TRUE
            Hubo_bloqueo := TRUE
    IF Falla THEN  BackTrack;
 END proper_defeat
```

Figura 7.14: Instrucciones JAM para comparar argumentos

Para comparar dos argumentos $\mathcal{A}$ y $\mathcal{B}$ por especificidad se utilizará el algoritmo de la figura 7.15, que asume que existe un sólo completamiento por argumento

Al construir un argumento, las submetas con `not` generán un hecho temporario, y se almacenarán en la THT, como si fueran una CPE. Los hechos del programa utilizados en la construcción de un argumento, también se guardarán en la THT, pero no genererán un hecho temporario. Esto último se logra incluyendo en el código de los hechos la instrucción JAM `save_subterms`, pero no la instrucción `save_fact`; de esta forma, se crea una entrada en la THT, pero no se genera código para el hecho temporario.

Por lo tanto la operación que verifica si un argumento $\mathcal{A}$ está basado en hechos o no, simplemente recorre el espacio de la THT donde están los elementos de $\mathcal{A}$, y verifica si fue usado un hecho o una submeta `not p`, o no.

7.5.1 Obtención de los conjuntos de activación

Para obtener los conjuntos de activación se utilizará el algoritmo de la figura 4.1, utilizando la información de la THT. Como el conjunto de hechos temporarios de la THT es

---

ALGORITMO **7.5.1** *Comparación por especificidad*
DATO DE ENTRADA: dos argumentos $\mathcal{A}$ y $\mathcal{B}$.
DATO DE SALIDA: TRUE si $\mathcal{A}$ es estríctamente más especifico que $\mathcal{B}$,
COMIENZO:
SI $\mathcal{A}$ está basado en hechos y $\mathcal{B}$ no lo está,
ENTONCES RETORNAR TRUE
SINÓ:

Calcular los conjuntos de activación de $\mathcal{A}$, y de $\mathcal{B}$.
SI se cumple que:
a) Para todo $C_a \in$ NTAct-sets($\mathcal{A}$) existe $C_b \in$ Act-sets($\mathcal{B}$) tal que $C_b \subseteq C_a$, y
b) existe un $C_b \in$ NTAct-sets($\mathcal{B}$) tal que no exista $C_a \in$ Act-sets($\mathcal{A}$), con $C_a \subseteq C_b$
ENTONCES RETORNAR TRUE
SINÓ RETORNAR FALSE.

FIN ALGORITMO

Figura 7.15: Comparación por especificidad

simplemente el recorrido en preorden del argumento, a partir de esta información no es posible obtener los conjuntos de activación. Es por esto, que en la THT se mantiene el hecho temporario padre de cada elemento de la THT.

Para mantener la información del apuntador al padre en la THT, se utiliza un registro global `padre` que tiene almacenado la posicion en la THT del padre del hecho temporario a guardar. La instrucción `save_subtems` además de guardar el potencial hecho temporario en la THT, guarda el valor de su padre. Luego actualiza el registro `padre` con el valor del hecho temporario en curso, asi cuando sea invocada de nuevo, tendrá el valor del padre. Como la ejecución de una consulta sigue el recorrido en preorden del argumento, y `padre` debe actualizarse cada vez que se termina de probar un subárbol del argumento, entonces el registro `padre` se guarda en el entorno de cada cláusula a fin de que se actualice con cada `allocate-deallocate`. Pero además, el backtracking interfiere con la actualización de `padre`, y es por esto que también es salvado en el choice point, a fin de recuperar el verdadero `padre`, cada vez que se hace backtraking. De esta forma, al terminar de construir un argumento, se tendrá en la THT el árbol de derivación correspondiente, representado con un apuntador al padre.

Además la instrucción `save_subtems` guarda el tipo de cláusula que generó el hecho temporario (CPE o CPR), con lo cuál se podrá determinar si el de conjunto de activación que se genera es trivial, o no trivial.

El algoritmo de obtención de los conjuntos de activación se corre sobre la THT (entre los topes correspondientes al argumento). Se utiliza el algoritmo de la figura 4.1, donde al usar una cola, ya no es necesario usar otra estructura para ir almacendo los conjuntos ya generados, porque al avanzar el frente de la cola, ya quedan guardados en la misma estructura.

7.6 La negación por falla en los PLR

Observesé que en los PLR, una consulta falla cuando no existe una justificación, lo cual puede ocurrir por varias razones: no hay argumento, el argumento está derrotado, o el argumento está bloqueado. Por lo tanto el comportamiento del `not` no será el mismo que en un programa en Prolog. Según la definición de un PLR, `not p` tiene éxito cuando no es posible justificar `p`, y falla cuando existe una justificación para `p`.

La semántica definida para el `not` hace que toda submeta de la forma `not p` sea una hoja dentro del árbol de derivación de un argumento, ya que si `not p` tiene éxito su argumento es el conjunto vacío. También será una hoja dentro del árbol de dialéctica, ya que nunca se buscarán derrotadores para una submeta `not p`.

Dada una submeta `not p`, la máquina abstracta debe buscar una justificación para `p`, lo cual involucrará construir un argumento $\mathcal{A}$ para `p`, y si existe, construir el correspondiente árbol de dialéctica a partir de $\mathcal{A}$. Pero nunca deberá buscar derrotadores para `not p`.

Ya se vió que el operador `naf` de negación por falla de Prolog, puede definirse en términos de Prolog, utilizando los predicados extra lógicos `call` y cut (`!`):

```
naf(X) :- call(X),!,fail
naf(_)
```

El operador `not` de los PLR puede definirse también con un PLR:

```
not(X) :- justify(X),!,fail
not(_)
```

con la diferencia que ahora `justify(X)` implica generar una justificación para `X`.

De la misma forma que el `naf` de Prolog fue incorporado como un operador predefinido a la WAM, el operador `not` también se lo incluyó como un operador predefinido en la JAM. La figura 7.16 muestra su código (compárese este código con el de la negación por falla de Prolog en la sección 7.2).

```
not/1:  try_me_else L
        allocate 2
        get_level Y2
        get_variable V1 A1
        put_value V1 A1
        call call/1
        prepare_to_defeat K
        put_const defeatK
        call naf/1
        cut Y2
        fail_and_backtrack
   L:   trust_me
        allocate 1
        save_subterms not/1 1
        get_variable V1 A1
        save_fact not/1
        deallocate
        proceed
```

Figura 7.16: Código JAM para el operador predifinido `not`

7.7 Obtención de la respuesta ante una consulta

Una consulta en PROLOGtiene dos respuestas posibles, SI o NO. Sin embargo, una consulta a un PLR puede tener cuatro respuestas posibles: SI, NO, INDECISO, y DESCONOCIDO.

En la JAM, si una consulta tiene éxito, el registro `falla` queda en FALSE, mientras que si falla, el registro `falla` queda en TRUE. El registro `falla` alcanzaba en la WAM para dar las dos respuestas de PROLOG, pero no es suficiente para identificar las cuatro respuestas de un PLR. Por lo tanto se agregaron a la máquina abstracta los siguientes registros que tendrán un valor TRUE, o FALSE.

- `hubo_argumento`: Indica si hubo al menos un argumento.
- `derrotador_propio`: Indica si el derrotador hallado para el argumento de la consulta es propio o no
- `hubo_bloqueo`: Indica si hubo un derrotador aceptable de bloqueo
- `hubo_inconsistencia`: Indica si no existe argumento por una inconsistencia.

Ninguno de estos de registros se guarda en el STACK, e inicialmente los cuatro están en FALSE.

```
IF NOT falla
   THEN WRITELN(' SI.')
   ELSE
     IF NOT hubo_argumento AND NOT Hubo_inconsistencia
        THEN WRITELN(' DESCONOCIDO.')
        ELSE
          IF derrotador_propio OR Hubo_inconsistencia
          THEN WRITELN(' NO. ')
          ELSE IF hubo_bloqueo
               THEN  WRITELN(' INDECISO.')
               ELSE  WRITELN(' DESCONOCIDO.');
```

Figura 7.17: Obtención de las respuestas en un PLR

La única vez que se modifica el registro `hubo_argumento`, es cuando se ejecuta la instrucción `prepare_to_defeat K`, y `K` vale 1, ya que es cuando se ha construído un argumento para la consulta y se está por buscar un derrotador, entonces se pone el registro `hubo_argumento` en TRUE.

El registro `derrotador_propio` se modifica únicamente en la instrucción `proper_defeat` (ver figura 7.14), se pone en TRUE sólo si el derrotador hallado es un derrotador propio del argumento de la consulta.

El registro `hubo_inconsistencia` se pone en TRUE cada vez que se ejecuta la instrucción `prove_consistency_else`, (*i.e.*, cuando se comienza a verificar la consistencia), y se pone en FALSE cuando se ejecuta la instrucción `end_consistency_proof`, (*i.e.*, cuando se ha culminado con éxito la prueba de consistencia). De esta forma si hay una falla mientras se está verificando la consistencia, el registro `hubo_inconsistencia` quedará en TRUE.

El registro `hubo_bloqueo`, es puesto en FALSE cada vez que se ejecuta la instrucción `prepare_to_defeat K`, y `K` vale 1. La instrucción `proper_defeat`, lo pone en TRUE sólo si el derrotador hallado es un derrotador de bloqueo del argumento de la consulta.

La figura 7.17 muestra como se obtiene la respuesta de una consulta, en función de los registros anteriores.

Capítulo 8

Conclusiones y resultados obtenidos

Tanto la implementación de métodos de razonamiento, como la representación de conocimiento, siguen siendo problemas abiertos, y constituyen áreas centrales dentro de investigación en Inteligencia Artificial (IA), ya que cualquier avance en estas áreas, redundará en un beneficio casi imnediato para muchas otras áreas de las Ciencias de la Computación. El estudio del uso la programación en lógica como herramienta de representación de conocimiento, demostró que a pesar de ser uno de los lenguajes de programación que más se adapta para representar conocimiento, presenta limitaciones para adaptarse a ciertas características del razonamiento del sentido común. En los últimos años se han desarrollado extensiones de la programación en lógica, pero de su estudio se concluyó que igual presentaban problemas. Después de realizar un estudio minucisoso sobre los nuevos métodos de razonamiento y de representación de conocimiento que produjo la IA en los últimos años, se eligió a la argumentación rebatible, como formalismo de partida para solucionar los problemas mencionados.

Por lo tanto, se desarrolló un lenguaje de programación en lógica rebatible como una extensión de la programación en lógica convencional. Este nuevo lenguaje permite capturar aspectos del razonamiento que son difíciles de expresar en la programación en lógica tradicional, y en las extensiones estudiadas. Los programas lógicos rebatibles (PLR), están formados por dos tipos de cláusulas, que permiten representar conocimiento estricto e información rebatible, y utilizan los conceptos de la argumentación rebatible a fin de poder decidir entre metas contradictorias. En los PLR, es posible utilizar dos tipos de negación: la negación por falla finita y la negación clásica, esto permite representar información in-

completa, y potencialmente inconsistente. Además, la hipótesis de mundo cerrado (CWA) se puede representar directamente como cláusulas de programa.

Una vez definido el nuevo lenguaje, se diseñó un conjunto de algoritmos, que permite construir por encadenamiento hacia atrás de cláusulas de programa, argumentos y contrargumentos, utilizando únicamente la información contenida en los argumentos involucrados. De esta forma, el costo de obtener una justificación esta relacionado únicamente a los argumentos que forman el árbol de dialéctica, y es independiente del tamaño del programa.

Los PLR incorporaron algunos conceptos que no estaban presentes en los sistemas de argumentación rebatible, como la negación por falla, y las presuposiciones. Por lo tanto, se extendió la definición del criterio de especificidad, utilizado en la argumentación rebatible, de tal manera que pueda comparar correctamente argumentos que contengan presuposiciones y el operador de negación por falla.

Se diseñó una máquina abstracta (llamada JAM), para la implementación de la programación en lógica rebatible. La JAM está diseñada como una extensión de la WAM, y está formada por un conjunto de intrucciones, una estructura de memoria, y un conjunto de registros, que permiten construir argumentos, contraargumentos y generar el árbol de dialéctica necesario para obtener una justificación.

Además del desarrollo de la nueva máquina abstracta, se implementó un compilador que toma como entrada un programa lógico rebatible (PLR), y produce como salida un programa en instrucciones de la JAM. Para realizar este trabajo, se combinaron técnicas convencionales de compilación, como el análisis recursivo descendente, con técnicas especiales para la programación en lógica. El compilador construído puede utilizarse también para traducir programas en Prolog a código WAM, y podría modificarse fácilmente para producir compiladores para otras extensiones de la programación en lógica.

Por último, se implementó la nueva máquina abstracta, como una máquina virtual, sobre la cúal es posible ejecutar programas lógicos rebatibles. La implementación del lenguaje se utilizó para ejecutar PLRs, que permitieron refinar la definición del lenguaje.

Por lo expresado antes, es posible afirmar, que los objetivos propuestos en esta tesis, fueron alcanzados.

Bibliografía

[Aït91] Aït-Kaci Hassan Warren's abstract machine, a tutorial reconstruction. MIT Press, 1991.

[AlfPer93] Alferes José J. Pereira Luis M. Contradiction: when avoidance equals removal (part I and II). Proc. of Extensions of Logic Programming, 4th International Workshop ELP'93 St. Andrews U.K. March 1993.

[Ashley89] Ashley, K. Defining salience in Case-Based arguments. Proceedings IICAI, 1989 .

[BatMel73] Battani G. y Meloni H., Interpreteur du Language de Programmation PROLOG, Groupe de Recherche en Intelligence Artificielle, Universit'e d'Aix-Marseille, 1973.

[BlaSub89] Blair H. and Subrahmanian V. Paraconsistent logic programming. Theoritical Computer Science, 68:135-154,1989.

[Bratko87] Bratko I. Prolog Programming for Artificial Intelligence. Adisson-Wesley 1987.

[Brewka89] Brewka G. Preferred Subtheories: An Extended Logical Framework for Default Reasoning. Gesellschaft für Mathematik und Datenverarbeitung. (in IJCAI, 1989).

[Brewka94] Brewka G. A Reconstruction of Rescher's Theory of Formal Disputation Based on Default Logic. 1994

[BreGor] Brewka G. y Gordon T. How to buy a Porsche: An approach to defeasible decision making (extended abstract).

[CGS93] C. I. Chesñevar, A. J. García y G. R. Simari, Sistemas expertos basados en argumentos , IV Encuentro Académico Tecnológico, Universidad Nacional del Nordeste, Resistencia, Chaco, 1993.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel y R. Pasero, Un Systeme de Communication Homme-Machine en Francais, Groupe de Recherche en Intelligence Artificielle, Universit'e d'Aix-Marseille, 1973

[Das92] Das S. Deductive Databases and Logic Programming. Addison-Wesley 1992

[Davis80] Davis, M. The mathematics of non-monotonic reasoning. Artificial Intelligence 13, 73-80, 1980

[DSS93] Davis, M., Shrobe, H. y Szolovits, P. What is a knowledge representation? AAAI, 1993

[Debray86] Debray S. K. Register Allocation in a Prolog Machine. Symposium on Logic Programming pp.267-275 IEEE 1986

[DoyPat91] Doyle, J. y Patil, R. Two theses cases of knowledge representation: language restrictions, taxonomic classification and the utility of representation services. Artificial Intelligence 48, 261-267, 1991.

[Doyle] Doyle, J. A truth maintenance system Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, USA.

[DubPra87] Dubois, D. y Prade, H. Necessity measures and resolution principle. IEEE Transactions on systems, man and cybernetics, SMC17 N 3, 1987

[Dung91] Dung Phan M. Negations as Hypotheses: an Abductive Foundation for Logic Programming. Proc. of 8th. Int. Conf. on Logic Programming (ICLP) 1991.

[Dung93a] Dung Phan M. An Argumentation Semantics for Programming with Explicit Negation. Proc. of 10th. Int. Conf. on Logic Programming (ICLP) 1993.

[Dung93b] Dung Phan M. On the Acceptability of Argumets and its Fudamental Role in Nonmonotonic Reasoning and Logic Programming. Proc. of Int. Join Conf. on Artificial Intelligence (IJCAI), Chambéry, France, 1993.

[Dung93c] Dung P. M. Representing actions in logic programming and its applications in database updates. Proceedings of the 10th Int. Conf. on Logic Programming ICLP, 1993.

[Elkan90] Elkan C. A rational reconstruction of nonmonotonic truth maintenance systems Artificial inteligence, 43, 210-234, 1990. November 1992.

[Elkan92] Elkan C. The qualification problem, the frame problem and nonmonotonic logic Dep. of Computer Science and Engineering, University of California, November 1992.

[Elkan93] Elkan, Charles. The paradoxical success of fuzzy logic. National Conference on Artificial Intelligence, julio de 1993.

[Elkan94a] Elkan Charles, Reply to comments on The Pardoxical Sucess of Fuzzy Logic, Dep. of Computer Science and Engineering, University of California, May 1994.

[Elkan94b] Elkan, Charles. Reply to Comments on "The paradoxical success of fuzzy logic". *IEEE Expert, agosto de 1994.*

[EthRei83] Etherington D. y Reiter R. On inheritance with exceptons. Proceedings AAAI, 1983

[Fillo95] Fillottrani, Pablo R. Sistemas de razonamiento no monótono y su relación con la semántica de las bases de datos deductivas. Tesis de Magister en Ciencias de la Computación, Bahía Blanca, Universidad Nacional del Sur, Octubre de 1995.

[FikNil71] Fikes, R. y Nilsson, N. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Inteligence, 2 189-208, 1971

[Gabbay82] Gabbay, D. Intuitionistic basis for non-monotonic logic. Lectures Notes in Computer Science, 6th Conference on Atomated Deduction, NY, USA, 1982

[GMN79] Gallaire, H., Minker, J. y Nicolas, J. Logic databases 1979

[GMN84] Gallaire, H., Minker, J. y Nicolas, J. Logic and databases: a deductive approach. 1984

[GCS93a] García A. J., Chesñevar C. I., and Simari G. R. *Bases de Argumentos: su mantenimiento y revisión.* XIX Conferencia Latinoamericana de Informática. Buenos Aires, August 1993.

[GCS93b] García A. J., C.I.Chesñevar, G.R.Simari. Making argument systems computationally attractive. *XIII Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computación* – Octubre de 1993.

[Garcia94] García A. J. Evitando falacias en los sistemas argumentativos. 1er. Workshop Argentino sobre Aspectos Teóricos de la Inteligencia Artificial, Universidad de Buenos Aires, Capital Federal, abril de 1994.

[GarSim95a] García A. J. Una aproximación a la programación en lógica rebatible, Segundo Workshop Argentino sobre Aspectos Teóricos de la Inteligencia Artificial (ATIA'95), Universidad Nacional del Sur, Bahía Blanca Octubre de 1995.

[GarSim95b] García A. J. y Simari G. R. Compilacion de programas lógicos que utiliza la negación por falla: una extensión a la máquina abstracta de Warren. 1er. Congreso Argentino de Ciencias de la Computación. Bahía Blanca, Noviembre de 1995.

[GarSim96a] García A. J. y Simari G. R. Diferentes formas de hipótesis de mundo cerrado en la programación en lógica rebatible. Segundo Congreso Argentino de Ciencias de la Computación (CACiC'96), Universidad Nacional de San Luis, Noviembre de 1996.

[GarSim96b] García A. J. y Simari G. R. El criterio de especificidad en la programación en lógica rebatible. Tercer Workshop Argentino sobre Aspectos Teóricos de la Inteligencia Artificial (ATIA'96), Universidad Nacional de San Luis, Noviembre de 1996.

[Garden] Gardenfors, P. Belif revision and nonmonotonic logic: Two sides of the same coin? Department of Philosophy, University of Lund, Sweden.

[Geffner89] Geffner, H. Inheritance = chaining + defeat (extended abstract). Proceedings of the Workshopon Formal Aspects of Semantic Networks, Catalina, 1989

[GelLif90] Gelfond M. and Lifschitz V. Logic Programs with Classical Negation. Proc. of 7th. Int. Conf. on Logic Programming (ICLP) 1990

[GelLif88] Gelfond M. and Lifschitz V. The stable model semantics for logic programmming. Proc. of 5th. Int. Conf. on Logic Programming and Symp. 1988

[George87] Georgeff, M. Planning. Ann. Rev. Sci. 2, 359-400, 1987.

[GoySho93] GoyalÑ. and Shoham Y. Reasoning Precisely with Vague Concepts. Proc. 11th. Int. Conf. Artificial Intelligence, 1993

[Green69] C. Green , Applications of Theorem Proving to Problen Solving. IJCAI-69 Washington, D.C., 1969 219-239.

[GKF93] Goransson, M., Krause, P. and Fox, J. Acceptability of arguments as logical uncertainty. ECSQARU, 1993

[Hage] Hage, J. Introducing dialogical reason based logic.

[Hansson94] Hansson Sven Ove, A textbook of Belief Dynamics, 1994.

[Hayes73] P. J. Hayes, Computation and Deduction. Prc. MFCS Conf. Czechoslovak Academy of Science, 1973.

[HuXia90] Hu Ziang, Xiang Rong Ci An Imporved Register Allocation Algorithm for the WAM Model. IEEE Proc. Tools for AI'90

[Inoue91] Inoue K. Extended Logic Programming with Default Assumptions. Proc. of 8th. Int. Conf. on Logic Programming (ICLP) 1991.

[JDM88] Janssens G. Demoen B. and Martën A. Improving the Register Allocation in WAM by reordening Unification. Proc 5th. Conf. Symp. Logic Programming pp.1388-1402 MIT Press 1988.

[Kleer86] Kleer, J. An assumption-based TMS. Artificial Inteligence 28,127-162, 1986.

[Konolige] Konolige, K. Defeasible argumentation in reaoning about events.

[KowSad90] Kowalski R. and Sadri F. Logic Programs with Exceptions. Proc. of 7th. Int. Conf. on Logic Programming (ICLP) 1990

[Kowa74] R. A. Kowalski, Predicate Logic as a Programming Language. Information Processing 74, Stockholm, North Hollannd, 1974, 569-574.

[KowToni] Kowalski, R. y Toni, F. Arguments and reconciliation.

[KurKir86] Kurosawa S. A. K. and Kiriyama K. A new Optimization Technique for a Prolog Compiler. IEEE COMPCON'86 pp.241-245

[Leves86] Levesque, H. Knowledge representation and reasoning. Ann. Rev. Comput. Sci. 1, 255-87, 1986.

[Leves90] Levesque, H. All I know: a study in autoepistemic logic. Artificial Intelligence 42, 263-309, 1990.

[LinSho89] Lin F. and Y Shoham. Argument Systems: a uniform basis for nonmonotonic reasoning, STAN-CS-89-1243, Stanford University, Department of Computer Science,1989.

[LinSho92] Lin F. and Y Shoham. Alogic of knowledge and justified assumptions. Artificial Intelligence 37, 271-289, 1992.

[Lloyd87] Lloyd J. W. Foundations of Logical Programming. 2nd. edition, Springer-Verlag 1987

[Loui87] Loui R. P. Defeats among arguments: a system of defeasible inference Comp. Intell. 3, 100-106, 1987

[Loui88] Loui R. P. The curse of frege. Standford University.

[Loui92] Loui R. P. Analogy, Decision, and Theory-Formation as Defeasible Reasoning. Department of Computer Science, Washington University, Saint Louis,

[Loui93] Loui R. P. Rationales and Argument Moves. Department of Computer Science, Washington University, Saint Louis,

[Love78] Loveland D. Automated Theorem Proving: A Logical Basis . North Holland, New York, 1978.

[Mac85] MacKenzie J. D. The dialectics of Logic. Philosophy Department, University of Wollongong, New South Wales. 1985

[MJT91] Ludek Matyska, Adriana Jergová, David Toman. Register Allocation in WAM. Proc. 8th. ICLP Paris 1991

[MJT92] Ludek Matyska, Adriana Jergová, David Toman, Petr, Mejzlík, Václav Racansky, Jirí Zlatuska. WAM Based Effincient Implementation of Logiic Programming Languages - Final Report. Reporte interno, IBM academic iniciative research project.

[McCar80] McCarthy J. Circunscription – A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13: 27–39,1980.

[McCar] McCarthy J. Mathematical logic in artificial intelligence.

[McCar86] McCarthy J. Application of circumscription to formalizing common-sense knowledge Artificial Intelligence 28, 89-116, 1986.

[McDDoy80] McDermott D. y J. Doyle. Non-Monotonic Logic I. Artificial Intelligence, 13: 41–72,1980.

[McDer82] McDermott D. Nonmonotonic logic II: nonmonotonic modal theories Journal of the Association for Computing Machinery 29, 33-57, 1982.

[Moore85] Moore R. C. Semantical Considerations on Nonmonotonic Logic Artificial Intelligence, 25:(1) 75–94,1985.

[Nute1] Nute D. Counterfactuals and the similarity of worlds. Journal of Philosophy 72:773-778

[Nute2] Nute D. Topics in Conditional Logic. Dordrecht Holland: Reidel Publishing Company

[Nute84] Nute D. Non-monotonic reasoning on conditional logic. ACMC Research Report 01-0002, 1984. University of Georgia.

[Nute86a] Nute D. LDR: a logic for defeasible reasoning. ACMC Research Report 01-0013, 1986 University of Georgia.

[Nute86b] Nute D. and Covington M. Inplicture, Disjuntion, and Non-monotonic Logic, ACMC Research Report 01-0015, 1986. University of Georgia.

[Nute86c] Nute D. and Lewis M. d-PROLOG: a user's manual. ACMC Research Report 01-0017, 1986 University of Georgia.

[Nute87] Nute D. Defeasible Reasoning, 1987, en Proceedings of the 20th Annual Hawaii Conference .

[Nute88] Nute D. Defeasible Reasoning. A Philosophical Analysis in PROLOG, 1988, Aspects of Artificial Intelligence.

[Nute92] Nute Donald, *Basic defeasible logic*, in *Intensional Logics for Programming*, Ed by Luis Fariñas del Cerro, Claredon Press – Oxford (c) 1992.

[Pollock87] J. L. Pollock. Defeasible Reasoning. Cognitive Science, 11:481–518, 1987.

[Pollock90] Pollock J. L. A Theory of Defeasible Reasoning, Int. J. Intelligence Systems, 1990 33-34

[Pollock92] Pollock J. L. How to reason defeasibly, Artificial Intelligence 57, 1992.

[Pollock94] Pollock J. L. Justification and Defeat. Artificial Intelligence 67, 1994 (337-407).

[Poole85a] Poole D. L. On the Comparison of Theories: Preferring the Most Specific Explanation. En Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pags 144–147, IJCAI, 1985.

[Poole85b] Poole D. L., R. Aleliunas y R. Goebel. THEORIST: A Logical Reasoning System for Defaults and Diagnosis Technical Report, Departament of Computer Science, University of Waterloo, Waterloo, Canada, 1985.

[Poole88] Poole D. L. A logical framework for default reasoning. Artificial Intelligence 36 (1988).

[Prakken] Prakken H. Logical Tools for Modelling Legal Arguments. Vrije University, Amsterdam, Holanda. Enero de 1993.

[RasMar89] Rasiowa, H. y Market, W. On reaching consensus by groups of intelligent agents. Proceedings Int. Symp. Methodologies Intelligent Systems, ISMIS, 1989.

[Rasiowa91] Rasiowa, H. Mechanical proof systems for logic: reaching consensus by groups of intelligen agents. International Journal of approximate reasoning 5, 415-432, 1991.

[Reiter80] Reiter R. A Logic For Default Reasoning. Artificial Intelligence, 13: 81–132,1980.

[Reiter87] Reiter R. N onmonotonic Reasoning. Ann. Rev. Comput. Sci. 2, 147-86, 1987.

[Rescher77] RescherÑ. Dialectics: A Controversy-Oriented Approach to the Theory of Knowledge. Ed. State University of New York Press, Albany, 1977.

[Roos92] Roos, N. A logic for reasoning with inconsistent knowledge Artificial Intelligence 57, 69-103, 1992.

[Rousel75] P .Rousel, PROLOG: Manuel de Reference et d'Utilization, Groupe de Recherche en Intelligence Artificielle, Universit'e d'Aix-Marseille, 1973.

[Shoam88] Shoham, Y. Chronological ignorance: experiments in nonmonotonic temporal reasoning, Artificial Intelligence 36, 279-331, 1988.

[SimLou92] Simari G. R. y R. P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. Artificial Intelligence, 53: 125–157,1992.

[SCG94] Simari G. R. , Chesñevar C. I., and García A. J. Focusing Inference in Defeasible Argumentation. IV Iberoamerican Congress on Artificial Intelligence, October 1994.

[SCG94] Simari G. R. , Chesñevar C. I., and García A. J. The Role of Dialectics in Defeasible Argumentation. XIV Int. Conf. of Chilean Computer Science Society, November 1994.

[SimGar95] Simari G. R. y García A. J. A Knowledge Representation Language for Defeasible Argumentation. XXI Conferencia Latinoamericana de Informática. Canela, Brasil, Agosto 1995.

[SimGar95] Simari G. R. and García A. J. Protocolos para arbitrar debates entre agentes inteligentes, *24as Jornadas Argentinas de Informática e Investigación Operativa*. Buenos Aires, agosto de 1995.

[SteSha86] Stearling L. y Shapiro E. The Art of Prolog Cambridge MA: MIT Press 1986

[Umrigar90] Umrigar Z.D. Finding Advantageous Orders for Argument Unification for the Prolog WAM Proc. NACLP'90 pp.80-96 MIT Press

[VanGel] Van Gelder, A. Negation as a failure using tight derivations for general logic programs Fundations of deductive databases adn logic programming, edited by Jack Minker, Morgan Kaufmann Publishers, Inc.

[VanRoy90] Van Roy Peter L. Can Logic Programmign Execute as Far as Imperative Programming? Ph.D Thesis 1990, Berkeley University.

[Vrees91] Vreeswijk G. The Feasibility of Defeat in Defeasible Reasoning, 1991, in Knowledge Representation '91.

[Vrees93] Vreeswijk G. Studies in Defeasible Argumentation (Ph.D. Thesis). Vrije University, Amsterdam, Holanda. March 1993.

[Warren77] Warren David H. D. Compiling Predicate Logic Programs, Volumenes 1 y 2, D.A.I. Research Reports Nro. 39. Mayo de 1977.

[Warren83] Warren David H. D. An Abstract Prolog Instruction Set. Thechnical Note 309, SRI International, Menlo Park, CA, October 1983.

[Warren88] Warren David H. D. Implementation of Prolog. Lectures Notes Tutorial Nro 3 5th International Comference and Symposium on Logic Programming, Seattle, WA 1988.

[Zadeh] Zadeh L. A. Commonsense and Fuzzy Logic. Department of Electrical Engienering and Computers Sciences, University of California.