Universidad Nacional del Sur

Tesis de Doctor en Ciencias de la Computación

Programación en Lógica Rebatible: Lenguaje, Semántica Operacional, y Paralelismo

Alejandro Javier García

Bahía Blanca Argentina

2000

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor 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 noviembre de 1997 y el 10 de noviembre de 2000, bajo la dirección del Dr. Guillermo R. Simari, Profesor Titular del Departamento de Ciencias de la Computación.

Mg. Alejandro Javier García
agarcia@cs.uns.edu.ar
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca, 13 de Noviembre de 2000.

Agradecimientos

En primer lugar, quiero expresar mi especial agradecimiento a Guillermo Simari, con quien he tenido el gusto de trabajar desde 1992, y quien me ofreció la oportunidad de realizar mi formación científica en Ciencias de la Computación bajo su dirección, compartiendo su conocimiento y su amistad. Esta tesis no hubiera sido posible sin el apoyo y la dedicación que supo ofrecerme.

Agradezco a mis compañeros del Departamento de Ciencias de la Computación por su amistad, el entorno de trabajo creado, y por su predisposición para confrontar y debatir ideas. Especialmente a Carlos Chesñevar, por todos estos años que hemos compartido aprendiendo y trabajando juntos, y por las jugosas discusiones que siempre mantenemos. A Marcelo Falappa, por su amistad incondicional. A Marcela Capobianco, Alejandro Stankevicius, y Diego Martinez, por todas las tardes de argumentación en nuestra oficina. A Pablo Fillottrani, Juan Carlos Augusto, Claudio Delriaux, y Laura Cecchi, compañeros del Laboratorio de Investigación en Inteligencia Artificial. Agradezco a quienes fueron mis profesores: Perla Señas, Tona Moroni, Rafael García, Marcelo Zanconi, Jorge Ardenghi, Silvia Castro, Sonia Rueda, y Elsa Estevez, por haber compartido su conocimiento y permitirme aprender de ellos. A todos los docentes con quienes tuve el gusto compartir las numerosas cátedras en las que he trabajado, en especial a Leo De Matteis y Sergio Martig. A todos ellos gracias por permitirme sentirme orgulloso de trabajar en nuestro Departamento.

Agradezco a las instituciones que confiaron en mí, otorgándome su apoyo económico para que pueda concretar este trabajo: la Universidad Nacional del Sur, el CONICET, la Fundación OSDIC y el FOMEC.

Agradezco a los siguientes investigadores con quienes tuve el honor de debatir ideas en persona, y contribuyeron directamente al desarrollo de esta tesis: Ronald P. Loui, Veronica Dahl, Henry Prakken, Bart Verheij, Simon Parsons, John Pollock, Francesca

Toni, y Micheal Gelfond. Y muy especialmente a Hassan Aït-Kaci, Manuel Hermenegildo, Grigoris Antoniou, y Paul Tarau, por haberme recibido en sus respectivas Universidades y permitirme trabajar con ellos. También agradezco a los integrantes del jurado que evaluaron este trabajo, Ronald Loui, Raimundo Forradelas y Fernando Tohme, por sus valiosos comentarios.

Quiero agradecer a mis padres por su cariño, dedicación y preocupación por mi formación como persona. Por el esfuerzo que ponen siempre en resignar sus propias satisfacciones por darle lo mejor a sus hijos. Un esfuerzo mucho mayor a la realización de cualquier tesis, y en el que hacen un excelente trabajo.

El agradecimiento más importante es para Cori, por permitirme compartir todo con ella y poder sentirme realmente completo.

Finalmente agradezco a toda mi familia, a los que ya no están pero permitieron que yo esté, a los que están siempre –incondicionales– para compartir las alegrías, y a los que fueron llegando para recordarnos que vale la pena.

Gracias, muchas gracias a todos.

Ale, diciembre de 2000.

Resumen

Esta tesis define formalmente a la Programación en Lógica Rebatible, y su semántica operacional. Define además, la computación en paralelo de Programas Lógicos Rebatibles, y una extensión de la Programación en Lógica Rebatible que permite trabajar con dos tipos de negación. Los resultados obtenidos son comparados con formalismos existentes y que están relacionados en gran medida con nuestro desarrollo.

El lenguaje definido puede ser utilizado básicamente en cualquier aplicación que requiera representación de conocimiento. Sus características le permiten modelar conocimiento que involucre información incompleta o potencialmente contradictoria. El mecanismo de inferencia sobre el cual está basado, permite decidir entre conclusiones contradictorias y adaptarse fácilmente a entornos cambiantes.

Las características antes mencionadas, hacen de los Programas Lógicos Rebatibles, una herramienta atractiva para representar el conocimiento y definir el razonamiento de agentes inteligentes deliberativos. La flexibilidad del lenguaje permite que los agentes trabajen en un entorno donde la información cambia rápidamente. Como parte de esta tesis, se describe una aplicación concreta que consiste de un sistema de múltiples agentes, con agentes especializados en extraer información del mercado de valores a través de Internet, y agentes deliberativos que toman decisiones en función de la información recolectada.

Índice General

Capítulo 1

Introducción

En los últimos años, varios investigadores centraron su atención sobre el área de Argumentación Rebatible, y su posible combinación con la Programación en Lógica. Las nuevas propuestas han intentado solucionar los problemas de cada una de estas áreas, utilizando los formalismos existentes en la otra, pero sin definir un sistema combinado. En esta tesis, desarrollamos un nuevo formalismo que reune características de las áreas citadas antes, capturando además los avances producidos por otros investigadores.

Esta tesis define formalmente a la Programación en Lógica Rebatible, y su semántica operacional. Define además, la computación en paralelo de Programas Lógicos Rebatibles, y una extensión de la Programación en Lógica rebatible que permite trabajar con dos tipos de negación. Se muestran también aplicaciones concretas del lenguaje definido, como la implementación de un sistema de multiples agentes inteligentes en el dominio del mercado de valores por Internet. Los resultados obtenidos son comparados con formalismos existentes y que están relacionados en gran medida con nuestro desarrollo.

El contenido de esta tesis conjuga además aspectos de varias áreas de Ciencias de la Computación, las cuales han tenido un notable desarrollo en los últimos años: *Programación en Lógica, Argumentación Rebatible, Agentes Inteligentes*, y *Paralelismo en Programación en Lógica*. Muchos de los resultados obtenidos durante el desarrollo de esta tesis han sido publicados en los siguientes artículos [GS98, GS99c, GS99b, GS99a, GGTS00].

1.1 Contribuciones

En primer lugar, se define formalmente el lenguaje de la Programación en Lógica Rebatible, como una extensión de los desarrollos del área de programación en lógica [Lif96, DK95, TK95, GS97], y bajo la óptica de los últimos avances en el área de argumentación rebatible [Pol96a, AMB00, Lou97b, BDKT97, Dun95, PS97]. Se demuestran propiedades del lenguaje definido, se define un nuevo criterio de comparación de argumentos, y se consideran nuevas restricciones sobre las lineas de argumentación.

Se presenta además una semántica operacional para la Programación en Lógica Rebatible. La idea de una semántica operacional [Mey90], es expresar el significado de un lenguaje, definiendo un mecanismo que permita determinar el efecto de todo programa escrito en ese lenguaje. Esta descripción operacional, consiste de un dispositivo formal, o "autómata intérprete", capaz de ejecutar formalmente cualquier programa. Dicho autómata intérprete puede variar desde una descripción matemática abstracta hasta una descripción concreta mediante otro lenguaje de programación. Aquí tomaremos la segunda opción. El lenguaje que utilizaremos para realizar esta descripción operacional será PROLOG, debido a su cercanía conceptual con la Programación en Lógica Rebatible, y por tratarse de un lenguaje bien conocido por la comunidad científica en argumentación rebatible.

Otra de las contribuciones de esta tesis, es el estudio de formas implícitas de paralelismo en la argumentación rebatible. El uso de paralelismo en Programación en Lógica ha sido objeto de estudio en los últimos 20 años [Gup94, GKA$^+$94, Her86a, Her86b, Gre87, DeG84]. Sin embargo, el uso de paralelismo en Sistemas de Argumentación Rebatible no ha sido estudiado hasta el momento, y constituye una nueva área de investigación, que surge del desarrollo de esta tesis. La definición del cómputo en paralelo de Programas Lógicos Rebatibles, captura las formas de paralelismo implícito estudiadas para la Programación en Lógica, y define nuevas fuentes de paralelismo que pueden explotarse en un sistema de argumentación rebatible.

Finalmente otra contribución importante es un nuevo tratamiento de la negación default en los programas lógicos rebatibles. El uso de la negación resulta necesario en cualquier sistema de representación de conocimiento. La Programación en Lógica Rebatible inicialmente contempla el uso de un operador de negación llamado *negación fuerte*, que permite la representación de conocimiento contradictorio. Aunque la negación fuerte

fue utilizada por todos los sistemas de Argumentación Rebatible desde sus comienzos, el uso de negación fuerte en Programación en Lógica, fue introducida recientemente, en la definición de la Programación Lógica Extendida dada por Gelfond y Lifschitz [GL90]. Sin embargo, los formalismos de Programación en Lógica han utilizado desde sus inicios, otro operador de negación conocido como *negación default*, y que permite la representación de información incompleta. El uso de negación default en los sistemas de Argumentación Rebatible es muy reciente [Dun93b, GS95, PS97], y aún continúa en estudio. Es esta tesis se define la Programación en Lógica Rebatible Extendida que permite utilizar los dos tipos de negación. A diferencia de desarrollos anteriores [SG95, Gar97, GS99c], en esta tesis se utiliza un enfoque diferente, definiendo a la negación default como una nueva forma de ataque dentro del formalismo argumentativo.

1.2 Aplicaciones

La Programación en Lógica Rebatible es un lenguaje que puede ser utilizado básicamente en cualquier aplicación que requiera representación de conocimiento. Sus características le permiten modelar conocimiento que involucre información incompleta o potencialmente contradictoria. El mecanismo de inferencia sobre el cual está basado permite decidir entre conclusiones contradictorias, y adaptarse fácilmente a entornos cambiantes. Esto es, aplicaciones donde se agrega o quita información en forma dinámica, y por lo tanto conclusiones obtenidas previamente pueden quedar invalidadas.

Las características antes mencionadas, hacen de los Programas Lógicos Rebatibles, una herramienta atractiva para representar el conocimiento y definir el razonamiento de *agentes inteligentes deliberativos*. La flexibilidad del lenguaje permite que los agentes trabajen en un entorno donde la información cambia rápidamente. Describiremos en el capítulo 8, una aplicación concreta donde se utilizó la Programación en Lógica Rebatible para construir agentes inteligentes deliberativos. La aplicación consiste de un sistema de múltiples agentes, con agentes especializados en extraer información del mercado de valores a través de Internet, y agentes deliberativos, que toman decisiones en función de la información recolectada. Otras aplicaciones se encuentran actualmente en desarrollo.

1.3 Organización de la tesis

Esta tesis está organizada de la siguiente manera:

Capítulo 2: En este capítulo se define la *Programación en Lógica Rebatible*. En primer lugar se introduce la sintaxis del lenguaje de los programas lógicos rebatibles, que permite representar información tentativa y potencialmente contradictoria. Luego se presenta un formalismo de *argumentación rebatible* para definir un procedimiento de prueba para el lenguaje.

Capítulo 3: Se muestra como representar en nuestro lenguaje, ejemplos que se han convertido en un estándar para demostrar las virtudes, o defectos, de un formalismo de razonamiento no monótono, o de un sistema de argumentación rebatible.

Capítulo 4: En este capítulo se presenta la semántica operacional de la Programación en Lógica Rebatible. Cada uno de los elementos del procedimiento de prueba de la Programación en Lógica Rebatible será descripto como un programa en PROLOG.

Capítulo 5: Se muestra como la definición del cómputo en paralelo de Programas Lógicos Rebatibles, captura las formas de paralelismo implícito estudiadas para la Programación en Lógica, y define nuevas fuentes de paralelismo que pueden explotarse en un sistema de argumentación rebatible.

Capítulo 6: En este capítulo se mostrará la importancia de extender la programación en lógica rebatible, agregando la negación default. Se extenderán las reglas de programa para poder utilizar los dos tipos de negaciones, y se definirá la forma en que la negación default será considerada.

Capítulo 7: El objetivo de este capítulo es mostrar resumidamente desarrollos similares, comparándolos con nuestra propuesta. En la primera parte del capítulo se describirán sistemas de Argumentación Rebatible, y en la segunda parte, extensiones de la Programación en Lógica similares a nuestra propuesta.

Capítulo 8: Se mostrará como puede utilizarse la Programación en Lógica Rebatible para definir el razonamiento de *agentes inteligentes deliberativos*. La flexibilidad del lenguaje permitirá que los agentes trabajen en un entorno donde la información cambia rápidamente.

Capítulo 9: Finalmente se detallan los resultados y conclusiones obtenidas.

Capítulo 2

Programas Lógicos Rebatibles

En este capítulo se define la *Programación en Lógica Rebatible* (en inglés Defeasible Logic Programming) y que abreviaremos DeLP. En primer lugar se introduce la sintaxis del lenguaje de los programas lógicos rebatibles, que permite representar información tentativa y potencialmente contradictoria. Luego se presenta un formalismo de *argumentación rebatible* para definir un procedimiento de prueba para DeLP. Este formalismo fue propuesto originalmente por Simari y Loui en [Sim89, SL92], y luego extendido por García, Chesñevar y Simari en [SCG94, GSC98]. No obstante, algunos conceptos serán nuevamente extendidos o modificados en esta tesis.

Un programa lógico rebatible estará formado por un conjunto de *hechos*, *reglas estrictas*, y *reglas rebatibles*. Permitiendo estas últimas la representación de información tentativa. Toda conclusión del programa deberá ser sustentada por algún *argumento* que pueda construirse utilizando las reglas y los hechos del programa. Cuando se utilicen reglas rebatibles para derivar una conclusión C, esta conclusión será tentativa, y podrá ser refutada por información que la contradiga. Para decidir cuando una conclusión C puede aceptarse a partir de un programa lógico rebatible, se realizará un *análisis dialéctico*, construyendo argumentos a favor y en contra de la conclusión C.

Un argumento que sustenta una conclusión C podrá ser atacado por otros argumentos *derrotadores* que lo contradigan y que puedan construirse a partir del programa. Dichos derrotadores podrán a su vez ser atacados, y así sucesivamente, generando una secuencia de argumentos llamada *línea de argumentación*. Cada línea de argumentación deberá satisfacer ciertas propiedades, a fin de evitar que se generen argumentaciones falaces. El

proceso completo considera para cada argumento todos sus posibles argumentos derrotadores, lo cual, en lugar de una única línea de argumentación, genera un conjunto de líneas representadas con un *árbol de dialéctica*. Este análisis dialéctico determinará si la conclusión en cuestión está *garantizada* o no, a partir del programa.

2.1 Conceptos Preliminares

En la sección siguiente se introducirá el lenguaje de la Programación en Lógica Rebatible (Defeasible Logic Programming o DeLP). En esta sección introduciremos conceptos y terminología estándar de lógica y programación en lógica, que son necesarios para definir nuestro lenguaje.

Definición 2.1 (Signatura) [Fil01]
Una signatura es una tupla $\sigma = \langle \mathcal{V}, Func, Pred \rangle$ *donde* $\mathcal{V}$*,* $Func$*, y* $Pred$ *son conjuntos finitos que representan variables, funciones y predicados respectivamente.*

Dada una signatura σ, una función llamada "*aridad*" le asignará a cada elemento de $Func$ y $Pred$ un número natural. Si $f \in Func$ y $aridad(f) = 0$, entonces f se denominará *constante*, por otro lado, si $p \in Pred$, y $aridad(p) = 0$, entonces p se denominará *proposición*.

Definición 2.2 (Alfabeto) [Fil01]
El alfabeto generado a partir de una signatura σ*, consiste del conjunto de elementos miembros de la signatura, el símbolo de negación "*$\sim$*", y los símbolos de puntuación "(", ")", y ",".*

Definición 2.3 (Término) [Fil01]
Sea $\sigma = \langle \mathcal{V}, Func, Pred \rangle$ *una signatura. Un término* T*, y respectivamente sus compontes* $comp(T)$*, se definen inductivamente como sigue:*

- *toda variable* $V \in \mathcal{V}$*, es un término, y* $comp(V) = \{V\}$
- *toda constante* $c \in Func$ $(aridad(c) = 0)$*, es un término, y* $comp(c) = \{c\}$

- *si* $f \in Func$, $aridad(f) = n$ *con* $n \geq 1$, *y* $t_1, \ldots, t_n$ *son términos, entonces* $f(t_1, \ldots, t_n)$ *también es un término.*
  Se define $comp(f(t_1, \ldots, t_n)) = \{f\} \cup (\bigcup_{1 \leq i \leq n} comp(f_i))$.

Como se verá a continuación en DeLP los términos no utilizarán variables, por este motivo introducimos la siguiente definición.

Definición 2.4 (Término fijo) [Fil01]
Sea $\sigma = \langle \mathcal{V}, Func, Pred \rangle$ *una signatura, y* T *un término.* T *será un término fijo*[1] *si no contiene variables. Esto es,* $comp(T) \cap \mathcal{V} = \emptyset$.

Definición 2.5 (Átomo) [Fil01]
Sea $\sigma = \langle \mathcal{V}, Func, Pred \rangle$ *una signatura, y* $t_1, \ldots, t_n$ *términos. Si* $p \in Pred$ *con* $aridad(p) = n$, *entonces* $p(t_1, \ldots, t_n)$ *es un átomo. Un átomo fijo es un átomo* $p(t_1, \ldots, t_n)$, *donde todos sus términos* $t_1, \ldots, t_n$ *son fijos.*

Definición 2.6 (Literal) [Fil01]
Un literal "L" es un átomo "A" o un átomo negado "$\sim A$*", donde "*$\sim$*" representa la negación fuerte*[2]*. Un literal L se dirá negativo si es un átomo negado, y positivo en caso contrario. Un literal fijo es un átomo fijo o un átomo fijo negado.*

Los literales podrán ser átomos negados utilizando el símbolo "$\sim$" de la *negación fuerte*. No debe confundirse este tipo de negación, con la negación por falla (o negación default) que es utilizada en Programación en Lógica, y que normalmente se denota con el símbolo "not". La definición y propiedades de la negación fuerte pueden verse en [AP93].

Definición 2.7 (Complemento de un literal) [Fil01]
Sea L un literal y A un átomo, el complemento de L con respecto a la negación fuerte, denotado $\overline{L}$, *se define de la siguiente manera:*
si $L = A$, *entonces* $\overline{L} = \sim A$,
si $L = \sim A$, *entonces* $\overline{L} = A$.

[1] En inglés *ground term*.
[2] En inglés *strong negation*.

2.2 Sintaxis de los Programas Lógicos Rebatibles

Un programa lógico rebatible estará compuesto por *hechos*, *reglas estrictas*, y *reglas rebatibles*, cuya sintaxis se define a continuación.

Definición 2.8 (Hecho)
Un hecho es un literal fijo L. Esto es, un átomo fijo, o un átomo fijo negado.

Por ejemplo, un programa lógico rebatible podrá contener los hechos: "$caballo(pampero)$" y "$\sim pez(petete)$", para representar que "Pampero" es un caballo, y que "Petete" no es un pez.

Definición 2.9 (Regla Estricta)
Una regla estricta *es un par ordenado, denotado "Cabeza $\leftarrow$ Cuerpo", donde el primer elemento Cabeza, es un literal fijo, y el segundo elemento, Cuerpo, es un conjunto finito no vacío de literales fijos. Una regla estricta con cabeza L_0 y cuerpo $\{L_1, \ldots, L_n\}$ $(n > 0)$ se escribirá también como: $L_0 \leftarrow L_1, \ldots, L_n$.*

Definición 2.10 (Regla Rebatible)
Una regla rebatible *es un par ordenado, denotado "Cabeza $-\prec$ Cuerpo", donde el primer elemento Cabeza, es un literal fijo, y el segundo elemento, Cuerpo, es un conjunto finito no vacío de literales fijos. Una regla rebatible con cabeza L_0 y cuerpo $\{L_1, \ldots, L_n\}$ $(n > 0)$ se escribirá también como: $L_0 -\prec L_1, \ldots, L_n$.*

Queda claro de las definiciones anteriores, que DeLP utilizará únicamente literales fijos. En lo que resta de la tesis, a menos que se indique lo contrario, cuando se hable de un "literal", siempre se estará asumiendo que es un literal fijo. A continuación se muestran ejemplos de reglas estrictas y rebatibles, obsérvese que la negación fuerte puede utilizarse tanto en el cuerpo como en la cabeza de una regla.

$$\sim inocente \leftarrow culpable$$
$$ave(lucas) \leftarrow pato(lucas)$$
$$\sim gallina(pampero) \leftarrow \sim ave(pampero)$$
$$muerto \leftarrow \sim vive$$
$$\sim vuela(tina) \prec gallina(tina)$$
$$vuela(tina) \prec gallina(tina), asustada(tina)$$
$$buen\_tiempo \prec alta\_presión$$
$$llueve \prec nublado$$

Sintácticamente, el símbolo "$\prec$" es lo único que distingue a una regla rebatible de una estricta. Pragmáticamente, las *reglas rebatibles* se utilizarán para representar conocimiento rebatible, esto es, información tentativa que puede utilizarse en la medida que no exista información que la contradiga. Las *reglas estrictas* representarán conocimiento seguro y libre de excepciones, esto es, siempre que se crea en los literales que forman el cuerpo, se podrá creer con la misma seguridad en el literal que corresponde a la cabeza.

De esta manera, conocimiento como *"los patos son aves"* se representará utilizando una regla estricta:

$$ave \leftarrow pato$$

mientras que la información *"las aves usualmente vuelan"* se representará con la regla rebatible:

$$vuela \prec ave$$

La sintaxis de las *reglas estrictas* corresponde a la sintaxis de las *reglas básicas* (basic rules) definidas por Lifschitz en [Lif96]. En DeLP se las llama "estrictas" para acentuar la diferencia con las "rebatibles". Las reglas estrictas representan una conexión fuerte entre el cuerpo y la cabeza de la regla, mientras que las reglas rebatibles permiten expresar una conexión débil. Una regla rebatible "$a \prec b$" expresa que "*creer en b provee razones para creer en a*" [SL92]. Los símbolos " $\prec$ " y " $\leftarrow$ " denotan meta-relaciones entre conjuntos de literales, por lo tanto no existe ninguna interacción con los símbolos del lenguaje. En particular, no pueden ser contrapuestas.

La diferenciación entre reglas estrictas y rebatibles también es utilizada en el trabajo de *Defeasible Logic* de Donald Nute [Nut94], en extensiones del trabajo de Nute hechas por Billington, Antoniou, Maher y Governatori [AMB00, AM98], y en muchos de los

formalismos de argumentación rebatible [PS97, Vre97]. En todos los casos las reglas rebatibles se utilizan para expresar información tentativa.

Definición 2.11 (Programa Lógico Rebatible)
Un programa lógico rebatible es un conjunto $\mathcal{P}$, posiblemente infinito, de hechos, reglas estrictas y reglas rebatibles. En un programa $\mathcal{P}$ identificaremos con Θ al conjunto de hechos, con Ω al conjunto de reglas estrictas, y con Δ al conjunto de reglas rebatibles. Por conveniencia, también denotaremos con Π al conjunto $\Theta \cup \Omega$. Por lo tanto, en algunas ocasiones denotaremos a un programa lógico rebatible $\mathcal{P}$ con el par $\mathcal{P}=(\Pi, \Delta)$, y en los casos que se quiera identificar los hechos del programa explícitamente, utilizaremos la terna $\mathcal{P}=(\Theta, \Omega, \Delta)$.

Observación 2.1 *Es importante destacar, que aunque las reglas de un programa utilizan únicamente literales fijos, siguiendo la misma política que Lifschitz en [Lif96], en algunos ejemplos se utilizarán variables solamente como una forma de denotar* esquemas de reglas. *Dado un esquema de regla R, se define $Ground(R)$ como el conjunto de todas las instancias de R con literales fijos. Dado un programa $\mathcal{P}$ se define [Lif96]*

$$Ground(\mathcal{P}) = \bigcup_{R \in \mathcal{P}} Ground(R)$$

Es claro que $Ground(R)$ y $Ground(\mathcal{P})$ siguen las definiciones de regla y programa lógico rebatible dadas antes. Para diferenciar las variables de los demás elementos del programa, estas serán denotadas con una letra mayúscula inicial.

Ejemplo 2.1 A continuación figura el programa $\mathcal{P}_{2.1}$, un programa lógico rebatible donde se representa información sobre aves.

$vuela(X) \prec ave(X)$
$\sim vuela(X) \prec gallina(X)$
$vuela(X) \prec gallina(X), asustada(X)$
$anida\_en\_arboles(X) \prec vuela(X)$
$gallina(tina)$
$pingüino(tweety)$

$ave(X) \leftarrow gallina(X)$
$ave(X) \leftarrow pingüino(X)$
$ave(X) \leftarrow paloma(X)$
$\sim vuela(X) \leftarrow pingüino(X)$
$paloma(loly)$
$asustada(tina)$

Ejemplo 2.2 El programa lógico rebatible $\mathcal{P}_{2.2}$ que sigue a continuación representa información sobre el mercado de valores. En el capítulo 8 se mostrará como utilizar DeLP para representar el conocimiento de agentes inteligentes en un sistema multi-agentes de este dominio de aplicación.

$$comprar\_acciones(T) -\prec buen\_precio(T)$$
$$\sim comprar\_acciones(T) -\prec buen\_precio(T), empresa\_riesgosa(T)$$
$$empresa\_riesgosa(T) -\prec en\_fusión(T)$$
$$empresa\_riesgosa(T) -\prec con\_deuda(T)$$
$$buen\_precio(acme)$$
$$en\_fusión(acme)$$

2.3 Derivaciones Rebatibles

A continuación se introducirá el concepto de *derivación rebatible*, con el cual se define qué literales pueden ser derivados utilizando las reglas de un programa lógico rebatible.

Definición 2.12 (Derivación Rebatible de un literal)
Sea $\mathcal{P} = (\Pi, \Delta)$ un programa y L un literal. Una derivación rebatible para L a partir de $\mathcal{P}$, consiste de una secuencia finita de literales fijos $L_1, L_2, \ldots, L_n = L$, provisto de que exista una secuencia de reglas estrictas o rebatibles $R_1, R_2, \ldots, R_n$ del programa $\mathcal{P}$, de tal forma que para cada literal L_i en la secuencia, tenemos que:

(a) L_i es un hecho, o

(b) existe en $\mathcal{P}$ una regla R_i con cabeza L_i y cuerpo $B_1, B_2, \ldots, B_k$ donde todo literal B_j del cuerpo ($1 \leq j \leq k$) es un elemento de la secuencia que precede a L_i.

La derivación se dice *rebatible*, porque aunque un literal L pueda ser derivado, puede existir en el programa información que contradiga a L, y entonces L no será aceptado como una creencia válida del programa.

Observación 2.2 *Como se expresó en la observación 2.1, en algunos ejemplos se utilizarán esquemas de reglas. Si un programa $\mathcal{P}$ tiene esquemas de reglas, para las derivaciones rebatibles se asume que se utiliza a $Ground(\mathcal{P})$.*

Por ejemplo, a partir del programa del Ejemplo 2.1, es posible obtener una derivación rebatible para el literal "$anida\_en\_arboles(loly)$", ya que existe la secuencia de literales: $paloma(loly)$, $ave(loly)$, $vuela(loly)$, $anida\_en\_arboles(loly)$. Que se obtiene utilizando las reglas:

$$ave(loly) \leftarrow paloma(loly)$$
$$vuela(loly) -\!\prec ave(loly)$$
$$anida\_en\_arboles(loly) -\!\prec vuela(loly)$$

En el caso del programa del Ejemplo 2.2, es posible obtener una derivación rebatible para el literal $\sim comprar\_acciones(acme)$ utilizando la siguiente secuencia de literales:

$$buen\_precio(acme)$$
$$en\_fusión(acme)$$
$$empresa\_riesgosa(acme)$$
$$\sim comprar\_acciones(acme)$$

Observación 2.3 *Si un programa lógico rebatible $\mathcal{P}$ no tiene hechos, entonces no puede obtenerse ninguna derivación rebatible.*

Dado un programa $\mathcal{P}$ y un literal h puede existir más de una derivación rebatible para h. Inclusive pueden existir derivaciones utilizando sólo reglas rebatibles, o sólo reglas estrictas. Se define a continuación un caso especial de derivación, que será de utilidad más adelante.

Definición 2.13 (Derivación Estricta) *Sea $\mathcal{P} = (\Pi, \Delta)$ un programa y L un literal para el cual existe una derivación rebatible $L_1, L_2, \ldots, L_n = L$. Diremos que L tiene una* derivación estricta *si todas las reglas de programa utilizadas para obtener la secuencia $L_1, L_2, \ldots, L_n = L$ son reglas estrictas.*

Como se verá más adelante, las derivaciones estrictas sustentan conclusiones que no podrán ser refutadas, ya que se basan en reglas que no son rebatibles.

Ejemplo 2.3 El literal "$\sim vuela(tweety)$" tiene una derivación estricta a partir del programa del ejemplo 2.1, ya que su derivación utiliza solamente el hecho "$pingüino(tweety)$" y la regla estricta "$\sim vuela(tweety) \leftarrow pingüino(tweety)$".

Como las reglas de programa permiten utilizar literales negados en la cabeza, entonces es posible derivar literales complementarios. Por ejemplo, a partir del programa del Ejemplo 2.1 existen derivaciones rebatibles para "$vuela(tina)$" y "$\sim vuela(tina)$", y en el Ejemplo 2.2 los literales "$comprar\_acciones(acme)$" y "$\sim comprar\_acciones(acme)$" pueden ambos derivarse rebatiblemente. Esto nos lleva a la siguiente definición.

Definición 2.14 (Conjunto de Reglas Contradictorio) *Un conjunto de reglas es* contradictorio *si y solo si, a partir de ese conjunto es posible obtener derivaciones rebatibles para un literal L y su complemento $\overline{L}$.*

De esta manera, se podrá representar información contradictoria en un programa lógico rebatible. Considere el programa $\mathcal{P}_{2.1} = (\Pi_{2.1}, \Delta_{2.1})$ del ejemplo 2.1. El conjunto $\Pi_{2.1}$ no es contradictorio, sin embargo $\Pi_{2.1} \cup \Delta_{2.1}$ es un conjunto contradictorio, ya que pueden derivarse rebatiblemente los literales $vuela(tina)$ y $\sim vuela(tina)$. En el caso del programa $\mathcal{P}_{2.2} = (\Pi_{2.2}, \Delta_{2.2})$ del ejemplo 2.2, el conjunto $\Pi_{2.2} \cup \Delta_{2.2}$, también es contradictorio.

El uso de la negación fuerte enriquece la expresividad del lenguaje, y permite la representación de información contradictoria. Como las reglas rebatibles permiten representar información tentativa, en la mayoría de los programas $\mathcal{P}=(\Pi, \Delta)$, el conjunto $\Pi \cup \Delta$ será contradictorio. Sin embargo, el conjunto Π, de hechos y reglas estrictas, es usado para representar información no-rebatible, y por lo tanto debe poseer cierta coherencia interna.

Observación 2.4 *En todo programa lógico rebatible $\mathcal{P}=(\Pi, \Delta)$, se asume que el conjunto Π es un conjunto no contradictorio.*

Nótese que en todos los ejemplos dados, el conjunto Π cumple esta condición.

Observación 2.5 *Sea $\mathcal{P}=(\Pi, \Delta)$ un programa lógico rebatible, dado que el conjunto Π se asume no contradictorio, no existen dos literales complementarios que tengan derivaciones estrictas a partir de $\mathcal{P}$.*

2.4 Argumentación Rebatible

Supongamos que se decide utilizar la noción de derivación rebatible como procedimiento de prueba, para decidir que literales pueden aceptarse como una creencia válida a partir

de un programa lógico rebatible $\mathcal{P}$. Entonces por ejemplo, a partir del programa $\mathcal{P}_{2.1}$ del ejemplo 2.1, tanto el literal "$vuela(tina)$", como "$\sim vuela(tina)$" serían aceptados, ya que ambos literales poseen una derivación rebatible a partir de $\mathcal{P}_{2.1}$. Este ejemplo muestra, que la noción de derivación rebatible no es conveniente como procedimiento de prueba de un literal. Por lo tanto, para decidir si un literal L está aceptado a partir de un programa, se realizará un *análisis global* más profundo que involucre toda la información relevante del programa. Considerando, por ejemplo, aquella información que contradice a L.

Existen en la literatura diferentes propuestas para decidir entre literales complementarios. Por ejemplo, en la "Programación Lógica Extendida" propuesta por Gelfond y Lifschitz [GL90], si a partir de un programa es posible derivar un par de literales complementarios, entonces a partir de ese programa puede derivarse el conjunto *Lit* de todos los literales posibles, sin considerar más análisis. Otros formalismos más recientes [Nut94, AMB00, AM98, PS97, KMD94, DK95], utilizan prioridades entre las reglas del programa para decidir entre dos reglas con literales complementarios en sus cabezas. En el capítulo 7 se incluye un análisis comparativo más completo entre DeLP y los formalismos antes citados.

En lo que resta de este capítulo se definirá un *procedimiento de prueba* que proveerá a DeLP de un mecanismo de análisis global para decidir que literales serán aceptados a partir de un programa. Este procedimiento de prueba está basado en el formalismo de *argumentación rebatible* propuesto originalmente por Simari y Loui en [Sim89, SL92], y luego extendido en [SCG94, GSC98]. Algunos conceptos serán nuevamente extendidos o adaptados al desarrollo de esta tesis. Dicho procedimiento de prueba permitirá definir *argumentos* para un literal y *contra-argumentos* para los argumentos obtenidos. Luego, un *análisis dialéctico* permitirá decidir cuando un literal está *aceptado*. A continuación presentamos la noción de estructura de argumento definida en [SL92], y adaptada a los programas lógicos rebatibles.

Definición 2.15 (Estructura de argumento)
Sea h un literal y $\mathcal{P}=(\Pi, \Delta)$ un programa lógico rebatible, una estructura de argumento *para h es un par $\langle \mathcal{A}, h\rangle$, donde $\mathcal{A}$ es conjunto de reglas rebatibles de Δ, tal que:*

1. *existe una derivación rebatible para h a partir de $\Pi \cup \mathcal{A}$.*
2. *$\Pi \cup \mathcal{A}$ es no contradictorio, y*

3. *$\mathcal{A}$ es minimal, es decir, no existe un subconjunto propio $\mathcal{A}'$ de $\mathcal{A}$ tal que $\mathcal{A}'$ satisface las condiciones (1) y (2).*

Ejemplo 2.4 A partir del programa del ejemplo 2.2 es posible construir la estructura de argumento $\langle \mathcal{A}, \sim comprar\_acciones(acme) \rangle$, donde

$$\mathcal{A} = \left\{ \begin{array}{l} \sim comprar\_acciones(acme) -\!\prec buen\_precio(acme), empresa\_riesgosa(acme) \\ empresa\_riesgosa(acme) -\!\prec en\_fusión(acme) \end{array} \right\}$$

A partir del programa del Ejemplo 2.1, es posible obtener estas dos estructuras de argumento para el literal "$vuela(tina)$":

$$\langle \{vuela(tina) -\!\prec ave(tina)\}, \text{vuela(tina)} \rangle$$
$$\langle \{vuela(tina) -\!\prec gallina(tina), asustada(tina)\}, \text{vuela(tina)} \rangle$$

También es posible obtener una estructura de argumento para "$\sim vuela(tina)$"

$$\langle \{\sim vuela(tina) -\!\prec gallina(tina)\}, \text{vuela(tina)} \rangle$$

Sin embargo, no es posible obtener un argumento para el literal "$vuela(tweety)$", porque la derivación rebatible de "$vuela(tweety)$" unida al conjunto Π es un conjunto contradictorio (observe que "$\sim vuela(tweety)$" tiene una derivación estricta como indica el ejemplo 2.3).

Definición 2.16 (Igualdad de estructuras de argumentos) *Dos estructuras de argumento $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ son iguales si y sólo si, $\mathcal{A}_1 = \mathcal{A}_2$, y $h_1 = h_2$.*

Definición 2.17 (Sub-estructura de argumento) *Una estructura de argumento $\langle \mathcal{B}, q \rangle$ es una sub-estructura de argumento de $\langle \mathcal{A}, h \rangle$ si y sólo si, $\mathcal{B} \subseteq \mathcal{A}$.*

Ejemplo 2.5 Considérese el siguiente programa:

$$\sim d \leftarrow b, e$$
$$e$$
$$a \leftarrow b$$
$$b -\!\prec c$$
$$c$$
$$\sim a -\!\prec d$$
$$d -\!\prec e$$

Aquí, la estructura de argumento $\langle\{b \prec c\}, a\rangle$ es una sub-estructura de argumento de $\langle\{b \prec c\}, \sim d\rangle$ y viceversa. Sin embargo, $\langle\{b \prec c\}, a\rangle$ no es igual a $\langle\{b \prec c\}, \sim d\rangle$, ya que difieren en el literal que sustentan.

Observación 2.6 *La unión de estructuras de argumento, no es una estructura de argumento. Dadas dos estructuras de argumento $\langle\mathcal{A}_1, h_1\rangle$ y $\langle\mathcal{A}_2, h_2\rangle$, el conjunto $\mathcal{A}_1 \cup \mathcal{A}_2$ puede ser contradictorio, puede ser no minimal, o las dos cosas simultaneamente como muestra el siguiente ejemplo.*

Ejemplo 2.6 Considérese el siguiente programa:

$$
\begin{aligned}
h_1 &\leftarrow b \\
h_2 &\leftarrow b \\
h &\leftarrow h_1, h_2 \\
p &\leftarrow h_1 \\
\sim p &\leftarrow h_2 \\
c & \\
d & \\
b &\prec c \\
b &\prec d
\end{aligned}
$$

y las siguientes estructuras de argumento:
$\langle\mathcal{A}_1, h_1\rangle = \langle\{b \prec c\}, h_1\rangle$
$\langle\mathcal{A}_2, h_2\rangle = \langle\{b \prec d\}, h_2\rangle$
A pesar que a partir de Π y el conjunto $\mathcal{A} = \{(b \prec c), (b \prec d)\}$[3] se obtiene una derivación rebatible para el literal h, el conjunto $\mathcal{A}$ no puede ser una estructura de argumento para h, ya que no es minimal y $\Pi \cup \mathcal{A}$ es contradictorio.

Proposición 2.1 *Sea $\mathcal{P}=(\Pi, \Delta)$ un programa lógico rebatible. Existe una derivación estricta para un literal h a partir de $\mathcal{P}$, si y sólo si, existe una única estructura de argumento $\langle\mathcal{A}, h\rangle$, donde $\mathcal{A}=\emptyset$.*
Demostración: si existe una derivación estricta para h a partir de Π, entonces se verifica la condición (1) de la definición de argumento con $\Pi \cup \emptyset$. Debido a que el conjunto Π es

[3]Por motivos de claridad en la notación, en los casos que sea neceario, se utilizarán paréntesis para separar una regla de otra dentro de un conjunto.

no contradictorio, entonces también se verifica trivialmente la condición (2). Finalmente, como no puede existir un subconjunto propio de $\mathcal{A}=\emptyset$, se cumple la condición (3) y por lo tanto $\langle\{\}, h\rangle$ es una estructura de argumento. Dicha estructura es única debido a que cualquier otro conjunto de reglas rebatibles que cumpla las condiciones (1) y (2) será un superconjunto de $\emptyset$. La demostración en el otro sentido es trivial. □

Proposición 2.2 *Sea $\mathcal{P}$ un programa lógico rebatible. Sea L un literal para el cual existe una derivación estricta a partir de $\mathcal{P}$. Entonces no puede existir un argumento para $\overline{L}$.*
Demostración: trivial a partir de la definición de argumento, ya que no se cumpliría la condición 2. □

2.5 Desacuerdo, ataque y contra-argumentación

Dos literales complementarios representan información estrictamente contradictoria. La definición siguiente generaliza este concepto.

Definición 2.18 (Literales en Desacuerdo)
Sea $\mathcal{P}=(\Pi, \Delta)$ un programa, y Π el conjunto de reglas estrictas y hechos del programa. Dos literales h y h_1 están en desacuerdo*, si y sólo si el conjunto $\Pi \cup \{h, h_1\}$ es contradictorio.*

El ejemplo más simple de literales en desacuerdo son dos literales complementarios como "p" y "$\sim p$", ya que $\{p, \sim p\} \cup \Pi$ es contradictorio, cualquiera sea el conjunto Π. Sin embargo, como muestra el siguiente ejemplo, dos literales que no sean complementarios, como "p" y "q", también pueden estar en desacuerdo, si unidos al conjunto Π permiten derivar dos literales complementarios.

Ejemplo 2.7 Dado el conjunto $\Pi = \{(b), (a \leftarrow b, p), (\sim a \leftarrow c), (c \leftarrow q)\}$, los literales "$p$" y "$q$" están en desacuerdo, ya que a partir de $\{p, q\} \cup \Pi$ es posible derivar tanto al literal "a", como al literal "$\sim a$".

Definiremos a continuación una noción de conflicto entre dos estructuras de argumento basada en la negación fuerte. Esto es, dos argumentos estarán en conflicto cuando sustenten conclusiones en desacuerdo.

Definición 2.19 (Contra-argumento o ataque)
Sean $\langle \mathcal{A}_1, h_1 \rangle$ *y* $\langle \mathcal{A}_2, h_2 \rangle$ *dos estructuras de argumento obtenidas a partir de un programa* $\mathcal{P}$. *Decimos que* $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta *a* $\langle \mathcal{A}_2, h_2 \rangle$ *en el literal* h, *si y sólo si, existe una sub-estructura de argumento* $\langle \mathcal{A}, h \rangle$ *de* $\langle \mathcal{A}_2, h_2 \rangle$ *tal que* h *y* h_1 *están en desacuerdo. El argumento* $\langle \mathcal{A}, h \rangle$ *se llama* sub-argumento de desacuerdo, *y el literal* h *será el* punto de contra-argumentación.

Si $\langle \mathcal{A}_1, h_1 \rangle$ *contra-argumenta* a $\langle \mathcal{A}_2, h_2 \rangle$, entonces también se dirá que $\langle \mathcal{A}_1, h_1 \rangle$ *ataca* a $\langle \mathcal{A}_2, h_2 \rangle$, o que $\langle \mathcal{A}_1, h_1 \rangle$ es un contra-argumento para $\langle \mathcal{A}_2, h_2 \rangle$.

Ejemplo 2.8 A partir del programa del ejemplo 2.2 es posible construir la estructura de argumento $\langle \mathcal{A}_2, comprar\_acciones(acme) \rangle$, con

$$\mathcal{A}_2 = \left\{ \; comprar\_acciones(acme) \prec buen\_precio(acme) \; \right\}$$

y la estructura $\langle \mathcal{A}_1, \sim comprar\_acciones(acme) \rangle$, con

$$\mathcal{A}_1 = \left\{ \begin{array}{l} \sim comprar\_acciones(acme) \prec buen\_precio(acme), empresa\_riesgosa(acme) \\ empresa\_riesgosa(acme) \prec en\_fusión(acme) \end{array} \right\}$$

La estructura de argumento $\langle \mathcal{A}_1, \sim comprar\_acciones(acme) \rangle$ es un contra-argumento para la estructura $\langle \mathcal{A}_2, comprar\_acciones(acme) \rangle$. En este caso, el subargumento de desacuerdo es el mismo $\langle \mathcal{A}_2, comprar\_acciones(acme) \rangle$, y el punto de contra-argumentación es "$comprar\_acciones(acme)$". Nótese además que recíprocamente, $\langle \mathcal{A}_2, comprar\_acciones(acme) \rangle$ es un contra-argumento para $\langle \mathcal{A}_1, \sim comprar\_acciones(acme) \rangle$.

Observación 2.7 *En la definición 2.19, puede ocurrir que el subargumento* $\langle \mathcal{A}, h \rangle$ *sea el propio* $\langle \mathcal{A}_2, h_2 \rangle$, *y en este caso se dirá que* $\langle \mathcal{A}_1, h_1 \rangle$ *contra-argumenta o ataca* directamente *a* $\langle \mathcal{A}_2, h_2 \rangle$. *Pero también puede ocurrir que* $\langle \mathcal{A}, h \rangle$ *sea un subargumento propio de* $\langle \mathcal{A}_2, h_2 \rangle$, *esto es, el ataque se produzca en un punto* h *interior de* $\langle \mathcal{A}_2, h_2 \rangle$, *entonces se dirá que* $\langle \mathcal{A}_1, h_1 \rangle$ *contra-argumenta o ataca* indirectamente *a* $\langle \mathcal{A}_2, h_2 \rangle$. *La Figura 2.1 muestra esto dos casos.*

Observación 2.8 (Notación gráfica) *En la Figura 2.1 y en otras a continuación, se utilizará una notación gráfica definida en [SCG94] para estructuras de argumento. En dicha notación, una estructura de argumento es representado con un triángulo, en cuyo vértice superior se indica el literal que sustenta. Una etiqueta en la base del triángulo dá un nombre de referencia a la estructura de argumento. Un triángulo dentro de otro mayor representa una sub-estructura de argumento, y una línea recta entre dos vértices de triángulos representa ataque entre estructuras de argumento (conectando los literales en desacuerdo). Por ejemplo, en la Figura 2.1 la estructura de argumento* $\langle \mathcal{A}_2, h_2 \rangle$ *contiene a una sub-estructura de argumento* $\langle \mathcal{A}, h \rangle$*, y la estructura de argumento* $\langle \mathcal{A}_1, h_1 \rangle$ *es un contra-argumento que ataca indirectamente en el punto* h*, con sub-argumento de desacuerdo* $\langle \mathcal{A}, h \rangle$*.*

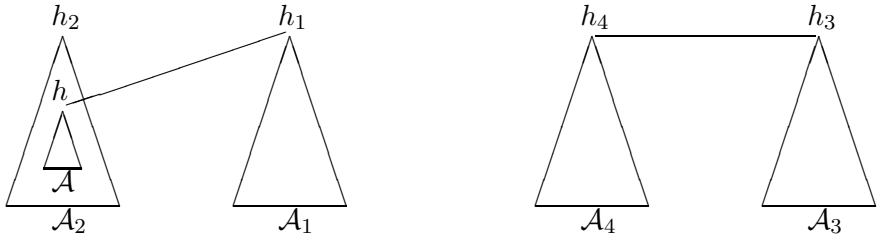

Figura 2.1: Ataque indirecto (izquierda) y ataque directo (derecha)

Ejemplo 2.9 Dado un programa $\mathcal{P}$=(Π, Δ) donde
Π= $\{(c), (d), (\sim p \leftarrow h)\}$
Δ= $\{(a \prec p), (p \prec c), (b \prec \sim p), (h \prec d)\}$
pueden obtenerse las siguientes estructuras de argumento:
$\langle \mathcal{A}, a \rangle = \langle \{(a \prec p), (p \prec c)\}, a \rangle$
$\langle \mathcal{B}, b \rangle = \langle \{(b \prec \sim p), (h \prec d)\}, b \rangle$
$\langle \mathcal{A}_1, p \rangle = \langle \{p \prec c\}, p \rangle$
$\langle \mathcal{B}_1, h \rangle = \langle \{h \prec d\}, h \rangle$
Como puede verse, los literales p y h están en desacuerdo, y además $\langle \mathcal{A}_1, p \rangle$ es un contra-argumento para $\langle \mathcal{B}, b \rangle$ en el literal h, siendo $\langle \mathcal{B}_1, h \rangle$ el contra-argumento de desacuerdo.

Observación 2.9 *Si un programa lógico rebatible* $\mathcal{P}$*, no posee ningún hecho negado y ninguna regla con un átomo negado en la cabeza, entonces* $\mathcal{P}$ *es un conjunto de reglas no contradictorio y ningún argumento tendrá contra-argumentos. Lo mismo ocurre si todos los hechos, y todas las reglas tienen átomos negados en la cabeza.*

Observación 2.10 *De la observación anterior se desprende lo siguente. Sea $Co(\mathcal{P})$ el conjunto de literales que son hechos o cabezas de reglas de $\mathcal{P}$. Si $Co(\mathcal{P})$ no posee literales complementarios, entonces ningún argumento tiene un contra-argumento.*

Las dos observaciones anteriores permitirían realizar un análisis estático de un programa a fin de evitar la búsqueda de contra-argumentos.

Observación 2.11 *Si $\langle \mathcal{A}_1, h_1 \rangle$ es un contra-argumento para $\langle \mathcal{A}_2, h_2 \rangle$, entonces el conjunto $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \Pi$ es contradictorio, ya que existen dos literales en desacuerdo. Sin embargo, la recíproca no es cierta, como muestra el siguiente ejemplo. Considere un programa formado por las siguiente reglas:*

$h_1 -\prec c1, c2$	$p \leftarrow c1, d1$
$h_2 -\prec d1, d2$	$\sim p \leftarrow c2, d2$
$c1 -\prec t1$	$t1$
$c2 -\prec t2$	$t2$
$d1 -\prec t3$	$t3$
$d2 -\prec t4$	$t4$

A partir del programa anterior pueden obtenerse las siguientes estructuras de argumento:
$\langle \mathcal{A}_1, h_1 \rangle = \langle \{(h_1 -\prec c1, c2), (c1 -\prec t1), (c2 -\prec t2)\}, h_1 \rangle$
$\langle \mathcal{A}_2, h_2 \rangle = \langle \{(h_2 -\prec d1, d2), (d1 -\prec t3), (d2 -\prec t4)\}, h_1 \rangle$
En este caso, aunque el conjunto $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \Pi$ es contradictorio, $\langle \mathcal{A}_1, h_1 \rangle$ no es un contra-argumento para $\langle \mathcal{A}_2, h_2 \rangle$, y $\langle \mathcal{A}_2, h_2 \rangle$ no es un contra-argumento para $\langle \mathcal{A}_1, h_1 \rangle$.

La siguiente proposición muestra que aquellos literales que poseen una derivación estricta no pueden ser refutados, ya que no puede existir un contra-argumento que los ataque.

Proposición 2.3 *Sea $\langle \mathcal{A}, h \rangle$ una estructura de argumento donde $\mathcal{A} = \emptyset$, esto es, h tiene una derivación estricta, entonces no existe un contra-argumento para $\langle \mathcal{A}, h \rangle$.*
Demostración: Si existiera un contra-argumento $\langle \mathcal{A}_2, h_2 \rangle$ para $\langle \mathcal{A}, h \rangle$, entonces $\mathcal{A}_2 \cup \mathcal{A} \cup \Pi$ sería un conjunto contradictorio (observación 2.11). Como $\mathcal{A} = \emptyset$, entonces $\mathcal{A}_2 \cup \Pi$ sería un conjunto contradictorio, lo cual violaría la condición (2) de la definición de argumento, y por lo tanto $\langle \mathcal{A}_2, h_2 \rangle$ no sería un argumento. □

También es importante destacar que una estructura de argumento $\langle \mathcal{A}, h \rangle$, con $\mathcal{A} = \emptyset$, es decir, formada con una derivación estricta, no puede ser contra-argumento de ninguna otra estructura $\langle \mathcal{B}, q \rangle$, ya que $\langle \mathcal{B}, q \rangle$ no puede existir por definición. Este hecho es formalizado en la siguiente proposición.

Proposición 2.4 *Sea $\langle \mathcal{A}, h \rangle$ una estructura de argumento donde $\mathcal{A} = \emptyset$, esto es, h tiene una derivación estricta, entonces no existe argumento $\langle \mathcal{B}, q \rangle$ tal que $\langle \mathcal{A}, h \rangle$ es un contra-argumento para $\langle \mathcal{B}, q \rangle$.*
Demostración: Si existiera $\langle \mathcal{B}, q \rangle$, tal que $\langle \mathcal{A}, h \rangle$ fuera un contra-argumento para $\langle \mathcal{B}, q \rangle$, entonces $\mathcal{B} \cup \mathcal{A} \cup \Pi$ sería un conjunto contradictorio (observación 2.11). Como $\mathcal{A} = \emptyset$, entonces $\mathcal{B} \cup \Pi$ sería un conjunto contradictorio, lo cual violaría la condición (2) de la definición de argumento, y por lo tanto $\mathcal{B}$ no sería un argumento. □

La siguiente proposición muestra que existe una simetría en la noción de contra-argumentación.

Proposición 2.5 *Si una estructura de argumento $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta a otra $\langle \mathcal{A}_2, h_2 \rangle$ en el punto h, con el subargumento de desacuerdo $\langle \mathcal{A}, h \rangle$, entonces también se cumple que $\langle \mathcal{A}, h \rangle$ contra-argumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el punto h_1.*
Demostración: como $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta a $\langle \mathcal{A}_2, h_2 \rangle$ en el punto h, entonces h y h_1 están en desacuerdo, por lo tanto, $\langle \mathcal{A}, h \rangle$ contra-argumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el punto h_1, ya que existe un subargumento de $\langle \mathcal{A}_1, h_1 \rangle$, que es el mismo $\langle \mathcal{A}_1, h_1 \rangle$, tal que h y h_1 están en desacuerdo. □

Observación 2.12 *Si $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta a $\langle \mathcal{A}_2, h_2 \rangle$ en el punto h_2, esto es en el literal que sustenta $\mathcal{A}_2$, entonces $\langle \mathcal{A}_2, h_2 \rangle$ contra-argumenta a $\langle \mathcal{A}_1, h_1 \rangle$.*

La noción de contra-argumentación establece cuando un argumento es atacado por otro capturando la noción de conflicto entre argumentos a través de la negación fuerte. Sin embargo, como muestran la proposición 2.5 y la observación 2.12, siempre que un argumento $\mathcal{A}$ contra-argumenta a otro $\mathcal{B}$, existe un subargumento de $\mathcal{B}$ que es un contra-argumento para $\mathcal{A}$. Esta simetría no permite indicar cuando un argumento se impone sobre el otro. Para esto, es indispensable una forma de comparar argumentos, de manera de poder indicar cuando un argumento es mejor que otro.

2.6 Comparación de argumentos

Un problema central en los formalismos de argumentación rebatible, es la definición de un criterio que permita comparar argumentos. En el caso de los sistemas más abstractos de argumentación rebatible, usualmente se asume un orden pre-establecido entre todos los argumentos posibles. Esta es la solución propuesta por Phan Dung [Dun93b] donde se abstrae completamente de la noción de argumento y ataque, y un *argumentation framework* (AF) es un par $(Args, attack)$, donde $Args$ es un conjunto de argumentos y $attack$ es una relación binaria sobre $Args$, que establece que argumento ataca a cual. En el sistema argumentativo abstracto de Bondarenko, Dung, Kowalski y Toni [KT96, BDKT97], y el trabajo de Gerard Vreeswijk [Vre93], también se asume un orden dado entre los argumentos.

La propuesta de otros formalismos consiste en asumir un orden definido explícitamente entre las reglas del programa (prioridades). Para resolver un conflicto, las reglas de mayor prioridad son preferidas. Esta es la estrategia utilizada en d-Prolog [Nut88], Defeasible Logic [Nut92], en las extensiones de Defeasiable Logic realizadas por Billington, Antoniou, Maher y Gobernatori [AMB00, AM98], y en la programación en lógica sin negación por falla [KMD94, DK95]. La propuesta de Prakken y Sartor en [PS97], permite incluso razonar rebatiblemente sobre las prioridades entre reglas. Más detalles sobre estos formalismos serán presentados en el capítulo 7.

Otra alternativa, en lugar de asumir un orden dado junto con el programa, es utilizar el criterio de especificidad definido originalmente por David Poole [Poo85]. La especificidad permite comparar reglas o argumentos sin la necesidad de especificar explícitamente las prioridades.

En esta sección se presentan dos propuestas para comparar argumentos, una basada en especificidad, y la otra basada en prioridades entre reglas.

2.6.1 Especificidad Generalizada

A continuación se presenta un criterio de comparación entre dos estructuras de argumento, llamado *especificidad generalizada*, el cuál está basado en la especificidad definida en [Poo85, Lou87, SL92]. Informalmente, una estructura de argumento $\langle \mathcal{A}_1, h_1 \rangle$ será preferida a otra estructura $\langle \mathcal{A}_2, h_2 \rangle$, si se cumple alguna de las dos condiciones siguientes:

1. $\langle \mathcal{A}_1, h_1 \rangle$ usa mayor información que $\langle \mathcal{A}_2, h_2 \rangle$, o
2. $\langle \mathcal{A}_1, h_1 \rangle$ es más directo (usa menos reglas) que $\langle \mathcal{A}_2, h_2 \rangle$

A continuación figura un ejemplo que muestra los dos casos, y luego se incluye la definición formal.

Ejemplo 2.10 Considere el programa $\mathcal{P}_{2.1}$ del Ejemplo 2.1, la estructura de argumento

$$\langle \{vuela(tina) -\prec gallina(tina), asustada(tina)\}, vuela(tina) \rangle$$

será preferida a

$$\langle \{\sim vuela(tina) -\prec gallina(tina)\}, \sim vuela(tina) \rangle$$

ya que la primera usa mas información al basarse en "$gallina(tina)$" y "$asustada(tina)$", mientras que la segunda sólo se basa en "$gallina(tina)$". Por su parte, la estructura de argumento

$$\langle \{\sim vuela(tina) -\prec gallina(tina)\}, \sim vuela(tina) \rangle$$

será preferida a

$$\langle \{vuela(tina) -\prec ave(tina)\}, vuela(tina) \rangle$$

por ser más directa, ya que esta última usa una regla estricta adicional para poder inferir "$ave(tina)$" a partir de "$gallina(tina)$". Observe que estas dos últimas estructuras de argumento se basan en la misma información: "$gallina(tina)$".

Definición 2.20 (Especificidad generalizada) *Sea $\mathcal{P}$=(Θ, Ω, Δ) un programa lógico rebatible, donde Ω es el conjunto de reglas estrictas. Sea $\mathcal{F}$ el conjunto de literales que tienen una derivación rebatible a partir de $\mathcal{P}$. Una estructura de argumento $\langle \mathcal{A}_1, h_1 \rangle$ es* estrictamente más específica *que otra estructura de argumento $\langle \mathcal{A}_2, h_2 \rangle$ (denotado $\langle \mathcal{A}_1, h_1 \rangle \succ \langle \mathcal{A}_2, h_2 \rangle$) si se verifican las dos condiciones siguientes:*

1. *para todo conjunto* $H \subseteq \mathcal{F}$,
   si existe una derivación rebatible para h_1 *a partir de* $\Omega \cup H \cup \mathcal{A}_1$ (H activa a $\mathcal{A}_1$),
   y no existe una derivación estricta para h_1 *a partir de* $\Omega \cup H$,
   entonces existe una derivación rebatible para h_2 *a partir de* $\Omega \cup H \cup \mathcal{A}_2$, ($H$ activa a $\mathcal{A}_2$)
2. *existe al menos un conjunto* $H' \subseteq \mathcal{F}$ *tal que:*
   existe una derivación rebatible para h_2 *a partir de* $\Omega \cup H' \cup \mathcal{A}_2$ (H' activa a $\mathcal{A}_2$),
   no existe una derivación estricta de h_2 *a partir de* $\Omega \cup H'$, *y*
   no existe una derivación rebatible de h_1 *a partir de* $\Omega \cup H' \cup \mathcal{A}_1$ (H' no activa a $\mathcal{A}_1$).

Los conjuntos $\mathcal{A}_1$ y $\mathcal{A}_2$ contienen reglas rebatibles, y Ω reglas estrictas sin hechos, por lo tanto es claro (Observación 2.3) que no puede existir ninguna derivación rebatible a partir de $\Omega \cup \mathcal{A}_1$, o a partir de $\Omega \cup \mathcal{A}_2$. Los conjuntos H y H' contienen literales, los cuales son tomados como hechos para obtener las derivaciones rebatibles. La condición 1 establece que para todo conjunto de literales (hechos) $H \subseteq \mathcal{F}$, si H permite derivar h_1 utilizando reglas de $\Omega \cup \mathcal{A}_1$, donde al menos una regla de $\mathcal{A}_1$ es utilizada (ya que no existe una derivación estricta de h_1 a partir de $\Omega \cup H$), entonces ese mismo conjunto H permite derivar rebatiblemente a h_2 utilizando los reglas de $\Omega \cup \mathcal{A}_2$. En la condición 2, se busca que exista al menos un conjunto $H' \subseteq \mathcal{F}$, que permita derivar rebatiblemente a h_2 pero que no permita derivar a h_1.

Ejemplo 2.11 Siguiendo con el ejemplo 2.10, todo conjunto H que active a la estructura de argumento

$$\langle \mathcal{A}_1, h_1 \rangle = \langle \{vuela(tina) \prec gallina(tina), asustada(tina)\}, vuela(tina) \rangle$$

debe contener los literales "$gallina(tina)$" y "$asustada(tina)$", y por lo tanto H activará a

$$\langle \mathcal{A}_2, h_2 \rangle = \langle \{\sim vuela(tina) \prec gallina(tina)\}, \sim vuela(tina) \rangle$$

Pero sin embargo, el conjunto $H' = \{\ gallina(tina)\ \}$ activa a $\langle \mathcal{A}_2, h_2 \rangle$, pero no activa a $\langle \mathcal{A}_1, h_1 \rangle$. Por lo tanto $\langle \mathcal{A}_1, h_1 \rangle$ es estrictamente más específico que$\langle \mathcal{A}_2, h_2 \rangle$.

En el caso de la estructura de argumento

$$\langle \mathcal{A}_3, h_3 \rangle = \langle \{\sim vuela(tina) \prec gallina(tina)\}, \sim vuela(tina) \rangle$$

todo conjunto H que la activa debe contener al literal "$gallina(tina)$" con lo cual también activa a

$$\langle \mathcal{A}_4, h_4 \rangle = \langle \{vuela(tina) \prec ave(tina)\}, vuela(tina) \rangle$$

Pero sin embargo, el conjunto $H' = \{\ ave(tina)\ \}$ activa a $\langle \mathcal{A}_4, h_4 \rangle$, pero no activa a $\langle \mathcal{A}_3, h_3 \rangle$. Por lo tanto $\langle \mathcal{A}_3, h_3 \rangle$ es estrictamente más específico que$\langle \mathcal{A}_4, h_4 \rangle$.

2.6.2 Otras formas de comparación de argumentos

Como se indicó antes, en algunos sistemas se asume un orden explícito (o prioridades) entre las reglas del programa. En dichos sistemas, para decidir entre dos literales contradictorios, las dos reglas que tienen a estos literales en sus cabezas son comparadas. El literal que prevalece es el que está en la regla de mayor prioridad.

La formulación de DeLP impide que se puedan comparar reglas individuales, ya que se deben comparar argumentos entre sí. Para poder utilizar un orden entre reglas de un programa, a continuación se define un criterio de comparación entre argumentos que toma como base las prioridades entre las reglas.

Asumiremos que el orden se establece entre reglas rebatibles, ya que no debería haber reglas estrictas mejores que otras. En algunos formalismos, el orden se establece únicamente entre dos reglas que tienen literales complementarios en sus cabezas. Aquí no pondremos esa restricción.

Definición 2.21 *Sea $\mathcal{P}$ un programa lógico rebatible y ">" un orden parcial definido explícitamente sobre las reglas rebatibles de $\mathcal{P}$. Sean $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ dos estructuras de argumento obtenidas a partir de $\mathcal{P}$. $\langle \mathcal{A}_1, h_1 \rangle$ será preferido sobre $\langle \mathcal{A}_2, h_2 \rangle$ cuando:*

1. *exista al menos una regla $r_a \in \mathcal{A}_1$, y una regla $r_b \in \mathcal{A}_2$, tal que $r_a > r_b$, y*
2. *no exista ninguna regla $r'_b \in \mathcal{A}_2$, y $r'_a \in \mathcal{A}_1$, tal que $r'_b > r'_a$.*

Ejemplo 2.12 Considere el siguiente programa,

$$comprar\_acciones(T) -\!\prec buen\_precio(T)$$
$$\sim comprar\_acciones(T) -\!\prec empresa\_riesgosa(T)$$
$$empresa\_riesgosa(T) -\!\prec con\_deuda(T)$$
$$con\_deuda(acme)$$
$$buen\_precio(acme)$$

donde el orden entre reglas sólo incluye:
$(\sim comprar\_acciones(T) -\!\prec empresa\_riesgosa(T)) > (comprar\_acciones(T) -\!\prec buen\_precio(T))$

A partir de este programa puede construirse la estructura de argumento $\langle \mathcal{A}, comprar\_acciones(acme)\rangle$,

$$\mathcal{A}= \{\ comprar\_acciones(acme) -\!\prec buen\_precio(acme)\ \}$$

y el contra-argumento $\langle \mathcal{B}$,∼comprar_acciones(acme)$\rangle$,

$$\mathcal{B} = \left\{ \begin{array}{l} \sim comprar\_acciones(acme) -\!\prec empresa\_riesgosa(acme) \\ empresa\_riesgosa(acme) -\!\prec con\_deuda(acme) \end{array} \right\}$$

Donde $\langle \mathcal{B}$,∼comprar_acciones(acme)$\rangle$, es mejor que $\langle \mathcal{A}$,comprar_acciones(acme)$\rangle$, ya que $\mathcal{B}$ tiene un regla que es mejor que una de $\mathcal{A}$, y no ocurre lo mismo a la inversa.

Observación 2.13 *Como siempre se comparará un argumento con su contra-argumento, entonces, si un literal h tiene una derivación estricta, (i. e., $\langle \mathcal{A}, h\rangle$ tiene $\mathcal{A}=\emptyset$), entonces $\langle \mathcal{A}, h\rangle$ no será comparado con otro argumento ya que por la proposición 2.3, no puede existir un contra-argumento para $\langle \mathcal{A}, h\rangle$.*

2.7 Derrota y líneas de argumentación

Como se vió en las sección anterior, es posible definir distintos criterios de comparación entre argumentos. De esta forma, se podrán comparar argumentos y contra-argumentos. Si $\langle \mathcal{A}_1, h_1\rangle$ es un contra-argumento para $\langle \mathcal{A}_2, h_2\rangle$, es necesario analizar si el ataque de $\langle \mathcal{A}_1, h_1\rangle$ sobre $\langle \mathcal{A}_2, h_2\rangle$ es lo suficientemente fuerte como para rebatirlo. Como caso particular, $\langle \mathcal{A}_1, h_1\rangle$ puede atacar directamente a $\langle \mathcal{A}_2, h_2\rangle$ en el punto h_2, con lo cual deberían

compararse las estructuras $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$. En general, si $\langle \mathcal{A}_1, h_1 \rangle$ ataca indirectamente en el punto h, siendo $\langle \mathcal{A}, h \rangle$ el sub-argumento de desacuerdo en $\langle \mathcal{A}_2, h_2 \rangle$ (Figura 2.1), se deben comparar el argumento atacante $\langle \mathcal{A}_1, h_1 \rangle$ y el subargumento de desacuerdo $\langle \mathcal{A}, h \rangle$. En ambas situaciones puede darse uno de esto cuatro casos:

(a) $\langle \mathcal{A}_1, h_1 \rangle$ es mejor que $\langle \mathcal{A}, h \rangle$,

(b) $\langle \mathcal{A}, h \rangle$ es mejor que $\langle \mathcal{A}_1, h_1 \rangle$,

(c) $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}, h \rangle$ son incomparables,

(d) $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}, h \rangle$ son equivalentes.

Como la comparación entre argumentos puede definirse de varias maneras, en lo que sigue asumiremos que existe un orden parcial "$\succ$" que permite distinguir cuando una estructura de argumento es mejor que otra. Por ejemplo, si $\langle \mathcal{A}_1, h_1 \rangle$ es mejor que $\langle \mathcal{A}, h \rangle$, denotaremos "$\langle \mathcal{A}_1, h_1 \rangle \succ \langle \mathcal{A}, h \rangle$".

Definición 2.22 (Derrotador Propio) [SCG94]
Sean $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ dos estructuras de argumento. La estructura $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador propio *para $\langle \mathcal{A}_2, h_2 \rangle$ en el literal h si, existe un sub-argumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_2, h_2 \rangle$, tal que $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta a $\langle \mathcal{A}_2, h_2 \rangle$ en el literal h, y $\langle \mathcal{A}_1, h_1 \rangle$ es mejor que $\langle \mathcal{A}, h \rangle$ con respecto al criterio de comparación que se utilice ($\langle \mathcal{A}_1, h_1 \rangle \succ \langle \mathcal{A}, h \rangle$).*

Cabe destacar que si $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador propio para $\langle \mathcal{A}_2, h_2 \rangle$ en un punto h interior al argumento $\langle \mathcal{A}_2, h_2 \rangle$, entonces $\langle \mathcal{A}_1, h_1 \rangle$ debe ser mejor que $\langle \mathcal{A}, h \rangle$, pero no necesita ser mejor que $\langle \mathcal{A}_2, h_2 \rangle$. La definición anterior captura los casos (a) y (b) mencionados mas arriba, la siguiente definición capturará los otros dos casos.

Definición 2.23 (Derrotador de bloqueo) [SCG94]
Sean $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ dos estructuras de argumento. La estructura $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador de bloqueo *para $\langle \mathcal{A}_2, h_2 \rangle$ en el literal h si, existe un sub-argumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_2, h_2 \rangle$, tal que $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta a $\langle \mathcal{A}_2, h_2 \rangle$ en el literal h, y ni $\langle \mathcal{A}_1, h_1 \rangle$ es mejor que $\langle \mathcal{A}, h \rangle$, ni $\langle \mathcal{A}, h \rangle$ es mejor que $\langle \mathcal{A}_1, h_1 \rangle$, con respecto al criterio de comparación que se utilice ($\langle \mathcal{A}_1, h_1 \rangle \not\succ \langle \mathcal{A}, h \rangle$, y $\langle \mathcal{A}, h \rangle \not\succ \langle \mathcal{A}_1, h_1 \rangle$).*

Finalmente, si $\langle \mathcal{A}_1, h_1 \rangle$ es un contra-argumento para $\langle \mathcal{A}_2, h_2 \rangle$ en un punto h, $\langle \mathcal{A}_1, h_1 \rangle$ será considerado un *derrotador* de $\langle \mathcal{A}_2, h_2 \rangle$, si $\langle \mathcal{A}_1, h_1 \rangle$ no es más débil que $\langle \mathcal{A}, h \rangle$. Formalmente,

Definición 2.24 (Derrotador) [SCG94]
Sean $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ dos estructuras de argumento. La estructura $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador *para $\langle \mathcal{A}_2, h_2 \rangle$ en el literal h si,*
(a) $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador propio para $\langle \mathcal{A}_2, h_2 \rangle$, o
(b) $\langle \mathcal{A}_1, h_1 \rangle$ es un derrotador de bloqueo para $\langle \mathcal{A}_2, h_2 \rangle$.

Observación 2.14 *Si $\langle \mathcal{A}_1, h_1 \rangle$ derrota a $\langle \mathcal{A}_2, h_2 \rangle$, entonces $\langle \mathcal{A}_1, h_1 \rangle$ derrota a toda estructura $\langle \mathcal{B}, q \rangle$ que contenga a $\langle \mathcal{A}_2, h_2 \rangle$ como subargumento.*

Dado el conjunto de estructuras de argumento que pueden obtenerse de un programa $\mathcal{P}$, la relación de derrota entre argumentos sólo puede establecer un orden entre dos argumentos en conflicto. Sin embargo, el estado de un argumento $\langle \mathcal{A}_1, h_1 \rangle$ con respecto al programa $\mathcal{P}$, dependerá de la interacción de $\langle \mathcal{A}_1, h_1 \rangle$ con el resto de los argumentos que pueden obtenerse.

Por ejemplo, si a partir de un programa $\mathcal{P}$ se obtiene una estructura de argumento $\langle \mathcal{A}_0, h_0 \rangle$ que no tiene derrotadores, entonces un agente que disponga de $\mathcal{P}$ para razonar, podrá "creer" en el literal h_0. Sin embargo, puede ocurrir que $\langle \mathcal{A}_0, h_0 \rangle$ tenga un derrotador $\langle \mathcal{A}_1, h_1 \rangle$, con lo cual existirán razones para invalidar la creencia en h_0. Pero también puede ocurrir que la estructura $\langle \mathcal{A}_1, h_1 \rangle$ tenga a su vez un derrotador $\langle \mathcal{A}_2, h_2 \rangle$, reinstaurando la creencia en h_0.

La secuencia de interacciones puede ir más lejos, y a su vez puede existir un derrotador $\langle \mathcal{A}_3, h_3 \rangle$ para $\langle \mathcal{A}_2, h_2 \rangle$, que vuelve a invalidar la creencia en h_0, y asi siguiendo. Lo cual da origen una secuencia de argumentos $[\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle \langle \mathcal{A}_3, h_3 \rangle, \ldots]$ que se llamará *línea de argumentación*, donde cada elemento es un derrotador de su predecesor.

Definición 2.25 (Línea de Argumentacion) [SCG94]
Sea $\mathcal{P}$ un programa lógico rebatible, y $\langle \mathcal{A}_0, h_0 \rangle$ una estructura de argumento obtenida a partir de $\mathcal{P}$. Una línea de argumentación *a partir de $\langle \mathcal{A}_0, h_0 \rangle$, es una secuencia de estructuras de argumentos (obtenidas a partir de $\mathcal{P}$) denotada $\Lambda = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle \langle \mathcal{A}_3, h_3 \rangle, \ldots]$, donde para cada elemento de la secuencia $\langle \mathcal{A}_i, h_i \rangle$, el elemento siguiente $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ es un derrotador para $\langle \mathcal{A}_i, h_i \rangle$.*

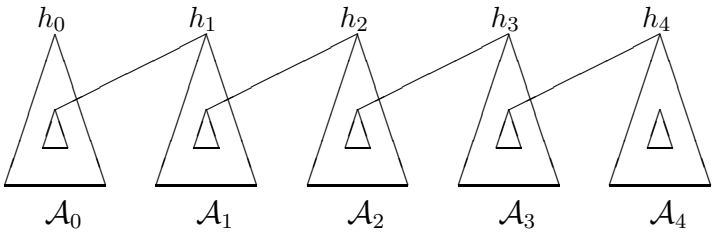

Figura 2.2: Línea de argumentación

El primer elemento de una línea de argumentación $[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle \ldots]$ es una estructura que sustenta a un literal h_0. El segundo elemento la línea de argumentación derrota al primero y por consiguiente interfiere con la creencia en h_0. El tercer elemento derrota al segundo y por consiguiente, indirectamente sustenta a h_0, y asi siguiendo. De esta manera, en una línea de argumentación se distinguirán dos conjuntos disjuntos de argumentos: argumentos de soporte para h_0 y de interferencia para h_0.

Definición 2.26 (Argumentos de soporte y de interferencia) [SCG94]
Sea $\mathcal{P}$ un programa lógico rebatible, y $\Lambda = [\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle \ldots]$ una línea de argumentación. El conjunto de argumentos de soporte está formado por los elementos de posiciones pares de la secuencia $\Lambda_S = \{ \langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_2, h_2\rangle, \langle \mathcal{A}_4, h_4\rangle, \ldots \}$, mientras que el conjunto de argumentos de interferencia está formado por los elementos de posiciones impares $\Lambda_I = \{\langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_3, h_3\rangle, \ldots\}$.

En principio, se podría establecer el siguiente criterio para decidir sobre la creencia en un literal h_0 a partir de un programa $\mathcal{P}$: si la línea de argumentación a partir de $\langle \mathcal{A}_0, h_0\rangle$, tiene un número impar de estructuras de argumento, entonces la creencia en h_0 está justificada, de lo contrario, si tiene un número par, o no existe argumento para h_0, entonces no lo está. Sin embargo, como muestra el ejemplo 2.13, una línea de argumentación no es suficiente para analizar la interacción entre argumentos, ya que para cada estructura de argumento pueden existir más de un derrotador, dando origen a un *árbol de derrotadores*.

Ejemplo 2.13 Sea $\mathcal{P}$ un programa de donde se pueden obtener las estructuras de argumento $\langle \mathcal{A}_0, h_0\rangle$, $\langle \mathcal{A}_1, h_1\rangle$, $\langle \mathcal{A}_2, h_2\rangle$, y $\langle \mathcal{A}_3, h_3\rangle$. Supóngase que $\langle \mathcal{A}_1, h_1\rangle$ y $\langle \mathcal{A}_2, h_2\rangle$ derrotan a $\langle \mathcal{A}_0, h_0\rangle$, y que $\langle \mathcal{A}_3, h_3\rangle$ derrota a $\langle \mathcal{A}_2, h_2\rangle$, entonces existen dos líneas de argumentación a partir de $\langle \mathcal{A}_0, h_0\rangle$:

$[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle]$

$[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_2, h_2\rangle, \langle \mathcal{A}_3, h_3\rangle]$

En la primera línea hay un número par, y en la segunda un número impar de estructuras de argumento.

Antes de definir un criterio que analice todas las posibles líneas de argumentación, se estudiará en detalle los posibles problemas que pueden aparecer considerando una sola línea de argumentación.

2.8 Restricciones sobre una línea de argumentación

2.8.1 Argumentos que se derrotan a sí mismos

Un argumento $\mathcal{A}$ se dice "self-defeating" (se derrota a sí mismo), cuando el argumento $\mathcal{A}$ es un derrotador para el mismo $\mathcal{A}$. Esta situación no es deseable por dos motivos: (1) el argumento es contradictorio con él mismo, con lo cual no podrá sostener o invalidar una creencia, y (2) podría generar una línea de argumentación infinita: $[\langle \mathcal{A}, h\rangle, \langle \mathcal{A}, h\rangle, \langle \mathcal{A}, h\rangle, \ldots]$. Por ejemplo dado el programa:

$$\begin{array}{ll} a \prec b & \sim c \leftarrow a \\ b \prec c & d \\ c \prec d & \end{array}$$

si $\mathcal{A}$= $\{(a \prec b), (b \prec c), (c \prec d)\}$ fuera un argumento para el literal a, entonces $\mathcal{A}$ sería un derrotador para $\mathcal{A}$ en el punto c ya que "$\sim c \leftarrow a$".

Algunos acercamientos a la argumentación rebatible permiten la existencia de argumentos "self-defeating" y tienen que controlar este tipo de situaciones. Sin embargo, en DeLP, un argumento nunca puede ser un derrotador para sí mismo como se muestra en la proposición siguiente. Obsérvese que en el ejemplo anterior el conjunto $\mathcal{A}$ no es un argumento, ya que $\Pi \cup \mathcal{A}$ es contradictorio.

Proposición 2.6 *Una estructura de argumento $\langle \mathcal{A}, h\rangle$ no puede ser un derrotador para sí mismo.*

Demostración: Supóngase que existe un argumento $\langle \mathcal{A}, h\rangle$ que es un derrotador para sí

mismo, entonces debería existir un subargumento $\langle \mathcal{B}, q\rangle$ de $\langle \mathcal{A}, h\rangle$ tal que los literales h y q estén en desacuerdo, esto es, $\Pi \cup \{h, q\}$ sería un conjunto contradictorio. Pero si esto ocurriera, como h y q tienen derivaciones rebatibles a partir de $\Pi \cup \mathcal{A}$, entonces el conjunto $\Pi \cup \mathcal{A}$ sería contradictorio, lo cual contradice la hipótesis de que $\langle \mathcal{A}, h\rangle$ es un argumento (condición 2 de la definición de argumento). □

2.8.2 Derrotadores recíprocos

Como se vió en la proposición 2.5, la noción de contra-argumentación es simétrica, esto es, dos argumentos pueden ser contra-argumentos recíprocamente. En la noción de derrota, al incluir una comparación entre los argumentos, parece desaparecer esta simetría. Sin embargo, como se muestra a continuación, la definición de derrotador puede permitir la aparición de "derrotadores recíprocos" [Pra93].

Ejemplo 2.14 Considérese el siguiente programa:

$$\begin{array}{ll} d -\!\prec \sim b, c & b -\!\prec \sim d, a \\ \sim b -\!\prec a & \sim d -\!\prec c \\ a & c \end{array}$$

de donde se obtienen las siguentes estructuras de argumento: $\langle \mathcal{A}_2$,d$\rangle = \langle\{(d -\!\prec \sim b, c), (\sim b -\!\prec a)\}, d\rangle$, y $\langle \mathcal{A}_1$,b$\rangle = \langle\{(b -\!\prec \sim d, a), (\sim d -\!\prec c)\}, b\rangle$.
Utilizando al criterio de especificidad para comparar los argumentos, se tiene que (ver Figura 2.3), $\langle \mathcal{A}_2$,d$\rangle$ derrota a $\langle \mathcal{A}_1$,b$\rangle$, en el subargumento $\langle \mathcal{B}$,$\sim$d$\rangle$, pero además $\langle \mathcal{A}_1$,b$\rangle$ derrota a $\langle \mathcal{A}_2$,d$\rangle$ en el subargumento $\langle \mathcal{A}$,$\sim$b$\rangle$.

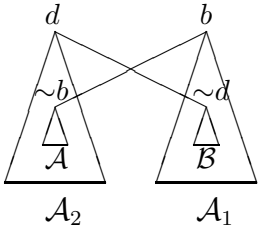

Figura 2.3: Derrotadores Recíprocos

Al igual que en el caso de 'self-defeating', los derrotadores recíprocos podrían generar una secuencia infinita $[\langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \ldots]$. Por lo tanto, los derrotadores recíprocos deben detectarse para no caer en argumentaciones infinitas.

2.8.3 Argumentación Circular

En una línea de argumentación, podría ocurrir que un argumento es reintroducido en la secuencia, lo cual produciría una argumentación circular. La Figura 2.4 muestra un ejemplo donde la estructura de argumento $\mathcal{A}$ es reintroducida como soporte de ella misma. Introducir un argumento que ya fue utilizado previamente en la línea de argumentación se considera una "falacia" desde el punto de vista de la argumentación, pero además computacionalmente produciría un ciclo dentro del proceso de argumentación. Por estos motivos es que se impedirá llegar a esta situación imponiendo la condición que *ningún argumento pueda reintroducirse en la misma línea de argumentación.*

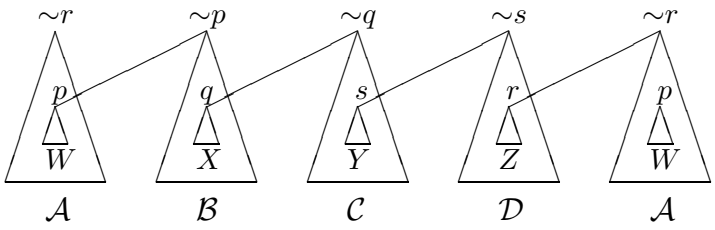

Figura 2.4: línea de argumentación circular

Obsérvese que cuando un argumento es reintroducido como soporte de él mismo, entonces el ciclo tiene un número par de argumentos. También puede ocurrir que el ciclo no involucre toda la línea, como por ejemplo en la línea

$$[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \langle \mathcal{A}_3, h_3\rangle, \langle \mathcal{A}_4, h_4\rangle, \langle \mathcal{A}_1, h_1\rangle, \ldots],$$

donde $\langle \mathcal{A}_0, h_0\rangle$ no es parte del ciclo. El argumento que es reintroducido puede también ser de interferencia como en la línea anterior, sin embargo, como el ciclo tiene un número par de argumentos $\langle \mathcal{A}_1, h_1\rangle$ es reintroducido para ayudarse a sí mismo.

Otro caso de argumentación circular (ciclo par) aparece cuando un subargumento de desacuerdo es reintroducido en la línea de argumentación. La figura 2.5 muestra un ejemplo de este caso: la estructura de argumento $\mathcal{B}$ es un derrotador para $\mathcal{A}$, con subargumento de desacuerdo W. En la misma línea, la estructura W es reintroducida para interferir con $\mathcal{B}$, lo cual dá pie para reintroducir a $\mathcal{B}$.

Aunque el ciclo puede detectarse al reintroducir $\mathcal{B}$, la situación falaz constituye el uso del subargumento de descuerdo como soporte de sí mismo. Por lo tanto se establecerá la siguiente restricción: *ningun argumento* $\langle \mathcal{A}_k, h_k\rangle$ *de una línea de argumentación* Λ *es un sub-argumento de desacuerdo de argumento* $\langle \mathcal{A}_i, h_i\rangle$ *previo en la línea* Λ *(*$i < k$*).*

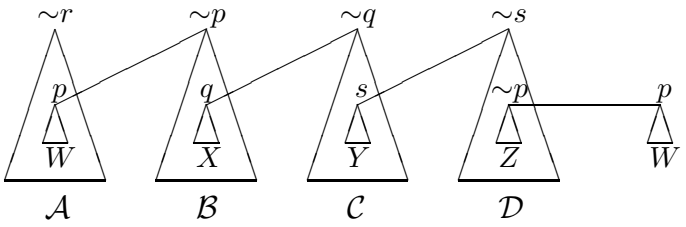

Figura 2.5: Argumentación circular por reintroducir un subargumento

2.8.4 Argumentación contradictoria

Otro tipo de ciclos en la línea de argumentación puede producirse si un argumento es reintroducido como interferencia para él mismo, formando un ciclo con un número impar de argumentos. En el ejemplo de la Figura 2.6 puede verse este problema, donde el argumento $\mathcal{A}$ es reintroducido, y se convierte en un argumento de soporte y de interferencia al mismo tiempo. Este problema es una generalización de "self-defeating arguments" ya que un argumento es reintroducido para interferir con él mismo.

Aunque el ciclo podría evitarse utilizando la condición de argumentación circular, de no introducir un mismo argumento en la línea, el verdadero problema está en aceptar que dos argumentos de soporte sean contradictorios. Obsérvese en la Figura 2.6, que el argumento $\mathcal{C}$, introducido como soporte de $\mathcal{A}$, es en realidad contradictorio con $\mathcal{A}$, ya que $\mathcal{A}$ puede ser utilizado como derrotador de $\mathcal{C}$.

La introducción de un argumento como $\mathcal{C}$ debe ser evitada, ya que se está defendiendo a $\mathcal{A}$ con un argumento que es contradictorio con $\mathcal{A}$. En general, debe haber una coherencia entre los argumentos de soporte de una línea de argumentación. Lo mismo con el conjunto de argumentos de interferencia. Por lo tanto se establecerá una condición de *concordancia* entre los argumentos de soporte (respectivamente de interferencia) en una línea de argumentación.

Definición 2.27 (Concordancia) *Sea* Π *el conjunto de reglas rebatibles y hechos de un programa* $\mathcal{P}$*. Un conjunto de estructuras de argumento* $\{\langle \mathcal{A}_i, h_i \rangle\}_{i=1}^{n}$ *se dice* concordante*, si el conjunto* $\Pi \cup \bigcup_{i=1}^{n} \mathcal{A}_i$ *no es contradictorio.*

De esta forma, con la condición de concordancia entre los argumentos de soporte, (e interferencia) el argumento $\mathcal{C}$ no puede ser utilizado en la línea de argumentación del ejemplo de la Figura 2.6.

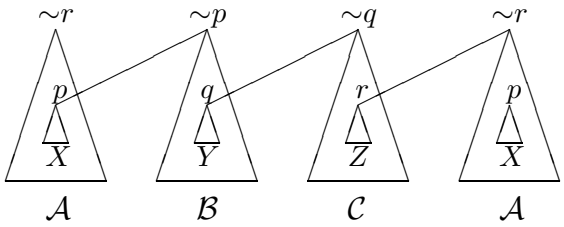

Figura 2.6: línea de argumentación contradictoria

2.8.5 "Argumentos flotantes"

Otro caso problemático de una línea de argumentación estudiado por Makinson y Schlechta [MS91] son los argumentos flotantes (floating arguments). Por ejemplo, si $\langle \mathcal{A}_3, h_3 \rangle$ derrota a $\langle \mathcal{A}_2, h_2 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$ derrota a $\langle \mathcal{A}_3, h_3 \rangle$, y tanto $\langle \mathcal{A}_3, h_3 \rangle$, como $\langle \mathcal{A}_2, h_2 \rangle$ son un derrotador propio de $\langle \mathcal{A}_1, h_1 \rangle$. Se darían las líneas $[\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \langle \mathcal{A}_3, h_3 \rangle]$ y $[\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_3, h_3 \rangle, \langle \mathcal{A}_2, h_2 \rangle]$ con lo cual indirectamente ambos derrotadores de $\langle \mathcal{A}_1, h_1 \rangle$, defienden a $\langle \mathcal{A}_1, h_1 \rangle$.

Según Makinson y Schlechta [MS91], *un argumento no puede defender a otro si también lo derrota*. En nuestro formalismo, este caso esta cubierto por la restricción impuesta a las líneas de argumentación contradictorias. Ya que el conjunto $\{\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_3, h_3 \rangle\}$ no es concordante, y también ocurre lo mismo con el conjunto $\{\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle\}$.

2.8.6 Bloqueo entre argumentos

El objetivo de la programación en lógica rebatible es realizar un análisis de la información presente en el programa a fin de decidir si creer o no en un literal. Por este motivo es que una estructura de argumento puede ser derrotada cuando existe información más relevante, o simplemente bloqueada cuando hay información que no puede compararse.

Supóngase que existen tres estructuras de argumento para el literal h, y dos estructuras para $\sim h$. No está en el espíritu de este sistema que se sustente la creencia del literal h simplemente porque existan más argumentos a su favor.

Como se dijo antes, en una línea de argumentación $[\langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \langle \mathcal{A}_3, h_3\rangle]$, una estructura de argumento $\langle \mathcal{A}_1, h_1\rangle$ puede considerarse no derrotada, si su derrotador $\langle \mathcal{A}_2, h_2\rangle$ está a su vez derrotado por otra estructura $\langle \mathcal{A}_3, h_3\rangle$. Sin embargo, ante la presencia de derrotadores de bloqueo podría ocurrir que un literal prevalezca simplemente porque está sustentado por más de un argumento.

Considérese el siguiente programa:

$$
\begin{array}{l}
peligroso(X) \prec tigre(X) \\
\sim peligroso(X) \prec bebe(X) \\
\sim peligroso(X) \prec mascota(X) \\
tigre(kitty) \\
bebe(kitty) \\
mascota(kitty)
\end{array}
$$

A partir del programa anterior, pueden obtenerse dos estrucuturas de argumento para "$\sim peligroso(kitty)$", con los siguientes conjuntos

$\mathcal{A}$= { $\sim peligroso(kitty) \prec bebe(kitty)$ }, y

$\mathcal{C}$= { $\sim peligroso(kitty) \prec mascota(kitty)$ }

y una estructura de argumento para "$peligroso(kitty)$ con

$\mathcal{B}$= { $peligroso(kitty) \prec tigre(kitty)$ }.

En lo que resta del ejemplo p será una abreviatura de $peligroso(kitty)$. Si se utiliza el criterio de especificidad, $\langle \mathcal{A}, \sim p\rangle$ y $\langle \mathcal{C}, \sim p\rangle$ son derrotadores de bloqueo para $\langle \mathcal{B}, p\rangle$, y $\langle \mathcal{B}, p\rangle$ es un derrotador de bloqueo tanto para $\langle \mathcal{A}, \sim p\rangle$ como para $\langle \mathcal{C}, \sim p\rangle$. En las siguientes dos líneas de argumentación

$[\langle \mathcal{A}, \sim p\rangle, \langle \mathcal{B}, p\rangle, \langle \mathcal{C}, \sim p\rangle\ ]$

$[\langle \mathcal{C}, \sim p\rangle, \langle \mathcal{B}, p\rangle, \langle \mathcal{A}, \sim p\rangle\ ]$

los argumentos que sustentan a "p" ($\sim peligroso(kitty)$) finalizan victoriosos. Sin embargo, como todos los argumentos son de bloqueo, en realidad se está haciendo prevalecer una posición sólo porque hay más argumentos a favor. Obsérvese que en ninguna de las dos líneas puede utilizarse a $\langle \mathcal{B}, p\rangle$ para derrotar al último argumento de la línea, porque se produciría una argumentación circular. Además si existiera un cuarto argumento que bloquee a $\langle \mathcal{C}, \sim p\rangle$ y $\langle \mathcal{A}, \sim p\rangle$, entonces hubiera resultado en un "empate".

El problema aqui es permitir bloquear a un argumento que a su vez está bloqueando a otro. Por lo tanto, para resolver el problema, se impondrá la condición que un argumento usado como derrotador de bloqueo, no puede a su vez ser bloqueado, sólo puede ser derrotado.

2.8.7 Línea de argumentación aceptable

Una línea de argumentación será aceptable si satisface las restricciones definidas en las secciones anteriores.

Definición 2.28 (línea de argumentación aceptable)
Una línea de argumentación $\Lambda = [\langle \mathcal{A}_1, h_1\rangle, \ldots, \langle \mathcal{A}_i, h_i\rangle, \ldots, \langle \mathcal{A}_n, h_n\rangle]$ *será aceptable si:*

1. Λ *es una secuencia finita.*
2. *El conjunto* Λ_S*, de estructuras de argumento de soporte de* Λ*, es concordante, y el conjunto* Λ_I*, de estructuras argumentos de interferencia de* Λ*, también es concordante.*
3. *Ningún argumento* $\langle \mathcal{A}_k, h_k\rangle$ *de* Λ *es un sub-argumento de una estructura de argumento* $\langle \mathcal{A}_i, h_i\rangle$ *que aparece previamente en* Λ *(*$i < k$*).*
4. *Para toda estructura* $\langle \mathcal{A}_i, h_i\rangle$ *de* Λ *tal que* $\langle \mathcal{A}_i, h_i\rangle$ *es un derrotador de bloqueo de* $\langle \mathcal{A}_{i-1}, h_{i-1}\rangle$*, si* $\langle \mathcal{A}_{i+1}, h_{i+1}\rangle$ *existe, entonces* $\langle \mathcal{A}_{i+1}, h_{i+1}\rangle$ *es un derrotador propio de* $\langle \mathcal{A}_i, h_i\rangle$*.*

De esta manera para analizar si creer en un literal h_0 a partir de un programa $\mathcal{P}$, se puede construir una línea de argumentación aceptable comenzando con la estructura de argumento $\langle \mathcal{A}_0, h_0\rangle$, y luego ir introduciendo derrotadores en la secuencia mientras se satisfagan las condiciones establecidas en la definición anterior. Sin embargo, como muestra el ejemplo 2.15 a continuación, una línea de argumentación aceptable no es suficiente para analizar la interacción entre argumentos, ya que para cada estructura de argumento pueden existir más de un derrotador.

2.9 Árboles de dialéctica y literales garantizados

El ejemplo siguiente muestra que en general para cada estructura de argumento puede existir más de un derrotador. La presencia de múltiples derrotadores para una estructura de argumento produce una ramificación de líneas de argumentación, dando origen a un *árbol de derrotadores* que se denominará *árbol de dialéctica* y se definirá a continuación. En este árbol, cada camino desde la raíz hasta una hoja corresponde a un línea de argumentación.

Ejemplo 2.15 Considérese el siguiente programa lógico rebatible:

$a \prec b$	$\sim d \prec k$	$\sim b \prec c, f$	$\sim f \prec i$
$b \prec c$	e	$f \prec g$	i
c	$\sim b \prec e$	g	$\sim h \prec k$
$\sim b \prec c, d$	$h \prec j$	k	
$d \prec g$	j	$\sim f \prec g, h$	

A partir de este programa, es posible construir una estructura de argumento $\langle \mathcal{A}, a \rangle$, con $\mathcal{A}$=$\{$ $(a \prec b)$, $(b \prec c)$ $\}$. Si se utiliza el criterio de especificidad para comparar estructuras de argumentos, es posible construir tres derrotadores para $\langle \mathcal{A}, a \rangle$ que atacan indirectamente en el literal b:

- $\langle \mathcal{B}_1, \sim b \rangle$, con $\mathcal{B}_1$= $\{$ $(\sim b \prec c, d)$ $\}$
- $\langle \mathcal{B}_2, \sim b \rangle$, con $\mathcal{B}_2$= $\{$ $(\sim b \prec c, f)$, $(f \prec g)$ $\}$
- $\langle \mathcal{B}_3, \sim b \rangle$, con $\mathcal{B}_3$= $\{(\sim b \prec e)$ $\}$

La estructura $\langle \mathcal{B}_1, \sim b \rangle$, a su vez, tiene un derrotador $\langle \mathcal{C}_1, \sim d \rangle$ con $\mathcal{C}_1$= $\{\sim d \prec k\}$. La estructura $\langle \mathcal{B}_2, \sim b \rangle$ tiene dos derrotadores: $\langle \mathcal{C}_2, \sim f \rangle$ con $\mathcal{C}_2$= $\{(\sim f \prec i)\}$, y $\langle \mathcal{C}_3, \sim f \rangle$ con $\mathcal{C}_3$= $\{(\sim f \prec g, h)$, $(h \prec j)\}$. Finalmente, la estructura $\langle \mathcal{C}_3, \sim f \rangle$ tiene el derrotador $\langle \mathcal{D}_1, \sim h \rangle$, donde $\mathcal{D}_1$= $\{(\sim h \prec k)\}$.

En el ejemplo anterior existen cuatro líneas de argumentación aceptables a partir de $\langle \mathcal{A}, a \rangle$:

$\Lambda_1 = [\langle \mathcal{A}, a\rangle, \langle \mathcal{B}_1, \sim b\rangle, \langle \mathcal{C}_1, \sim d\rangle\ ]$
$\Lambda_2 = [\langle \mathcal{A}, a\rangle, \langle \mathcal{B}_2, \sim b\rangle, \langle \mathcal{C}_2, \sim f\rangle\ ]$
$\Lambda_3 = [\langle \mathcal{A}, a\rangle, \langle \mathcal{B}_2, \sim b\rangle, \langle \mathcal{C}_3, \sim f\rangle, \langle \mathcal{C}_3, \sim h\rangle\ ]$
$\Lambda_4 = [\langle \mathcal{A}, a\rangle, \langle \mathcal{B}_3, \sim b\rangle\ ]$

En realidad, estas cuatro líneas pueden representarse como un árbol, donde la raíz está etiquetada con $\langle \mathcal{A}, a\rangle$ y los nodos internos representan derrotadores de su nodo padre. Las hojas del árbol corresponden a estructuras de argumento sin derrotadores. Este tipo de árboles se llama árbol de dialéctica, y se define a continuación. La Figura 2.7 muestra el árbol correspondiente a este ejemplo.

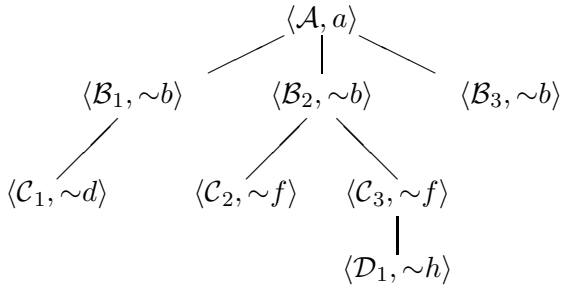

Figura 2.7: Árbol de dialéctica del ejemplo 2.15

Definición 2.29 (Arbol de dialéctica)
Sea $\langle \mathcal{A}_0, h_0\rangle$ una estructura de argumento obtenida a partir de un programa $\mathcal{P}$. Un árbol de dialéctica para $\langle \mathcal{A}_0, h_0\rangle$, a partir de $\mathcal{P}$, se denota $\mathcal{T}_{\langle \mathcal{A}_0, h_0\rangle}$, y se construye de la siguiente forma:

1. *La raíz del árbol es etiquetada con $\langle \mathcal{A}_0, h_0\rangle$.*
2. *Sea N un nodo del árbol etiquetado $\langle \mathcal{A}_n, h_n\rangle$, y $[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \ldots, \langle \mathcal{A}_n, h_n\rangle]$ la secuencia de etiquetas del camino que va desde la raíz hasta el nodo N.*
   Sean $\langle \mathcal{B}_1, q_1\rangle$, $\langle \mathcal{B}_2, q_2\rangle$, ..., $\langle \mathcal{B}_k, q_k\rangle$ todos los derrotadores de $\langle \mathcal{A}_n, h_n\rangle$.
   Para cada derrotador $\langle \mathcal{B}_i, q_i\rangle$ $(1 \leq i \leq k)$, tal que, la línea de argumentación $[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \ldots, \langle \mathcal{A}_n, h_n\rangle, \langle \mathcal{B}_i, q_i\rangle]$ sea aceptable, existe un nodo hijo N_i de N etiquetado con $\langle \mathcal{B}_i, q_i\rangle$.
   Si no existe ningún derrotador $\langle \mathcal{B}_i, q_i\rangle$ en tales condiciones, entonces el nodo N es una hoja.

Como puede observarse en la Figura 2.7, los nodos hoja del árbol de dialéctica corresponden a argumentos no derrotados. En cambio, un nodo interno que tiene como hijo un nodo hoja, corresponderá a un argumento derrotado. Siguiendo con este análisis, desde las hojas, hacia la raíz, los nodos en un árbol de dialéctica se pueden marcar como "D" *derrotado*, o "U" *no derrotado*[4]. A continuación se especifica el procedimiento de marcado:

Procedimiento 2.1 (Marcado de un árbol de dialéctica)
Sea $\mathcal{T}_{\langle \mathcal{A}, h\rangle}$ un árbol de dialéctica para $\langle \mathcal{A}, h\rangle$. Un árbol de dialéctica marcado, denotado $\mathcal{T}^*_{\langle \mathcal{A}, h\rangle}$ puede obtenerse marcando cada nodo en $\mathcal{T}_{\langle \mathcal{A}, h\rangle}$ de la siguiente forma:

1. *Todas las hojas de $\mathcal{T}_{\langle \mathcal{A}, h\rangle}$ se marcan con "U" en $\mathcal{T}^*_{\langle \mathcal{A}, h\rangle}$.*
2. *Sea N un nodo interno de $\mathcal{T}_{\langle \mathcal{A}, h\rangle}$. El nodo N se marca con "U" si todo nodo hijo de N está marcado con "D", y N se marca con "D" si existe al menos un nodo hijo de N marcado con "U".*

Este procedimiento sugiere un proceso desde las hojas a la raíz (bottom-up) para marcar los nodos, y determinar el estado de la raíz. La Figura 2.8 muestra el árbol de la figura 2.7 después de aplicarle el procedimiento de marcado.

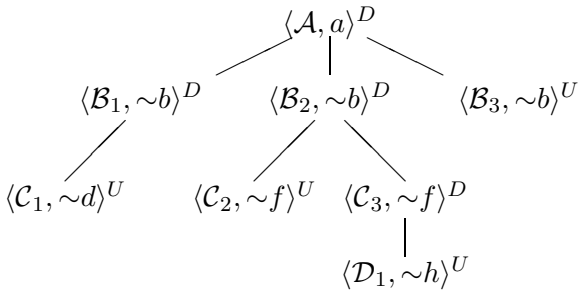

Figura 2.8: Árbol de dialéctica marcado del ejemplo 2.15

Un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, h\rangle}$ representa el análisis dialéctico en el cuál todos los argumentos construibles a partir de un programa $\mathcal{P}$ son considerados a fin de decidir el status de un argumento $\langle \mathcal{A}, h\rangle$. En un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, h\rangle}$, los nodos marcados con "U" corresponden a argumentos no derrotados y los nodos marcados con

[4]En inglés *undefeated*

"D" a estructuras de argumento derrotadas. Por lo tanto, el status de un argumento $\langle \mathcal{A}, h \rangle$ será "derrotado" si la raíz de $\mathcal{T}^*_{\langle \mathcal{A}, h \rangle}$ queda marcada con "D", o "garantizado"[5] si la raíz tiene como marca "U".

La siguiente definición es el concepto que resume todo el análisis dialéctico en la programación en lógica rebatible. Decir que un literal "h" está garantizado a partir de un programa $\mathcal{P}$, significa que un razonador que utilice el programa $\mathcal{P}$, podrá creer en "h".

Definición 2.30 (Literales garantizados)
Sea $\mathcal{P}=(\Pi, \Delta)$, un programa lógico rebatible, y h un literal. Sea $\langle \mathcal{A}, h \rangle$ una estructura de argumento para h, y $\mathcal{T}^*_{\langle \mathcal{A}, h \rangle}$ el árbol de dialéctica marcado asociado a $\langle \mathcal{A}, h \rangle$. El literal h está garantizado *si la raíz de $\mathcal{T}^*_{\langle \mathcal{A}, h \rangle}$ está marcada con "U".*

A partir de la noción de literal garantizado, es posible definir un operador modal de creencia "B" de la siguiente manera: "Bh" si y solo si "h" es un literal garantizado. De esta manera, la expresión Bh debe leerse "cree en h". Con la ayuda de este operador, es posible definir las respuestas que daría un intérprete de programas lógicos rebatibles, o un razonador que utilice un programa lógico rebatible para representar su conocimiento, y la noción de literal garantizado para convalidar sus creencias.

Definición 2.31 (Respuestas a una consulta) *Dado un programa lógico rebatible $\mathcal{P}=(\Pi, \Delta)$. Sea h un literal que representa una consulta para un programa $\mathcal{P}$. La respuesta a h será*

- SI*: si Bh (cree en h)*
- NO*: si $B\overline{h}$ (cree en $\overline{h}$)*
- INDECISO*: si $\neg Bh$ y $\neg B\overline{h}$ (no cree en h ni en $\sim h$)*
- DESCONOCIDO*: si h no pertenece a la signatura del programa.*

La siguiente proposición muestra que en particular los literales que son hechos de un programa $\mathcal{P}$, o tienen una derivación estricta a partir de $\mathcal{P}$, siempre están garantizados a partir de $\mathcal{P}$.

[5] En inglés *warrant*.

Proposición 2.7 *Sea $\mathcal{P} = (\Pi, \Delta)$ un programa lógico rebatible. Si el literal h tiene una derivación estricta, entonces h está garantizado.*
Demostración: Si un literal h posee una derivación estricta, entonces por la Proposición 2.1 existe una estructura de argumento $\langle \mathcal{A}, h \rangle$, con $\mathcal{A} = \emptyset$. Por la Proposición 2.3 no existe contra-argumento para $\mathcal{A}$, con lo cual, el árbol de dialéctica $\mathcal{T}^*_{\langle \mathcal{A}, h \rangle}$ tiene un único nodo raíz y hoja, que es marcado como nodo "U". Por lo tanto, h está garantizado. □

El ejemplo 2.16 muestra un programa donde pueden realizarse consultas que cubren las cuatro respuestas posibles.

Ejemplo 2.16 Considérese el siguiente programa:

$vuela(X) \prec ave(X)$	$vuela(X) \prec pato(X)$
$ave(X) \leftarrow pingüino(X)$	$\sim vuela(X) \prec bebe(X)$
$pingüino(tweety)$	$bebe(lucas)$
$\sim vuela(X) \prec pingüino(X)$	$pato(lucas)$
$llega\_a\_una\_isla(X) \prec vuela(X)$	$llega\_a\_una\_isla(X) \prec nada(X)$

A continuación se muestran algunas consultas, su respuesta y una breve explicación.

Consulta:	Respuesta:	Observaciones:
$ave(tweety)$	SI	garantizado (derivación estricta)
$pingüino(tweety)$	SI	garantizado (es un hecho)
$\sim vuela(tweety)$	SI	garantizado: B $\sim vuela(tweety)$ (no derrotado)
$vuela(tweety)$	NO	argumento derrotado y B $\sim vuela$
$\sim vuela(lucas)$	INDECISO	argumento bloqueado
$vuela(lucas)$	INDECISO	argumento bloqueado
$nada(lucas)$	DESCONOCIDO	no pertenece a la signatura del programa
$llega\_a\_una\_isla(tweety)$	INDECISO	($\neg B$ $llega\_a\_una\_isla(tweety)$ y $\neg B$ $\sim llega\_a\_una\_isla(tweety)$)

En el último caso, la estructura de argumento $\langle \mathcal{A}, llega\_a\_una\_isla(tweety) \rangle$, con

$$\mathcal{A} = \left\{ \; llega\_a\_una\_isla(tweety) \prec vuela(tweety) \; \right\}$$

es derrotada indirectamente en el punto $vuela(tweety)$, por el derrotador propio:

$$\mathcal{B} = \left\{ \sim vuela(tweety) \prec ping\ddot{u}ino(tweety) \right\}$$

Por lo tanto no se puede garantizar $llega\_a\_una\_isla(tweety)$. Como tampoco se puede garantizar $\sim llega\_a\_una\_isla(tweety)$, la respuesta es INDECISO.

Observación 2.15 *En este formalismo, Bp implica $\neg B{\sim}p$ (creer en p implica no creer en $\sim p$), o dicho de otra manera, no se puede creer en "$\sim p$" y "p" al mismo tiempo. Lo cual puede demostrarse facilmente. Supongamos que se cree en "$\sim p$" y "p" al mismo tiempo, esto es, Bp y $B{\sim}p$. Esto lleva directamente a un absurdo, porque si $\sim p$ esta garantizado, entonces es un derrotador para p y viceversa.*

Sin embargo, la recíproca ($\neg B{\sim}p$ implica Bp) no es cierta, ya que puede darse el caso de no creer ni en p, ni en $\sim p$. Por ejemplo en el programa $\mathcal{P} = (\emptyset, \{a \prec b\})$, no se cree en "$a$", y tampoco se cree en "$\sim a$".

2.10 Cómputo de árboles de dialéctica con poda

Considérese el árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, a\rangle}$ de la Figura 2.9 (izquierda). Dicho árbol tiene un parecido con los árboles *mini-max* usados en Inteligencia Artificial para la implementación de búsqueda adversaria en juegos de dos participantes. En lugar de marcar los nodos con 1 o -1, como en *mini-max*, en el caso de los árboles de dialéctica los nodos se marcan como "D", o "U".

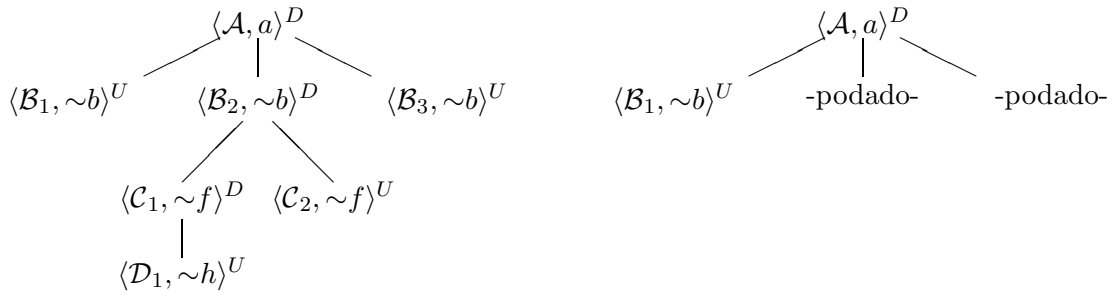

Figura 2.9: Árbol de dialéctica marcado (izquierda) y poda del mismo (derecha)

Es importante notar que durante el proceso de marcado, algunos nodos no necesitan ser tomados en cuenta para obtener la marca de su nodo padre. Esto es, podrían ser tanto "U" como "D", sin modificar la marca de la raíz. Por ejemplo en la Figura 2.9 (izquierda), el nodo hijo de más a la izquierda de la raíz es "U", con lo cual la raíz es "D", sin importar la marca que tienen los demás hijos de la raíz. Este tipo de nodos que no contribuyen a la marca del padre, pueden no ser tenidos en cuenta, con lo cuál se puede "podar" el sub-árbol que tienen por raíz. El proceso de "poda" es similar a la poda "α-β" de un árbol *mini-max*. Por lo tanto, durante el proceso de marcado, una vez que un nodo es marcado como "U", la marca de su padre será "D" y todos sus hermanos pueden podarse. La Figura 2.9 muestra a la izquierda el árbol de dialéctica mardado para el argumento $\mathcal{A}$ del Ejemplo 2.15 y a la derecha el árbol podado recorriendo en profundidad.

```
warrant(H):-          find_argument(H,A),
                      \+ defeated(A,[A]).
defeated(A,Line):-    find_defeater(A,D),
                      acceptable(D,Line),
                      \+ defeated(D,[D|Line]).
```

Figura 2.10: Especificación del proceso de dialéctica con poda

La especificación de la Figura 2.10, muestra como verificar si un literal H está garantizado, utilizando el proceso dialéctica con poda. La especificación se realizó utilizando un programa PROLOG donde \+ representa la negación default. Siguiendo la especificación, en primer lugar, se busca si existe una estructura de argumento A para H que no esté derrotada. Para verificar si una estructura A está derrotada, se busca un derrotador D que permita continuar con una línea de argumentación aceptable, y si existe, se verifica (recursivamente) que el derrotador D en cuestión no esté derrotado. Obsérvese que la aceptabilidad de la línea de argumentación se verifica dinámicamente, a medida que se agregan argumentos a la línea. El árbol de dialéctica es construído en profundidad (Depth-first), podando aquellas ramas que no es necesario considerar.

La especificación completa de la Programación en Lógica Rebatible será dada al describir la semántica operacional en el capítulo 4. El ejemplo 2.17 a continuación muestra muestra como se construyen las líneas de argumentación siguiendo la especificación de la Figura 2.10.

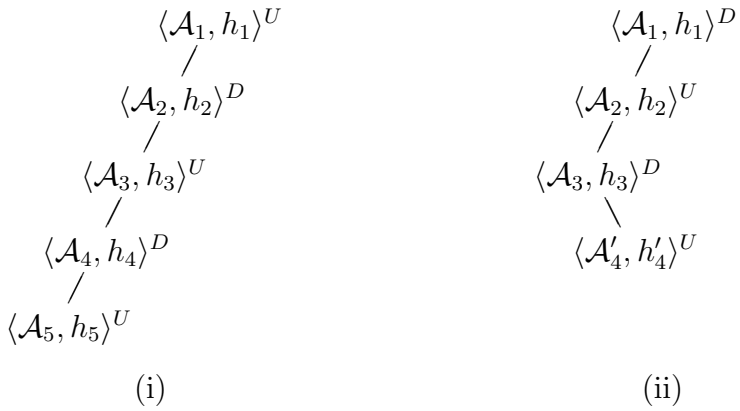

Figura 2.11: Líneas de argumentación del ejemplo 2.17

Ejemplo 2.17 Supóngase que al intentar verificar si un literal h_1 está garantizado, se ha encontrado la estructura de argumento $\langle \mathcal{A}_1, h_1 \rangle$, y se ha construído la línea de argumentación $[\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \langle \mathcal{A}_3, h_3 \rangle, \langle \mathcal{A}_4, h_4 \rangle, \langle \mathcal{A}_5, h_5 \rangle]$ como muestra la Figura 2.11 (i).

Hasta aquí, la línea de argumentación encontrada termina con una estructura de argumento de soporte ($\langle \mathcal{A}_5, h_5 \rangle$), y por lo tanto el procedimiento de marcado establecería – al menos hasta este punto – que la estructra $\langle \mathcal{A}_1, h_1 \rangle$ es un nodo "U". Pero como podrían existir más derrotadores a considerar, el proceso debe continuar, expandiendo otras líneas de argumentación.

En primer lugar, podrían existir derrotadores para $\langle \mathcal{A}_4, h_4 \rangle$, sin embargo, encontrar más derrotadores para $\langle \mathcal{A}_4, h_4 \rangle$, no cambiará el status de $\langle \mathcal{A}_4, h_4 \rangle$, que ya es derrotado. Por lo tanto, si existen otros hijos de $\langle \mathcal{A}_4, h_4 \rangle$, estos pueden ser podados.

El análisis anterior hecho para $\langle \mathcal{A}_4, h_4 \rangle$ no puede aplicarse a $\langle \mathcal{A}_3, h_3 \rangle$, ya que es un nodo "U", y si existiera otro derrotador para $\langle \mathcal{A}_3, h_3 \rangle$, entonces podría pasar a "D". Por lo tanto es necesario explorar si existen otros derrotadores de $\langle \mathcal{A}_3, h_3 \rangle$, generando si existiesen, otras líneas de argumentación.

Supongamos que el derrotador $\langle \mathcal{A}'_4, h'_4 \rangle$ para $\langle \mathcal{A}_3, h_3 \rangle$ es encontrado. Entonces, como muestra la Figura 2.11 (ii), se crea una nueva línea de argumentación. Si $\langle \mathcal{A}'_4, h'_4 \rangle$ no tuviera derrotadores, entonces como muestra la figura, el status del nodo raíz cambia a "D".

Utilizando el mismo criterio que antes, no es necesario analizar nuevos derrotadores para $\langle \mathcal{A}_3, h_3 \rangle$, obteniendo una nueva poda. Sin embargo, es necesario analizar si existen otros derrotadores para $\langle \mathcal{A}_2, h_2 \rangle$, con lo cual el status de $\langle \mathcal{A}_1, h_1 \rangle$ podría cambiar.

2.11 Conclusiones

Un programa lógico rebatible $\mathcal{P}=(\Pi, \Delta)$, estará compuesto por *hechos*, *reglas estrictas*, y *reglas rebatibles*. El conjunto Π de los *hechos* y las *reglas estrictas* se utiliza para representar conocimiento libre de excepciones, y se asume no contradictorio. Sin embargo, el conjunto Δ de las reglas rebatibles, permite representar información tentativa y potencialmente contradictoria,

Existe entonces una gran diferencia entre un literal que se deriva en forma estricta a partir de un programa, y un literal que se deriva utilizando reglas rebatibles. A continuación se resumen las propiedades que cumple un literal que puede derivarse estríctamente a partir de Π.

Sea $\mathcal{P}=(\Pi, \Delta)$ un programa, y h un literal que tiene una derivación estricta a partir de $\mathcal{P}$, entonces, se verifican las siguiente propiedades:

- existe una única estructura de argumento $\langle \mathcal{A}, h \rangle$, donde $\mathcal{A}=\emptyset$ (proposición 2.1),
- no puede existir un argumento para $\overline{h}$ (proposición 2.2),
- h no puede ser atacado con un contra-argumento (proposición 2.3),
- no existe estructura argumento $\langle \mathcal{B}, q \rangle$ tal que $\langle \mathcal{A}, h \rangle$ sea un contra-argumento para $\langle \mathcal{B}, q \rangle$. (proposición 2.4),
- h está garantizado (proposición 2.7).

En otras palabras, la información que puede derivarse en forma estricta, es aceptada sin necesidad de mayor análisis. Sin embargo, en aplicaciones reales es más común disponer de información tentativa y generalmente contradictoria. Cuando se utilizan reglas rebatibles para derivar una conclusión C, esta conclusión será tentativa, y podrá ser refutada por información que la contradiga.

El procedimiento de prueba que hemos utilizado para los programas lógicos rebatibles, se basa en las nociones de estructura de argumento y contra-argumento definidas en [SL92], y adaptadas a los programas lógicos rebatibles. Para comparar argumentos se describió la definición de especificidad generalizada, y fue definido un nuevo criterio de comparación entre argumentos utilizando prioridades entre reglas. Las restricciones de una línea de argumentación definidas en [SCG94], también fueron extendidas en esta tesis, para consider nuevos casos.

En la sección 2.9 se introdujo un operador de creencia "B" que permite definir las creencias de un agente que razona utilizando DeLP. Dicho operador también permite definir las respuestas que daría un interprete de programas lógicos rebatibles. En el capítulo 4 se mostrará como implementar dicho intérprete.

La Programación en Lógica Rebatible es un lenguaje que puede ser utilizado básicamente en cualquier aplicación que requiera representación de conocimiento. Sus características le permiten modelar conocimiento que involucre información incompleta o potencialmente contradictoria. El mecanismo de inferencia sobre el cual está basado permite decidir entre conclusiones contradictorias, y adaptarse fácilmente a entornos cambiantes. Esto es, aplicaciones donde se agrega o quita información en forma dinámica, y por lo tanto conclusiones obtenidas previamente pueden quedar invalidadas.

Las características antes mencionadas hacen de los Programas Lógicos Rebatibles, una herramienta atractiva para representar el conocimiento y definir el razonamiento de *agentes inteligentes deliberativos*. La flexibilidad del lenguaje permite que los agentes trabajen en un entorno donde la información cambia rápidamente. En el capítulo 8, se mostrará una aplicación concreta donde se utilizó la Programación en Lógica Rebatible para construir agentes inteligentes deliberativos.

Capítulo 3

Representación de conocimiento utilizando DeLP

Existen en la literatura, ejemplos que se han convertido en un estándar para demostrar las virtudes o defectos de un formalismo de razonamiento no monótono o de un sistema de argumentación rebatible. En este capítulo se mostrará como representar estos ejemplos en DeLP, y como se comporta DeLP con ellos. Se incluirán también ejemplos propios, que muestran características especiales de DeLP.

A menos que se indique explícitamente lo contrario, en este capítulo se asume que en los ejemplos se utiliza el criterio de especificidad para comparar las estructuras de argumentos.

3.1 Ejemplos

Ejemplo 3.1 (Tweety no vuela)
Quizás el ejemplo más difundido de la literatura y utilizado para introducir la noción de no monotonicidad sea el que representa el siguiente programa lógico rebatible, donde se establece cierta información tentativa: "las aves normalmente vuelan", y luego información segura: "los pingüinos son aves", "los pingüinos no vuelan", y "Tweety es un pingüino".

$$\mathcal{P}_{3.1} = \left\{ \begin{array}{l} vuela(X) -\!\prec ave(X) \\ ave(X) \leftarrow pingüino(X) \\ \sim vuela(X) \leftarrow pingüino(X) \\ pingüino(tweety) \end{array} \right\}$$

A partir de este programa es posible obtener una derivación estricta para el literal $\sim vuela(tweety)$. La proposición 2.7 nos asegura entonces que $\sim vuela(tweety)$ está garantizado, ya que que no podrá construirse un contra-argumento (ver proposición 2.3). Observe además, que el literal $ave(tweety)$ también tiene una derivación estricta.

El literal $vuela(tweety)$, aunque posee una derivación rebatible, no posee un argumento, ya que se viola la condición 2 de la definición 2.15. Por lo tanto un razonador que utilice DeLP creerá en que "Tweety no vuela" a partir de este programa.

Ejemplo 3.2 (Tina vuela)
En el programa $\mathcal{P}_{3.2}$ a continuación, a diferencia del ejemplo 3.1, entrán en conflicto dos derivaciones rebatibles. El ejemplo tambíen muestra como se construye una línea de argumentación y se muestran los dos casos considerados por la relación de especificidad.

$$\mathcal{P}_{3.2} = \left\{ \begin{array}{l} vuela(X) -\!\prec ave(X) \\ \sim vuela(X) -\!\prec gallina(X) \\ vuela(X) -\!\prec gallina(X), asustada(X) \\ ave(X) \leftarrow gallina(X) \\ gallina(tina) \\ asustada(tina) \end{array} \right\}$$

A partir de este programa el literal "$vuela(tina)$" tiene la siguiente estructura de argumento que lo sustenta

$$\langle \{vuela(tina) -\!\prec ave(tina)\}, vuela(tina) \rangle$$

por su parte, la estructura anterior tiene el siguiente derrotador propio:

$$\langle \{\sim vuela(tina) -\!\prec gallina(tina)\}, \sim vuela(tina) \rangle$$

La línea de argumentación se completa con la siguiente estructura que es un derrotador propio de la anterior y reinstala la creencia en que "Tina vuela"

$$\langle \{vuela(tina) \prec gallina(tina), asustada(tina)\}, vuela(tina) \rangle$$

Como no hay más argumentos para considerar, "$vuela(tina)$" está garantizado.

Ejemplo 3.3 (Pacifista)
El programa $\mathcal{P}_{3.3}$ a continuación es utilizado en la literatura para mostrar el caso en que no se puede decidir entre conclusiones contradictorias. Muchos sistemas resuelven este caso con extensiones múltiples:

$$\mathcal{P}_{3.3} = \left\{ \begin{array}{l} pacifista(X) \prec quakero(X) \\ \sim pacifista(X) \prec republicano(X) \\ quakero(nixon) \\ republicano(nixon) \end{array} \right\}$$

de donde se obtienen las siguentes estructuras de argumento:

$$\langle \{pacifista(nixon) \prec quakero(nixon)\}, pacifista(nixon) \rangle$$
$$\langle \{\sim pacifista(nixon) \prec republicano(nixon)\}, \sim pacifista(nixon) \rangle$$

Como ambas estructuras están basadas en diferente información, ninguna es más específica, y por lo tanto, cada una es un derrotador de bloqueo para la otra. De esta forma, en DeLP ni "$pacifista(nixon)$", ni "$\sim pacifista(nixon)$" están garantizados, con lo cual, no es posible creer en ninguna de las dos conclusiones, y la respuesta a cualquiera de los literales será "INDECISO".

Ejemplo 3.4 (Prioridades)
Considere nuevamente el programa $\mathcal{P}_{3.3}$, y suponga que se establece una prioridadad entre las reglas expresando que:

$$(pacifista(X) \prec quakero(X)) > (\sim pacifista(X) \prec republicano(X))$$

Utilizando ahora el criterio de comparación de argumentos que considera la prioridad entre las reglas, la estructura de argumento

$$\langle \{pacifista(nixon) \prec quakero(nixon)\}, pacifista(nixon) \rangle$$

es un derrotador propio para la estructura

$$\langle\{\sim pacifista(nixon) \prec republicano(nixon)\}, \sim pacifista(nixon)\rangle$$

y por lo tanto, ahora el literal "$pacifista(nixon)$" está garantizado.

Ejemplo 3.5 (Oferta y demanda)
Este ejemplo muestra la importancia de comparar argumentos completos, en lugar de reglas individuales. Si sólo se comparan dos reglas con cabezas complementarias, entonces se puede interpretar que una es más específica que la otra, mientras que en realidad no lo es. Considere el programa $\mathcal{P}_{3.5}$,

$$\mathcal{P}_{3.2} = \left\{ \begin{array}{l} aumentan\_los\_precios \prec aumenta\_la\_demanda) \\ \sim aumentan\_los\_precios \prec aumenta\_la\_demanda, aumenta\_la\_oferta \\ aumenta\_la\_oferta \prec aumenta\_la\_demanda \\ aumenta\_la\_demanda \end{array} \right\}$$

A partir de este programa se pueden construir la siguiente estructura de argumento que sustenta la conclusión "$aumentan\_los\_precios$"

$$\langle\{aumentan\_los\_precios \prec aumenta\_la\_demanda\}, aumentan\_los\_precios\rangle$$

y el contra-argumento $\langle\mathcal{A}, \sim aumentan\_los\_precios\rangle$, con

$$\mathcal{A}= \left\{ \begin{array}{l} \sim aumentan\_los\_precios \prec aumenta\_la\_demanda, aumenta\_la\_oferta \\ aumenta\_la\_oferta \prec aumenta\_la\_demanda \end{array} \right\}$$

A pesar que la segunda estructura de argumento parece estar basada en mayor información, no es más específica que el primero porque en realidad el literal $aumenta\_la\_oferta$ es derivado a partir del literal $aumenta\_la\_demanda$. Por lo tanto los dos argumentos se basan en la misma información, y cada uno es un derrotador de bloqueo del otro.

Ejemplo 3.6 (Argumento "zombie")
Supóngase que se tienen dos argumentos $\mathcal{A}$ y $\mathcal{B}$ que se bloquean mutuamente (como en el Ejemplo 3.3) y además el argumento $\mathcal{A}$ derrota a otro argumento $\mathcal{C}$. En muchos sistemas de argumentación rebatible, la indecisión del bloqueo mutuo se propaga y el argumento

$\mathcal{C}$ no es aceptado (ver [PV00]). En esos casos, Makinson y Schlechta en [MS91] llaman a $\mathcal{A}$ un "argumento zombie": no está 'vivo' porque es derrotado por $\mathcal{B}$, pero no está 'totalmente muerto' porque derrota a $\mathcal{C}$. El siguiente programa muestra que en DeLP esto no ocurre, y $\mathcal{C}$ es aceptado.

El programa $\mathcal{P}_{3.6}$ a continuación es una adaptación del un ejemplo presentado en [PV00].

$$\mathcal{P}_{3.6} = \left\{ \begin{array}{l} tiene\_un\_arma(X) -\!\prec vive\_en\_chicago(X) \\ \sim tiene\_un\_arma(X) -\!\prec vive\_en\_chicago(X), pacifista(X) \\ vive\_en\_chicago(nixon) \\ pacifista(X) -\!\prec quakero(X) \\ \sim pacifista(X) -\!\prec republicano(X) \\ quakero(nixon) \\ republicano(nixon) \end{array} \right\}$$

En este caso existe una estructura de argumento que sustenta a $tiene\_un\_arma(nixon)$

$$\langle \{tiene\_un\_arma(nixon) -\!\prec vive\_en\_chicago(nixon)\}, tiene\_un\_arma(nixon) \rangle$$

la cual es derrotada por $\langle \mathcal{A}, \sim tiene\_un\_arma(nixon) \rangle$

$$\mathcal{A} = \left\{ \begin{array}{l} \sim tiene\_un\_arma(nixon) -\!\prec vive\_en\_chicago(nixon), pacifista(nixon) \\ pacifista(nixon) -\!\prec quakero(nixon) \end{array} \right\}$$

Esta última tiene como sub-estructura de argumento a:

$$\langle \{pacifista(nixon) -\!\prec quakero(nixon)\}, pacifista(nixon) \rangle$$

que es bloqueada por:

$$\langle \{\sim pacifista(nixon) -\!\prec republicano(nixon)\}, \sim pacifista(nixon) \rangle$$

Lo cual reinstaura la primera estructura de argumento. La estructura de argumento para $pacifista(nixon)$ no puede utilizarse nuevamente porque produce un línea de argumentación circular. Por lo tanto, el literal $tiene\_un\_arma(nixon)$ está garantizado.

Ejemplo 3.7 (Armas en Chicago)
El programa $\mathcal{P}_{3.7}$ a continuación es un extensión del programa $\mathcal{P}_{3.6}$,

$$\mathcal{P}_{3.7} = \left\{ \begin{array}{l} tiene\_un\_arma(X) -\prec vive\_en\_chicago(X) \\ \sim tiene\_un\_arma(X) -\prec vive\_en\_chicago(X), pacifista(X) \\ vive\_en\_chicago(nixon) \\ pacifista(X) -\prec quakero(X) \\ \sim pacifista(X) -\prec republicano(X) \\ quakero(nixon) \\ republicano(nixon) \\ vive\_en\_chicago(mary) \\ pacifista(mary) \\ vive\_en\_chicago(sally) \\ \sim tiene\_un\_arma(X) -\prec chico(X) \\ vive\_en\_chicago(tomy) \\ chico(tomy) \end{array} \right\}$$

En este caso, como $pacifita(mary)$ es un hecho, entonces se puede construir la estructura de argumento $\langle \mathcal{B}, \sim tiene\_un\_arma(mary) \rangle$, donde

$$\mathcal{B} = \{ \sim tiene\_un\_arma(mary) -\prec vive\_en\_chicago(mary), pacifista(mary) \}$$

Esta estructura no es derrotada por ninguna otra, ya que es más especifica que su único contra-argumento. Por lo tanto se cree que Mary no tiene un arma.

La única información sobre *Sally* es que "vive en Chicago", y por lo tanto la creencia a partir de este programa es "tiene un arma", ya que el argumento

$$\langle \{ tiene\_un\_arma(sally) -\prec vive\_en\_chicago(sally) \}, tiene\_un\_arma(sally) \rangle$$

no tiene contra-argumentos.

En el caso de $Tomy$, el argumento

$$\langle \{ tiene\_un\_arma(tomy) -\prec vive\_en\_chicago(tomy) \}, tiene\_un\_arma(tomy) \rangle$$

es bloqueado por el contra-argumento

$$\langle \{ \sim tiene\_un\_arma(tomy) -\prec chico(tomy) \}, \sim tiene\_un\_arma(tomy) \rangle$$

y por lo tanto no hay ninguna creencia garantizada.

3.2 Líneas de argumentación aceptables

En el caso de los derrotadores recíprocos, considerados en la sección 2.8.2, las respuestas serán INDECISO, indicando su condición de bloqueo mutuo. A continuación se re-escribe el ejemplo 2.14 donde se muestra este caso.

Ejemplo 3.8 (Derrotadores Recíprocos)[SCG94]
Considérese el programa $\mathcal{P}_{3.8}$

$$\mathcal{P}_{3.8} = \left\{ \begin{array}{ll} d -\!\prec \sim b, c & b -\!\prec \sim d, a \\ \sim b -\!\prec a & \sim d -\!\prec c \\ a & c \end{array} \right\}$$

de donde se obtienen las siguentes estructuras de argumento:

$$\langle \mathcal{A}_2,\mathrm{d}\rangle = \langle \{(d -\!\prec \sim b, c), (\sim b -\!\prec a)\}, d\rangle$$
$$\langle \mathcal{A}_1,\mathrm{b}\rangle = \langle \{(b -\!\prec \sim d, a), (\sim d -\!\prec c)\}, b\rangle$$

Utilizando al criterio de especificidad para comparar los argumentos, se tiene que (ver Figura 2.3), $\langle \mathcal{A}_2,\mathrm{d}\rangle$ derrota a $\langle \mathcal{A}_1,\mathrm{b}\rangle$, en el subargumento $\langle \mathcal{B},\sim\mathrm{d}\rangle$, pero además $\langle \mathcal{A}_1,\mathrm{b}\rangle$ derrota a $\langle \mathcal{A}_2,\mathrm{d}\rangle$ en el subargumento $\langle \mathcal{A},\sim\mathrm{b}\rangle$.

El árbol de dialéctica para $\langle \mathcal{A}_2,\mathrm{d}\rangle$ posee una única línea de argumentación aceptable: [$\langle \mathcal{A}_2,\mathrm{d}\rangle$, $\langle \mathcal{A}_1,\mathrm{b}\rangle$] (el ciclo es evitado, al no permitir reintroducir el primer argumento), y entonces el nodo raíz es marcado como "D" (derrotado). Por lo tanto, se cumple que $\neg Bd$ y $\neg B{\sim}d$ (no cree en d ni en $\sim d$), y la respuesta a la consulta d es INDECISO.

Análogamente, el nodo raíz para el árbol de dialéctica de $\langle \mathcal{A}_1,\mathrm{b}\rangle$ es marcado como "D", y la respuesta para b es también INDECISO.

Ejemplo 3.9 (Importancia de la concordancia)
Como se vió en varios de los ejemplos anteriores, en DeLP es posible que a partir de un programa no se pueda creer en un literal, ni en su negación, generando una situación de indecisión. Sin embargo, en la observación 2.15 se mostró que en este formalismo, no se puede creer en "$\sim p$" y "p" al mismo tiempo, ya que esto lleva directamente a un absurdo, porque si "$\sim p$" esta garantizado, entonces es un derrotador para "p" y viceversa.

El siguiente programa, muestra que si DeLP no tuviera la condición de concordancia en las líneas de argumentación, entonces podría ocurrir que "p" y "$\sim p$" estén garantizados.

$$\mathcal{P}_{3.9} = \left\{ \begin{array}{l} p -\!\prec d, h \\ \sim p -\!\prec d, h, \sim a \\ \sim a -\!\prec e \\ a -\!\prec e, f \\ f -\!\prec d \\ \sim p \leftarrow f \\ d \\ h \\ e \end{array} \right\}$$

donde pueden construirse las siguientes estructuras de argumento:

$$\langle \mathcal{B}, p \rangle = \langle \{p -\!\prec d, h\}, p \rangle$$
$$\langle \mathcal{C}, \sim p \rangle = \langle \{(\sim p -\!\prec d, h, \sim a), (\sim a -\!\prec e)\}, \sim p \rangle$$
$$\langle \mathcal{A}, a \rangle = \langle \{(a -\!\prec e, f), (f -\!\prec d)\}, a \rangle$$

Como $\langle \mathcal{B}, p \rangle$ es derrotado por $\langle \mathcal{C}, \sim p \rangle$, que a su vez es derrotado internamente por $\langle \mathcal{A}, a \rangle$, se podría pensar que la línea de argumentación que se forma es: $[\langle \mathcal{B}, p \rangle, \langle \mathcal{C}, \sim p \rangle, \langle \mathcal{A}, a \rangle]$ y por lo tanto "p" estaría garantizado, i. e., Bp.

Obsérvese además que $\langle \mathcal{C}, \sim p \rangle$ es derrotado por $\langle \mathcal{A}, a \rangle$, que a su vez es derrotado por $\langle \mathcal{B}, p \rangle$ (usando la regla estricta "$\sim p \leftarrow f$"), con lo cual parecería que se podría construir la línea de argumentación que se forma es: $[\langle \mathcal{C}, \sim p \rangle, \langle \mathcal{A}, a \rangle, \langle \mathcal{B}, p \rangle]$ y el literal "$\sim p$" estaría garantizado, i. e., $B{\sim}p$.

Afortunadamente en DeLP esto no puede ocurrir, ya que las dos líneas anteriores no son aceptables. Observe que $\langle \mathcal{B}, p \rangle$ y $\langle \mathcal{A}, a \rangle$ no son concordantes, y que además $\langle \mathcal{B}, p \rangle$ y $\langle \mathcal{C}, \sim p \rangle$ tampoco son concordantes.

Ejemplo 3.10 (Acciones)
El programa $\mathcal{P}_{3.10}$ que sigue a continuación muestra como puede representarse información sobre el mercado de valores en DeLP.[1]

$$\mathcal{P}_{3.10} = \left\{ \begin{array}{l} buy(T) -\!\prec good\_price(T) \\ \sim buy(T) -\!\prec good\_price(T), price\_decreasing(T) \\ price\_decreasing(T) -\!\prec lastsale(T,P), previousclose(T,Pre), P < Pre \\ good\_price(acme) -\!\prec lastsale(acme,P), P < 100 \\ good\_price(ti) -\!\prec lastsale(ti,P), P < 20 \\ sell\_stock(T) -\!\prec advice(T,sell) \\ \sim sell\_stock(T) -\!\prec advice(T,sell), speculate(T) \\ speculate(T) -\!\prec negative\_profit(T) \\ \sim speculate(T) -\!\prec negative\_profit(T), too\_risky(T) \\ too\_risky(T) -\!\prec market(down) \\ too\_risky(T) -\!\prec breaking(T) \\ negative\_profit(T) \leftarrow pricepaid(T,P), lastsale(T,L), (L < P) \\ lastsale(acme,90) \\ previousclose(acme,80) \\ lastsale(ti,10) \\ previousclose(ti,30) \\ pricepaid(acme,101) \\ advice(acme,sell) \\ market(down) \end{array} \right\}$$

Este ejemplo será analizado en detalle en el capítulo 8.

[1] La implementación de DeLP proveerá las operaciones aritmético-lógicas básicas como $<$, $+$, etc. (ver sección 4.5.3).

Capítulo 4

Semántica Operacional

En este capítulo se presenta la semántica operacional de DeLP. La idea de una semántica operacional [Mey90, Ten91, Hen90, NN90], es expresar el significado de un lenguaje, definiendo un mecanismo que permita determinar el efecto de todo programa escrito en ese lenguaje. Esta descripción operacional, consiste de un dispositivo formal, o "autómata intérprete", capaz de ejecutar formalmente cualquier programa. El autómata intérprete puede variar desde una descripción matemática abstracta, hasta una descripción concreta mediante otro lenguaje de programación (también llamada "máquina virtual").

La ventaja de disponer una semántica operacional [Mey90], es tener una descripción concreta del lenguaje. Esta semántica usualmente está orientada a programadores, ya que las descripciones operacionales son muy cercanas a un programa real. Como ventaja adicional, a partir de la semántica operacional, es muy fácil construir un intérprete para ejecutar programas. Esto hace que la semántica operacional sea una herramienta atractiva para verificar nuevos lenguajes mucho antes que los compiladores para ese lenguaje sean escritos. Si se desea describir una semántica operacional en forma concreta, el lenguaje para realizar las descripciones debe ser lo más simple posible, y el que mejor se adapte al proceso a ser descripto.

El lenguaje elegido para realizar esta descripción operacional será PROLOG, debido a su cercanía conceptual con DeLP, y por tratarse de un lenguaje bien conocido por la comunidad científica en argumentación rebatible. La descripción se realizará de manera incremental, especificando primero los conceptos básicos como la noción de *derivación rebatible* y *argumento*, mostrando luego la especificación de las relaciones entre argumen-

tos (*contra-argumentos y derrotadores*), describiendo finalmente conceptos más complejos como la construcción de árboles de dialéctica y líneas de argumentación aceptables.

A continuación se muestra nuevamente la especificación en PROLOG del proceso de dialéctica con poda (ver Figura 2.10). Recordemos que un literal está garantizado, cuando el argumento que lo sustenta es la raíz de un árbol de dialéctica, y dicha raíz está marcada como un nodo "U".

```
warrant(H):-          find_argument(H,A),
                      \+ defeated(A,[A]).
defeated(A,Line):-    find_defeater(A,D),
                      acceptable(D,Line),
                      \+ defeated(D,[D|Line]).
```

Figura 4.1: Especificación del proceso de dialéctica con poda

Siguiendo esta especificación, un literal H está garantizado si existe una estructura de argumento A para H que no esté derrotada. Para verificar si una estructura A está derrotada, se busca un derrotador D que permita continuar con una línea de argumentación aceptable, y si existe, se verifica (recursivamente) que el derrotador D en cuestión no esté derrotado. Obsérvese que la aceptabilidad de la línea de argumentación se verifica dinámicamente, a medida que se agregan argumentos a la línea. El árbol de dialéctica es construído en profundidad (depth-first), podando las ramas que no es necesario considerar (ver Ejemplo 2.17).

En las secciones siguientes, se detallarán las especificaciones de los predicados `find_argument(H,A)`, `find_defeater(A,D)`, y `acceptable(D,Line)`. Pero en primer lugar, se especificará como obtener una derivación rebatible, que constituye la noción base para obtener estructuras de argumento.

4.1 Derivaciones Rebatibles

La definición 2.12 establece cuando un literal L posee una derivación rebatible a partir de un programa $\mathcal{P}$. A continuación se introduce una versión equivalente de dicha definición, donde la secuencia de literales es obtenida partiendo del literal L y por encadenamiento hacia atrás de las reglas de programa. Esta nueva definición, será utilizada como base

para la especificación del proceso para obtener una derivación rebatible que se incluye en esta sección.

Definición 4.1 (versión 'backward' de la Definición 2.12)
Sea $\mathcal{P} = (\Pi, \Delta)$ *un programa y* L *un literal fijo. Una derivación rebatible para* L *a partir de* $\mathcal{P}$ *consiste de una secuencia finita de literales, obtenidos de la siguiente manera:*

(a) si L *es un hecho en* Π*, la secuencia* S *que contiene a* L *es la derivación rebatible para* L*.*

(b) si L *no es un hecho en* Π*, y existe en* $\mathcal{P}$ *una regla* R *con cabeza* L *y cuerpo* $B_1, B_2, \ldots B_k$*, tal que, para cada literal* B_i *del cuerpo existe una derivación rebatible* S_i*, entonces la secuencia* S *definida por* $S_1, S_2, \ldots, S_k, L$ *es una derivación rebatible para* L*.*

Para la especificación operacional se asume que existen los siguientes operadores para representar reglas estrictas, rebatibles y la negación fuerte. `op(1101,xfx,-<)`, `op(1101,xfx,<-)`, y `op(191,fx,∼)`. El predicado `op/3` de PROLOG, define entonces la precedencia y la asociatividad de cada operador. De esta forma, un hecho h será representado entonces como "`h <- true`".

La figura 4.2 muestra la especificación del proceso para obtener una derivación rebatible "`D`" para un literal "`Q`". La especificación es una versión del clásico "meta-intérprete" de PROLOG puro, modificada para que utilice reglas estrictas y rebatibles. Cabe destacar, que a pesar que una derivación rebatible es una secuencia de literales, la especificación siguiente acumula las reglas utilizadas en la lista que aparece como tercer parámetro de `defeasible_derivation/3`. Los literales correspondientes pueden obtenerse luego, extrayendo las cabezas de las reglas acumuladas en la lista "`D`".

La especificación de la Figura 4.2, sigue la implementación sugerida por la definición 4.1, y consta de cuatro partes:

1. existe una derivación rebatible del átomo `true`, que es la secuencia vacía,

2. existe la derivación de una conjunción, si existe para el primer elemento `A`, y existe para el resto de la conjunción `B`,

```
find_defeasible_derivation(Q,D):-defeasible_derivation(Q,[],D).

defeasible_derivation(true,D,D) :-!.
defeasible_derivation((A,B),CurrentDerivation,ABDerivation) :-!,
           defeasible_derivation(A,CurrentDerivation,ADerivation),
           defeasible_derivation(B,ADerivation,ABDerivation).
defeasible_derivation(H,CurrentDerivation,[Rule|BodyDerivation]) :-
           (H <- B),  Rule = (H <- B)
           defeasible_derivation(B,CurrentDerivation,BodyDerivation).
defeasible_derivation(H,CurrentDerivation,[Rule|BodyDerivation]) :-
           (H -< B),  Rule = (H -< B)
           defeasible_derivation(B,CurrentDerivation,BodyDerivation).
```

Figura 4.2: Especificación de Derivación Rebatible

3. existe la derivación de un literal H que es cabeza de una regla estricta, si existe la derivación del cuerpo de la regla, y finalmente,

4. existe la derivación de un literal H que es cabeza de una regla rebatible, si existe la derivación del cuerpo de la regla.

Es importante destacar, que aunque las reglas estrictas y rebatibles son seleccionadas en el orden que están especificadas en el programa lógico rebatible, las reglas estrictas son elegidas antes que las rebatibles. Con lo cuál, si existe una derivación estricta para un literal, esta será encontrada primero.

4.2 Construcción de argumentos

Una estructura de argumento $\langle \mathcal{A}, h \rangle$, se construirá buscando una derivación rebatible de la forma descripta en la sección anterior, pero verificando que el conjunto de literales T de la derivación no sea contradictorio con el conjunto Π de reglas estrictas y hechos del programa.

La derivación y la verificación de contradicciones se realizará simultáneamente, siguiendo la siguiente idea: cada vez que se obtiene la derivación del cuerpo de una regla R que se utiliza en la derivación (`BodyDerivation` en la Figura 4.2), se verifica si la cabeza H de la regla R es contradictoria con el conjunto Π. Una cabeza de regla H será contradictoria con el conjunto Π, si su complemento $\overline{H}$ puede derivarse rebatiblemente a partir de Π, y el conjunto T de los literales acumulados en la derivación hasta ese momento. Este proceso será detallado más adelante.

Durante la construcción del argumento, si una cabeza H de una regla es certificada como no contradictoria, entonces el literal H pasará a formar parte de la secuencia T de la derivación rebatible, y además H será considerado como un *hecho temporario*. Se lo llama así, porque mientras se está construyendo un argumento puede utilizárselo como un hecho, pero desaparece una vez que termina este proceso. Los hechos temporarios también serán utilizados para evitar la re-derivación de literales para los cuales ya se obtuvo una derivación.

La Figura 4.3 muestra la especificación del proceso de obtención de una estructura de argumento "`A`" para un literal "`Q`". El predicado `argument/3` es muy similar al predicado `defeasible_derivation/3` de la Figura 4.2, con la diferencia que antes de buscar una regla, verifica si el literal no fue derivado antes y existe como hecho temporario.

Los hechos temporarios corresponden a las cabezas de las reglas almacenadas en la lista que representa la derivación en curso, y esto evita tener que incluir una estructura de datos para almacenarlos. Cuando se está construyendo la derivación rebatible de un literal "`H`" y se descubre que `H` es un hecho temporario, esto significa que la derivación rebatible de `H` ya fue encontrada con anterioridad, y no es necesario calcularla ya que está almacenada en la lista. Esto evita buscar derivaciones innecesariamente, y además mantiene la lista sin reglas repetidas.

Puede verse además en la especificación, que una vez que un literal "`H`" ha sido probado, se utiliza el predicado "`verify_non_contradictory/2`" para verificar que no sea contradictorio. Este predicado es especificado en la Figura 4.4 que se incluye más adelante.

Cabe destacar que los verdaderos hechos de un programa no necesitan transformarse en un hecho temporario, ya que de por si son hechos. En el caso de la especificación de la figura 4.3, los hechos quedan almacenados en la lista que guarda la secuencia de la derivación, y por ende son recuperados como hechos temporarios. El proceso podría

```
find_argument(Q,A):- argument(Q,[],A).
argument(true,Derivation,Derivation) :-!.
argument((A,B),CurrentDerivation,ABDerivation) :-!,
         argument(A,CurrentDerivation,ADerivation),
         argument(B,ADerivation,ABDerivation).
argument(true,Derivation,Derivation) :- temporary_fact(H,Derivation),!.
argument(H,CurrentDerivation,Argument) :-
      (
         (H <- B), Rule = (H <- B)           %   there is a strict rule
       ;                                     % or
         (H -< B), Rule = (H -< B)           %   there is a defeasible rule
      ),
       argument(B,CurrentDerivation,BodyDerivation),
       Argument = [Rule|BodyDerivation],
       verify_non_contradictory(H,Argument).
temporary_fact(H,[( H <- _ )|_]):-!.  % H is a temporary fact if H is a Head
temporary_fact(H,[( H -< _ )|_]):-!.  % of a rule in the current derivation
temporary_fact(H,[_|Rest]):-temporary_fact(H,Rest).
```

Figura 4.3: Especificación de la construcción de estructuras de argumentos

optimizarse, si se busca un hecho en el programa (`H <- true`), antes de buscar un hecho temporario en la lista.

Ejemplo 4.1 Considérese el siguiente programa lógico rebatible:

$escape\_flying(X) \prec fly(X), scared(X)$ $\quad$ $scared(tina)$

$fly(X) \prec bird(X)$ $\quad$ $bird(X) \leftarrow penguin(X)$

$scared(tweety)$ $\quad$ $bird(X) \leftarrow hen(X)$

$hen(tina)$ $\quad$ $\sim fly(X) \leftarrow penguin(X)$

$penguin(tweety)$ $\quad$ $penguin(chilly)$

A partir del programa anterior, el literal "$escape\_flying(tina)$" tiene la siguiente derivación rebatible:

$$\left\{ \begin{array}{l} escape\_flying(tina) -\!\prec fly(tina), scared(tina) \\ fly(tina) -\!\prec bird(tina) \\ bird(tina) \leftarrow hen(tina) \\ hen(tina) \\ scared(tina) \end{array} \right\}$$

A fin de verificar que la derivación rebatible no sea contradictoria junto con el conjunto Π, cada vez que una regla de cabeza H es utilizada en la derivación, se verificará que el conjunto $H \cup \Pi \cup T$ no sea contradictorio, donde T es el conjunto de hechos temporarios obtenidos hasta el momento.

Al comienzo de una derivación, T es vacío, y cada vez que una regla es utilizada y verificada que no sea contradictoria, su cabeza H es agregada a T. En este ejemplo, durante la derivación del literal "$escape\_flying(tina)$", la primer cabeza de regla que es verificada es "$hen(tina)$", como no es contradictoria con el conjunto $\Pi \cup \emptyset$, se la incorpora como un hecho temporario. La siguiente tabla muestra el orden en el que los nuevos hechos temporarios son incorporados a T:

Cabeza de regla	conjunto T de hechos temporarios	¿contradictorio?
$hen(tina)$	$\emptyset$	no
$bird(tina)$	$\{\ hen(tina)\ \}$	no
$fly(tina)$	$\{hen(tina), bird(tina)\ \}$	no
$escape\_flying(tina)$	$\{\ hen(tina), bird(tina), fly(tina)\ \}$	no

Como ninguna regla genera un hecho temporario contradictorio con $\Pi \cup T$, entonces la derivación rebatible obtenida es no contradictoria.

Sin embargo, como se muestra a continuación, la derivación rebatible de "$escape\_flying(tweety)$" es contradictoria:

$$\left\{ \begin{array}{l} escape\_flying(tweety) -\!\prec fly(tweety), scared(tweety) \\ fly(tweety) -\!\prec bird(tweety) \\ bird(tweety) \leftarrow penguin(tweety) \\ scared(tweety) \\ penguin(tweety) \end{array} \right\}$$

La siguiente tabla muestra la traza con el orden en que los hechos temporarios fueron generados hasta encontrar la contradicción. Esto es, cuando se verifica al literal $fly(tweety)$, ya que a partir de $\{penguin(tweety), bird(tweety)\ \}$ y el conjunto Π, se puede derivar $\sim fly(tweety)$.

Cabeza de regla	conjunto T de hechos tempoarios	¿contradictorio?
$penguin(tweety)$	$\emptyset$	no
$bird(tweety)$	$\{\ penguin(tweety)\ \}$	no
$fly(tweety)$	$\{penguin(tweety), bird(tweety)\ \}$	yes

Por la tanto la derivación rebatible de "$escape\_flying(tweety)$" no permite la generación de un argumento.

Lamentablemente el proceso descripto antes no es suficiente para detectar contradicciones. Es decir, dado un literal g, si existe una derivación para $\overline{g}$ a partir de $\Pi \cup T$ entonces hay una contradicción. Pero como muestra el siguiente ejemplo, puede no haber una derivación para $\overline{g}$ y sin embargo existir una contradicción.

Ejemplo 4.2 Considérese el siguiente programa:

$$\begin{array}{ll} a -\!\prec b, c & \sim p \leftarrow b \\ b & p \leftarrow a \\ c & \end{array}$$

el conjunto $\mathcal{A} = \{\ a -\!\prec b, c\ \}$ no puede ser un argumento, ya que a partir de $\Pi \cup \mathcal{A}$ pueden derivarse los literales "p" y "$\sim p$". Sin embargo, si se utiliza el proceso sugerido más arriba, la contradicción no sería encontrada, ya que al utilizar la regla "$a -\!\prec b, c$" no se encuentra una derivación de "$\sim a$" (el complemento de la cabeza).

El ejemplo 4.2 muestra que para detectar una contradicción, no alcanza con intentar derivar al complemento de la cabeza de una regla utilizando Π y los hechos temporarios. Una alternativa costosa es calcular la clausura de $\Pi \cup \mathcal{A}$. Sin embargo, como se muestra a continuación, existe una forma de detectar las contradicciones utilizando encadenamiento hacia atrás y sin necesidad de generar la clausura de $\Pi \cup \mathcal{A}$. Para esto, durante la búsqueda de contradicciones se permitirá utilizar a las reglas estrictas en forma "invertida", como muestra la siguiente definición.

Definición 4.2 (Reglas estrictas invertidas) *Sea $\mathcal{P}$ un programa lógico rebatible, dada una regla estricta R:*

$$p \leftarrow a_1, \ldots, a_i, \ldots, a_n.$$

una regla estricta invertida *I se obtiene a partir de R, colocando en la cabeza de I al complemento de algún literal del cuerpo de R, y al complemento de la cabeza de R en el cuerpo de I:*

$$\overline{a_i} \hookleftarrow a_1, \ldots, \overline{p}, \ldots, a_n.$$

El símbolo " $\hookleftarrow$ " distingue a las reglas invertidas de las reglas estrictas. Es importante notar, que por cada regla estricta R, existen tantas reglas invertidas como literales hay en el cuerpo de R. Es importante recalcar, que las reglas invertidas no pueden utilizarse para obtener derivaciones, ya que de hacerlo se estaría asumiendo la contrapositiva de las reglas estrictas.

El lema 4.1 muestra que ahora utilizando las reglas invertidas es posible verificar si un literal es contradictorio o no, utilizando un proceso de razonamiento hacia atrás.

Lema 4.1 *Sea $\mathcal{P}$=(Π, Δ) un programa lógico rebatible y h un literal. Sea Π un conjunto no contradictorio de hechos y reglas estrictas, T el conjunto de hechos temporarios, e I el conjunto de reglas invertidas de $\mathcal{P}$. El conjunto $\Pi \cup T \cup \{h\}$ es contradictorio si y sólo si existe una derivación estricta para $\overline{h}$ a partir de $\Pi \cup T \cup I \cup \{h\}$.*
Demostración:
($\Rightarrow$) Supóngase que el conjunto $\Pi \cup T \cup \{h\}$ es contradictorio, se desea probar entonces que existe una derivación para $\overline{h}$ a partir del conjunto $\Pi \cup T \cup I \cup \{h\}$.
Si el conjunto $\Pi \cup T \cup \{h\}$ es contradictorio, entonces existe una derivación rebatible para un par de literales complementarios p y $\overline{p}$. En el caso particular que se cumpla $p = h$, entonces $\overline{p} = \overline{h}$, y el lema en esta dirección queda probado.
Veamos ahora el caso en que $p \neq h$. El conjunto Π es no contradictorio por definición, y el conjunto $\Pi \cup T$ tampoco es contradictorio. Por lo tanto a partir de $\Pi \cup T$, no es posible obtener derivaciones para p y para $\overline{p}$. Sin embargo, puede ocurrir que:
(a) a partir de $\Pi \cup T$ no existe derivación para $\overline{p}$ y tampoco para p.
(b) a partir de $\Pi \cup T$ existe derivación para $\overline{p}$ pero no para p.
(c) a partir de $\Pi \cup T$ existe derivación para p pero no para $\overline{p}$.
Caso (a): Dado que existe una derivación rebatible de p a partir del conjunto $\Pi \cup T \cup \{h\}$, pero no a partir de $\Pi \cup T$, entonces el literal h es necesario para la derivación de p. Podría

ocurrir por ejemplo que Π contenga una regla estricta "$p \leftarrow h, D$", y D es un conjunto de literales que poseen una derivación rebatible a partir de $\Pi \cup T \cup \{h\}$. Si tal regla estricta existe, entonces con ella puede obtenerse la siguiente regla invertida: "$\overline{h} \hookleftarrow \overline{p}, D$". Dado que D y $\overline{p}$ pueden derivarse rebatiblemente a partir de $\Pi \cup T \cup \{h\}$, entonces la regla invertida permite obtener una derivación para $\overline{h}$ a partir de $\Pi \cup T \cup I \cup \{h\}$.
Sin embargo, en general no existirá una sola regla estricta como la indicada antes, sino que habrá una secuencia de reglas estrictas como la siguiente:

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

donde los B_i son conjuntos de literales que poseen una derivación rebatible a partir de $\Pi \cup T \cup \{h\}$.
Generando las reglas invertidas apropiadas, se obtiene la siguiente secuencia:

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

dado que tanto $\overline{p}$, como los literales de los conjuntos B_i se derivan a partir de $\Pi \cup T \cup \{h\}$, entonces utilizando las reglas invertidas de la secuencia anterior existe una derivación para $\overline{h}$ a partir de $\Pi \cup T \cup I \cup \{h\}$.

Caso (b): $\overline{p}$ se deriva a partir de $\Pi \cup T$, pero p no se deriva de $\Pi \cup T$. La demostración es idéntica que para el caso (a).

Caso (c): p se deriva a partir de $\Pi \cup T$, pero $\overline{p}$ no se deriva de $\Pi \cup T$. La demostración es similar al caso (a) pero intercambiando p por $\overline{p}$, y viceversa.

($\Leftarrow$)
Supongamos ahora que existe una derivación de $\overline{h}$ a partir del conjunto $\Pi \cup T \cup I \cup \{h\}$. Debemos probar que el conjunto $\Pi \cup T \cup \{h\}$ es contradictorio. Esto es, existe una derivación rebatible para un par de literales complementarios p y $\overline{p}$, a partir del conjunto $\Pi \cup T \cup \{h\}$.

Como caso partircular, si $\overline{h}$ puede derivarse a partir del conjunto $\Pi \cup T \cup \{h\}$ entonces el lema queda probado. En general, será necesario utilizar una secuencia de reglas invertidas

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

donde el literal $\overline{c_n}$ y los literales de los conjuntos D_i pueden derivarse a partir del conjunto $\Pi \cup T \cup \{h\}$. Luego, usando las reglas estrictas que dieron origen a estas reglas invertidas, puede obtenerse una derivación para el literal c_n:

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

Por lo tanto, a partir del conjunto $\Pi \cup T \cup \{h\}$, es posible derivar rebatiblemente a c_n y $\overline{c_n}$, con lo cual se muestra que el conjunto es contradictorio. □

Definición 4.3 (Restricciones de uso de reglas invertidas) *Dado que una regla invertida no es parte de una derivación rebatible, y se usan sólo para buscar contradicciones, se las puede utilizar únicamente bajo las siguientes condiciones que surgen del lema 4.1:*

1. *Las reglas invertidas sólo pueden utilizarse al comenzar una búsqueda de una posible contradicción.*

2. *Una vez que una regla del programa es usada, ya no se podrán utilizar más reglas invertidas en lo que resta de la búsqueda.*

3. *Si una regla "$p \leftarrow a_1, a_2, \dots, a_n$" es invertida para ser usada como "$\overline{a_1} \hookleftarrow \overline{p}, a_2, \dots, a_n$". entonces los literales del cuerpo de la regla invertida (excluyendo a $\overline{p}$) "$a_2, \dots, a_n$" deberán derivarse únicamente utilizando reglas estrictas o hechos temporarios, esto es sin utilizar reglas invertidas.*

```
verify_non_contradictory(H,TF):-complement(H,G),
                                \+ find_contradiction(G,TF,inverted).
complement(~ A, A):- A \= ~ _ ,!.
complement(A, ~ A) :- A \= ~ _.
find_contradiction(true,_,_):- !.
find_contradiction((A,B),TF,I):- !, find_contradiction(A,TF,I),
                                 find_contradiction(B,TF,I).
find_contradiction(G,TF,_):- temporary_fact(G,TF), !.
find_contradiction(G,TF,_):- (G <- Body), find_contradiction(Body,TF,direct).
find_contradiction(G,TF,inverted):- find_inverted(G,I,Body),
                                find_contradiction(I,TF,inverted),
                                find_contradiction(Body,TF,direct).
% inverted strict rules are obtained scanning the body of regular rules.
find_inverted(G,I,N):-(Head <- Body),    % picks a strict rule
                       complement(G,C),   % C is the complement of G
                       new_body(C,Body,N),% if C is in Body then N=(Body-C)
                       complement(Head,I).% I is the complement of the head
new_body(G,(G,B),B):-!. %given a goal G and a Body, returns Body without G.
new_body(G,(Q,B),(Q,B1)):- !, new_body(G,B,B1).
new_body(G,G,true).
```

Figura 4.4: Verificación de derivaciones contradictorias

La Figura 4.4 muestra la especificación de como verificar si un literal es contradictorio con la derivación en curso. El predicado `verify_non_contradictory/2`, tiene éxito cuando no existe una derivación estricta para el complemento de H. El predicado `strict_derivation/3` es el encargado de buscar una derivación estricta que use, si es necesario, hechos temporarios o reglas invertidas que satisfagan las condiciones de la definición 4.3. Obsérvese, que en lugar de generar las reglas invertidas explícitamente, se las genera implícitamente utilizando las reglas estrictas. A fin de garantizar las condiciones de la definición 4.3, las reglas invertidas pueden utilizarse sólo cuando el tercer parámetro de `strict_derivation/3` es "`inverted`", una vez que este parámetro es puesto en "`direct`" ya no pueden utilizarse más.

4.3 Especificación del proceso dialéctico

La Figura 4.5 muestra la especificación del proceso que construye un árbol de dialéctica para determinar si un literal `Q` está garantizado (warrant). Para controlar que las líneas de argumentación sean aceptables, se lleva una lista donde se van acumulando las estructuras de argumento que corresponden a la línea de argumentación en curso. Dicha lista se inicializa en el predicado `warrant/2` con "`sup(A,Q)`", indicando que se ha encontrado un argumento `A` de soporte para `Q`. Los argumentos de interferencia será agregados en la lista utilizando la forma "`int(B,H)`", como se explicará más adelante en la sección 4.3.2.

```
warrant(Q,A):-                                  % Q is a warranted literal
  find_argument(Q,A,[]),                        % if A is an argument for Q
  \+ defeated(A,[sup(A,Q)]).                    % and A is not defeated

defeated(A,ArgLine):-                           % A is defeated
  find_defeater(A,arg(D,G),ArgLine),            % if there is a defeater D
  acceptable(arg(D,G),ArgLine,NewArgLine),% acceptable within the line
  \+ defeated(D,NewArgLine).                    % and D is not defeated

find_defeater(A,C,ArgLine):-                    %C is a defeater (proper or blocking)
  find_counterargument(A,C,SubA,ArgLine), %if C counterargues A attacking SubA
  \+ better(SubA,C).                            %and SubA is not better than C
```

Figura 4.5: Construcción de un árbol de dialéctica con poda

El predicado `find_defeater/2` llama a `find_counterargument/3` que busca una estructua de argumento `C` que contra-argumente al argumento `A`, con sub-argumento de desacuerdo `SubA`, y luego verifica que `SubA` no sea mejor que `C`. Esto es, `find_defeater/2` devuelve un derrotador sin distinguir si es propio o de bloqueo. A continuación se mostrará como se obtienen los contra-argumentos y como se controla que las líneas de argumentación sean aceptables.

4.3.1 Búsqueda de contra-argumentos

Para obtener un contra-argumento de una estructura de argumento $\langle \mathcal{A}, h \rangle$, se debe construir un argumento $\langle \mathcal{B}, q \rangle$, tal que el literal q esté en desacuerdo con algún literal p de

$\langle \mathcal{A}, h \rangle$ (donde p es el punto de contra-argumentación). Sea T el conjunto de hechos temporarios generados durante la construcción de $\langle \mathcal{A}, h \rangle$. Cada elemento de T es un potencial punto de contra-argumentación. Por ejemplo, dado el programa:

$$a \prec b$$
$$\sim a \prec c$$
$$b$$
$$c$$

la estructura de argumento $\langle \mathcal{A}, a \rangle = \langle \{a \prec b\}, a \rangle$ tiene el conjunto de hechos temporarios $T = \{a\}$, y la estructura $\langle \mathcal{B}, \sim a \rangle = \langle \{\sim a \prec c\}, \sim a \rangle$, es un contra-argumento en el punto a.

Por lo tanto, para obtener contra-argumentos para una estructura $\langle \mathcal{A}, h \rangle$, primero se buscarán argumentos para el complemento de cada elemento de T. Pero lamentablemente, como muestra el siguiente ejemplo, el conjunto T no es suficiente, ya que pueden existir contra-argumentos para literales que no pertenezcan a T.

Ejemplo 4.3 Dado el programa

$\sim c \leftarrow q, r$	$h \prec q, r$
a	$q \prec a$
b	$r \prec b$
$c \leftarrow p$	$p \prec a, b$

La estructura de argumento $\langle \mathcal{A}, h \rangle = \langle \{(h \prec q, r), (q \prec a), (r \prec b)\}, h \rangle$, tiene el conjunto de hechos temporarios $T = \{ r, q, h\}$. Sin embargo, el argumento $\langle \{p \prec a, b\}, c \rangle$ es un contra-argumento para $\langle \mathcal{A}, h \rangle$, donde el subargumento de desacuerdo es $\langle \{(q \prec a), (r \prec b)\}, \sim c \rangle$. Por lo tanto, el punto de contra-argumentación no es el complemento de un literal de T.

El ejemplo 4.3 muestra que pueden existir puntos de contra-argumentación que no están en T. El problema es análogo al que se presentó con la verificación de contradicciones, y tiene una solución similar: se utilizarán reglas invertidas con las restricciones establecidas por la definición 4.3.

```
find_counterargument(A,C,SubA):-        % C counterargues A in SubA
     pointsOfAttack(A,Points),           % if either:
   ( attackInnerPoint(Points,A,C,SubA)  % attacks an inner literal in A
   ; attackOuterPoint(Points,A,C,SubA)).% or attacks a literal out of A
pointsOfAttack([],[]).  % returns set of conseq. of rules that are no facts.
pointsOfAttack([(P -< _)|Rules],[P|Ps]):-pointsOfAttack(Rules,Ps).
pointsOfAttack([(P <- Q)|Rules],[P|Ps]):-Q\=true,pointsOfAttack(Rules,Ps).
pointsOfAttack([(P <- true)|Rules],Ps):-pointsOfAttack(Rules,Ps).
attackInnerPoint([P|_],A,arg(CounterA,CompP),arg(SubA,P)):-
    complement(P,CompP),            % There is a counterargument for A in P
    find_argument(CompP,CounterA),  % if there is an argument for CompP
    find_sub-argument(P,A,SubA).    % and SubA is a disagreement subargument
attackInnerPoint([_|Ps],A,C,S):- attackInnerPoint(Ps,A,C,S).
attackOuterPoint([P|_],A,arg(CounterA,Qc),arg(SubA,Lit)):-
   complement(P,CompP),                    % There is a counterargument in P
   find_inverted(CompP,NewQ,RestOfBody), % if using a inverted rule, and a
   strict_derivation(RestOfBody,A,direct),% strict derivation for RestOfBody
   ( find_argument(NewQ,CounterA),         % then there is an argument for NewQ
     complement(NewQ,Lit),  Qc = NewQ      % that disagree in the point Lit
   ;                                       % or
     complement(NewQ,G),                % else continue inverting from G
     attackOuterPoint([G],A,arg(CounterA,Qc),arg(SubA,Lit)) ),
   find_sub-argument(Lit,A,SubA).
attackOuterPoint([_|Ps],A,C,S):- attackOuterPoint(Ps,A,C,S).
```

Figura 4.6: Construcción de contra-argumentos

Proposición 4.1 *Sea* $\mathcal{P} = (\Pi, \Delta)$ *un programa lógico rebatible, e* I *el conjunto de reglas invertidas que pueden obtenerse de* $\mathcal{P}$*. Sea* $\langle \mathcal{A}, h \rangle$ *una estructura de argumento obtenida a partir de* $\mathcal{P}$*, y* T *su conjunto de hechos temporarios. Una estructura de argumento* $\langle \mathcal{B}, q \rangle$ *es un contra-argumento para* $\langle \mathcal{A}, h \rangle$*, en un punto* $\overline{q} \notin T$*, si y solo si para algún literal* $l \in T$ *existe una derivación rebatible para* $\overline{l}$ *a partir de* $\Pi \cup T \cup \mathcal{B} \cup I$*.* □

Para ilustrar el uso de esta proposición, obsérvese que en el Ejemplo 4.3, con $T = \{h, r, q\}$, las reglas $(c \leftarrow p)$, y $(p \prec a, b)$, y la regla invertida $(\sim q \hookleftarrow c, r)$, es posible

obtener una derivación para el complemento de q (literal que pertenece a T). Eliminando las reglas invertidas, se obtiene el contra-argumento buscado $\langle\{p -\prec a, b\}, c\rangle$.

La Figura 4.6 muestra la especificación en PROLOG del proceso de construcción de contra-argumentos. El predicado `pointsOfAttack/3` es el que obtiene el conjunto T. Luego, `attackInnerPoint/4` es el encargado de buscar contra-argumentos en puntos de T, y `attackOuterPoint/4` en puntos que no están en T, utilizando si es necesario reglas invertidas. Una vez que el punto de contra-argumentación es fijado, el predicado `find_sub_argument/3` especificado en la Figura 4.7, extrae el subargumento de desacuerdo.

```
find_sub\_argument(true,Arg,[]):-!.
find_sub\_argument((not _),Arg,[]):-!.
find_sub\_argument((Q1,Q2),Arg,SubArg):-!, find_sub\_argument(Q1,Arg,S1),
                    find_sub\_argument(Q2,Arg,S2),append(S1,S2,SubArg).
find_sub\_argument(Q,Arg,[Rule|Cs]):-
                   (
                      member((Q -< B),Arg), Rule = (Q -< B)
                   ;
                     (Q <- B), Rule = (Q <- B)
                   ),
                   find_sub\_argument(B,Arg,Cs).
```

Figura 4.7: Extracción del sub-argumento de desacuerdo

4.3.2 Control de aceptabilidad de las líneas de argumentación

En la especificación de la Figura 4.5, el predicado `defeated/2`, una vez que encuentra un derrotador "`D`" para el último argumento de la línea, debe verificar que se puede incorporar a "`D`" a la línea de argumentación en curso, sin producir los problemas mencionados en la sección 2.8.7. Para esto, toda línea de argumentación $\Lambda = [\langle\mathcal{A}_1, h_1\rangle, \ldots, \langle\mathcal{A}_i, h_i\rangle, \ldots, \langle\mathcal{A}_n, h_n\rangle]$ debe ser aceptable (definición 2.28), es decir, debe verificar:

```
acceptable(arg(NewArg,G),ArgLine,NewArgLine):-
        \+ circular(arg(NewArg,G),ArgLine),    % if Line is not circular
      ( ArgLine = [sup(A,Q)|_],                % then, the new arg is "int"
        NewArgLine = [int(NewArg,G)|ArgLine],! % when the previous is "sup"
      ;                                        % or
        ArgLine = [int(A,Q)|_],           % the new one is "sup"
        NewArgLine = [sup(NewArg,G)|ArgLine],! % when the last one is "int"
      ),
       \+ blockblock(NewArgLine).              % block to blocked check
circular(arg(A,_),[int(B,_)|_]):-is_contained(A,B),!.
circular(arg(A,_),[sup(B,_)|_]):-is_contained(A,B),!.
circular(arg(A,Q),[_|RestOfLine]):-circular(arg(A,Q),RestOfLine),!.
is_contained([],_):-!.
is_contained([Clause|Cs],Arg):- Clause = d_rule(Q, B), member(d_rule(Q, B),Arg),
                                is_contained(Cs,Arg).
is_contained([Clause|Cs],Arg):- Clause \= d_rule(_, _), is_contained(Cs,Arg).
blockblock(NewLine):- NewLine = [Arg1,Arg2,Arg3|_], %has 3 or more arguments
                 ( Arg1 = sup(A1,G1), Arg2 = int(A2,G2), Arg3 = sup(A3,G3),!
                 ; Arg1 = int(A1,G1), Arg2 = sup(A2,G2), Arg3 = int(A3,G3),!
                 ),
                 defeater_type(arg(A2,G2),arg(A3,G3),T), T=blocking,
                 defeater_type(arg(A1,G1),arg(A2,G2),T1), T1=blocking.
defeater_type(D,A,proper):-better(D,A),!.
defeater_type(_,_,blocking).
```

Figura 4.8: Verificación de líneas de argumentación aceptables

1. (concordancia) El conjunto Λ_S, de estructuras de argumento de soporte de Λ es concordante, y el conjunto Λ_I, de estructuras de argumentos de interferencia de Λ es también concordante.
2. (circularidad) Ningún argumento $\langle \mathcal{A}_k, h_k \rangle$ de Λ es un sub-argumento de una estructura de argumento $\langle \mathcal{A}_i, h_i \rangle$ que aparece previamente en Λ ($i < k$).
3. (bloqueo) Para toda estructura $\langle \mathcal{A}_i, h_i \rangle$ de Λ tal que $\langle \mathcal{A}_i, h_i \rangle$ es un derrotador de bloqueo de $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$, si $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ existe, entonces $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ es un derrotador propio de $\langle \mathcal{A}_i, h_i \rangle$.

```
argument(true,D,D,ArgLine) :-!.
argument((A,B),CurrentDerivation,ABDerivation,ArgLine) :-!,
          argument(A,CurrentDerivation,ADerivation,ArgLine),
          argument(B,ADerivation,ABDerivation,ArgLine).
argument(H,D,D,Line) :- temporary_fact(H,D), !.
argument(H,CurrentDerivation,Argument,ArgLine) :-
     ( (H <- B),  Rule = (H <- B)  ;  (H -< B),  Rule = (H -< B)  ),
     argument(B,CurrentDerivation,BodyDerivation,ArgLine),
     Argument = [Clause|BodyDerivation],
     verify_concordancy(H,Argument,Line).
verify_concordance(G,TF,ArgLine) :-
  ( ArgLine = [],  All_TF = TF              %If no Line, all temp. facts are TF
  ; ArgLine = [sup(_,_)|_],                 %else if the previous is sup then
    collect_int([int(TF,G)|ArgLine],All_TF) %   collect interference temp facts
  ; ArgLine = [int(_,_)|_],                 %else if the previous is int then
    collect_sup([sup(TF,G)|ArgLine],All_TF) %   collect supporting temp facts
  ), verify_non_contradictory(G,All_TF),!.  %  verification with All_TF
collect_int([],[]):-!.
collect_int([int(TF,_)|ArgLine],All_TF):-collect_int(ArgLine,RestTF),
                                         append(TF,RestTF,All_TF),!.
collect_int([sup(TF,_)|ArgLine],RestTF):-collect_int(ArgLine,RestTF),!.
collect_sup([],[]):-!.
collect_sup([sup(TF,_)|ArgLine],All_TF):-collect_sup(ArgLine,RestTF),
                                      append(TF,RestTF,All_TF),!.
collect_sup([int(TF,_)|ArgLine],RestTF):-collect_sup(ArgLine,RestTF),!.
```

Figura 4.9: Verificación de concordancia

En la Figura 4.8 se especifica como verificar que una línea no sea circular, y que un derrotador no bloquee a un derrotador de bloqueo. La verificación de la concordancia de los conjuntos de soporte y de interferencia se realizará al momento de generar un argumento y será explicado más adelante. La línea de argumentación estará representada con una lista con la siguiente estructura:

```
[sup(A1,h1),int(A2,h2),sup(A3,h3),int(A4,h4),...]
```

donde (`sup`) e (`int`) indican que la estructura de argumento es de soporte o de interferencia.

El predicado `acceptable/3` que se muestra a continuación, primero verifica que el nuevo argumento no genere una línea circular. Esto se realiza simplemente utilizando el predicado `circular(A,Line)`, que tendrá éxito si el conjunto de reglas rebatibles de "`A`" está contenido (`is_contained`) en alguno de las estructuras de argumentos de "`Line`". Si el argumento no produce una circularidad, entonces es agregado a la línea como `sup` o `int`, según corresponda. Finalmente, se verifica que un derrotador de bloqueo no sea usado para bloquear a uno de bloqueo. Para esto se utiliza el predicado `blockblock/1`, que compara los últimos 3 argumentos de la línea (si existen).

Para verificar la concordancia, el predicado `find_argument` lleva un parámetro más con la línea de argumentación en curso. El predicado `verify_concordance/3` (Figura 4.9), es una generalización de la búsqueda de contradicciones. Si el argumento que se está construyendo es de soporte, antes de generar un hecho temporario en la construcción del argumento, se verifica que no haya una contradicción con Π y los hechos temporarios de todos los argumentos de soporte de la línea en curso. Analogamente, si se construye un argumento de interferencia, se verifica con los hechos temporarios de los argumentos de interferencia.

4.4 Comparación de argumentos

En la especificación de la Figura 4.5 el predicado `find_defeater/3` utiliza al predicado `better/2` que comparar dos estructuras de argumentos, y debe indicar si el primer argumento es mejor que el segundo. Como se mostró en la sección 2.6 una alternativa para comparar argumentos es definir un órden entre reglas rebatibles del programa, mientras que la otra alternativa sugerida fué el criterio de especificidad generalizada (definición 2.20).

4.4.1 Usando prioridades sobre las reglas rebatibles

Incluimos a continuación nuevamente la definición 2.21, que indica como comparar argumentos utilizando prioridades entre reglas rebatibles

Definición 2.21 *Sea $\mathcal{P}$ un programa lógico rebatible y "$>$" un orden parcial definido explícitamente sobre las reglas rebatibles de $\mathcal{P}$. Sean $\langle \mathcal{A}_1, h_1 \rangle$ y $\langle \mathcal{A}_2, h_2 \rangle$ dos estructuras de argumento obtenidas a partir de $\mathcal{P}$. $\langle \mathcal{A}_1, h_1 \rangle$ será preferido sobre $\langle \mathcal{A}_2, h_2 \rangle$ cuando:*

1. *exista al menos una regla* $r_a \in \mathcal{A}_1$, *y una regla* $r_b \in \mathcal{A}_2$, *tal que* $r_a > r_b$, *y*
2. *no exista ninguna regla* $r'_b \in \mathcal{A}_2$, *y* $r'_a \in \mathcal{A}_1$, *tal que* $r'_b > r'_a$.

La Figura 4.10 muestra como especificar el predicado `better/2` utilizando la definición anterior. En la especificación se asume que el programa lógico rebatible incluye el orden parcial ">", representando con "`better_rule`(R_1, R_2)" a $R_1 > R_2$.

```
better(arg(A,_),arg(B,_)):-
          one_rule_better(A,B),    %there is a better rule in A
          \+ one_rule_better(B,A). %there is no better rule in B

one_rule_better([R1|_],L):- better_rule(R1,R2),member(R2,L),!.
one_rule_better([_|Rest],L):- one_rule_better(Rest,L).
```

Figura 4.10: Comparación de argumentos utilizando prioridades entre reglas

4.4.2 Especificidad generalizada

Para computar la especifidad generalizada puede utilizarse un resultado obtenido en [Gar97], que consiste en calcular los *conjuntos de activación* de cada estructura de argumento a compararse. A continuación incluimos las definiciones principales. El lector interesado puede encontrar más detalles en [Gar97].

Definición 4.4 (Completamiento) [Gar97]
Sea $\mathcal{P} = (\Theta, \Omega, \Delta)$ *un programa, y* $\langle \mathcal{A}, h \rangle$ *una estructura de argumento obtenida a partir de* $\mathcal{P}$. *Un completamiento de* $\mathcal{A}$, *denotado* $\mathcal{A}^c$, *es un conjunto de reglas estrictas y rebatibles,* $\mathcal{A}^c = \mathcal{A} \cup R$, *tal que* $R \subseteq \Pi$ *y existe una derivación rebatible de* h *a partir de* $\mathcal{A}^c \cup \Theta$.

Ejemplo 4.4
Sea $\mathcal{P}$=(Π, Δ), donde Π=$\{$ $(h \leftarrow a)$, $(b \leftarrow d)$, $(c \leftarrow d)$, (d), $(c)\}$, y Δ= $\{a \prec b, c\ \}$. Sea $\langle \mathcal{A}, h \rangle$= $\langle \{a \prec b, c\}, h \rangle$. Un completamiento de $\mathcal{A}$ es $\mathcal{A}^c$=$\{$ $(h \leftarrow a)$, $(a \prec b, c)$, $(b \leftarrow d)\}$.

Observación 4.1 *Dado un argumento $\mathcal{A}$ para h, no existe un único completamiento $\mathcal{A}^c$, ya que puede haber diferentes reglas en Π con las cuales construir $\mathcal{A}$. Por ejemplo, $\mathcal{A}^{c\prime}$=$\{(h \leftarrow a), (a \prec b, c), (b \leftarrow d), (c \leftarrow d)\}$ es otra forma de completar el argumento $\mathcal{A}$ del ejemplo anterior.*

Definición 4.5 (Conjunto de literales de $\mathcal{A}^c$) [Gar97]
Sea $\mathcal{A}^c$ un completamiento de $\mathcal{A}$, el conjunto de literales de $\mathcal{A}^c$ denotado $Lit(\mathcal{A}^c)$, es el conjunto de literales que aparecen en los antecedentes y consecuentes de toda regla de $\mathcal{A}^c$.

Definición 4.6 (Conjunto de activación) [Gar97]
Sea $\langle \mathcal{A}, h\rangle$ una estructura de argumento, $\mathcal{A}^c$ un completamiento, 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 existe una derivación rebatible de h a partir de $U \cup \mathcal{A}^c$, y además U es minimal con respecto a la inclusión de conjuntos (i. e., no existe $U' \subseteq U$ tal que exista una derivación rebatible de h a partir de $U' \cup \mathcal{A}^c$). Se denotará con Act-sets($\mathcal{A}^c$), al conjunto de todos los conjuntos de activación de $\mathcal{A}^c$.*

Definición 4.7 (Conjunto de activación no trivial) [Gar97]
Sea $\mathcal{P}$= (Θ, Ω, Δ), $\langle \mathcal{A}, h\rangle$ una estructura de argumento obtenida a partir de $\mathcal{P}$, y $\mathcal{A}^c$ un completamiento. 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 no existe una derivación rebatible de h a partir de $U \cup \Omega$. Se denotará con NTAct-sets($\mathcal{A}^c$) al conjunto de todos los conjuntos de activación no triviales de $\mathcal{A}^c$.*

Definición 4.8 (Especificidad) [Gar97]
Sea $\mathcal{P}$= (Θ, Ω, Δ) un programa, y $\langle \mathcal{A}, h\rangle$, $\langle \mathcal{B}, q\rangle$ dos estructuras de argumento obtenidas a partir de $\mathcal{P}$, $\mathcal{A} \neq \emptyset, \mathcal{B} \neq \emptyset$. Sean $\mathcal{A}^c$, $\mathcal{B}^c$ dos completamientos para $\langle \mathcal{A}, h\rangle$ y $\langle \mathcal{B}, q\rangle$ respectivamente. Se dirá que $\langle \mathcal{A}, h\rangle$ es estrictamente más específica *que $\langle \mathcal{B}, q\rangle$ si,*

1. *para todo conjunto $U \in$ NTAct-sets($\mathcal{A}^c$), existe una derivación rebatible de q a partir de $U \cup \Omega \cup \mathcal{B}^c$, y*
2. *existe un conjunto $U' \in$ NTAct-sets($\mathcal{B}^c$), tal que no existe una derivación rebatible de h a partir de $U' \cup \mathcal{A}^c$.*

4.5 Intérprete de Programas Lógicos Rebatibles

Como se dijo al comienzo del capítulo, a partir de la semántica operacional, es muy fácil construir un intérprete para ejecutar programas. Si se implementa un intérprete de programas lógicos rebatibles, siguiendo lo especificado en este capítulo, entonces existen ciertos detalles que deben tenerse en cuenta. En esta sección tratamos algunos de ellos.

4.5.1 Respuestas a una consulta

La figura 4.11 muestra la especificación del comportamiento de un intérprete para programas lógicos rebatibles, siguiendo la definición 2.31, y utilizando el predicado `warrant/2` especificado anteriormente en la Figura 4.5.

```
answer(Q,yes):-warrant(Q,_),!.
answer(Q,no):-complement(Q,CQ),warrant(CQ,_),!.
answer(Q,unknown):- \+ in_signature(Q),!.
answer(_,undecided).

in_signature(Q):-s_rule(Head,_),in_literal(Q,Head),!.
in_signature(Q):-d_rule(Head,_),in_literal(Q,Head),!.
in_signature(Q):-s_rule(_,Body),in_body(Q,Body),!.
in_signature(Q):-d_rule(_,Body),in_body(Q,Body),!.
in_body(Q,L):-in_literal(Q,L),!.
in_body(Q,(L,_)):-in_literal(Q,L),!.
in_body(Q,(_,B)):-in_body(Q,B),!.
in_literal(Q,Q):-!.              % Q is Q or
in_literal(Q,C):-complement(Q,C). % Q is the complement of C
```

Figura 4.11: Especificación de un intérprete

4.5.2 Verificación de contradicciones para reglas estrictas

Como fue explicado en la sección 2.3, el conjunto de reglas estrictas es usado para representar información no-rebatible, y por lo tanto debe poseer cierta coherencia interna. Es por esto que se asume, que en todo programa lógico rebatible, el conjunto Π (de hechos y reglas estrictas) es un conjunto no contradictorio.

La observación 2.5, establecía que en un programa lógico rebatible $\mathcal{P}= (\Pi, \Delta)$, como el conjunto Π se asume no contradictorio, entonces no existen dos literales complementarios que tengan derivaciones estrictas a partir de $\mathcal{P}$. Esto permite especificar una forma muy simple de verificar que no existan contradicciones en Π. La figura 4.12 muestra dicha especificación. Un intérprete de programas lógicos rebatibles puede utilizar esta verificación cada vez que carga un programa como muestra el predicado `load/1`.

```
contradictory_SSet:-
     pos_heads(P), neg_heads(N),    % collects Positive and Negative heads
     intersection(N,P,I), I \= [],  % if the intersection is not empty
     deriveApair(I).                % try to derive a pair of contrad. lit.

neg_heads(L):- findall(Head, (~Head <- _),L).
pos_heads(L):- findall(Head,( (Head -< _), Head \= ~_),L).

deriveApair([H|_]):-strict_derivation(H,[],strict),
                    strict_derivation(~H,[],strict),
                    write('ERROR! contradiction in strict information'),nl,!.
deriveApair([_|L]):-deriveApair(L).

load(FileName):-consult(FileName), \+ contradictory_SSet.
```

Figura 4.12: Verificación de Π no contradictorio

4.5.3 Built-ins

Para implementar operaciones aritméticas y relacionales, un intérprete de DeLP debe modificar la especificación de `argumento/3`, agregando una forma más de derivar un literal: esto es, ejecutando el built-in. La Figura 4.13 muestra la implementación.

```
argument(true,D,D) :-!.
argument((A,B),CurrentDerivation,ABDerivation) :-!,
         argument(A,CurrentDerivation,ADerivation),
         argument(B,ADerivation,ABDerivation).
argument(H,D,D,Line) :- temporary_fact(H,D), !.
argument(H,CurrentDerivation,Argument) :-
      (  (H <- B), Rule = (H <- B)            %   there is a strict rule
       ;                                      % or
         (H -< B), Rule = (H -< B)            %   there is a defeasible rule
 ;                                    % or
         let_prolog_prove(H), Rule=(H <-true) %   is a built-in
      ),
argument(B,CurrentDerivation,BodyDerivation,Line),
      Argument = [Clause|BodyDerivation],
verify_concordancy(H,Argument,Line).
let_prolog_prove(H):- is_a_built_in(H),call(H).
is_a_built_in(_ < _).                         % Built-in declaration
is_a_built_in(_ > _).
is_a_built_in(sum(_,_,_)).
is_a_built_in(difference(_,_,_)).
sum(S1,S2,Result):- Result is S1 + S2.        % Built-in implementation
difference(S1,S2,Result):- Result is S1 - S2.
```

Figura 4.13: Construcción de estructuras de argumentos con “built-ins”

4.6 Conclusiones

La semántica operacional descripta en este capítulo permite tener una descripción concreta de la Programación en Lógica Rebatible. Dicha descripción se realizó utilizando el lenguaje PROLOG, debido a su cercanía conceptual con DeLP.

El resultado fue una semántica clara y compacta, que describe de manera incremental todos los elementos involucrados en el análisis dialéctico de DeLP, especificando primero los conceptos básicos como la noción de derivación rebatible y argumento, mostrando luego la especificación de las relaciones entre argumentos, y describiendo finalmente conceptos más complejos como la construcción de árboles de dialéctica y líneas de argumentación aceptables.

De las especificaciones anteriores se obtuvo un Intérprete de Programas Lógicos rebatibles que fue utilizado para verificar la semántica, experimentar con nuevos ejemplos, y realizar las aplicaciones que se describen en el capítulo 8.

Capítulo 5

Computación en Paralelo de Programas Lógicos Rebatibles

El uso de paralelismo en Programación en Lógica ha sido objeto de estudio en los últimos 20 años [Gup94, GKA+94, Her86a, Her86b, Gre87, DeG84]. Sin embargo, el uso de paralelismo en Sistemas de Argumentación Rebatible no ha sido estudiado hasta el momento, y constituye uno de los aportes originales de esta tesis. Lo que se detalla en el presente capítulo ha sido publicado en los siguientes artículos [GS98, GS99b, GS99a].

En el caso de la Programación en Lógica el paralelismo puede explotarse *explícitamente* cuando el programador indica, mediante primitivas del lenguaje, qué partes del programa deben ser ejecutadas en paralelo. Pero también el paralelismo puede explotarse *implícitamente*, cuando es el compilador, o el intérprete del lenguaje, el que decide directamente que partes del programa pueden paralelizarse, sin intervención del programador. Para el objetivo de esta tesis, sólo es de interés el paralelismo implícito.

Existen diferentes formas de paralelismo que pueden explotarse implícitamente: paralelismo-OR, paralelismo-AND y también paralelismo en la unificación (ver [Gup94, Her86a]). Como se mostrará en este capítulo, estas tres formas de paralelismo pueden ser explotadas en DeLP. Pero además, existen nuevas fuentes de paralelismo implícito que pueden explotarse en un sistema de argumentación rebatible:

1. diferentes argumentos para un literal h pueden computarse en paralelo,
2. los derrotadores de una estructura $\langle \mathcal{A}, h \rangle$ pueden buscarse en paralelo,

3. diferentes líneas de argumentación de un árbol de dialéctica pueden construirse en paralelo.
4. árboles de dialéctica para diferentes estructuras de argumento para un literal h pueden analizarse en paralelo.

Todas estas estrategias permiten acelerar el proceso de dialéctica y permiten realizar un proceso de argumentación paralelizado.

Para estudiar el paralelismo para DeLP, comenzaremos por los diferentes tipos de paralelismo propuestos para la Programación en Lógica. Estos serán descriptos brevemente a continuación. Se mostrará luego que todas las formas de paralelismo implícito identificadas para la Programación en Lógica, pueden explotarse directamente para computar *derivaciones rebatibles*.

Como fue discutido en el capítulo 2, obtener una derivación rebatible es sólo el comienzo del proceso dialéctico. Por lo tanto, en las secciones siguientes se estudiarán nuevas fuentes de paralelismo que pueden explotarse implícitamente en DeLP y que no están presentes en la programación en lógica.

5.1 El paralelismo en la Programación en Lógica

La programación en lógica es un paradigma basado en un subconjunto de la Lógica de Predicados de Primer Orden [Llo87, Lif96]. Nos referiremos en esta sección a la Programación en Lógica Definida, donde una *regla de programa* es un par ordenado, por convención denotado '$Cabeza \leftarrow Cuerpo$'. La primer componente $Cabeza$ es un átomo y la segunda componente $Cuerpo$ es un conjunto finito de átomos. Una regla con cabeza A y cuerpo $\{Q_1, \ldots, Q_m\}$, puede escribirse también como "$A \leftarrow Q_1, \ldots, Q_n$". Si el cuerpo es un conjunto vacío, la regla se denota "$A \leftarrow true$". Una *meta definida* o *consulta* para un programa lógico es una regla sin cabeza "$\leftarrow Q_1, \ldots, Q_n$". Un *Programa Lógico Definido* es un conjunto de reglas de programa.

Para responder consultas con respecto a un programa lógico definido, es necesario disponer de un *evaluador* que utilice un algoritmo de resolución. En Programación en Lógica existen muchos procedimientos de refutación, que son refinamientos del procedimiento original de Robinson [Rob65]. Se incluye en esta sección la definición de *resolución-SLD* ("Linear resolution with Selection function for Definite rules") dada en [Llo87].

Definición 5.1 *Sea G una consulta $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ y C una regla definida $A \leftarrow B_1, \ldots, B_q$. Se dice que G' se* deriva *de G y C usando el unificador más general θ si se cumplen las siguientes condiciones:*

- *A_m es un átomo, llamado el átomo seleccionado de G.*
- *θ es el unificador más general para A_m y A.*
- *G' es la consulta $\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k$*

Utilizando la terminología de resolución, se dice que G' es una resolvente *de G y C.*

Definición 5.2 *Sea P un programa definido y G una consulta. Una derivación-SLD consiste de una secuencia (finita o infinita) G_0=$G, G_1, \ldots$ de consultas y una secuencia $C_1, C_2, \ldots$ de variantes de reglas del programa P, y una secuencia $\theta_1, \theta_2, \ldots$ de unificadores más generales, tal que G_{i+1} se deriva de G_1 y C_{i+1} usando θ_{i+1}.*

Definición 5.3 *Una refutación-SLD para $P \cup \{G\}$ es una derivación-SLD finita de $P \cup \{G\}$ que tiene la regla vacía como última consulta en la derivación. Si G_n es la regla vacía, entonces la refutación tiene longitud n.*

De esta manera, el proceso de computación del evaluador consiste en seleccionar una submeta G en el cuerpo de una consulta, y buscar una cabeza de una regla $H \leftarrow B$ en el programa tal que H unifique con G. Si la unificación tiene éxito, entonces en la consulta, se reemplaza la submeta seleccionada G, por el cuerpo B de la regla, aplicando la sustitución unificadora al resto de las submetas de la consulta. El proceso de selección de submetas y reducción es aplicado hasta que no queden más submetas para reducir, en cuyo caso la consulta es exitosa.

Observación 5.1 *A diferencia de DeLP, en la programación en lógica las reglas de programa tienen variables y se utiliza la unificación de literales en el proceso de derivación. Como se mostrará más adelante, esta diferencia no afecta a la definición de los diferentes tipos de paralelismo en DeLP.*

El evaluador de un programa lógico tiene cierta libertad (no-determinismo) para seleccionar átomos y reglas [Her86a, Gup94]. Por un lado, dada una consulta " $\leftarrow Q_1, \ldots, Q_n$", el evaluador tiene la libertad de seleccionar cual átomo Q_i tomar primero. Por otro lado, si existen varias reglas $R_1, R_2, \ldots, R_k$ del programa, tal que la cabeza de estas reglas unifiquen con el átomo seleccionado Q_i, el evaluador tiene otra vez la libertad de seleccionar una de estas reglas para seguir computando. De esta manera, si la regla elegida no lleva a una solución, otras reglas puede intentarse, hasta llegar a una solución o intentar con todas las reglas. Esta libertad en el evalùador da origen a dos tipos de no-determinismo:

- no-determinismo-1: Si existen varias reglas cuyas cabezas unifican con la submeta seleccionada, no está determinada cual regla elegir. La política del evaluador, llamada *regla de búsqueda* (search rule), resulta en dos sub-tipos de no-determinismo:
  "don't care" non-determinism: una vez que una regla es elegida, el sistema se limita a esa elección, y
  "don't know" non-determinism: más de una de las posibles elecciones pueden ser elegidas en la búsqueda de una solución.
- no-determinismo-2: si la consulta tiene varias submetas, no está determinada cuál de ellas elegir para su ejecución. La política del evaluador es llamada *regla de computación* (computation rule).

Ejemplo 5.1 Considérese el siguiente programa:

$$
\begin{aligned}
&p(X,Y) \leftarrow q(X), r(Y)\\
&q(X) \leftarrow t(X)\\
&t(X) \leftarrow s(X)\\
&t(X) \leftarrow v(X)\\
&t(4) \leftarrow true\\
&r(X) \leftarrow v(X)\\
&s(1) \leftarrow true\\
&s(2) \leftarrow true\\
&v(3) \leftarrow true\\
&t(4) \leftarrow true
\end{aligned}
$$

Supóngase un evaluador en el que la regla de búsqueda elige de arriba hacia abajo, y la regla de computación elige de izquierda a derecha. Da-

da la consulta "$\leftarrow p(X,Y)$", el evaluador generaría la siguiente secuencia:	
$\leftarrow p(X,Y)$	elige $p(X,Y)$ y unificando con la regla "$p(X,Y) \leftarrow q(X), r(Y)$" obtiene:
$\leftarrow q(X), r(Y)$	elige $q(X)$ y unificando con la regla "$q(X) \leftarrow t(X)$" obtiene:
$\leftarrow t(X), r(Y)$	elige $t(X)$ y unificando con la regla "$t(X) \leftarrow s(X)$" obtiene:
$\leftarrow s(X), r(Y)$	elige $s(X)$ y unificando con la regla "$s(1) \leftarrow true$" obtiene:
$\leftarrow r(Y)$	elige $r(y)$ y unificando con la regla "$r(Y) \leftarrow v(Y)$" obtiene:
$\leftarrow v(Y)$	elige $v(y)$ y unificando con la regla "$v(3) \leftarrow true$" obtiene:
$\leftarrow$	llega a la consulta vacía y da éxito

Estos dos tipos de no-determinismo dan origen a los dos principales tipos de paralelismo en programación en lógica: paralelismo-OR, y paralelismo-AND.

1. el *paralelismo-OR* se origina cuando una submeta unifica con más de una regla de programa y las submetas de los cuerpos de esas reglas son ejecutadas en paralelo.
2. el *paralelismo-AND* se origina cuando varias submetas en una consulta, o en el cuerpo de una regla, son ejecutadas en paralelo.

Ejemplo 5.2 Considérese nuevamente el programa del ejemplo 5.1:

$$
\begin{aligned}
p(X,Y) &\leftarrow q(X), r(Y)\\
q(X) &\leftarrow t(X)\\
t(X) &\leftarrow s(X)\\
t(X) &\leftarrow v(X)\\
t(4) &\leftarrow true\\
r(X) &\leftarrow v(X)\\
s(1) &\leftarrow true\\
s(2) &\leftarrow true\\
v(3) &\leftarrow true
\end{aligned}
$$

Si el evaluador utiliza paralelismo OR y AND, entonces dada la consulta "$\leftarrow p(X,Y)$", se inicia la la siguiente secuencia:

$\leftarrow p(X,Y)$	usa la regla "$p(X,Y) \leftarrow q(X), r(Y)$"
$\leftarrow q(X), r(Y)$	(paralelismo-AND) dispara en paralelo "$\leftarrow q(X)$" y "$\leftarrow r(Y)$"

a partir de aquí la secuencia se parte en dos ejecuciones en paralelo:

secuencia	para " $\leftarrow q(X)$":	secuencia	para " $\leftarrow r(Y)$":
$\leftarrow q(X)$	usa "$q(X) \leftarrow t(X)$"	$\leftarrow r(Y)$	usa "$r(Y) \leftarrow v(Y)$"
$\leftarrow t(X)$	(paralelismo-OR) tres reglas unifican	$\leftarrow v(Y)$	usa "$v(3) \leftarrow true$"
	y las tres son disparadas en paralelo	$\leftarrow$	da éxito
$\leftarrow$	la regla "$t(4) \leftarrow true$" da éxito		

Ambas secuencias dan éxito y por lo tanto la consulta original " $\leftarrow p(X, Y)$", da éxito, habiendo unificado $X = 4$ $Y = 3$. Las otras dos ejecuciones en paralelo disparadas por el paralelismo-OR, también darán éxito con otras unificaciones para la variable X.

Tal como indica Gopal Gupta en [Gup94], existen numerosas propuestas para extender el lenguaje de la Programación en Lógica con constructores explícitos para lograr paralelismo, tal es el caso de: *Delta Prolog, CS-Prolog, Shared prolog, Parlog, GHC*, y *Concurrent Prolog*. En dichos sistemas, el programador tiene que pensar y programa explícitamente en términos de procesos que se ejecutan en paralelo, sacrificando la mayoría de las veces la declaratividad del lenguaje.

Sin embargo, en la Programación en Lógica, el paralelismo puede explotarlo directamente el propio evaluador en tiempo de ejecución, liberando al programador de la tarea de paralelizar el código, y conservando la declaratividad del lenguaje. Gopal Gupta menciona en [Gup94] cuatro formas principales de paralelismo explotable en forma implícita:

1. Paralelismo-OR.
2. Paralelismo-AND Independiente.
3. Paralelismo-AND Dependiente.
4. Paralelismo en la unificación.

El paralelismo-OR está relacionado con la búsqueda a lo ancho (breadth-first search), ya que las ramas OR del árbol AND-OR son evaluadas en paralelo. Un sistema de paralelismo-OR tiende a ser más completo que PROLOG, ya que las soluciones que están a la derecha de una rama infinita pueden computarse.

El paralelismo-AND independiente tiene lugar cuando las submetas de una consulta no tienen variables en común (por ejemplo $\leftarrow p(X), q(Y), r(Z)$), con lo cual las submetas pueden ejecutarse en paralelo independientemente una de otra. El paralelismo-AND dependiente aparece cuando dos o más submetas comparten una variable, (por ejemplo

$\leftarrow p(X), q(X), r(X)$). En este caso la ejecución de las submetas no puede ser independiente, ya que en un sistema secuencial, la unificación de X en la primer submeta debe ser utilizada por las demás. Los métodos de paralelismo-AND dependiente tienen en consideración esta situación. El paralelismo en la unificación se logra si los distintos argumentos de un predicado son unificados en paralelo.

Según Gopal Gupta en [Gup94] estas cuatro formas de paralelismo son ortogonales una a la otra, i. e., cada una puede explotarse sin afectar a las demás. Sin embargo, todavía no se ha implementado ningún sistema paralelo con estas características, y continúa siendo objeto de investigación.

Otros tipos de paralelismo de más bajo nivel también han sido identificados en [Her86a]:

- *Stream Parallelism*: cuando más de un proceso puede evaluar una estructura de datos compleja en forma incremental, trabajando en paralelo con el proceso que la está produciendo.
- *Search Parallelism*: cuando un programa puede ser dividido en conjuntos disjuntos de reglas, da tal manera que la búsqueda de reglas se realice en paralelo en cada uno de estos conjuntos.

5.2 Argumentos en Paralelo

La construcción de estructuras de argumentos es un tema central dentro de la programación en lógica rebatible. Como se mostrará en esta sección, varias formas de paralelismo podrán ser utilizadas durante el proceso de construcción de un argumento.

Para obtener una estructura de argumento $\langle \mathcal{A}, h \rangle$, a partir de un programa $\mathcal{P}$=(Π, Δ) (ver definición 2.15), debe obtenerse una derivación rebatible, tal que el conjunto $\Pi \cup \mathcal{A}$ no sea contradictorio. A continuación, se analizará en primer lugar como obtener derivaciones rebatibles en paralelo, y luego se mostrará que es posible realizar la verificación de contradicciones en paralelo a las derivaciones.

5.2.1 Derivaciones Rebatibles en Paralelo

Para obtener una estructura de argumento para un literal h, es necesario en principio construir una derivación rebatible para h. El concepto de derivación rebatible fue introducido en la definición 2.12, y luego especificado en la sección 4.1.

Como puede verse en la especificación de la Figura 4.2, el proceso para computar *derivaciones rebatibles* posee los dos tipos de no-determinismo especificados anteriormente para un evaluador de Programación en Lógica. Por lo tanto, todas las formas de paralelismo implícito identificadas para la Programación en Lógica, pueden explotarse directamente para computar derivaciones rebatibles.

Al computar derivaciones rebatibles debe tenerse en cuenta que los programas lógicos rebatibles contienen dos tipos de reglas y literales que utilizan la negación fuerte. Sin embargo, esto no representa un problema para los tipos de paralelismo mencionados en la sección anterior. En el caso del paralelismo-OR, este puede extenderse sin problemas para que considere la ejecución en paralelo de ambos tipos de reglas.

Los literales con negación fuerte tampoco representan un problema en el caso de paralelsimo-AND, ya que como se dijo en el capítulo 4, el símbolo de negación fuerte es considerado como parte del literal por el mecanismo de derivación rebatible. De esta manera, en una regla como "$h -\!\prec \sim p, q, r, \sim s$" las consultas del cuerpo pueden evaluarse en paralelo sin necesidad de distinguir entre literales negados o sin negar. Cabe recordar, que el uso de la negación cobra relevancia sólo cuando se buscan contradicciones o contra-argumentos.

5.2.2 Verificación de contradicciones en Paralelo

Al construir una estructura de argumento $\langle \mathcal{A}, h \rangle$, debe verificarse que el conjunto $\Pi \cup \mathcal{A}$ no sea contradictorio. Como se mencionó en la sección 4.2, este proceso puede hacerse simultáneamente con la derivación rebatible. Por lo tanto, es posible verificar la concordancia de un literal (ver Figura 4.9) en paralelo con la derivación de otra parte del argumento.

5.2.3 Cómputo de argumentos en paralelo

El paralelismo-OR puede explotarse en DeLP para obtener todas las posibles estructuras de argumento para un literal h. En una implementación secuencial, cuando mas de una regla de programa puede utilizarse, entonces para obtener más de un argumento debe realizarse *backtracking*. Si se utiliza paralelismo-OR, y asumiendo un número ilimitado de procesadores (o procesos) todos los argumentos pueden computarse en paralelo sin la necesidad de utilizar backtracking.

Si todas las estructuras de argumento se computan en paralelo, entonces el cálculo de minimalidad puede simplificarse utilizando la siguiente estrategia: *cada vez que un subconjunto propio de un posible argumento C es encontrado en paralelo, se descarta al superconjunto C.*

5.3 Árboles de Dialéctica en Paralelo

Considérese el árbol de dialéctica marcado de la Figura 5.1. La raíz de dicho árbol está marcada como un nodo "D" debido a que existe un derrotador marcado como nodo "U".

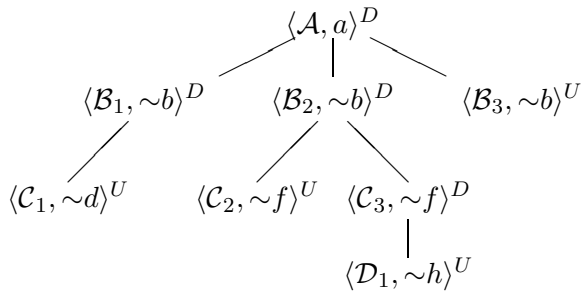

Figura 5.1: Árbol de dialéctica marcado del ejemplo 2.15

En una implementación secuencial, si el árbol de dialéctica es generado en profundidad (*depth-first*), entonces todo el árbol de la Figura 5.1 debe ser explorado de izquierda a derecha hasta descubrir el derrotador $\mathcal{B}_3$ (si se recorre de derecha a izquierda es fácil encontrar un ejemplo que muestre lo mismo). Si en cambio, el árbol de dialectica es generado por niveles (*breadth-first*), entonces no se necesitaría recorrer todo el árbol. No obstante, el principal problema de generar árboles por niveles, es que se requiere mucho más espacio en memoria que si se generan en profundidad.

Dada un estructura de argumento, sus derrotadores pueden computarse en paralelo. Por lo tanto, cada línea de argumentación del árbol de dialéctica puede computarse en paralelo, si cada vez que un nodo del árbol es generado, sus derrotadores son buscados en paralelo. Generar el árbol de dialéctica en paralelo tiene varias ventajas: acelera el proceso de dialéctica, permite distribuir el análisis dialéctico en diferentes procesos (o máquinas sobre una red) y además, aunque cada línea de argumentación se genera en profundidad, el computo global puede verse como una generación por niveles.

Si el árbol de dialéctica es computado en paralelo, entonces el control debe distribuirse sobre los nodos. De esta forma, el proceso de marcado de los nodos como "U" o "D" debe hacerse mediante un mecanismo de pasaje de mensajes entre los nodos. A continuación se introduce un modelo para construir un árbol de dialéctica en paralelo en forma distribuída. Uno de los objetivos del modelo es delegar la mayor cantidad de tarea a los procesos hijos. La descripción de este proceso distribuído se realizará en dos partes: (a) la generación del árbol de dialéctica y (b) la estrategia utilizada para decidir la marca "U" o "D" que tiene cada nodo. Como el modelo puede adaptarse a una implementación con varios procesadores o un procesador que corra procesos concurrentemente, los términos *proceso* y *procesador* se consideran como sinónimos.

a) Generación de un árbol de dialéctica en paralelo: Luego que un procesador P haya generado un argumento $\mathcal{A}$ las siguientes acciones deben llevarse a cabo:

1. El conjunto $Co(\mathcal{A})$ de potenciales puntos de contra-argumentación deben ser obtenidos por el procesador local P.
2. Para cada literal $l \in Co(\mathcal{A})$ la búsqueda de argumentos para el complemento $\overline{l}$ se realizará en parallelo. Estos argumentos son potenciales derrotadores para $\mathcal{A}$, y algunos de ellos se convertirán en nodos hijos de $\mathcal{A}$ en el árbol de dialéctica.
3. A fin de no quedar en una espera ociosa, una vez que las búsquedas de derrotadores han sido disparadas en paralelo, el procesador local P puede encargarse de ejecutar una de las búsquedas. Las demás búsquedas serán realizadas por procesadores remotos. Si el procesador local termina su búsqueda y aún quedan tareas, entonces puede tomar otra tarea y continuar trabajando para no quedar ocioso.
4. Cuando un procesador (remoto o local) finaliza su tarea debe informar a su padre acerca del estado en que se encuentra el argumento que generó, esto es: éxito o fracaso según el proceso de marcado que se describe en el inciso (b) a continuación.

5. Una vez que todos los procesadores hijos han terminado, el procesador P que disparó las tareas evalúa el estado de sus derrotadores y genera el estado del argumento $\mathcal{A}$ generado por él, para informar a su nodo padre.

b) Marcado de un árbol en forma distribuída: a continuación se indica como evaluar la marca de cada nodo en forma distribuída.

1. Si un proceso P que había construído un argumento $\mathcal{A}$ recibe de **algún** proceso hijo (que estaba buscando un derrotador $\mathcal{B}$ para $\mathcal{A}$), el mensaje que ha podido generar el derrotador $\mathcal{B}$ con éxito (esto es, es un nodo "U"), entonces:
   (1) el argumento $\mathcal{A}$ se convierte automáticamente en derrotado (nodo "D"),
   (2) el proceso P debe enviar inmediatamente un mensaje a su nodo padre informando que el argumento que ha construído es un nodo "D", y
   (3) como el estado del argumento $\mathcal{A}$ ya está definido como nodo "D", entonces los demás procesos hijos de P pueden ser "podados" y evitar computos innecesarios. Para esto, el proceso P envía mensajes a los demás nodos hijos para abortar su ejecución.
2. Si **todos** los procesos hijos de P que buscan derrotadores para $\mathcal{A}$ informan que han fallado (porque no pueden construir el derrotador, o el derrotador generado es un nodo "D") entonces el argumento $\mathcal{A}$ generado por P se convierte en un nodo "U" y el proceso P debe enviar un mensaje a su proceso padre indicando esta situación.

5.4 Conclusiones

En este capítulo se mostró como las tres formas de paralelismo básicas identificadas en la Programación en Lógica, pueden ser explotadas implícitamente en DeLP. Pero además, se definieron nuevas fuentes de paralelismo implícito que pueden explotarse en un sistema de argumentación rebatible:

1. diferentes argumentos para un literal h pueden computarse en paralelo,
2. los derrotadores de una estructura $\langle \mathcal{A}, h \rangle$ pueden buscarse en paralelo,
3. diferentes líneas de argumentación de un árbol de dialéctica pueden construirse en paralelo.

4. árboles de dialéctica para diferentes estructuras de argumento para un literal h pueden analizarse en paralelo.

Todas estas estrategias permiten acelerar el proceso de dialéctica y permiten realizar un proceso de argumentación paralelizado.

Se desarrolló un modelo que permite construir un árbol de dialéctica en paralelo en forma distruibuída. Como el árbol de dialéctica es computado en paralelo, entonces el control debe distribuirse sobre los nodos. De esta forma, el proceso de marcado de los nodos como "U" o "D" se realiza mediante un mecanismo de pasaje de mensajes entre los nodos.

Capítulo 6

Extensiones de la Programación en Lógica Rebatible

Consideraremos en este capítulo dos extensiones de la Programación en Lógica Rebatible. En primer lugar se definirá la *Programación en Lógica Rebatible Extendida* que permite utilizar dos tipos de negación: la negación fuerte que permite representar conocimiento contradictorio, y la negación default que permite representar información incompleta. Luego se se motrará como puede extenderse la Programación en Lógica Rebatible para permitir la representación de presuposiciones, esto es, reglas rebatibles sin cuerpo.

6.1 Negación Default en DeLP

La *negación default*[1], denotada habitualmente con el símbolo "*not*", es utilizada ampliamente en programación en lógica, bases de datos deductivas, y en formalismos de razonamiento no-monótono. Según Alferes, Pereira y Przymusinski, en [APP96], el nombre "negación default" surge debido a que "*not F*" se asume "by default". Esto es, "*not F*" es asumido ante la falta de *suficiente evidencia* en contra. El significado de "suficiente evidencia" depende de la semántica del formalismo usado.

Por ejemplo, en *Closed World Assumption* (CWA) de Reiter [Rei78], *not F* es asumido si F no es probable, o equivalentemente, si existe un modelo mínimo en el cual F es

[1] También llamada *negación por falla* en algunos contextos.

falso. En *Generalized CWA* de Minker [Min82, GPP89], y en *Circunscription* de MacCarthy [McC80], *not* F es asumido sólo si F es falso en todos los modelos mínimos. En el caso de *Predicate Completion* de Clark [Cla78], esta forma de negación se llama *negación por falla finita*, debido a que *not* F es derivable cuando los intentos de derivación de F fallan en forma finita.

Como fue discutido en los capítulos anteriores, DeLP utiliza la negación fuerte denotada con el símbolo "$\sim$". A diferencia de la negación default, un literal "$\sim$F" no puede ser asumido, sino que debe ser *probado*. En el caso de DeLP se cree en "$\sim$F" cuando el literal "$\sim$F" está *garantizado*. En este capítulo se mostrará la importancia de extender la programación en lógica rebatible, agregando la negación default. Se extenderán las reglas de programa para poder utilizar los dos tipos de negaciones, y se definirá la forma en que la negación default será considerada. Como se verá a continuación, en el caso de DeLP, *not* F será asumido si F no está garantizado, esto es, "suficiente evidencia en contra de *not* F" significará "F está garantizado". Parte de los conceptos desarrollados en el presente capítulo, fueron publicados en las actas del Workshop *DGNMR'99* [GS99c].

6.1.1 Motivación

Los programas lógicos rebatibles permiten la utilización de negación fuerte, tanto en el cuerpo como en la cabeza de las reglas. Por ejemplo, es posible representar información como "los inocentes no son culpables", y "los no carnívoros usualmente no son peligros", mediante las siguientes reglas:

$$\sim culpable \leftarrow inocente$$
$$\sim peligroso \prec \sim carnivoro$$

Sin embargo, para expresar que: "*es inocente si no hay evidencia que sea culpable*", la regla "$inocente \prec \sim culpable$" no resulta conveniente, ya que su significado es: "si puede probarse que no es culpable, entonces es inocente". En este caso conviene utilizar la negación "default". Lo mismo ocurre para representar "*buscar a alguien si está perdido y no puede probarse que ha muerto*". En ambos casos la negación default se comporta mejor:

$$inocente \prec not\ culpable$$
$$buscar \prec perdido, not\ muerto$$

Pero existen otros casos donde la negación default no es la adecuada. Por ejemplo, si se quiere reprensentar la información "*aterrizar si la pista no está ocupada*" con la regla "*aterrizar* $\leftarrow$ *not pista_ocupada*", la falta de información sobre el estado de la pista (niebla por ejemplo), permitiría disparar la regla aunque la pista esté verdaderamente ocupada. En este caso, debería usarse negación fuerte, expresando: "*aterrizar* $\leftarrow$ $\sim$*pista_ocupada*", esto es, aterrizar si existe "garantía" que la pista no está ocupada.

Obsérvese finalmente, que en algunos casos las dos negaciones son necesarias simultáneamente. Por ejemplo, para expresar "*no cruzar la vías si no puede probarse que no viene el tren*", puede utilizarse la siguiente regla[2]: $\sim$*cruzar_las_vias* $-\!\prec$ *not* $\sim$*viene_el_tren*.

Los ejemplos anteriores muestran que en un sistema de representación de conocimiento, es importante disponer de los dos tipos de negación. Por lo tanto, se extenderá la programación en lógica rebatible a fin de permitir el uso de negación default sobre los literales del cuerpo de las reglas rebatibles. De esta forma, la negación fuerte permitirá representar información potencialmente contradictoria, y la negación default se podrá usar para representar información incompleta.

6.1.2 Programas Lógicos Extendidos

A continuación se extiende la sintaxis de DeLP a fin de permitir el uso de la negación default solamente en el cuerpo de las reglas rebatibles. La razón de no utilizar negación default en reglas estrictas es doble. Por un lado, en una regla "$p \leftarrow$ *not* q", el consecuente p se basa en una suposición, lo cual, de alguna manera, estaría contradiciendo el hecho que la regla sea estricta. Por otro lado, si se permitiera negación default en las reglas estrictas, el conjunto Π podría resultar contradictorio, como se muestra el ejemplo 6.2 más adelante.

Definición 6.1 (Literal extendido) *Un literal extendido es un literal L, o un literal precedido por el símbolo de la negación default "not L".*

Definición 6.2 (Regla Rebatible Extendida)
Una regla rebatible extendida *es un par ordenado, denotado "Cabeza* $-\!\prec$ *Cuerpo", donde*

[2]Adaptación de un ejemplo elaborado por John MacCarthy

el primer elemento Cabeza, es un literal, y el segundo elemento, Cuerpo, es un conjunto finito no vacío de literales extendidos. Una regla rebatible extendida con cabeza L_0 y cuerpo $\{X_1, \ldots, X_n\}$ $(n > 0)$ se escribirá también como: $L_0 \prec X_1, \ldots, X_n$.

Un *programa lógico rebatible extendido* será entonces un conjunto posiblemente infinito de hechos, reglas estrictas y reglas rebatibles extendidas. Como antes, cuando sea necesario denotaremos $\mathcal{P}=(\Pi, \Delta)$, para identificar al conjunto Π de hechos y reglas estrictas, y al conjunto Δ de reglas rebatibles extendidas del programa $\mathcal{P}$.

Siguiendo con la idea de Alferes, Pereira y Przymusinski, en [APP96], el literal extendido "*not L*", *será asumido ante la falta de suficiente evidencia en su contra.* En nuestro lenguaje, esto significará que L no puede ser garantizado dentro del proceso de dialéctica. Es decir, si L está garantizado, entonces "*not L*" no puede asumirse, y por el contrario, si toda estructura de argumento $\langle \mathcal{A}, L \rangle$ es derrotada, entonces "*not L*" podrá asumirse. Este proceso será introducido en la sección siguiente.

Como en el caso de DeLP, asumir "*not L*" implica un análisis dialéctico, entonces ante un literal extendido "*not L*", la definición de derivación rebatible simplemente lo "asumirá", dejando que luego el proceso dialéctico decida si L está garantizado o no. A continuación figura la definición de derivación rebatible modificada para las reglas extendidas, como puede verse, los literales precedidos de negación default son simplemente ignorados.

Definición 6.3 (Derivación Rebatible con reglas extendidas)
Sea $\mathcal{P}= (\Pi, \Delta)$ un programa lógico rebatible extendido y L un literal. Una derivación rebatible para L a partir de $\mathcal{P}$ consiste de una secuencia finita de literales fijos $L_1, L_2, \ldots L_n = L$, provisto de que exista una secuencia de reglas (estrictas o rebatibles) extendidas $R_1, R_2, \ldots R_n$ del programa $\mathcal{P}$, de tal forma que para cada literal L_i en la secuencia tenemos que:

(a) L_i es un hecho, o

(b) existe en $\mathcal{P}$ una regla R_i con cabeza L_i y cuerpo $B_1, B_2, \ldots B_k$ donde todo elemento del cuerpo, que no es un literal precedido por la negación default, es un elemento de la secuencia que precede a L_i.

Ejemplo 6.1 Considere el siguiente programa lógico rebatible extendido:

$$buscar\_a(X) \prec perdido\_en\_el\_mar(X), not\ muerto(X)$$
$$perdido\_en\_el\_mar(popeye)$$
$$muerto(X) \prec \sim toma\_agua(X)$$
$$\sim toma\_agua(X) \prec perdido\_en\_el\_mar(X)$$
$$toma\_agua(X) \prec perdido\_en\_el\_mar(X), junta\_lluvia(X)$$
$$junta\_lluvia(X) \prec marino\_experimentado(X)$$
$$marino\_experimentado(popeye)$$

A partir de este programa existe una derivación rebatible para el literal "$buscar\_a(popeye)$" obtenida utilizando el hecho "$perdido\_en\_el\_mar(popeye)$" y la regla "$buscar\_a(popeye) \prec perdido\_en\_el\_mar(popeye), not\ muerto(popeye)$" El literal extendido "$not\ muerto(popeye)$" es simplemente asumido por la derivación rebatible. Como se mostrará a continuación, el proceso de dialéctica considerará luego si está bien asumirlo o no.

El ejemplo 6.2 a continuación, muestra una de las razones de no permitir el uso de negación default en las reglas fuertes.

Ejemplo 6.2 Supóngase que se permitiera la negación default en las reglas estrictas, entonces a partir del siguiente conjunto

$$\left\{ \begin{array}{l} inocente(X) \leftarrow not\ culpable(X) \\ \sim inocente(pepe) \end{array} \right\}$$

se podría derivar rebatiblemente un par de literales complementarios con respecto a la negación fuerte. Es decir, el conjunto Π sería contradictorio.

Considérese el programa $\mathcal{P}=(\emptyset, \{p \prec not\ p\})$. A partir de $\mathcal{P}$ es posible obtener una derivación rebatible para el literal "p". Sin embargo, no debería ser posible obtener una estructura de argumento para "p", habiendo asumido "$not\ p$". El ejemplo anterior puede generalizarse, y si el conjunto de reglas es $\mathcal{A}$={ $(a \prec b)$, $(b \prec c)$ $(d \prec not\ a)$ }, se estaría obteniendo un argumento para el literal "a", asumiendo "$not\ a$".

A continuación se introduce la definición de estructura de argumento modificada para evitar estos casos. La condición 2 incluye la restricción que impide la formación de los casos citados antes.

Definición 6.4 (Estructura de argumento para DeLP extendida)
Sea h un literal y $\mathcal{P}$=(Π, Δ) un programa lógico rebatible, una estructura de argumento *para h es un par $\langle \mathcal{A}, h \rangle$, donde $\mathcal{A}$ es conjunto de reglas rebatibles de Δ, tal que:*

1. *existe una derivación rebatible para h a partir de $\Pi \cup \mathcal{A}$.*
2. *si L es un literal de la derivación rebatible de h, entonces not L no pertenece al cuerpo de ninguna regla rebatible de $\mathcal{A}$,*
3. *$\Pi \cup \mathcal{A}$ es no contradictorio, y*
4. *$\mathcal{A}$ es minimal, es decir, no existe un subconjunto propio $\mathcal{A}'$ de $\mathcal{A}$ tal que $\mathcal{A}'$ satisface las condiciones anteriores.*

En el ejemplo a continuación se muestran algunos casos en los cuales las estructuras de argumento utilizan negación default.

Ejemplo 6.3 Considérese el siguiente programa:

$a \prec not\ p$	$d \prec not\ l$
$b \prec not\ q$	$l \prec e$
$c \prec not\ s$	$\sim l \prec e, t$
$q \prec t$	e
t	$f \prec g$
$\sim c \prec t$	$g \prec not\ f$

Utilizando este programa, el literal "p" no está garantizado, ya que no existe estructura de argumento para "p", entonces debería poder asumirse "*not* p", y por lo tanto el literal "a" debería estar garantizado.

El caso del literal "b" es diferente, ya que aunque existe una estructura de argumento $\langle \{b \prec not\ q\}, b \rangle$, el literal "$q$" está garantizado, (la estructura de argumento para "q" no tiene derrotadores), entonces no puede asumirse "*not* q", y por lo tanto, el literal "b" no debería estar garantizado.

En el caso del literal "c", aunque existe una estructura de argumento $\langle \{c \prec not\ s\}, c \rangle$, y puede asumirse "*not* s", (ya que el literal "s" no está garantizado), el literal "c" no está garantizado porque existe un derrotador de bloqueo que ataca al literal "c": $\langle \{\sim c \prec t\}, t \rangle$.

Finalmente, en el caso del literal "*d*", aunque existe una estructura de argumento para "*l*", el literal "*l*" no está garantizado porque existe un derrotador propio. Por lo tanto se debería poder asumir "*not l*" y el literal "*d*" debería estar garantizado.

Obsérvese que a partir de este programa, no existe argumento para el literal "*f*", ya que la derivación rebatible de "*f*", utiliza la regla "$g -\prec not\ f$", que no satisface la nueva condición en la definición de argumento.

6.1.3 Nueva noción de ataque entre argumentos

Hasta ahora, en DeLP, la única forma de *atacar* una estructura de argumento $\langle \mathcal{A}, h \rangle$, era construyendo otra estructura $\langle \mathcal{B}, q \rangle$ que contradiga (desacuerdo) a algún literal interno del conjunto $\mathcal{A}$. Esto es, el ataque estaba basado en el desacuerdo de literales contradictorios con respecto a la negación fuerte. Sin embargo, como sugiere el análisis del ejemplo 6.3, en los programas lógicos extendidos, los literales asumidos con negación default serán también un punto de ataque contra el argumento.

Según Phan M. Dung en [Dun93a], los literales negados con negación default, son "*suposiciones*" (assumptions) sobre las cuales la derivación está basada. La aceptación de la derivación (en nuestro caso el argumento) depende de la aceptación de dichas *suposiciones*. Esta política también es compartida por Henry Prakken y Giovanni Sartor en [PS97].

En [Dun93a], se define una noción de ataque entre argumentos llamada *ground attack* de la siguiente manera: un argumento $\mathcal{A}$ para un literal h, es un *ground attack* para $\mathcal{B}$, si $\mathcal{B}$ contiene una regla con un literal extendido *not* h en su cuerpo. Esto es, un argumento como $\mathcal{B}$, basado en la suposición *not* h, puede ser atacado con un argumento para h.

Extenderemos a continuación la noción de derrota para incorporar este tipo de ataque entre argumentos. A diferencia del concepto de derrota definido en los capítulos anteriores, como se ataca una suposición, los argumentos involucrados no son comparados. Formalmente:

Definición 6.5 (Ataque a una suposición)
Sea $\mathcal{P} = (\Pi, \Delta)$ *un programa lógico rebatible extendido, y sean* $\langle \mathcal{B}, q \rangle$ *y* $\langle \mathcal{A}, h \rangle$ *dos estructuras de argumento obtenidas a partir de* $\mathcal{P}$. $\langle \mathcal{A}, h \rangle$ *es un ataque a una suposición de* $\langle \mathcal{B}, q \rangle$, *si el literal not* h *es parte del cuerpo de una regla de* $\langle \mathcal{B}, q \rangle$.

Definición 6.6 (Derrotador en DeLP extendida)
Sean $\langle \mathcal{A}_1, h_1 \rangle$ *y* $\langle \mathcal{A}_2, h_2 \rangle$ *dos estructuras de argumento. La estructura* $\langle \mathcal{A}_1, h_1 \rangle$ *es un* derrotador *para* $\langle \mathcal{A}_1, h_1 \rangle$*, si se cumple alguna de las siguientes condiciones:*
(a) $\langle \mathcal{A}_1, h_1 \rangle$ *es un derrotador propio para* $\langle \mathcal{A}_2, h_2 \rangle$*, o*
(b) $\langle \mathcal{A}_1, h_1 \rangle$ *es un derrotador de bloqueo para* $\langle \mathcal{A}_2, h_2 \rangle$*, o*
(c) $\langle \mathcal{A}_1, h_1 \rangle$ *es un ataque a una suposición de* $\langle \mathcal{A}_2, h_2 \rangle$*.*

Ejemplo 6.4 Continuando con el ejemplo 6.1, puede construirse la estructura de argumento $\langle \mathcal{A}, buscar\_a(popeye) \rangle$, donde

$$\mathcal{A} = \left\{ \; buscar\_a(popeye) \prec perdido\_en\_el\_mar(popeye), not\ muerto(popeye) \; \right\}$$

Sin embargo, $\langle \mathcal{A}, buscar\_a(popeye) \rangle$ es atacada en la suposición, por el derrotador $\langle \mathcal{B}, muerto(popeye) \rangle$

$$\mathcal{B} = \left\{ \begin{array}{l} muerto(popeye) \prec \sim toma\_agua(popeye) \\ \sim toma\_agua(popeye) \prec perdido\_en\_el\_mar(popeye) \end{array} \right\}$$

Por su parte, la estructura de argumento $\langle \mathcal{B}, muerto(popeye) \rangle$, tiene un derrotador propio en el literal $\sim toma\_agua(popeye)$: $\langle \mathcal{C}, toma\_agua(popeye) \rangle$

$$\mathcal{C} = \left\{ \begin{array}{l} toma\_agua(popeye) \prec perdido\_en\_el\_mar(popeye), junta\_lluvia(popeye) \\ junta\_lluvia(popeye) \prec marino\_experimentado(popeye) \end{array} \right\}$$

Como $\langle \mathcal{C}, toma\_agua(popeye) \rangle$ no está derrotada, entonces $\langle \mathcal{B}, muerto(popeye) \rangle$ está derrotado, y el literal *not* $muerto(popeye)$ puede asumirse sin problemas. De esta manera, el literal "$buscar\_a(popeye)$" queda garantizado.

En el ejemplo 6.4 puede verse claramente que en DeLP extendida, aunque un literal p tenga una derivación rebatible, el literal extendido *not* p puede asumirse exitosamente, si p no está garantizado.

El siguiente resultado muestra que en DeLP la negación cumple con el *principio de coherencia* establecido en [AP93]:

Proposición 6.1 *Si "$\sim p$" está garantizado, entonces puede asumirse "not p".*
Demostración: si "$\sim p$" está garantizado, entonces "p" no está garantizado, y por lo tanto puede asumirse "*not* p". □

Es fácil ver que la recíproca no se cumple, esto es, si puede asumirse "*not* p" porque p no está garantizado, entonces no necesariamente $\sim p$ está garantizado. En el ejemplo 6.1 "*not* $muerto(popeye)$" pudo asumirse, pero sin embargo "$\sim muerto(popeye)$" no está garantizado.

6.1.4 Semántica Operacional de DeLP extendida

Al igual que en el capítulo 4, presentaremos la semántica operacional de DeLP extendida, especificando en PROLOG cada uno de lo elementos que fueron extendidos, e incorporados en la sección anterior.

6.1.5 Construcción de argumentos extendidos

En la Figura 6.1 se muestra la especificación de la construcción de argumentos considerando reglas rebatibles con negación default. Se incluye solamente el predicado `argument/3`, ya que los demás no necesitan ser modificados.

Obsérvese que en el predicado `argument/3`, sólo se realizaron dos cambios. Se agregó la segunda línea donde se ignora a los literales extendidos con negación default, con el mismo tratamiento que se le dá al átomo `true`. También se agregó una llamada al predicado `verify_assumptions/2`, que como muestra su especificación, verifica la nueva condición que se incorpora a la definición de argumento. Los predicados auxiliares `bodies/2`, y `assumptions/2` son especificados en la Figura 6.3 más adelante.

6.1.6 Ataque a una suposición

A fin de incluir el nuevo tipo de ataque, la especificación del predicado `find_defeater/3` de la Figura 4.5, debe cambiarse como muestra la Figura 6.2 El predicado `find_attack2assumption/3`, es el encargado de armar una lista con los nuevos puntos de ataque en los literales extendidos con negación default, y luego `attack_assumptions/3`

```
argument(true,D,D) :-!.
argument((not Lit),D,D) :- !.                          %%% DEFAULT NEGATION
argument((A,B),CurrentDerivation,ABDerivation) :-!,
         argument(A,CurrentDerivation,ADerivation),
         argument(B,ADerivation,ABDerivation).
argument(H,D,D,Line) :- temporary_fact(H,D), !.
argument(H,CurrentDerivation,Argument) :-
      (
         (H <- B), Rule = (H <- B)            %   there is a strict rule
       ;                                      % or
         (H -< B), Rule = (H -< B)            %   there is a defeasible rule
 ;                                    % or
         let_prolog_prove(H), Rule=(H <-true) %   is a built in
      ),
argument(B,CurrentDerivation,BodyDerivation,Line),
      Argument = [Clause|BodyDerivation],
verify_concordancy(H,Argument,Line).
verify_assumptions(H,Argument).

verify_assumptions(H,Derivation):-
          bodies(Derivation,Bodies), % collect bodies of defeasible rules
          assumptions(Bodies,Assump),% collect assumptions "not L" from bodies
          not member(not H,Assump).  % check if "not H" was assumed
```

Figura 6.1: Especificación de la construcción de estructuras de argumentos

recorre dicha lista buscando derrotadores. Los predicados auxiliares `bodies/2`, y `assumptions/2` son especificados en la Figura 6.3 más adelante.

6.1.7 Los dos tipos de negación son necesarios

En esta sección se analizará si ambos tipos de negación son necesarios, esto es, si es posible definir una en términos de la otra. Algunos investigadores como [KMD94, DK95, WYY97], han tratado de definir la negación default en términos de la negación fuerte. A

```
find_defeater(A,C,ArgLine):-              % C is a defeater (proper or blocking)
   (
   find_counterargument(A,C,SubA,ArgLine),% if C counterargues A attacking SubA
   \+ better(SubA,C)                      % and SubA is not better than C
  ;                                       % or
    find_attack2assumption(A,C,ArgLine)   % C attacks a default literal
   ).
find_attack2assumption(A,arg(C,Q),ArgLine):-
       bodies(A,Bodies),
       assumptions(Bodies,Assump),
       attack_assumption(Assump,arg(C,Q),ArgLine).
attack_assumption([not P|_],arg(C,P),Line):- find_argument(P,C,Line).
attack_assumption([_|Ps],C,L):-attack_assumption(Ps,C,L).
```

Figura 6.2: Especificación de ataque a una suposición

continuación se resumen las transformaciones propuestas y algunos contra-ejemplos que muestran que la transformación falla.

En el trabajo "Logic Programming without Negation as failure" [KMD94, DK95], se utiliza una relación de prioridad entre las reglas de programa. Las reglas de programa tienen una etiqueta (r_i) y las prioridades se indican referenciando a las reglas a través de dichas etiquetas. La forma que proponen de remover la negación default es utilizando la siguiente transformación. Una regla:

$$r_0 : p \leftarrow q, not\ s$$

se transforma en dos reglas:

$$r_1 : p \leftarrow q$$
$$r_2 : \sim p \leftarrow s$$
$$\text{con } r_1 < r_2.$$

La idea es que cuando el literal s no es derivable, la regla r_2 no pueda usarse, y por lo tanto exista una derivación para p. Pero que cuando s sea derivable, p no sea derivable, debido a que la regla r_2 (que ahora puede usarse), bloquea a r_1 ya que $r_1 < r_2$. En

```
bodies([],[]).   % Collects body rules in a list
bodies([d_rule(_,Body)|Clauses],[Body|Bs]):- bodies(Clauses,Bs).
bodies([s_rule(_,Body)|Clauses],[Body|Bs]):-bodies(Clauses,Bs).

assumptions([],[]). % Collects  assumed literals "not L"
assumptions([(A,B)|Rest],Points):- assumptions([B|Rest],R),
                                   ( A=not _,  \+ member(A,R), Points=[A|R],!
                                  ;
                                    Points=R ).
assumptions([A|Rest],Points):- A\=(_,_), assumptions(Rest,R),
                                   ( A=not _,  \+ member(A,R), Points=[A|R],!
                                  ;
                                   Points=R ).
```

Figura 6.3: Predicados auxiliares

principio parece que la transformación funciona correctamente, pero el problema es que cuando s es derivable, el literal $\sim p$, que no estaba presente en el programa original, ahora también es derivable.

La aparición de este nuevo literal, podría causar que otras derivaciones sean bloqueadas. Esto puede verse claramente en DeLP. Considérese el siguiente ejemplo, donde el programa $\mathcal{P}$ a continuación es trasformado a $\mathcal{P}'$, utilizando la transformación citada. Por simplicidad, la prioridad entre las reglas es simulada utilizando el criterio de especificidad, agregando el literal q a la nueva regla

$\mathcal{P}$	$\mathcal{P}'$
$p -\prec q, not\ r$	$p -\prec q$
	$\sim p -\prec r, q$
q	q
r	r
$\sim h -\prec \sim p$	$\sim h -\prec \sim p$
$a -\prec h$	$a -\prec h$
$h -\prec t$	$h -\prec t$
t	t

A partir del programa $\mathcal{P}$, el literal p no está garantizado, pero tampoco el literal $\sim p$. Pero notesé, que el literal a sí está garantizado. Sin embargo, en el programa $\mathcal{P}'$ producto de la transformación, el literal $\sim p$ está garantizado, y el literal a no lo está.

En el trabajo de Wang, You y Yuan, "Logic Programing without Negation as failure Revisited" [WYY97], reportan otros proplemas de la transformación anterior, y presentan una nueva transformación. La regla "$p \leftarrow not\ q$" se tranforma a "$p \leftarrow \overline{q}$", donde "$\overline{q}$" es un nuevo símbolo en la signatura del programa. Además, para todo literal p del programa, se generan dos nuevas reglas: "$r_1 : p \leftarrow p$" y "$r_2 : \overline{p}$", con $r_2 < r_1$. Referimos al lector interesado en los detalles de esta transformación al trabajo original en [WYY97].

A continuación se muestra un ejemplo donde un programa lógico rebatible $\mathcal{P}$, es transformado a $\mathcal{P}'$, según lo explicado por [WYY97]:

$\mathcal{P}$	$\mathcal{P}'$
$p -\!\prec not\ q$	$p -\!\prec \overline{q}$
	$\overline{q}$
	$\overline{p}$
	$p \leftarrow p$
	$q \leftarrow q$

Es fácil ver, que en los programas transformados, se derivan nuevos literales, con la problemática que ya fue mencionada antes. Además una regla como "$p \leftarrow p$" podría generar ciclos durante el proceso de derivación.

Otra posibilidad sugerida para DeLP, fue la de reemplazar todo literal extendido $not\ q$, por una *presuposición* "$\sim q -\!\prec$ ", como se muestra en los programas $\mathcal{P}_1$ y $\mathcal{P}_2$ mas abajo.

$\mathcal{P}_1$	$\mathcal{P}_2$
$p -\!\prec not\ q$	$p -\!\prec \sim q$
	$\sim q -\!\prec$

De esta manera, mientras q no esté garantizado, el literal p estará garantizado, (por ejemplo ante la presencia de q). Sin embargo, nuevamente un nuevo elemento, inexistente en el programa original (en este caso $\sim q$), puede ser garantizado en el programa transformado $\mathcal{P}_2$.

Finalmente, otra posibilidad sugerida fue reemplazar "$p -\!\prec not\ q$" con tres reglas: "$p -\!\prec$ ", "$\sim p -\!\prec q$" and "$p -\!\prec q$". Sin embargo, esta transformación también generaría

nuevos literales que no estaban presentes en el programa original, con los riesgos ya considerados.

Los casos analizados muestran que las transformaciones que incorporan nuevas reglas, o nuevos literales a un programa, producen efectos colaterales problemáticos.

6.2 Extendiendo DeLP con presuposiones

Un programa lógico rebatible $\mathcal{P}= (\Theta, \Omega, \Delta)$ está compuesto por un conjunto de hechos Θ, un conjunto de reglas estrictas Ω, y un conjunto de reglas rebatibles Δ. Tanto el cuerpo de las reglas estrictas, como el de las rebatibles está formado por un conjunto no vacío de literales. Un hecho, puede pensarse como una regla estricta sin cuerpo. Sin embargo, la sintaxis de DeLP no considera la representación de reglas rebatibles sin cuerpo.

Uno de los primeros investigadores en considerar reglas rebatible sin cuerpo fue Donald Nute [Nut87], quien las llamó *presuposiciones*[3]. Actualmente las presuposiciones son utilizadas en muchos formalismos de argumentación rebatible.

Extenderemos a continuación la sintaxis de la Programación en Lógica Rebatible, para permitir la representación de presuposiciones. De la misma forma que el símbolo " $-\prec$ " es lo que distingue sintácticamente a una regla rebatible de una estricta, en DeLP, una presuposicón será representada con un literal seguido del símbolo " $-\prec$ ", para diferenciarla sintácticamente de un hecho.

Definición 6.7 (Presuposición)
Una presuposición se denotará "$L -\prec$ ", donde L es un literal.

Asi como una regla rebatible "$a -\prec b$" expresa que "*creer en b provee razones para creer en a*" [SL92], representando una conexión débil entre b y a. Una presupoción "$p -\prec$ " expresa que "existen razones para creer en p", con lo cual representa una afirmación mucho más débil que un hecho. Según Nute "$p -\prec$ " también puede leerse como "presumiblemente p".

A continuación figuran ejemplos de presuposiciones:

[3]En inglés *presumptions*

$$empresa\_riesgosa(acme) \prec$$
$$buena\_persona(pepe) \prec$$
$$cambiará\_el\_viento \prec$$
$$inocente(acusado) \prec$$
$$pingüino(petete) \prec$$

Extendemos a continuación la definición de Programa Lógico Rebatible para considerar las presuposiciones.

Definición 6.8 (Programa Lógico Rebatible)
Un programa lógico rebatible será entonces un conjunto $\mathcal{P}$, posiblemente infinito, de hechos, reglas estrictas, reglas rebatibles, y presuposiciones. En un programa $\mathcal{P}$ identificaremos con Θ al conjunto de hechos, con Ω al conjunto de reglas estrictas, con Δ al conjunto de reglas rebatibles, y con Φ al conjunto de presuposiciones. Por conveniencia, también denotaremos con Π al conjunto $\Theta \cup \Omega$, y con Δ^+ al conjunto $\Delta \cup \Phi$. Por lo tanto, en algunas ocasiones denotaremos a un programa lógico rebatible $\mathcal{P}$=(Π, Δ^+), y en otros casos $\mathcal{P}$=$(\Theta, \Omega, \Delta, \Phi)$.

Las presuposiciones representan una afirmación mucho más débil que un hecho, y puede también pensarse en ellas como "hechos rebatibles". Extenderemos ahora la definición de derivación rebatible para considerar presuposiciones.

Definición 6.9 (Derivación Rebatible de un literal)
Sea $\mathcal{P}$= (Π, Δ^+) un programa y L un literal. Una derivación rebatible para L a partir de $\mathcal{P}$, consiste de una secuencia finita de literales $L_1, L_2, \ldots, L_n = L$, provisto de que exista una secuencia de reglas estrictas o rebatibles $R_1, R_2, \ldots, R_n$ del programa $\mathcal{P}$, de tal forma que para cada literal L_i en la secuencia, tenemos que:

(a) L_i es un hecho, o

(b) L_i es una presuposición, o

(c) existe en $\mathcal{P}$ una regla R_i con cabeza L_i y cuerpo $B_1, B_2, \ldots, B_k$ donde todo literal B_j del cuerpo ($1 \leq\ j \leq k$) es un elemento de la secuencia que precede a L_i.

La noción de estructura de argumento también debe extenderse para considerar las poresuposiciones.

Definición 6.10 (Estructura de argumento)
Sea h un literal y $\mathcal{P}$=(Π, Δ^+) un programa lógico rebatible, una estructura de argumento *para h es un par $\langle \mathcal{A}, h \rangle$, donde $\mathcal{A}$ es conjunto de reglas rebatibles y presuposiciones de Δ^+, tal que:*

1. *existe una derivación rebatible para h a partir de $\Pi \cup \mathcal{A}$.*
2. *$\Pi \cup \mathcal{A}$ es no contradictorio, y*
3. *$\mathcal{A}$ es minimal, es decir, no existe un subconjunto propio $\mathcal{A}'$ de $\mathcal{A}$ tal que $\mathcal{A}'$ satisface las condiciones (1) y (2).*

Las presuposiciones aumentan la capacidad de representar conocimiento en DeLP, como se muestra en el ejemplo 6.5 a continuación.

Ejemplo 6.5 Considere el siguiente programa:

$$\mathcal{P}_{6.5} = \left\{ \begin{array}{l} comprar\_acciones(T) -\!\prec subirán\_las\_acciones(T) \\ \sim comprar\_acciones(T) -\!\prec empresa\_riesgosa(T) \\ subirán\_las\_acciones(T) -\!\prec saca\_nuevo\_producto(T) \\ empresa\_riesgosa(T) -\!\prec huelga\_empleados(T) \\ huelga\_empleados(acme) -\!\prec \\ saca\_nuevo\_producto(acme) -\!\prec \end{array} \right\}$$

Apartir del programa anterior es posible construir las estructuras de argumento $\langle \mathcal{A}, comprar\_acciones(acme) \rangle$ y $\langle \mathcal{B}, \sim comprar\_acciones(acme) \rangle$, donde:

$$\mathcal{A} = \left\{ \begin{array}{l} comprar\_acciones(acme) -\!\prec subirán\_las\_acciones(acme) \\ subirán\_las\_acciones(acme) -\!\prec saca\_nuevo\_producto(acme) \\ saca\_nuevo\_producto(acme) -\!\prec \end{array} \right\}$$

$$\mathcal{B} = \left\{ \begin{array}{l} \sim comprar\_acciones(acme) -\!\prec empresa\_riesgosa(acme) \\ empresa\_riesgosa(acme) -\!\prec huelga\_empleados(acme) \\ huelga\_empleados(acme) -\!\prec \end{array} \right\}$$

En este caso, cada estructura es un derrotador de bloqueo de la otra.

Las demás nociones del formalismo de argumentación rebatible, utilizan a la noción de estructura de argumento como primitiva, y por lo tanto no es necesiaro extenderlas. No obstante, la comparación entre argumentos merece un análisis detallado. La siguiente definición será de utilidad en lo que resta de las sección.

Definición 6.11 (Estructura basada en hechos, o basada en presuposiciones) *Una estructura de argumento estará basada en hechos, si al menos un hecho fue utilizado en su derivación rebatible. De lo contrario, si se utilizaron sólo presuposiciones, se dirá que está basada en presuposiciones.*

Cuando se comparan dos argumentos, se deberían preferir argumentos que se basan en hechos, a los que se basan en presuposiciones. Como se mostró en la sección 2.6 una alternativa para comparar argumentos es definir un órden entre reglas rebatibles del programa. Sin embargo, un orden entre reglas rebatibles no soluciona el problema anterior, ya que no permite expresar que un hecho es mejor que una presuposición.

La otra alternativa para comparar argumentos, sugerida en la sección 2.6, es el criterio de especificidad generalizada (definición 2.20). No obstante, como fue mostrado en [Gar97], la definición 2.20 tampoco se comporta adecuadamente cuando una estructura de argumento contiene presuposiciones. Por ejemplo, si en el programa $\mathcal{P}_{6.5}$ la presuposición "$huelga\_empleados(acme) -\!\prec$" se reemplaza por el hecho "$huelga\_empleados(acme)$", entonces debería ocurrir que la estructura $\langle \mathcal{B}, \sim comprar\_acciones(acme) \rangle$ sea preferida, ya que esta se sustenta en información segura (el hecho), mientras que $\langle \mathcal{A}, comprar\_acciones(acme) \rangle$ se sustenta en una afirmación débil (la presuposición). Lamentablemente, la definición 2.20 no captura esta diferencia, y establece que ningún argumento es mejor que el otro. Con el fin de mantener esta sección autocontenida, incluimos a continuación el análisis y las soluciones encontradas en [Gar97].

La Figura 6.4 muestra un conjunto de casos utilizados en [Gar97] para analizar el comportamiento del criterio de especificidad en argumentos con presuposiciones. En dicha figura, cada recuadro representa un caso diferente donde se muestran dos estructuras de argumento[4] a ser comparadas. Cada caso corresponde a un programa lógico rebatible diferente $\mathcal{P}_i = (\Pi, \Delta^+{}_i)$, donde el conjunto de hechos $\Pi = \{h, a\}$, es el mismo para todos,

[4]Para simplificar la figura sólo se muestra el conjunto de reglas rebatibles de cada estructura de argumento.

$\mathcal{A}_1 = \{ \quad c_1 \prec h \ \}$ $\mathcal{B}_1 = \{ \quad \sim c_1 \prec \ \}$	$\mathcal{A}_2 = \{ \quad c_2 \prec a, h, s$ $s \prec \ \}$ $\mathcal{B}_2 = \{ \quad \sim c_2 \prec a\}$	$\mathcal{A}_3 = \{ \quad c_3 \prec s, h$ $s \prec \ \}$ $\mathcal{B}_3 = \{ \quad \sim c_3 \prec h \ \}$
$\mathcal{A}_4 = \{ \quad c_4 \prec s, h$ $s \prec \ \}$ $\mathcal{B}_4 = \{ \quad \sim c_4 \prec s$ $s \prec \ \}$	$\mathcal{A}_5 = \{ \quad c_5 \prec s, r$ $s \prec$ $r \prec \ \}$ $\mathcal{B}_5 = \{ \quad \sim c_5 \prec r$ $r \prec \ \}$	$\mathcal{A}_6 = \{ \quad c_6 \prec h \ \}$ $\mathcal{B}_6 = \{ \quad \sim c_6 \prec s, r$ $s \prec$ $r \prec \ \}$
$\mathcal{A}_7 = \{ \quad c_7 \prec \ \}$ $\mathcal{B}_7 = \{ \quad \sim c_7 \prec s$ $s \prec \ \}$	$\mathcal{A}_8 = \{ \quad c_8 \prec s$ $s \prec \ \}$ $\mathcal{B}_8 = \{ \quad \sim c_8 \prec q$ $q \prec s$ $s \prec \ \}$	$\mathcal{A}_9 = \{ \quad c_9 \prec p$ $p \prec h \ \}$ $\mathcal{B}_9 = \{ \quad \sim c_9 \prec \ \}$
$\mathcal{A}_{10} = \{ \quad c_{10} \prec h\}$ $\mathcal{B}_{10} = \{ \quad \sim c_{10} \prec s$ $s \prec \ \}$	$\mathcal{A}_{11} = \{ \quad c_{11} \prec p$ $p \prec h \ \}$ $\mathcal{B}_{11} = \{ \quad \sim c_{11} \prec s$ $s \prec \}$	$\mathcal{A}_{12} = \{ \quad c_{12} \prec \ \}$ $\mathcal{B}_{12} = \{ \quad \sim c_{12} \prec \ \}$
$\mathcal{A}_{13} = \{ \quad c_{13} \prec h\}$ $\mathcal{B}_{13} = \{ \quad \sim c_{13} \prec p, q$ $p \prec s \quad q \prec r$ $s \prec$ $r \prec \ \}$	$\mathcal{A}_{14} = \{ \quad c_{14} \prec h\}$ $\mathcal{B}_{14} = \{ \quad \sim c_{14} \prec p$ $p \prec s$ $s \prec \ \}$	$\mathcal{A}_{15} = \{ \quad c_{15} \prec \ \}$ $\mathcal{A}_{15} = \{ \quad \sim c_{15} \prec s, r$ $s \prec$ $r \prec \ \}$

Figura 6.4: Ejemplos de argumentos con presuposiciones

mientras que el conjunto $\Delta^+{}_i$ difiere, y está formado por las reglas de los argumentos de cada recuadro. Por ejemplo, el programa lógico rebatible del primer caso es $\mathcal{P}_1 = (\Pi, \Delta^+{}_1)$, con $\Delta_1 = \{(c_1 \prec h), (\sim c_1 \prec) \}$.

En la figura 6.5 se muestra el resultado esperado para cada uno de los ejemplos de la figura 6.4, y el resultado que se obtiene al utilizar la definición 2.20 de especificidad generalizada. Para simplifica la figura 6.5, si en una columna "i" figura "$\succ$", significa que $\mathcal{A}_i \succ \mathcal{B}_i$. Por por ejemplo, en la columna "3" figura $\succ$, y $\not\approx$, indicando que el resultado esperado es $\mathcal{A}_3 \succ \mathcal{B}_3$, mientras que el resultado de aplicar la definición 2.20 es $\mathcal{A}_3 \not\approx \mathcal{B}_3$.

La definición de especificidad generalizada será extendida entonces para que se com-

Casos de la Figura 6.4:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Resultado esperado	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$
Resultaso con Def. 2.20	$\succ$	$\succ$	$\not\approx$	$\succ$	$\not\approx$	$\succ$	$\not\approx$	$\not\approx$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\not\approx$

Figura 6.5: Resultados esperados y resultados obtenidos con la definición2.20

porte adecuadamente con argumentos que utilizan presuposiciones. La siguiente noción será utilizada en la nueva definición.

Definición 6.12 *Dada una estructura de argumento $\langle \mathcal{A}, h\rangle$, denotaremos con $RR(\mathcal{A})$, al conjunto de reglas rebatibles de $\mathcal{A}$. Por lo tanto, $RR(\mathcal{A})$ no contiene presuposiciones.*

Por ejemplo, considerando algunas de las estructuras de argumento de la Figura 6.4, $RR(\mathcal{A}_2) = \{\ c_2 -\!\prec a, h, s\}$, $RR(\mathcal{A}_1) = \mathcal{A}_1$, $RR(\mathcal{B}_1) = \emptyset$.

Observación 6.1 *Dada una estructura de argumento $\langle \mathcal{A}, h\rangle$, $RR(\mathcal{A})$ puede ser vacío si $\mathcal{A}$=$\emptyset$ (derivación estricta), o $\mathcal{A}\neq \emptyset$ y $\mathcal{A}$ sólo contiene presuposiciones.*

Definición 6.13 (Especificidad generalizada extendida) *Sea $\mathcal{P}$=$(\Theta, \Omega, \Delta, \Phi)$ un programa lógico rebatible con presuposiciones, donde Ω es el conjunto de reglas strictas. Sea $\mathcal{F}$ el conjunto de literales que tienen una derivación rebatible a partir de $\mathcal{P}$. Una estructura de argumento $\langle \mathcal{A}_1, h_1\rangle$ es* estrictamente más específica *que otra estructura de argumento $\langle \mathcal{A}_2, h_2\rangle$ (denotado $\langle \mathcal{A}_1, h_1\rangle \succ \langle \mathcal{A}_2, h_2\rangle$) si se verifica alguna de las siguientes tres condiciones:*

(a) *$RR(\mathcal{A}_1) \neq \emptyset$, $RR(\mathcal{B}_2) \neq \emptyset$ y se cumplen las dos condiciones siguientes:*

1. *para todo conjunto $H \subseteq \mathcal{F}$,*
   si existe una derivación rebatible para h_1 a partir de $\Omega \cup H \cup RR(\mathcal{A}_1)$,
   y no existe una derivación estricta para h_1 a partir de $\Omega \cup H$,
   entonces existe una derivación rebatible para h_2 a partir de $\Omega \cup H \cup RR(\mathcal{A}_2)$,
2. *existe al menos un conjunto $H' \subseteq \mathcal{F}$ tal que:*
   existe una derivación rebatible para h_2 a partir de $\Omega \cup H' \cup RR(\mathcal{A}_2)$,
   no existe una derivación estricta de h_2 a partir de $\Omega \cup H'$, y
   no existe una derivación rebatible de h_1 a partir de $\Omega \cup H' \cup RR(\mathcal{A}_1)$.

(*b*) $\langle \mathcal{A}_1, h_1 \rangle$ *está basada en hechos, y* $\langle \mathcal{A}_2, h_2 \rangle$ *está basada en presuposciones.*

(*c*) *Las estructuras* $\langle \mathcal{A}_1, h_1 \rangle$ *y* $\langle \mathcal{A}_2, h_2 \rangle$ *están ambas basadas en presuposicones, y se cumple* $RR(\mathcal{A}_1) = \emptyset$ *y* $RR(\mathcal{B}_2) \neq \emptyset$*,*

Como pueder verse, en la nueva versión de especificidad de la definición 6.13, la condición (a) corresponde exactamente a la definición 2.20 donde se utiliza $RR(\mathcal{A})$ en lugar de $\mathcal{A}$, para evitar que las presuposiciones generen un efecto no deseado. Obsérvese que la condición $RR(\mathcal{A}_1) \neq \emptyset$,y $RR(\mathcal{A}_2) \neq \emptyset$, garantiza que los conjunto de reglas rebatibles no sean vacíos, lo cual afectaría de forma no deseada al resultado.

En los casos en los cuales la condición (a) falle, con lo cuál $\langle \mathcal{A}_1, h_1 \rangle$ no sería más específico que $\langle \mathcal{A}_2, h_2 \rangle$. La condición (b) permitirá preferir las estructuras de argumento basadas en hechos, antes que los basadas en presuposiciones. Por ejemplo, en el caso "1" de la figura 6.4, $RR(\mathcal{B}_1) = \emptyset$, y por lo tanto la condición (a) no puede utilizarse, sin embargo, por la condición (b), $\mathcal{A}_1$ es más específico que $\mathcal{B}_1$ porque el primero está basado en hechos, y el segundo en presuposiciones. En el caso "10" de la figura 6.4, según la condición (a) ningún argumento es más específico que el otro, sin embargo, por la condición (b), $\mathcal{A}_{10}$ es más específico que $\mathcal{B}_{10}$ porque el primero está basado en hechos, y el segundo en presuposiciones.

La condición (c), permite diferenciar entre dos estructuras basadas en presuposiciones, donde será preferida aquella que sólo tenga presuposiciones. Por ejemplo, en el caso "7" de la figura 6.4, $RR(\mathcal{A}_7) = \emptyset$ y por lo tanto la condición (a) no puede utilizarse, pero además los dos argumentos son basados en presuposiciones, con lo cuál la condición (b) tampoco es aplicable, sin embargo, por la condición (c), $\mathcal{A}_7$ es más específico que $\mathcal{B}_7$ porque $RR(\mathcal{A}_7) = \emptyset$ y $RR(\mathcal{A}_7) \neq \emptyset$.

La Figura 6.6 presenta los resultados de los casos de la Figura 6.4 con la nueva definición.

Casos de la Figura 6.4:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Resultado esperado	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$
Resultaso con Def. 6.13	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$	$\not\approx$	$\succ$	$\succ$	$\succ$

Figura 6.6: Resultados obtenidos con la definición 6.13

6.3 Conclusiones

El uso de la negación es necesario en cualquier sistema de representación de conocimiento. Es este capítulo se definió la "*Programación en Lógica Rebatible Extendida* que permite utilizar dos tipos de negación: la negación fuerte que permite representar conocimiento contradictorio, y la negación default que permite representar información incompleta. A diferencia de desarrollos anteriores [SG95, Gar97, GS99c], en esta tesis se define a la negación default como una nueva forma de ataque dentro del formalismo argumentativo.

En este capítulo se motró además como puede extenderse fácilmente la Programación en Lógica Rebatible para permitir la representación de presuposiciones. Solamente fueron modificadas las definiciones de "derivación rebatible" y "estructura de argumento". Se incluyó además una extensión del criterio de especificidad generalizada, para comparar argumentos que poseen presuposiciones. Las demás nociones del formalismo de argumentación rebatible, utilizan a la noción de estructura de argumento como primitiva, y por lo tanto no necesitan ser extendidas.

Capítulo 7

Comparación con otros formalismos

La Programación en Lógica Rebatible está en la confluencia de dos áreas independientes: la Argumentación Rebatible, y la Programación en Lógica. Por este motivo, los trabajos que se comentarán a continuación provienen de ambas áreas. En la primera parte del capítulo se describirán sistemas de Argumentación Rebatible, y en la segunda parte, extensiones de la Programación en Lógica similares a nuestra propuesta. No es objetivo de este capítulo realizar un *survey* sobre argumentación rebatible, sino simplemente describir y comparar aquellos formalismos cercanos a DeLP. Referimos al lector intersado a dos excelentes trabajos que resumen el estado del arte en argumentación rebatible [CML99, PV00].

7.1 Razonamiento Rebatible (John Pollock)

John Pollock fue uno de los primeros investigadores en desarrollar un formalismo de argumentación rebatible [Pol70, Pol74, Pol87, Pol91b, Pol91a, Pol92b, Pol92a, Pol95, Pol96b]. Originalmente desarrolló su teoría como una contribución en epistemología, y luego se orientó a la inteligencia artificial, desarrollando finalmente un sistema en Lisp que implementa su teoría, llamado OSCAR [Pol95]. Como la mayoría de los formalismos de argumentación rebatible, define las nociones de argumento, ataque entre argumentos, comparación, derrotador, y un proceso para determinar las conclusiones garantizadas. Sin embargo, como se mostrará a continuación, OSCAR es sustancialmente diferente a DeLP.

En OSCAR el lenguaje subyacente es la lógica de primer orden. Los argumentos se definen como secuencias de *razones*, las cuales pueden ser *conclusivas* o *prima facie*, siendo estas últimas las que hacen rebatible a un argumento. Un argumento basado en un conjunto de hechos INPUT, es una secuencia $\sigma_1, \ldots, \sigma_n$, donde cada elemento σ_i es una tupla $\langle X_i, p_i, v_i \rangle$ cuya primer componete X_i es un conjunto de suposiciones, p_i es una proposición, y v_i es el *grado de justificación* del argumento en ese punto de la secuencia. Para formar un argumento, un elemento puede ser agregado a la secuencia utilizando diferentes reglas de formación. Por ejemplo, considerando las siguientes razones:
(1) $bird(X)$ es una razón *prima facie* de fuerza v para $flies(X)$
(2) $bat(X)$ es una razón *prima facie* de fuerza u para $flies(X)$
y el conjunto INPUT= { bird(t) $\vee$ bat(t) }, la conclusión $flies(t)$ puede obtenerse rebatiblemente mediante el siguiente argumento:

1.	$\langle \emptyset, bird(t) \vee\ bat(t), \infty \rangle$	$bird(t) \vee\ bat(t) \in$ INPUT
2.	$\langle \{bird(t)\}, bird(t), \infty \rangle$	suposición
3.	$\langle \{bird(t)\}, flies(t), v \rangle$	usando 2 y la razón *prima facie* (1)
4.	$\langle \{bat(t)\}, bat(t), \infty \rangle$	suposición
5.	$\langle \{bat(t)\}, flies(t), u \rangle$	usando 4 y la razón *prima facie* (2)
6.	$\langle \emptyset, flies(t), min\{v, u\} \rangle$	de 3 y 5

Pollock introduce dos tipos diferentes de derrotador: *rebutting defeater* y *undercutting defeater*, el primero coincide con el concepto de derrotador definido en el capítulo 2, donde se utiliza la negación fuerte para encontrar los conflictos, el segundo tipo, es un concepto de derrotador que ataca a una regla de formación del argumento y no a un literal. Los argumentos son comparados utilizando *probabilidades* asociadas. El punto débil en el uso de las probabilidades, como en muchos otros sistemas, es que en la práctica resulta difícil asociar probabilidades a hechos, reglas y argumentos, y como propagar las probabilides durante la argumentación.

Para evaluar el estado de los argumentos, OSCAR utiliza un grafo, llamado *inference graph*, donde los argumentos bajo análisis son relacionados. Según Pollock, el razonamiento humano es rebatible en dos sentidos diferentes: "*synchronically defeasible*", cuando una conclusión puede no estar garantizada relativo a un número grande de datos (inputs), y "*diachronically defeasible*" cuando una conclusión puede retractarse, simplemente como el resultado de más razonamiento, sin que existan nuevos datos (inputs). Por lo tanto,

en OSCAR un argumento puede estar 'justificado' en un momento del proceso de razonamiento, e 'injustificado' más tarde, sin que exista nuevo conocimiento agregado en el análisis. Sin embargo, un argumento está 'garantizado' cuando el razonador llega a un estado en el cual el argumento parmanecerá sin derrotarse. En OSCAR se utiliza un procedimiento 'bottom-up' para computar argumentos justificados y garantizados.

Los elementos enunciadas hasta aqui hacen muy difícil la comparación del trabajo de Pollock con DeLP, sin embargo el proceso de análisis del status de los argumentos que incluímos a continuación, tiene un gran parecido con nuestro concepto de árbol de dialéctica marcado.

La asignación de "derrotado" o "no derrotado" a un conjunto de argumentos S, es una *asignación parcial de status de derrota* si satisface las siguientes condiciones:

1. A todos los argumentos de S obtenidos sólo utilizando elementos del conjunto INPUT se les asigna el status "no derrotados".

2. A un argumento $\mathcal{A} \in S$ se le asigna el status "no derrotado", si y sólo si:
   (a) todos sus sub-argumentos propios tiene status "no derrotado" y
   (b) todos los argumentos de S que derrotan a $\mathcal{A}$ tienen el status "derrotado".

3. A un argumento $\mathcal{A} \in S$ se le asigna el status "derrotado", si y sólo si:
   (a) al menos uno de sus sub-argumentos propios tiene status "no derrotado", o
   (b) al menos un argumento de S que derrota a $\mathcal{A}$ tiene el status "no derrotado".

Una *asignación de status de derrota* se define entonces como una asignación parcial de status de derrota que es maximal con respecto a la inclusión de conjuntos.

Dado un conjunto de argumentos S la asignación de status de derrota no es única. Es facil de ver que si un conjunto S tiene dos argumentos que se derrotan mutuamente, entonces existen dos asignaciones de status. Pollock define una noción de consecuencia escéptica de la siguiente manera: sea S un conjunto de argumentos y $\mathcal{A} \in S$. Entonces, relativo al conjunto S,

1. un argumento $\mathcal{A}$ es "no derrotado", si toda asignación de status posible le asigna a $\mathcal{A}$ el status "no derrotado".

2. un argumento $\mathcal{A}$ es "definitivamente derrotado"[1], si ninguna asignación de status le asigna a $\mathcal{A}$ el status "no derrotado".

3. un argumento $\mathcal{A}$ es "provisionalmente derrotado", si no se cumple ninguna de las dos anteriores, esto es, hay asignaciones de status que le asignan "derrotado" y otras que no.

Siguiendo la terminología de [PV00] el sistema de Pollock sigue la política de "*multiple status-assigments approach*". Esto representa una diferencia sustancial con respecto a DeLP, donde se utiliza la política de "*unique status-assigments approach* ". Un análisis más profundo sobre las diferencias entre las propuestas de asignaciones de status únicas y múltiples, puede verse en [PV00].

7.2 Argumentación Rebatible (Ron Loui)

En 1987 Ron Loui presentó el trabajo *Defeat among arguments: a system of defeasible inference* [Lou87] donde se introdujo un sistema de razonamiento no-monotónico con reglas rebatibles. Este sistema posee muchos conceptos que han sido heredados por DeLP.

En [Lou87], una regla rebatible se denota "$a \succ b$", y se interpreta como "a es una razón rebatible para b". El sistema formal está definido a partir de una base de datos $\langle EK, R\rangle$, donde EK es conocimiento evidencial, y R un conjunto de reglas rebatibles, con las cuales se forman argumentos. Un argumento para una sentencia P es un grafo dirigido acíclico con "raíz" P, donde los nodos están etiquetados con sentencias, la raíz está etiquetada con P y si N es un nodo interno que tiene por hijos a $H_1, \ldots, H_k$, entonces la regla rebatible $H_1, \ldots, H_k \succ N$ pertenece a R.

En el citado trabajo se presentan varias formas de comparar argumentos (grafos),

- A_1 usa más evidencia que A_2
- A_1 es más específico que A_2
- A_1 es más directo que A_2
- A_1 tiene premisas preferidas a las de A_2,.

[1] En inglés *defeated outright*

Estas relaciones permiten definir los siguientes conceptos:

- Un argumento A_1 interfiere con A_2, si sustenta conclusiones inconsistentes, y además A_1 usa más evidencia que A_2, o A_2 no usa más evidencia que A_1, pero se cumple que A_1 es más específico que A_2, o A_1 es más directo que A_2, o A_1 tiene premisas preferidas a las de A_2.

- Un argumento A_1 derrota a A_2 si A_1 interfiere con A_2, y no ocurre que A_2 interfiere con A_1.

- Un argumento es no-derrotado si no existe argumento que lo derrote.

- Un argumento A_1 justifica a una sentencia P si A_1 es no-derrotado, y para todo nodo interno N de A_1 tal que exista un argumento A_2 para T, tal que N y T sean inconsistentes, existe un argumento A_3 que derrota a A_2.

Obséverse que muchas de las relaciones son muy similares a las establecidas en DeLP, sin embargo, la noción de justificación sólo tiene tres niveles de argumentación. Este sistema fue luego extendido y modificado considerablemente en [SL92] donde el concepto de justificación se basa en el trabajo de Pollock [Pol87].

Posteriormente, en [Lou91a, Lou93b], Ron Loui investigó explícitamente a la dialéctica como fundamento formal del razonamiento no-monotónico y el razonamiento rebatible. En estos trabajos, las reglas que entran en conflicto se consideran como políticas. En [Lou93a] Ron Loui presenta cuatro "reinos" del razonamiento rebatible. En los dos primeros: "defeasible statistical reasoning" y " analogical reasoing" la derrotabilidad aparece porque nueva información se agrega al sistema. En los otros dos: "best explanations", y "decision-theoretic reasoning", la derrotabilidad es el resultado de una deliberación más profunda o más computación con la misma evidencia.

En [Lou94] introduce juegos de argumentación (argument games) para investigar como razonar con recursos limitados. Finalmente en los trabajos *Rationales and Argument Moves* [LN95], *Alchourrón and Von Wright on Conflict among Norms* [Lou97a], y *Progress on Room 5: A Testbed for Public Interactive Semi-Formal Legal Argumentation* [Lou97b], Ron Loui se orientó a capturar patrones de razonamiento en el ámbito legal.

7.3 Lógica Rebatible (Donald Nute)

En esta sección se presentan los sistemas formales desarrollados por Donald Nute en los trabajos [Nut86, Nut87, Nut92, Nut94], que llevaron al desarrollo de d-Prolog [Nut88, CNV97]. El lenguaje d-Prolog fue el primero en introducir un sistema de programación en lógica con reglas rebatibles, y utilizar el criterio de especificidad para decidir entre metas contradictorias.

La lógica rebatible de Nute no es un formalismo de *argumentación*, ya que no existe en esta lógica una noción de argumento. La noción de conflicto se define entre dos reglas con consecuentes contradictorios. A diferencia de DeLP en lugar de comparar argumentos para decidir entre literales contradictorios, en esta lógica se comparan las reglas del programa que están en conflicto. Para esto, la lógica necesita de un orden definido explícitamente entre las reglas. De esta manera una regla r_1 derrotará a otra r_2, si están en conflicto, y r_1 es mejor que r_2 de acuerdo al orden dado entre las reglas. Al no existir una noción de argumento, no es posible definir un análisis dialéctico entre argumentos, ni controlar la aceptabilidad de las líneas de argumentación. En [PV00] se muestran algunos problemas relacionados con esto último.

El lenguaje de la lógica rebatible descripta por Donald Nute tiene grandes similitudes con DeLP: utiliza la misma definición de literal, la negación fuerte "$\sim$", reglas estrictas $A \rightarrow p$, y reglas rebatiblels $A \Rightarrow p$ (donde A es un conjunto finito de literales y p un literal). Sin embargo, Nute incorpora un tercer tipo de regla "$A \rightsquigarrow p$", llamado "regla derrotador". Una regla derrotador "$A \rightsquigarrow p$" se interpreta como : "Si A entonces podría ser que p". El rol de las reglas derrotadores es solamente interferir con el proceso inferencia cuando se us un regla rebatible. Las reglas derrotadores nunca sostienen una inferencia directamente. Estas reglas permiten definir algo parecido a los *undercutting defeater* de Pollock.

En *Defeasible Reasoning: a Philosophical analysis in Prolog* [Nut88] Nute explica con mayor detalle el porqué de la inclusión de reglas derrotadores. Lo que intenta capturar es lo que en inglés se denominan *might conditionals*. Por ejemplo, "alquilar una casa implica obtener un beneficio económico", es una regla rebatible. Sin embargo, "alquilar una casa implica que podría tener problemas con los inquilinos " no es una regla rebatible, ya que sólo está indicando que podría darse el caso. Para este tipo de reglas es que introduce las reglas derrotadores, la regla anterior es usada para impedir un razonamiento, pero no para

producir una conclusión. Son reglas que juegan un rol puramente negativo. Obviamente la responsabilidad de elegir entre representar conocimiento con una regla rebatible o una regla derrotador, no es del sistema, sino de quien escriba las reglas.

Ejemplo 7.1 El ejemplo anterior de los inquilinos quedaría expresado de la siguiente forma:

$$alquilo\_casa \Rightarrow ingresa\_dinero$$
$$problemas\_casa \Rightarrow \sim ingresa\_dinero$$
$$malos\_inquilinos \rightsquigarrow problemas\_casa$$

La lógica rebatible de Nute ha evolucionado a lo largo de los años. Fue definida inicialmente en [Nut87], luego modificada en [Nut88], y finalemente generalizada a una familia de lógicas rebatibles en [Nut92, Nut94]. El lenguaje definido antes, es lo único que se ha mantenido sin cambios. Sin embargo, la teoría de prueba fue modificada sustancialmente en cada trabajo.

En [Nut87] Nute define una *Teoría rebatible* T como un par (K, R), donde K es un conjunto de sentencias y R un conjunto de reglas. Define una derivación estricta como una secuencia de únicamente reglas estrictas, y a una derivación rebatible, como una secuencia de reglas estrictas o rebatibles, sobre las cuales establece un serie de condiciones para capturar de alguna manera las nociones de argumento, y derrota.

Posteriormente en [Nut92], la teoría de prueba se basa en un conjunto K de literales en lugar de sentencias y las pruebas rebatibles son descriptas como árboles, donde cada nodo está rotulado con (1) una teoría rebatible y (2) una D-fórmula f seguida de un "+" cuando f es derivable, o un "−" cuando f es demostrablemente no derivable. En la sección siguiente mostramos una extensión del trabajo de Nute donde se presenta una formalización alternativa a dichos árboles de prueba.

Según Nute, la lógica no monótona de McDermott y Doyle, circunscripción de McCarthy, default logic de Reiter, y la lógica autoepistémica de Moore, no son lógicas rebatibles. Los trabajos de Pollock, Geffner y Pearl, y Loui, si representan propuestas de lógicas rebatibles para el razonamiento no monótono. En un trabajo posterior [Nut94] expande un poco más este tema diciendo:

"Dado el default $\frac{P\colon \mathbf{M}Q}{R}$, en toda extensión Σ que lo contenga, o bien $P \in \Sigma$, o $\sim Q \in \Sigma$, o $R \in \Sigma$. Pero no puede existir una extensión donde P es

verdadero, Q es consistente, pero no contenga a R. Por lo tanto los defaults de Reiter son reglas estrictas (no rebatibles) con un antecedente muy particular."

Según Nute esto pasa también en circunscripción, lógica no monótona, y lógica autoepistémica. El siguiente ejemplo podría mostrar más claramente el problema.

Supongamos que se tiene la siguiente información:

1. Los moluscos normalmente tienen valvas.
2. Los cefalópodos son moluscos, que normalmente no tienen valvas.
3. Los cefalópodos son moluscos.
4. El individuo *pepe* es un molusco.

Esto puede expresarse en la lógica de Nute y con Defaults de la siguiente forma:

Lógica Rebatible	Defaults
$molusco(X) \Rightarrow valva(X)$	$\frac{molusco(X):valva(X)}{valva(X)}$
$cefalopodo(X) \Rightarrow \sim valva(X)$	$\frac{cefalopodo(X):\sim valva(X)}{\sim valva(X)}$
$cefalopodo(X) \rightarrow molusco(X)$	$cefalopodo(X) \rightarrow molusco(X)$
$molusco(pepe)$	$molusco(pepe)$

A partir del conocimiento anterior y utilizando cualquiera de los sistemas, es posible derivar que *pepe* tiene valva ($valva(pepe)$).

Pero supongamos que ahora nos informan que "*pepe*" es un cefalópodo. Esto es agregamos el hecho $cefalopodo(pepe)$. En el sistema de Nute, podremos derivar $\sim valva(pepe)$ pero no será posible derivar $valva(pepe)$.

Sin embargo, en la lógica default, aparecen dos extensiones. Esto se debe a que en las reglas default no hay conexión entre cefalópodo y molusco. Para que haya una sola extensión, hay que modificar el primer default para que diga: "$molusco(X) : valva(X) \wedge \sim cefalopodo(X)/valva(X)$ ". Esto es, explicitar la excepción de ser un cefalópodo en el propio default. Observese que si agregamos la información: "el Nautilus es un cefalópodo que sí tiene valva", y "pepe es un Nautilus", entonces se volverá a dar el mismo problema.

Esto lleva a un conocido problema de representación de conocimiento, que es la explicitación de todas las excepciones. La lógica rebatible de Nute, al igual que DeLP no presentan este problema, porque las excepciones posibles las calcula el sistema en forma dinámica.

7.4 Extensiones de la Lógica Rebatible

Recientemente los investigadores Grigoris Antoniou, David Billington, Michel Maher y Guido Governatori, de la Universidad de Griffith en Brisbane, Australia, and producido nuevos resultados extendiendo el trabajo de Nute. Sin embargo, al igual que las lógicas de Nute, no definen el concepto de argumento ni realizan un análisis dialéctico. A continuación se comenta dicho trabajo, más detalles pueden encontrase en [AM98, MG99, AMB00].

En la extensión de la lógica rebatible, se conserva el mismo lenguaje, pero cambia la teoría de prueba. Por lo tanto, en primer lugar se introduce la teoría de prueba propuesta, y luego se mostrarán las diferencias con DeLP.

Una teoría rebatible D es una tupla $(F, R, >)$, donde F es un conjunto finito de literales (hechos), R un conjunto finito de reglas estrictas (R_s), rebatibles (R_d) y reglas derrotadores, y "$>$" una relación de superioridad explícita sobre las reglas de R.

Una derivación es una secuencia finita elementos $[P_1, \ldots, P_n]$, donde cada elemento P_i de la secuencia es un literal q marcado de alguna de las siguientes cuatro formas:

- $+\Delta q$, que significa que q es definitivamente probable en D.
- $-\Delta q$, que significa que se prueba que q no es definitivamente probable en D.
- $+\partial q$ que significa que q es probable rebatiblemente en D.
- $-\partial q$ que significa que se prueba que q no es probable rebatiblemente en D.

En esta lógica es posible obtener derivaciones estrictas y rebatibles. Como se verá a continuación, aunque el concepto de derivación estricta ($+\Delta q$) coincide con el de DeLP, el concepto de derivación rebatible es muy distinto, ya que involucra un análisis entre reglas contradictorias. La noción de derivación rebatible ($+\partial q$) no debe compararse con la noción de derivacion rebatible ni la noción de argumento de DeLP, sino que debe compararse con

el concepto de literal garantizado. Como se verá a continuación, la definición completa de la teoría de prueba en esta lógica rebatible es mucho mas compacta que la definición dada para DeLP.

Dada un teoría $D = (F, R, >)$, y dada la secuencia $[P_1, \ldots, P_i]$ de una derivación, a continuación se muestra como obtener el elememento P_{i+1}:

- $+\Delta q = P_{i+1}$, si se cumple una de las dos condiciones siguientes:
  (a) $q \in F$ (q es un hecho), o
  (b) existe una regla estricta $r \in R_s$ con consecuente q, tal que para todo antecedente a de r se cumple que $+\Delta a \in [P_1, \ldots, P_i]$

El caso $+\Delta q$ coincide con el concepto que derivación estricta de q en DeLP, y el caso $-\Delta q$ sería equivalente a no tener una derivación estricta de q en DeLP.

- $-\Delta q = P_{i+1}$, si se cumple una de las dos condiciones siguientes:
  (1) $q \notin F$ (q no es un hecho), y
  (2) para toda regla estricta $r \in R_s$ con consecuente q, existe un antecedente a de r tal que $-\Delta a \in [P_1, \ldots, P_i]$

Sin embargo, el caso $+\partial q$ a continuación, que describe una derivación rebatible, involucra varios aspectos: (b.1) existe una derivación, (b.2) control de contradicción, y (b.3) se buscan reglas para el complemento $\overline{q}$, que no estén a su vez derrotadas. Esto es, para cada literal q en la derivación rebatible, se hace un análisis de dos niveles (ataque y contra-ataque): (1) se busca una regla r con cabeza q, (2) se busca si hay una regla s con cabeza $\overline{q}$ para atacar a r, y (3) finalmente se analiza si existe una regla t con cabeza q que sea mejor, para contra-atacar a s. La regla t podría ser la misma r en el caso que $r > s$. El caso (a) indica simplemente que si existe una derivación estricta, entonces no hay nada que verificar.

- $+\partial q = P_{i+1}$ si se cumple una de las dos condiciones siguientes:
  (a) $+\Delta q \in [P_1, \ldots, P_i]$ (tiene una derivación estricta), o
  (b.1) existe una regla estricta o rebatible $r \in R_s \cup R_d$ con consecuente q, tal que para todo antecedente a de r se cumple que $+\partial a \in [P_1, \ldots, P_i]$, tal que
  (b.2) $-\Delta\overline{q} \in [P_1, \ldots, P_i]$, y además

(b.3) para toda regla s de R con consecuente $\sim q$ (estricta, rebatible, o derrotador)
(b.3.1) existe un antecendente a' de s $-\partial a' \in [P_1, \ldots, P_i]$ (s no puede usarse), o
(b.3.2) existe otra regla $t \in R_s \cup R_d$ con consecuente q, tal que $t > s$ (i. e., t mejor que s), donde para todo antecedente a de t se cumple que $+\partial a \in [P_1, \ldots, P_i]$ (obs: t podría ser la misma r).

A continuación figura la definición de $-\partial q$ necesaria en la definición anterior:

- $-\partial q = P_{i+1}$ si se cumplen las dos condiciones siguientes:
  (1) $-\Delta q \in [P_1, \ldots, P_i]$ (no tiene una derivación estricta), o
  (2.1) para toda regla estricta o rebatible con consecuente q, existe un antecedente a de r tal que $-\partial a \in [P_1, \ldots, P_i]$, o de lo contrario
  (2.b) $+\Delta \overline{q} \in [P_1, \ldots, P_i]$, o de lo contrario
  (2.c) existe una regla s de R con consecuente $\sim q$ y se cumplen las dos condiciones siguientes
  (2.c.1) para todo antecedente a' de s $+\partial a' \in [P_1, \ldots, P_i]$, y además
  (2.c.2) para toda regla $t \in R_s \cup R_d$ con consecuente q, existe un antecedente a de t tal que $-\partial a \in [P_1, \ldots, P_i]$, o sino, se cumple que $t \not> s$.

7.4.1 Comparación con DeLP

Al igual que la lógica rebatible de Donald Nute, esta extensión no es un formalismo de *argumentación*, ya que no existe en esta lógica una noción de argumento. La noción de conflicto se define entre dos reglas con consecuentes contradictorios. A diferencia de DeLP en lugar de comparar argumentos para decidir entre literales contradictorios, en esta lógica se comparan las reglas del programa que están en conflicto. Al no existir una noción de argumento, como se verá a continuación, no es posible definir un análisis dialéctico entre argumentos, ni controlar la aceptabilidad de las líneas de argumentación.

El lector ha podido observar que existen diferencias sintácticas en la notación de las reglas en esta lógica y las reglas de DeLP. Esto es, "$b, c \rightarrow a$" equivale a "$a \leftarrow b, c$" y "$b, c \Rightarrow a$" equivale a "$a \prec b, c$". Sin embargo, afortunadamente no existen diferencias en el significado de las reglas. En los ejemplos que utilizaremos para comparar ambos formalismos, utilizaremos la notación sintáctica de la lógica rebatible.

En DeLP no es posible derivar en forma estricta dos literales compelementarios. Sin embargo, como en la lógica rebatible no hay ninguna restricción sobre las reglas estrictas, pueden obtenerse derivaciones estrictas para literales complementarios. Por ejemplo a partir del conjunto $F = \{a, b\}$ y las reglas:

$$a \rightarrow q$$
$$b \rightarrow \sim q$$

se cumple "$+\Delta q$" y "$+\Delta \sim q$". Por lo tanto también se cumple "$+\partial q$" y "$+\partial \sim q$".

Si las reglas estrictas anteriores son debilitadas y transformadas a rebatibles,

$$a \Rightarrow q$$
$$b \Rightarrow \sim q$$

con el mismo $F = \{a, b\}$, ahora ninguna de las cuatro derivaciones del ejemplo anterior se verifican, y se cumple "$-\Delta q$", "$-\Delta \sim q$", "$-\partial q$", "$-\partial \sim q$".

Como se ha dicho antes, el concepto de "$+\partial$" no equivale a una derivación rebatible de DeLP. Utilizando la terminología de DeLP, "$+\partial$" condensa parte de la definición de argumento, de derrotador, realiza un análisis de dos niveles de derrota, entre reglas (ataque y contra-ataque). Como puede verse en la definición de "$+\partial$", la condición (b.1) pide que exista una secuencia de reglas con antecedentes probados con "$+\partial$", lo cual exige un análisis de ataque y contra-ataque en todas las reglas usadas en la derivación. La noción de derivación rebatible en DeLP no realiza ningún análisis.

La condición (b.2) de "$+\partial$", garantiza que no haya una derivación estricta del complemento de q. Esto es, si $+\Delta\ \overline{q}$, entonces no puede probarse $+\partial q$, o equivalente si $+\Delta\ \overline{q}$, entonces $-\partial q$. Como se mostró en la proposición 2.2, en DeLP ocurre lo mismo, si existe una derivación estricta de $\overline{q}$, entonces no existe argumento para q, y por lo tanto q no estará garantizado.

Sin embargo, en DeLP el análisis es más exhautivo, como muestra el siguiente ejemplo:

Ejemplo 7.2 Considérese el conjunto $F = \{a, e\}$, y las reglas

$$a \Rightarrow q$$
$$q \rightarrow \sim e$$

Aquí se verifica "$+\partial q$" (obsérvese que no hay reglas con consecuente $\sim q$, o $\sim a$). Sin embargo, en DeLP no existe estructura de argumento para q, porque la derivación rebatible es contradictoria con el conjunto de hechos y reglas rebatibles (puede derivarse e y $\sim e$)

En la lógica rebatible, la búsqueda de reglas atacantes también está limitada, ya que considera sólo literal complementarios. En cambio, en DeLP se utiliza un concepto más general: *literales en desacuerdo*. Recordemos que una estructura de argumento $\langle \mathcal{A}_1, h_1 \rangle$ contra-argumenta (ataca) a otra $\langle \mathcal{A}_2, h_2 \rangle$, si existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_2, h_2 \rangle$, tal que los literales h_1, y h estén en desacuerdo. Esto es, $\Pi \cup \{h_1, h\}$ es un conjunto contradictorio. El ejemplo a continuación muestra las diferencias:

Ejemplo 7.3 Considérese el conjunto $F = \{a, b\}$, y las reglas

$$a \Rightarrow q$$
$$b \Rightarrow e$$
$$q \rightarrow \sim e$$

Aquí se verifica "$+\partial q$" (obsérvese que no hay reglas con consecuente $\sim q$, o $\sim a$). Sin embargo, en DeLP el literal q no está garantizado, ya que la estructura de argumento $\langle \{q \prec a\}, q \rangle$, tiene el derrotador de bloqueo $\langle \{\sim e \prec b\}, \sim e \rangle$. Observe que los literales q y $\sim e$ están en desacuerdo.

Ejemplo 7.4 Considérese el conjunto $F = \{c, d\}$, y las reglas

$$c, d \Rightarrow b \ (r_1)$$
$$d \Rightarrow b \ (r_2)$$
$$b \Rightarrow a \ (r_3)$$
$$b \Rightarrow \sim a \ (r_4)$$

Donde $r_1 > r_2$, entonces se verifica "$+\partial c$" (hecho), "$+\partial d$" (hecho), "$+\partial b$" (gana por ser $r_1 > r_2$) , "$-\partial \sim b$" (pierde por ser $r_1 > r_2$), "$+\partial a$" (no hay ataque porque "$-\partial \sim b$"), y por último "$-\partial \sim a$". A diferencia de aquí, en DeLP utilizando especificidad el literal a no está garantizado. Si en DeLP se utiliza las prioridades entre reglas, entonces a está garantizado.

Si se agrega que $r_4 > r_3$, en la lógica rebatible no se alteran los resultados anteriores, ya que sigue cumpliéndose "$-\partial \sim b$"). Esto implica que las prioridades en la base de

una derivación, tienen más relevancia que las prioridades cerca de la conclusión de la derivación. En DeLP, utilizando prioridades entre reglas, con $r_4 > r_3$ y $r_1 > r_2$, a no estaría garantizado.

Reescribrimos a continuación el ejemplo 2.14 donde se mostraba el problema de los derrotadores recíprocos. Recordemos que en DeLP en este caso los literales $d, b, \sim b, y \sim d$ no están garantizados. Como muestra el ejemplo a continuación, en la lógica rebatible se entra en un ciclo infinito en la verificación de "$+\partial\sim b$".

Ejemplo 7.5 Considérese el conjunto $F = \{c, a\}$, y las reglas

$$\sim b, c \Rightarrow d \ (r_1)$$
$$\sim d, a \Rightarrow b \ (r_2)$$
$$a \Rightarrow \sim b \ (r_3)$$
$$c \Rightarrow \sim d \ (r_4)$$

Para verificar si se cumple "$+\partial\sim b$", como existe r_3 y su antecedente a es un hecho, se cumple "$+\partial a$". Entonces se debe buscar una regla atacante con consecuente b. La regla r_2 cumple estas condiciones, pero tiene que ocurrir que sus dos antecedentes se puedan derivar. Como "$+\partial a$", hay que verificar si se cumple "$+\partial\sim d$", lo cual nos llevara a un ciclo como mostramos a continuación.

Para verificar si se cumple "$+\partial\sim d$", como existe r_4 y su antecedente c es un hecho, se cumple "$+\partial c$". Entonces se debe buscar una regla atacante con consecuente d. La regla r_1 cumple estas condiciones, pero tiene que ocurrir que sus dos antecedentes se puedan derivar. Como "$+\partial c$", hay que verificar si se cumple "$+\partial\sim b$", pero esto nos lleva a un ciclo, ya que es lo que originalmente queriamos verificar.

7.5 Argumentation Frameworks (Dung)

En [Dun95] P. Dung propone un marco argumentativo general, donde se abstrae completamente de la noción de argumento y derrota. En contraste con DeLP la propuesta de Dung asume la existencia de un conjunto de argumentos y una relación binaria que define la derrota entre argumentos.

Para Dung, un *argumentation framework* (AF) es un par $(Args, defeat)$, donde $Args$ es un conjunto de argumentos y $defeat$ es una relación binaria sobre $Args$.

En sus trabajos, propone varias nociones de "extensiones" para capturar diferentes tipo de consecuencias rebatibles. Resumimos aquí los principales conceptos:

- Un conjunto S de argumentos es *libre de conflicto* si ningún argumento de S es derrotado por otro argumento de S.
- Un argumento A es *aceptable* con respecto a un conjunto S de argumentos si cada argumento B que derrota a A es a su vez derrotado por un argumento de S.
- Un conjunto S de argumentos es *admisible* si cada argumento de S es aceptable con respecto a S.
- Un conjunto S de argumentos libre de conflicto, es una *extensión estable* si todo argumento que no está en S es derrotado por algún argumento de S.
- Un conjunto S de argumentos libre de conflicto, es una *extensión preferida* si S es el conjunto admisible maximal (con respecto a inclusión de conjuntos).
- $F(S) = \{A \in Args | A$es aceptable con respecto a $S\}$
- Un conjunto admisible S de argumentos es una *extensión completa* si cada argumento que es acceptable con respecto a S pertenece a S.

Según Prakken & Vreeswijk todas las semánticas de Dung tienen problemas con argumentos "self-defeating" (ver [PV00] p.38). Resultados formales de los "argumentation frameworks":

Utilizando la terminología y las nociones definidas por Dung, dada un línea de argumentación Λ, tanto el conjunto de argumentos de soporte Λ_S, como el conjunto de argumentos de interferencia Λ_I, son conjuntos libres de conflicto (conflict free). Sin embargo, en general ninguno de estos dos conjuntos es admisible, ya que no consideran todos las reglas derrotadores posibles para cada argumento.

Sea $\mathcal{T}_{Support}$ el conjunto de todos los argumentos de soporte de todas las líneas de argumentación de un árbol de dialéctica $\mathcal{T}$. Como no existe ninguna restricción entre

líneas de argumentación, podría darse el caso que el conjunto $\mathcal{T}_{Support}$ no sea "conflict free".

Lo anterior muestra que las líneas de argumentación no son un buen elemento para comparar con el sistema de Dung. Un árbol de dialéctica, considera únicamente lo argumentos que afectan el estatus de un argumento en particular, mientras que en el sistema de Dung, se construye el conjunto de todos los argumentos que son garantizados.

7.6 Otros sistemas de argumentación rebatible

Inspirados en el *razonamiento legal*, H. Prakken y G. Sartor en [PS96, PS97] desarrollaron un sistema de argumentación que usa el lenguaje de la programación en lógica extendida. Introducen una *teoría de prueba dialéctica*, basada en el sistema argumentativo abstracto de Dung [Dun95]. Como la propuesta está inspirada en el razonamiento legal, el protocolo de disputa es diferente a los árboles de dialéctica utilizados en DeLP. La prueba de una fórmula tiene la forma de un *árbol de diálogo* donde cada rama del árbol es un diálogo entre un *proponente* y un *oponente*. Proponente y oponente poseen diferentes reglas para introducir argumentos en la disputa, lo que lleva a una situación asimétrica entre ambos. Mas tarde, Prakken en [Pra97] modificó este sistema para utilizar el lenguaje de la Lógica Default en lugar de los programas lógicos extendidos.

Otros trabajos de argumentación rebatible han sido realizados por Verheij [Ver96, Ver98], Vreeswijk [Vre97], R. Kowalski y F. Toni[KT96] , Bondarenko [BDKT97], y Loui [Lou97b]. El lector interesado encontrará muy interesantes los siguientes surveys de argumentación rebatible, escritos por Prakken & Vreesvijk [PV00], y Chesñevar, Maguitman & Loui [CML99].

7.7 Sistemas de Programación en Lógica

En las secciones anteriores se describen propuestas que surgieron del área de razonamiento rebatible, y razonamiento no monótomo. Dentro del área de programación en lógica, aunque un poco más recientemente, se han desarrollado propuestas relacionadas con nuestro trabajo. Presentamos a continuación algunas de ellas.

7.7.1 Programación en Lógica Extendida (Gelfond & Lifschitz)

Aunque la negación fuerte fue utilizada desde el comienzo por los sistemas de argumentación rebatible, lo mismo no ocurrió con la programación en lógica. El uso de negación fuerte en la cabeza de reglas, fue introducida en la definición de la Programación Lógica Extendida dada por Gelfond y Lifschitz [GL90].

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 "$\sim$".

Los progamas lógicos extendidos presentan importantes ventajas para la representación de conocimiento en programación en lógica. Sin embargo, en la propuesta de [GL90], cuando a partir de un programa $\mathcal{P}$ es posible derivar un par de literales complementarios, entonces a partir $\mathcal{P}$ se puede derivar el conjunto *Lit* de todos los literales posibles. Como el conocimiento de un agente está típicamente compuesto por información tentativa y contradictoria, la representación de conocimiento utilizando programas lógicos extendidos lleva usualmente a programas contradictorios, en los cuales se deriva todo el lenguaje.

La semántica de los PLE, se define con un *conjunto de respuestas*, que representa 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 7.1 Conjunto de respuestas (preliminar) *[GL90]*
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 7.2 Conjunto de respuestas *[GL90]*
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*. El conjunto de respuestas queda caracterizado por* $S = \alpha(\Pi^S)$.

La definición anterior está 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.

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. Según esta definición, un programa que contiene un not, nunca tiene un conjunto de respuestas que es igual a Lit. Puede ocurrir, y de hecho pasará en la mayoría de los casos en que un programa no exista conjunto de respuestas (ver ejemplo 7.7). A continuación figuran algunos ejemplos

1. Sea $\Pi_2 = \{(\ \sim P \leftarrow true), (\ P \leftarrow \sim Q)\ \}$ [2] entonces $\alpha(\Pi_2) = \{\sim P\}$.
2. Sea $\Pi_3 = \{\ (\ Q \leftarrow \sim P)\ (\ \sim P \leftarrow true)\ \}$ $\alpha(\Pi_3) = \{Q, \sim P\}$.
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 $\sim Q \leftarrow true$ (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= $\{B\}$ sí lo es.
5. Sea $\Pi_5 = \{\ P \leftarrow not\ P\}$. Este programa no tiene un conjunto de respuestas.

La siguiente proposición muestra que ocurre en los programas contradictorios.

[2] Gelfond y Lifschitz utilizan una cláusula de la forma "$P \leftarrow$" para indicar una regla incondicional (un hecho), a fin de evitar confusiones en la lectura, en este trabajo se utilizará "$P \leftarrow$ true".

Proposición 7.1 *[GL90] Todo programa contradictorio (i. e., su conjunto de respuestas tiene un par de literales complementarios) tiene exactamente un conjunto de respuestas: Lit (el conjunto de todos los literales del programa).* □

Ejemplo 7.6 Sea Π el siguiente PLE:

$$
\begin{aligned}
&vuela \leftarrow ave.\\
&\sim vuela \leftarrow pingüino.\\
&ave \leftarrow pingüino.\\
&pingüino \leftarrow \text{true}.\\
&corre \leftarrow mamifero, hambre, comida, cerca.
\end{aligned}
$$

Como el programa es contradictorio, el conjunto de respuestas es *Lit*, y a partir de este programa, es posible derivar los literales $mamifero, comida, cerca, hambre$ y , $corre$ como efecto colateral de la contradicción.

Los programas lógicos extendidos no resultan atractivos para representar conocimiento con información contradictoria, la propuesta de Gelfond y Lifschitz motivo a muchos investigadores de programación en lógica a buscar soluciones a esta problema. A continuación presentamos algunas de ellas.

7.7.2 PLE con Default Assumptions (Katsumi Inoue)

Katsumi Inoue en *Extended Logic Programming with Default Assumptions* [Ino91], fue uno de los primeros en extender la propuesta de Gelfond y Lifschitz [GL90], para intentar solucionar el problema de los programas contradictorios. 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).

Definición 7.3 Sistema de conocimiento *[Ino91]*
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 $T \cup E$.

Ejemplo 7.7 [Ino91] Dado el PLE

$$Q \leftarrow \sim P(a), \sim P(b)$$
$$\sim Q \leftarrow true$$

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 debería ocurrir es tener dos conjuntos de respuestas consistentes: $\{\sim Q, \sim P(a)\ \}$ y $\{\sim Q, \sim P(b)\}$, ignorando las cláusulas que causan inconsistencia.

Sin embargo, según nuestro criterio, 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 7.4 *[Ino91] 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 7.2 *[Ino91] 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 agreguen cláusulas, o se saque alguna con not. Considerar todas las cláusulas como defaults, lleva (igual que en el ejemplo anterior) a programas incoherentes. En su trabajo Inoue aclara "nuestro objetivo principal es por lo tanto, resolver programas incoherentes y no contradictorios."

Definición 7.5 *[Ino91] 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 7.3 *[Ino91]*
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 $\{P\}$.

Definición 7.6 *[Ino91] 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 7.8 [Ino91]
Sea $H = \{$ vuela(X) ← ave(X) $\}$, y T el conjunto formado por:

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

Aunque $T \cup H$ es contradictorio, existe una extensión base de $\mathcal{K}$: {ave(lucas), vuela(lucas), ping(petete), ave(petete), ∼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 que existe en esta propuesta es que no hay definido un criterio que permita decidir entre múltiples extensiones, ni tampoco un criterio de comparación entre las hipótesis. Por lo tanto, parte del problema de los programas contradictorios continúa presente.

7.7.3 Programación en Lógica sin Negación por Falla (Dimopoulos & Kakas)

La Programación en Lógica sin Negación por Falla (Logic Programming without Negation as Failure o LPwNF) es un formalismo propuesto en [KMD94], y luego estudiado en [DK95]. Como su nombre lo indica, el lenguaje de LPwNF no utiliza la negación por falla, pero si la negación fuerte, en la cabeza y en el cuerpo de las reglas.

Un programa en LPwNF consiste de un conjunto de reglas "$L_0 \prec L_1, \ldots, L_n$", donde $L_0, \ldots, L_n$ son literales. Para decidir entre literales complementarios se utiliza, al igual que en las lógicas rebatibles, un orden ">" definido explícitamente entre las reglas de programa (prioridades).

Las reglas de programa tienen una etiqueta (r_i) y las prioridades se indican refenciando a las reglas a través de dichas etiquetas. La forma que proponen de simular la negación por falla, es utilizando la siguiente transformación. Una regla:

$$r_0 : p \leftarrow q, not\ s$$

se transforma en dos reglas:

$$r_1 : p \leftarrow q$$
$$r_2 : \sim p \leftarrow s$$
$$\text{con } r_1 < r_2.$$

Sin embargo, como se describió en el capítulo 6, esta solución trae aparejado otros problemas nuevos.

En el trabajo "*Defeasible Logic versus Logic Programming without Negation as Failure*" [AMB00] se compara LPwNF y la lógica rebatible presentada en la sección 7.4. Como parte de dicha comparación, en [AMB00] se presenta el siguiente resultado:

Teorema: *Sea q un literal que puede probarse escépticamente a partir de un programa P en LPwNF, entonces a partir de* $T(P)$[3] *se verifica "*$+\partial q$*"*

Por lo tanto, la comparación realizada en la sección 7.4 entre DeLP y la Lógica Rebatible, puede hacerse extensiva a LPwNF.

7.7.4 Otras propuestas relacionadas

En [KMD94] fue introducida una semántica para la negación por falla llamada "acceptability semantics", la cual está basada en trabajos previos sobre negación default y Programanción en Lógica Abductiva [EK89, Dun91, KKT93]. El lenguaje que definen utiliza sólo negación por falla. Por lo tanto, la gran diferencia con DeLP, es que esta propuesta no utiliza negación fuerte para los literales.

Aquí, conjuntos de literales negados con la negación default son considerados extensiones del programa, y se define una noción de "ataque" entre extensiones. Una extensión H es "aceptable" si toda extensión A que ataca a H no es aceptable. Un operador de punto fijo es definido para la aceptabilidad. Luego, introducen una teoría general de aceptabilidad, basada en una relación binaria de "ataque", que en particular para la Programación en Lógica se define utilizando *prioridades* entre las reglas del programa.

Posteriormente, Francesca Toni y Antoni Kakas en [TK95], desarrollaron un procedimiento de prueba para la semántica de aceptabilidad definida en [KMD94]. En su propuesta utilizan una estructura de árbol similar a los árboles de dialectica usados por DeLP. En ambas propuestas, los nodos hijos de un nodo N representan un ataque para N. Sin embargo, como la aplicación de los árboles en [TK95], es para computar la semántica de la negación default, los nodos de un árbol en [TK95] son conjuntos de literales negados llamados "abducibles".

Aunque en los trabajos [KMD94] y [TK95], se definen nociones de ataque y árboles de derrotadores, la ausencia de negación fuerte en esta propuesta, marca una gran diferencia con DeLP, dificultando la comparación de ambos formalismos en ejemplos concretos.

En [GS97], Gelfond y Son desarrollan un sistema para investigar la metodología de razonamiento con "prioritized defaults" en un lenguaje de Programación en Lógica, bajo la semántica de conjuntos de respuestas (answer set). El sistema permite la representación de

[3] $T(P)$ es la transformación sintáctica del programa P en LPwNF, a la lógica rebatible

reglas estrictas y rebatibles, y requiere de un orden explícito entre las reglas del programa. La forma en que son obtenidas inferencias rebatibles, es muy similar al trabajo de Antoniou *et.al.* [AMB00], y al de PLwNF [DK95].

Capítulo 8

Aplicaciones

En este capítulo se mostrarán algunas aplicaciones de DeLP. Como fue descripto antes, DeLP es un lenguaje de representación de conocimiento, que permite representar información tentativa, y potencialmente contradictoria. Su mecanismo de inferencia, basado en la argumentación rebatible, le permite decidir entre conclusiones contradictorias. Además, DeLP puede trabajar en un entorno cambiante donde se agrega o quita información en forma dinámica, ya que conclusiones garantizadas puede quedar invalidadas si nueva información es agregada al programa.

Se mostrará a continuación, como puede utilizarse DeLP para definir el razonamiento de *agentes inteligentes deliberativos*. La flexibilidad de DeLP permitirá que los agentes trabajen en un entorno donde la información cambia rápidamente.

8.1 Agentes inteligentes deliberativos

Describiremos a continuación una aplicación concreta de DeLP para construir agentes inteligentes deliberativos, en un entorno dinámico. El contenido de esta sección es parte del artículo "Deliverative Stock Market Agents using Jinni and Defeasible Logic Programming" [GGTS00] donde hemos desarrollado un ambiente para la implementación de Sistemas Multi-agentes, que utiliza DeLP. Se describirá a continuación únicamente la contribución del autor que fue el desarrollo de agentes inteligentes deliberativos que representan su conocimiento con reglas estrictas y rebatibles, y razonan a partir de él utilizando DeLP.

La implementación de este ambiente combina DeLP con el sistema Jinni (Java INference engine and Networked Interactor) [Tar99a, Tar99b]. Jinni provee la plataforma que permite construir agentes inteligentes autónomos, y DeLP provee a los agentes la capacidad de razonar utilizando argumentación rebatible.

Trabajar con el mercado de valores requiere de un constante monitoreo de datos, y la abilidad de consultar información histórica. Un agente trabajando en tal entorno, usualmente lleva registro de las fluctuaciones de un 'portfolio' particular, y basándose en la información actual del mercado, tiene que decidir que acción tomar, como por ejemplo si vender o comprar acciones. Para tomar estas deciciones, los agentes de bolsa formulan argumentos y contra-argumentos, basados en su conocimiento del dominio, que generalmente tiene la forma de reglas tentativas o rebatibles.

En el ambiente implementado, agentes especializados se encargarán de monitorear el mercado de valores usando Internet. De esta manera, accediendo a páginas especializadas, podrán recolectar información actualizada sobre los valores de las acciones, índices de los mercados, factores de riesgo, y otros datos de interés. Dentro del ambiente, otros agentes con capacidad deliberativa, se especializan en la toma de desiciones. Estos agentes deliberativos están formados por un módulo de razonamiento usando DeLP, que provee a los agentes la posibilidad de formular argumentos, contra-argumentos y garantizar sus conclusiones en base al análisis dialéctico de DeLP.

El conocimiento de los agentes deliberativos será expresado con reglas estrictas y rebatibles, y los hechos podrán ser obtenidos a partir de la información recogida del mercado por otros agentes especializados. El módulo de razonamiento con DeLP le permite a los agentes formalizar sus creencias utilizando el concepto de literal garantizado.

Ejemplo 8.1 El conocimiento que un agente tiene del dominio, podría esta representado con el siguiente programa lógico rebatible[1]

$$buy(T) -\prec good\_price(T)$$
$$\sim buy(T) -\prec good\_price(T), price\_decreasing(T)$$
$$price\_decreasing(T) -\prec lastsale(T, P), previousclose(T, Pre), P < Pre$$
$$good\_price(acme) -\prec lastsale(acme, P), P < 100$$

[1]El símbolo "<" en las reglas del programa es manejado por DeLP como un built-in como se describió en la sección 4.5.3.

Si la información obtenida del mercado por los agentes especializados indica que, "$lastsale(acme, 70)$" y "$proviousclose(acme, 85)$", esto es, el valor de la última venta de "*Acme*" fue 70, y que el cierre anterior fue de 85, se puede generar la estructura de argumento: $\langle \mathcal{A}, buy(acme) \rangle$, con

$$\mathcal{A} = \left\{ \begin{array}{l} buy(acme) -\!\prec good\_price(acme) \\ good\_price(acme) -\!\prec lastsale(acme, 70), 70 < 100 \end{array} \right\}$$

y la estructura $\langle \mathcal{B}, \sim buy(acme) \rangle$, con

$$\mathcal{B} = \left\{ \begin{array}{l} \sim buy(acme) -\!\prec good\_price(acme), price\_decreasing(acme) \\ price\_decreasing(acme) -\!\prec lastsale(acme, 70), previousclose(acme, 85), 70 < 85 \end{array} \right\}$$

Ambas son un contra-argumento de la otra y utilizando el criterio de especificidad para comparar argumentos, la última es un derrotador propio de la primera. Por lo tanto el literal "$buy(acme)$" no estará garantizado, y el literal "$\sim buy(acme)$" si lo estará. Por lo tanto, la creencia del agente será "no comprar acciones de Acme".

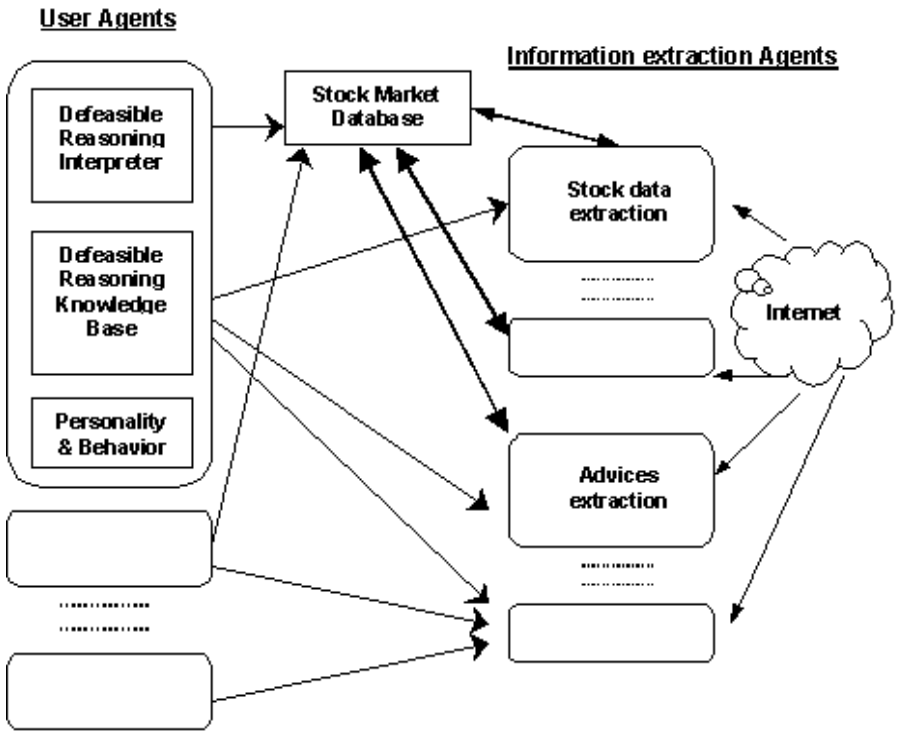

Figura 8.1: Esquema del sistema multi-agentes

La Figura 8.1 muestra el esquema del sistema, que consiste de agentes especializados en extraer información de páginas de internet (Information Extraction Agentes), una base de

datos con la información recolectada, y agentes deliberativos (User Agents) que analizan la información recuperada, buscando cumplir con la tarea que les fue asignada. Los agentes podrían estar corriendo en la misma máquina, o en máquinas diferentes sobre Internet. La comunicación entre los agentes se realiza a través de un *blackboard* que provee Jinni. Como el objetivo de esta sección es describir la aplicación de DeLP a la implementación de agente deliberativos, referimos al lector interesado en los detalles de los agentes de extracción de información, y la comunicación entre agentes a [GGTS00].

Cada agente deliberativo (user agent) consiste de:

1. un módulo con el conocimiento propio del agente, expresado como un programa lógico rebatible,
2. un módulo de razonamiento implementado como un intérprete de DeLP, y
3. la especificación de los objetivos y acciones que debe realizar el agente.

La base de datos del sistema (*Stock Market Data Base*) contiene la información recuperada por los agentes, almacenada como átomos de PROLOG, en un blackboad de Jinni. De esta manera, los agentes deliberativos podrán acceder a dicha base de datos, en busca de información, y generar su propia información histórica si lo desean.

8.1.1 Agentes inteligentes con razonamiento rebatible

Un agente deliberativo se implementará especificando las *metas* que debe cumplir, y las *acciones* que debe realizar cuando las metas se hallan cumplido. Por ejemplo, la meta de un agente podría ser "comprar acciones de la compañia C", y la acción asociada "avisar al usuario que es un buen momento para comprar". A fin de permitir la interacción entre los agentes, DeLP y Jinni, los agentes se implementan en BINPROLOG[2].

La Figura 8.2 muestra la implementación en BINPROLOG, de un agente llamado `agent0`. La primera parte del código implementa el comportamiento del agente, y la segunda, representa con un programa lógico rebatible, el conocimiento que tiene el agente del dominio. La meta de `agent0` es "comprar acciones de Acme" ($buy(acme)$), y la acción

[2]BINPROLOG es un intérprete de Prolog, implementado al igual que Jinni por el Dr. Paul Tarau, que provee primitivas para implementar agentes autónomos y comunicarse con los blackboards de Jinni (ver `http://www.binnetcorp.com/BinProlog`)

asociada es informar en el momento que se pueda satisfacer dicha meta. La forma en que `agent0` decide que su meta está garantizada, es razonando con DeLP a través de la llamada `answer(buy(acme),X)`. El predicado `answer/2`, se refiere exactamente a la especificación de la Figura 4.11. El resto del código del comportamiento del agente está suficientemente comentado en la figura. Cabe recordar que el predicado de PROLOG estándar "`repeat/0`", siempre tiene éxito al ser llamado y al hacer backtracking. Por lo tanto, toda llamada que precede al repeat nunca es alcanzada por el backtracking. En nuestro ejemplo, mientras el `X` no sea `yes`, el código hace backtracking hasta `repeat`, vuelve a dormir 5 segundos, y vuelve a llamar a DeLP. Si `X` es devuelto en `yes`, entonces el código de `agent0` tiene éxito, y no hay más backtracking.

```
% ....... Agent behaviour in BinProlog......
 agent0:- set_host('csci.unt.edu'),  % set the stock data base address
          repeat,   % loop and ask every 5 seconds until answer is yes
          sleep(5),
          answer(buy(acme),X),   % call to DLP with query "buy(acme)"
          X = yes,             % if the answer is yes, stop looping
          write('I believe that is good moment for buying T.I. stocks'),nl.

%........ Agent knowledge in DeLP........
buy(T) -< good_price(T).
~buy(T) -< good_price(T), price_decreasing(T).
price_decreasing(T) -< lastsale(T,P), previousclose(T,Pre), P < Pre .
good_price(acme) -< lastsale(acme,P), P < 100.
```

Figura 8.2: Implementation de un Agente Deliberativo en BINPROLOG

Supóngase que se ejecuta "`agent0`" y que en la base de datos del sistema se encuentran los hechos: "$lastsale(acme, 70)$", y "$previousclose(acme, 85)$". Entonces, como fue mostrado en el ejemplo 8.1, utilizando el programa lógico rebatible que representa el conocimiento de `agent0`, y los hechos anteriores, existe un argumento para $buy(acme)$, que esta derrotado. Por lo tanto la llamada a `answer(buy(acme),X)`, devolverá en `X=no`, y `agent0` seguirá en su bucle, durmiendo y consultando a DeLP.

Supongamos que se actualiza la base de datos con "$lastsale(acme, 90)$". Con esta nueva información, ya no puede derivarse "$price\_decreasing(acme)$", y no puede formarse el argumento $\langle \mathcal{B}, \sim buy(acme) \rangle$ (ver ejemplo 8.1). Por lo tanto, el argumento para "$buy(acme)$" no es derrotado, y la respuesta a "$buy(acme)$" es "`yes`". De esta manera, el agente informará que, según su conocimiento, cree que es un buen momento para comprar las acciones.

La Figura 8.3 muestra un agente más general que el anterior, donde la meta del agente es pasada como el parámetro "`Goal`". De esta manera, pueden crearse fácilmente agentes con el mismo comportamiento, para diferentes metas, simplemente llamando a `agent/1` con diferente parámetro.

```
agent(Goal):-set_host('csci.unt.edu'),  % select one of possible hosts
             repeat,   % loop and ask every 5 seconds until answer is yes
               sleep(5),
               answer(Goal,Answer),  % call to DLP Interpreter
               Answer = yes,
             write('The goal: '), write(Goal), write(' can be achieved '),nl,
             write('because: '),write(Argument),nl.
```

Figura 8.3: Implementation de un agente con una meta genérica

Para responder a las consultas de los agentes, el intérprete DeLP, utiliza:

1. el programa lógico rebatible que representa el conocimiento del agente, y
2. la información del mercado, almacenada en la base de datos, en un blackboard remoto.

Para permitir que el argumento se construya con información que está en una base de datos remota, el intérprete de DeLP utiliza las primitivas de BinProlog para acceder al blackboard remoto. Para esto, el código de generación de argumentos incluye una implementación análoga a lo especificado para "built-ins" en la sección 4.5.3. La Figura 8.4 muestra esta implementación, donde el predicado "`look_in_blackboard/1`" hace la consulta al blackboard remoto.

```
look_in_blackboard(H):-set_host('csci.unt.edu'),remote_run(H).
argument(true,D,D) :-!.
argument((A,B),CurrentDerivation,ABDerivation) :-!,
         argument(A,CurrentDerivation,ADerivation),
         argument(B,ADerivation,ABDerivation).
argument(H,CurrentDerivation,Argument) :-
      ( temporary_fact(H,CurrentDerivation),   % is a temporary fact
        Rule = none,  B = true,!               %
       ; (H <- B), Rule = (H <- B)             % or there is a strict rule
       ; (H -< B), Rule = (H -< B)             % or there is a defeasible rule
       ; look_in_blackboad(H), Rule=(H <-true) % or is a fact in a blackboard
       ; let_prolog_prove(H), Rule=(H <-true)  % or is a built in
      ),
       argument(B,CurrentDerivation,BodyDerivation),
       verify_non_contradictory(H,BodyDerivation),
      (   Rule = none, Argument = BodyDerivation
        ; Rule \= none, Argument = [Rule|BodyDerivation]
      ).
```

Figura 8.4: Implementación de acceso a la base de datos en el blackboard

8.1.2 Definiendo la personalidad del agente

Como fue especificado en la sección 2.9, el operador de creencia "B" permite definir cuatro respuestas ante una consulta: YES, NO, UNDECIDED, y UNKNOWN. Utilizando estas respuestas es posible especificar diferentes personalidades para los agentes deliberativos.

Por ejemplo, un agente con una personalidad *cautelosa*, puede definirse si toma sus decisiones cuando sus creencias están garantizadas, esto es, cuando la respuesta a las consultas es YES. En cambio, puede definirse un agente con una personalidad más *osada* si actua cuando la respuesta es YES o UNDECIDED.

La confiabililidad de la fuente de información, también puede utilizarse para definir la personalidad del agente. Un agente que base sus creencias en información poco confiable,

o “rumores” sobre el mercado, será más osado aún que los anteriores. Más detalles sobre este tema pueden encontrarse en [GGTS00].

8.2 DeLP y RoboLog

A continuación se comenta otra aplicación llamada DeRobolog[3] y que actualmente se encuentra en desarrollo. Esta aplicación combina DeLP con Robolog[4], un lenguaje para implementar agentes en RoboCup[5].

RoboCup [Rob00a, Tea95, HAK+97] es la copa mundial de futbol con robots, una iniciativa internacional de investigación en sistemas multiagentes. RoboCup es un intento para proveer un ámbito de desarrollo donde diferentes tecnologías puedan integrarse y examinarse. Para esto, se eligió un escenario complejo como es un partido de fútbol, donde un conjunto de agentes inteligentes (equipo) coopera con un objetivo común (ganar), compitiendo con otro conjunto de agentes que tiene un objetivo distinto: vencer a su adversario. Dentro de RoboCup existe una “liga simulada” donde un servidor permite la simulación de un partido completo.

Robolog [Rob00b, MOS00c, SOMB00, MOS00a, MOS00b, MSOB99, SOMB99a, SOMB99b] es un lenguaje de especificación de agentes para RoboCup que está implementado en Prolog. RoboLog es un lenguaje de alto nivel que permite abstraer las primitivas de manejo de los agentes en RoboCup, permitiendo el desarrollo de agentes especificando su comportamiento con un programa en Prolog. Por lo tanto, como un sistema de razonamiento, Robolog posee las mismas limitaciones que Prolog. Con lo cual, Robolog no permite diseñar agentes deliberativos fácilmente.

Como la Programación en Lógica Rebatible permite especificar conocimiento y razonar a partir de él, al combinar DeLP y RoboLog se obtiene un sistema donde se pueden especificar fácilmente agentes para RoboCup que puedan razonar utilizando el formalismo de argumentación rebatible.

Se encuentra en desarrollo la implementación de dicha combinación, para obtener un sistema integrado que permita usar los mecanismos de razonamiento de DeLP desde la

[3] `http://www.uni-koblenz.de/~stolzen/Argentina`

[4] `http://www.uni-koblenz.de/ag-ki/ROBOCUP/SIMULATION/simulator.html`

[5] `http://www.robocup.org/`

interface de RoboLog. Las pruebas realizadas hasta el momento han dado resultados altamente satisfactorios. El prototipo corre en ambiente Linux, y bajo la implementación de SWI Prolog.

8.3 Conclusiones

Como se mostró en este capítulo, la Programación en Lógica Rebatible es un lenguaje que puede ser utilizado básicamente en cualquier aplicación que requiera representación de conocimiento. Sus características le permiten modelar conocimiento que involucre información incompleta o potencialmente contradictoria. El mecanismo de inferencia sobre el cual está basado permite decidir entre conclusiones contradictorias, y adaptarse fácilmente a entornos cambiantes. Esto es, aplicaciones donde se agrega o quita información en forma dinámica, y por lo tanto conclusiones obtenidas previamente pueden quedar invalidadas.

Las características antes mencionadas, hacen de los Programas Lógicos Rebatibles, una herramienta atractiva para representar el conocimiento y definir el razonamiento de *agentes inteligentes deliberativos*. La flexibilidad del lenguaje permite que los agentes trabajen en un entorno donde la información cambia rápidamente.

Capítulo 9

Conclusiones

El contenido de esta tesis conjuga aspectos de varias áreas de Ciencias de la Computación, las cuales han tenido un notable desarrollo en los últimos años: *Programación en Lógica, Argumentación Rebatible, Agentes Inteligentes*, y *Paralelismo en Programación en Lógica*.

La Programación en Lógica Rebatible fue definida como una extensión del lenguaje de la Programación en Lógica, utilizando un formalismo de argumentación rebatible como procedimiento de prueba del lenguaje. El lenguaje permite representar información tentativa a través de reglas rebatibles, e información potencialmente contradictoria, utilizando la negación fuerte. De esta manera, el lenguaje resulta atractivo para representar conocimiento en aplicaciones de agentes inteligentes deliberativos.

Se definió una semántica operacional que permite determinar el efecto de todo programa lógico rebatible. La ventaja de disponer de una semántica operacional, es tener una especificación concreta del lenguaje. Esta semántica usualmente está orientada a programadores, ya que las descripciones operacionales son muy cercanas a un programa real. El lenguaje elegido para realizar la descripción operacional fue Prolog, debido a la cercanía conceptual con DeLP, y a que Prolog es un lenguaje simple y bien conocido por la comunidad científica.

Cada uno de los elementos del formalismo de argumentación rebatible en que se basa DeLP fue descripto como un programa en Prolog. De esta manera, se describió como obtener derivaciones rebatibles, argumentos, contra-argumentos, derrotadores, líneas de argumentación aceptables, y finalmente como construír árboles de dialéctica. Como ventaja adicional, a partir de la semántica operacional, es muy fácil construir un intérprete

para ejecutar programas. En el final del capítulo 4 fue definido el esquema de un intérprete para Programas Lógicos Rebatibles, que responde a consultas indicando 4 tipos posibles de respuestas.

Como se mostró el capítulo 5, las tres formas de paralelismo básicas identificadas en la Programación en Lógica, pueden ser explotadas implícitamente en DeLP. Pero además, se definieron nuevas fuentes de paralelismo implícito que pueden explotarse en un sistema de argumentación rebatible:

1. diferentes argumentos para un literal h pueden computarse en paralelo,
2. los derrotadores de una estructura $\langle \mathcal{A}, h \rangle$ pueden buscarse en paralelo,
3. diferentes líneas de argumentación de un árbol de dialéctica pueden construirse en paralelo.
4. árboles de dialéctica para diferentes estructuras de argumento para un literal h pueden analizarse en paralelo.

Todas estas estrategias permiten acelerar el proceso de dialéctica y permiten realizar un proceso de argumentación paralelizado.

El uso de la negación es necesario en cualquier sistema de representación de conocimiento. Es esta tesis se define la Programación en Lógica Rebatible Extendida que permite utilizar dos tipos de negación: la negación fuerte que permite representar conocimiento contradictorio, y la negación default que permite representar información incompleta. A diferencia de desarrollos anteriores [SG95, Gar97, GS99c], en esta tesis se define a la negación default como una nueva forma de ataque dentro del formalismo argumentativo.

La Programación en Lógica Rebatible es un lenguaje que puede ser utilizado básicamente en cualquier aplicación que requiera representación de conocimiento. Sus características le permiten modelar conocimiento que involucre información incompleta o potencialmente contradictoria. El mecanismo de inferencia sobre el cual está basado permite decidir entre conclusiones contradictorias, y adaptarse fácilmente a entornos cambiantes. Esto es, aplicaciones donde se agrega o quita información en forma dinámica, y por lo tanto conclusiones obtenidas previamente pueden quedar invalidadas.

Las características antes mencionadas, hacen de los Programas Lógicos Rebatibles, una herramienta atractiva para representar el conocimiento y definir el razonamiento de

agentes inteligentes deliberativos. La flexibilidad del lenguaje permite que los agentes trabajen en un entorno donde la información cambia rápidamente. En el capítulo 8, se mostró una aplicación concreta donde se utilizó la Programación en Lógica Rebatible para construir agentes inteligentes deliberativos.

En el capítulo 7, los resultados obtenidos fueron comparados con formalismos existentes y que están relacionados en gran medida con nuestro desarrollo. De esta comparación puede verse, que la Programación en Lógica Rebatible, representa un avance significativo en el área de Argumentación Rebatible y Programación en Lógica.

Como trabajo futuro se planea seguir investigando en dos frentes. Por un lado se continuará cimentando la definición del aparato deductivo de DeLP, y por otro lado, se experimentarán con las aplicaciones descriptas en el capítulo 8. Se buscarán además nuevas aplicaciones, que permitan tanto la aplicación tecnológica de la Programación en Lógica Rebatible, como el refinamiento del formalismo desarrollado.

Bibliografía

[AM98] Antoniou, G., and Maher, D. B. M. J. Normal forms for defeasible logic. In *Proceedings of International Joint Conference ad Symposium on Logic Programming* (1998), MIT Press, pp. 160–174.

En este trabajo se presenta una forma de normalizar una teoría de la lógica rebatible, eliminando los hechos y separando completamente las reglas rebatibles de las strictas. Luego muestran que la relación de superioridad puede obviarse mediante una transformación de reglas (lamentablemente el ejemplo usado tiene errores y no muestra el resultado esperado). Finalmente muestran que es necesario tener los 3 tipos de reglas: strictas, rebatibles y derrotadores, ya que ninguna puede simularse con las otras dos.

[AMB00] Antoniou, G., Maher, M. J., and Billington, D. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming 42* (2000), 47–57.

En este trabajo comparan una reciente version de la lógica rebatible de Donald Nute con la programación en lógica sin negación por falla. El resultado indicado es que la primera captura todo lo que la segunda puede expresar, pero LPwNF no puede capturar la idea de "teams rules".

[AP93] Alferes, J. J., and Pereira, L. M. Contradiction: when avoidance equals removal (part i and ii). In *Proc. of Extensions of Logic Programming, 4th International Workshop ELP'93* (March 1993), St. Andrews U.K.

[APP96] Alferes, J. J., Pereira, L. M., and Przymusinski, T. Strong and explicit negation in non-monotonic reasoning and logic programming. In *Logics in Artificial Intelligence (JELIA'96)* (1996), Springer, pp. 143–163.

El trabajo sugiere inicalmente que la utilización del término "negación clasica" en el trabajo de [GL90] no es correcta, ya que la negación clásica posee propiedades que no se cumplen en dicho trabajo. A raíz de esto sugieren distinguir entre diferentes tipos de negación: negación default, negación fuerte, negación simétrica, y negación explícita. Definen una lógica modal para definir y desarrollar propiedades sobre estos tipos de negación..

[BDKT97] Bondarenko, A., Dung, P., Kowalski, R., and Toni, F. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence 93* (1997), 63–101.

[Cla78] Clark, K. Negation as failure. In *Logic and Data Bases*, Gallaire and Minker, Eds. Plenum Press, New York, 1978, pp. 293–332.

[CML99] Chesñevar, C. I., Maguitman, A., and Loui, R. P. Logical models of arguments. *ACM Computing Surveys* (1999).

Es un excelente survey del estado del arte en argumentación rebatible..

[CNV97] Covington, M. A., Nute, D., and Vellino, A. *Prolog Programming in Depth*. Prentice-Hall, 1997.

Uno de los capítulos del libro es dedicado integramente a explicar en detalle d-Prolog..

[DeG84] DeGroot, D. Restricted And-Parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984* (1984), pp. 471–478.

[DK95] Dimopoulos, Y., and Kakas, A. Logic programming without negation as failure. In *Proceedings of 5th. International Symposium on Logic Programming* (Cambridge, MA, 1995), MIT Press, pp. 369–384.

Este trabajo estudia una extension de la programación en lógica donde no se utiliza la negación por falla. Es una continuación de [KMD94]. Las reglas de los programas pueden compararse utilizando una relación de orden irreflexiva y antisimétrica provista por el programador. El trabajo define, además de la sintaxis, un criterio de ataque entre derivaciones y un procedimiento de prueba que considera ataques entre reglas. Se presenta ademas una aplicación al razonamiento

temporaral y una comparación con otros enfoques de la programación en lógica extendida.

[Dun91] DUNG, P. M. Negation as hypothesis: An abductive foundation for logic programs. In *Proceedings of the 8th. International Conference on Logic Programming* (Paris, France, 1991), MIT Press.

[Dun93a] DUNG, P. M. An argumentation semantics for logic programming with explicit negation. In *Proc. ICLP'93* (1993), MIT Press, pp. 616–630.

El trabajo parte del lenguaje de la programación en lógica extendida (PLE) de Gelfond & Lifschitz [GL90], y define una teoría de argumentación donde poder definir semánticas para la PLE, generalizando las semánticas de la programación en lógica normal. Presenta una teoría de argumentación (argumentation framework) donde los argumentos son conjuntos de "suposiciones", esto es conjunto de literales negados por la negación default ("not"). Define dos formas de atacar los argumentos: via la negación default (ground attack), y via la negación fuerte (Reduction Ad Adsurdum)..

[Dun93b] DUNG, P. M. On the Acceptability of Arguments and its Fundamental Role in Nomonotonic Reasoning and Logic Programming. In *Proc. of the 13th. International Joint Conference in Artificial Intelligence (IJCAI), Chambéry, Francia* (1993).

Presenta una teoría de arguementación abstracta, definiendo un *argumentation framework* (AF) como un par $(Args, attacks)$, donde $Args$ es un conjunto de argumentos y $attacks$ es una relación binaria sobre $Args$. Los argumentos quedan sin especificar, aunque en [Dun93a] lo especifica. Un conjunto S de argumentos es *libre de conflicto* si ningún argumento de S es derrotado por otro argumento de S. Un argumento A es *aceptable* con respecto a un conjunto S de argumentos si cada argumento B que derrota a A es a su vez derrotado por un argumento de S. Un conjunto S de argumentos es *admisible* si cada argumento de S es aceptable con respecto a S. Un conjunto S de argumentos libre de conflicto, es una *extensión estable* si todo argumento que no está en S es derrotado por algún argumento de S. Un conjunto S de argumentos libre de conflicto, es una *extensión preferida* si S es el conjunto admisible maximal (con respecto a inclusión de conjuntos). $F(S) = \{A \in Args | A es aceptable c.r.a. S\}$ Un conjunto admisible

S de argumentos es una *extensión completa* si cada argumento que es acceptable con respecto a S pertenece a S..

[Dun95] Dung, P. M. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming and n-person games. *Artificial Intelligence 77* (1995), 321–357.

Es una versión extendida el trabajo [Dun93b], donde se agrega un análisis de la argumentación en los *juegos de N-personas* desarrollados por Von Neuman y Morgenstern, y un análisis del *Stable Marriage Problem* desde la óptica de los arguementation frameworks..

[EK89] Eshghi, K., and Kowalski, R. A. Abduction compared with negation as failure. In *Proceedings of the 6th. International Conference on Logic Programming* (Lisbon, Portugal, 1989), MIT Press.

[Fil01] Fillottrani, P. R. *Semántica para la Negación en Programas Lógicos Extendidos (Tésis Doctoral en desarrollo)*. Universidad Nacional del Sur, Bahía Blanca, Argentina, 2001.

[Gar97] García, A. J. Defeasible logic programming: Definition and implementation. Master's thesis, Dep. de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, July 1997.

[GGTS00] García, A. J., Gollapally, D., Tarau, P., and Simari, G. Deliberative stock market agents using jinni and defeasible logic programming. In *Proceedings of ESAW'00 Engineering Societies in the Agents' World, Workshop of ECAI 2000* (August 2000).

El trabajo desarrolla un ambiente para la implementación de Sistemas Multiagentes, que permite mantener agentes especializados en extraer información del mercado de valores a través de Internet, y agentes deliberativos, que representan su conocimiento con reglas estrictas y rebatibles, y razonan a partir de él utilizando DeLP. La implementación de este ambiente combina DeLP (Defeasible Logic Programming) con el sistema Jinni (Java INference engine and Networked Interactor) [Tar99a]. Jinni provee la plataforma que permite construir agentes inteligentes autónomos, y DeLP provee a los agentes la capacidad de razonar utilizando argumentación rebatible..

[GKA+94] Gupta, G., Khayri, Ali, A., Hermenegildo, M., and Carlsson, M. Parallel execution of prolog programs: A survey. Tech. rep., Department of Computer Science, New Mexico State University, 1994. http://www.cs.nmsu.edu/lldap/pub_para/survey.html.

[GL90] Gelfond, M., and Lifschitz, V. Logic programs with classical negation. In *ICLP* (1990), D. Warren and P. Szeredi, Eds., MIT Press, pp. 579–597.

[GPP89] Gelfond, M., Przymusinski, H., and Przymusinski, T. On the relationship between circumscription and negation as failure. *Artificial Intelligence 38* (1989), 75–91.

[Gre87] Gregory, S. *Parallel Logic Programming in PARLOG. The language and its implementation.* Addison-Wesley, 1987.

[GS95] García, A. J., and Simari, G. R. Compilación de programas lógicos que utilizan la negación por falla. In *Proceedings of the I Congreso Argentino en Ciencias de la Computación* (October 1995), Univ. Nac. de Sur, pp. 27–38.

[GS97] Gelfond, M., and Son, T. C. Reasoning with prioritized defaults. In *Lecture Notes in Artificial Intelligence 1471, Selected Papers from the Workshop on Logic Programming and Knowledge Representation* (1997), pp. 164–223.

Desarrollan un sistema para investigar la metodología de razonamiento con "priortized defaults" en el lenguaje de programas lógicos, bajo la semantica de conjuntos de respuestas (answer set). El sistema permite la representación de reglas estrictas y rebatibles, y de un orden entre las reglas. La forma en que son obtenidas inferencias rebatibles, es muy similar al trabajo de Antoniou *et.al.* refAntoniouMaherBillington00. Sin embargo no presentan ninguna comparación de ambos trabajos..

[GS98] García, A. J., and Simari, G. R. Sources of parallelism in defeasible logic programming. In *Proceedings of the IV Congreso Argentino en Ciencias de la Computación* (October 1998).

Este trabajo presenta por primera vez un estudio de diferentes tipos de paralelismo que pueden explotarse en forma implícita en la programación en lógica rebatible. Esto es, sin la intervención explícita del programador..

[GS99a] García, A. J., and Simari, G. R. Parallel conctruction of dialectical trees for defeasible logic programming. In *Proceedings of the VI Workshop of Aspectos Teóricos de la Inteligencia Artificial* (May 1999), Universidad Nacional de San Juan.

Este trabajo muestra como construir árboles de dialéctica en paralelo. Para esto se define un modelo de construcción del árbol en forma distrubuída, y luego especifica como implementarlo usando una máquina abstracta paralela..

[GS99b] García, A. J., and Simari, G. R. Parallel defeasible argumentation. *Journal of Computer Science and Technology Special Issue: Artificial Intelligence and Evolutive Computation. http://journal.info.unlp.edu.ar/ 1*, 2 (1999), 45–57.

Este trabajo es una vesión extendida y mejorada de [GS98]. Se muestran diferentes formas de paralelismo explotable en forma implícita en DeLP. Se analiza como pueden pueden explotarse las técnicas de paralelismo existentes para la programación en lógica, y se agrega la construcción de argumentos en paralelo, la búsqueda de derrotadores en paralelo, y la generación de árboles de dialéctica en paralelo..

[GS99c] García, A. J., and Simari, G. R. Strong and default negation in defeasible logic programming. In *Proc. Fourth Dutch-German Workshop on Nonmonotonic Reasoning Techniques and Their Applications, DGNMR'99* (March 1999).

En este trabajo se analiza el uso de dos tipos de negación en la programación en lógica rebatible..

[GSC98] García, A. J., Simari, G. R., and Chesñevar, C. I. An argumentative framework for reasoning with inconsistent and incomplete information. In *Workshop on Practical Reasoning and Rationality* (August 1998), 13th biennial European Conference on Artificial Intelligence (ECAI-98).

En este trabajo se presenta un sistema de argumentación rebatible que permite trabajar con información inconsistente e incompleta. Permitiéndese el uso de dos tipos de negación..

[Gup94] Gupta, G. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Publishers, 1994.

[HAK+97] H.Kitano, Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. Robocup: The robot world cup initiative. In *in Proc. of The First International Conference on Autonomous Agent (Agents-97)* (Marina del Ray, 1997), T. A. Press, Ed.

[Hen90] Hennessy, M. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

[Her86a] Hermenegildo, M. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

Presenta una implementación de paralelismo AND restringido.

[Her86b] Hermenegildo, M. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming* (July 1986), no. 225 in Lecture Notes in Computer Science, Imperial College, Springer-Verlag, pp. 25–40.

[Ino91] Inoue, K. Extended logic programming with default assumptions. In *Proc of 8th. ICLP* (1991).

[KKT93] Kakas, A. C., Kowalski, R. A., and Toni, F. Abductive logic programming. *Journal of Logic and Computation 2* (1993), 719–770.

[KMD94] Kakas, A. C., Mancarella, P., and Dung, P. M. The acceptability semantics for logic programs. In *Proceedings of the 11th. International Conference on Logic Programming* (Santa Margherita, Italy, 1994), MIT Press, pp. 504–519.

Presentan una semántica para la negación por falla basada en la idea que los literales negados por *not* son posibles extensiones al programa lógico. Se basan en el trabajo [Dun91]. Definen una noción de "ataque" entre conjuntos de literales con *not* y una noción de aceptabilidad con un operador de punto fijo. También

presentan una teoría más general de aceptabilidad, propiedades de la noción de ataque. Finalmente introducen la programacion en lógica sin negación por falla con las mismas consideraciones que en [DK95].

[KT96] Kowalski, R. A., and Toni, F. Abstract argumentation. *Artificial Intelligence and Law 4*, 3-4 (1996), 275–296.

[Lif96] Lifschitz, V. Foundations of logic programs. In *Principles of Knowledge Representation*, G. Brewka, Ed. CSLI Pub., 1996, pp. 69–128.

Es un trabajo muy bueno donde se condensa y ordena los últimos avances en programación en lógica. Se define primeramente los *programas básicos* con "negación clásica" y un operador de concecuencia. Luego define el cálculo SLD para programas básicos. En segundo lugar define a los programas normales (sin negación). Luego introduce la negación por falla, presenta la semántica de conjuntos de respuestas, de [GL90], presenta los programas *tight* y estratificados, y define el cálculo SLDNF. A continuación define los programas con reglas esquemáticas (con variables) y extiende el cálculo SLDNF para trabajar con variables. Finalmente introduce los programas disyuntivos. interesante: Answer set: pp 23, esquemas: PP 39.

[Llo87] Lloyd, J. W. *Foundations of Logic Programmming*. Springer-Verlag, 1987.

El libro presenta de una manera muy clara y ordenada los fundamentos de la Programación en Lógica. Introduce el lenguaje de los programas lógicos partiendo del cálculo de predicados. Define los programas lógicos definidos (sin negación) y presenta una semántica de punto fijo. Luego presenta los programas lógicos normales (con negación default).

[LN95] Loui, R. P., and Norman, J. Rationales and argument moves. *Artificial Intelligence and Law 3* (1995), 159–189.

[LNS+93] Loui, R. P., Norman, J., Stiefvater, K., Merrill, A., Costello, A., and Olson, J. Computing Specificity. Tech. Rep. CS-TR93-03, Departament of Computer Science, Washington University, St.Louis, 1993.

[Lou87] Loui, R. P. Defeat Among Arguments: A System of Defeasible Inference. *Computational Intelligence 3*, 3 (1987), 100–106.

Este trabajo presenta un sistema de razonamiento no-monotónico con reglas rebatibles. En este sistema ante la aparición de múltiples extensiones, se realiza un ordenamiento utilizando consideraciones sintácticas. El sistema trata la derrota de la forma en que es tomada por los filósofos en epistomología. En [Lou87], una regla rebatible se denota "$a \succ b$", y se interpreta como "a es una razón rebatible para b". El sistema formal está definido a partir de una base de datos $\langle EK, R\rangle$, donde EK es conocimiento evidencial, y R el conjunto de reglas rebatibles. Las reglas rebatibles permiten formar argumentos.

[Lou89] LOUI, R. P. Defeat Among Arguments II. Tech. Rep. WUCS-89-6, Washington University, Department of Computer Science, January 1989.

[Lou91a] LOUI, R. P. Ampliative inference, computation and dialectic. In *AI and Philosophy*, R.Cummins and J.Pollock, Eds. 1991.

[Lou91b] LOUI, R. P. Argument and belief: Where we stand in keynesian tradition. *Minds and Machines (Special issue: defeasible reasoning) 1*, 4 (November 1991).

[Lou93a] LOUI, R. P. Analogy, decision, and theory-formation as defeasible reasoning. Tech. rep., Departament of Computer Science, Washington University, St.Louis, 1993.

El trabajo presenta cuatro "reinos" del razonamiento rebatible. En los primeros dos: "defeasible statistical reasoning" y " analogical reasoing" la derrotabilidad aparese porque nueva información se agrega al sistema. En los otros dos: "best explanations", y "decision-theoretic reasoning", la derrotabilidad es el resultado de una deliberación más profunda o más computación con la misma evidencia..

[Lou93b] LOUI, R. P. Process and policy: resource-bounded, non-demonstrative reasoning. *Computational Intelligence 2*, 5 (1993).

[Lou94] LOUI, R. P. Argument and arbitration games. Tech. rep., *First Workshop On Computational Dialectics*, Seattle, USA, August 1994.

[Lou95] LOUI, R. P. Report on the workshop in computational dialectics. *AI Magazine* (August 1995).

[Lou97a] Loui, R. P. Alchourrón and Von Wright on Conflict among Norms. In *Defeasible Deontic Logic*, D.Ñute, Ed., vol. 263. Synthese Library, 1997, pp. 345–353.

[Lou97b] Loui, R. P. et al. Progress on Room 5: A Testbed for Public Interactive Semi-Formal Legal Argumentation. In *Proc. of the 6th. International Conference on Artifcial Intelligence and Law* (July 1997).

[McC80] McCarthy, J. Circumscription – A Form of Non-monotonic Reasoning. *Artificial Intelligence 13* (1980), 27–39,171–172.

[Mey90] Meyer, B. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.

[MG99] Maher, M. J., and Governatori, G. A semantic decomposition of defeasible logics. *AAAI* (1999), 299–305.

Investiga las logicas rebatibles usando una técnica que descompone la semántica de la lógica en dos partes: una especificación de la estructura del razonamiento rebatible y una semántica para el meta-lenguaje en el cual se escribe la especificación. Muestran que la lógica rebatible de Nute corresponde a la semántica de Kunen, y desarrollan una lógica rebatible a partir de la semántica bien fundada..

[Min82] Minker, J. On indefinite data bases and the closed world assumption. In *Proceedings of the 6th. conf. on Automated Deduction* (1982), Springer Verlag, pp. 292–308.

[MOS00a] Murray, J., Obst, O., and Stolzenburg, F. RoboLog Koblenz. In *RoboCup-99: Robot Soccer WorldCup III*, M. Veloso, E. Pagello, and H. Kitano, Eds., LNAI. Springer, 2000. Team description. To appear.

[MOS00b] Murray, J., Obst, O., and Stolzenburg, F. RoboLog Koblenz 2000. In *Proceedings of the Workshop for the Robocup European Championship* (2000), W. van der Hoek, Ed., Vrije Universiteit Amsterdam. Team description.

[MOS00c] Murray, J., Obst, O., and Stolzenburg, F. Towards a logical approach for soccer agents engineering. In *Proceedings of the 4th Interna-*

tional Workshop on RoboCup (Melbourne, 2000), T. Balch, P. Stone, and G. Kraetzschmar, Eds. To appear.

[MS91] Makinson, D., and Schlechta, K. Floating conclusions and zombie paths: two deep difficulties in the directly skeptical approach to defeasible inference nets. *Artificial Intelligence 48* (1991), 199–209.

[MSOB99] Murray, J., Stolzenburg, F., Obst, O., and Bremer, B. RoboLog Koblenz: Complex agent scripts implemented in logic. In *Proceedings of the Workshop RoboCup during KI'99 in Bonn* (1999), S. Sablatnög and S. Enderle, Eds., pp. 12–25. SFB 527 Report 1999/12, Universität Ulm.

[NN90] Nielson, H., and Nielson, F. *Semantics with Applications, a Formal Introduction.* John Wiley & Sons, 1990.

[Nut86] Nute, D. Ldr: a logic for defeasible reasoning. research report 01-0013. Tech. rep., Advanced Computational Methods Center (ACMC). University of Georgia., 1986.

Describe una lógica llamada LDR1, la cuál permite inferencias monótonas y no monótonas. Introduce las reglas rebatibles, diferenciándolas de las reglas absolutas. Incluye un desarrollo formal del sistema LDR. Por último analiza los casos de disyunciones exclusivas e inclusivas. Sugiere que una disyunción "$a \vee b$" sea reescrita como "a$\rightarrow \sim$b" y "b$\rightarrow \sim$a". Luego discute la posibilidad de usar reglas rebatibles reescribiendo "$a \vee b$"como "a$\Rightarrow \sim$b" y "b$\Rightarrow \sim$a",.

[Nut87] Nute, D. Defeasible reasoning. In *Proc. of the 20th annual Hawaii Int. Conf. on System Sciences.* (1987).

Aquí presenta la lógica LDR y su semántica. Aunque en el abstract promete mostrar su implementación, sólo cita donde encontrarla. Introduce los conceptos básicos del razonamiento rebatible, comenta diferentes enfoques al tema (entre ellos el probabílistico) presenta la lógica condicional VW, y luego describe el sistema formal LDR. Por último da una semántica formal para LDR..

[Nut88] Nute, D. Defeasible reasoning: a philosophical analysis in prolog. In *Aspects of Artificial Intelligence*, J. H. Fetzer, Ed. Kluwer Academic Pub., 1988, pp. 251–288.

En este trabajo intenta mostrar dos cosas: primero, como puede implementarse un programa en prolog que capture el razonamiento rebatible (d-Prolog), y además que un programa en Prolog puede ser utilizado para realizar un análisis filosófico, en lugar de los métodos tradicionales como son los sistemas formales y las semánticas formales. Al comienzo del trabajo explica los métodos de análisis filosófico, el uso de la lógica, de la programación en lógica, y que es Prolog. Luego da una introducción al razonamiento rebatible, reglas absolutas, rebatibles, y la necesidad de usar reglas como derrotadores. A medida que introduce los conceptos implementa en Prolog un programa para realizar inferencias absolutas y rebatibles. Por último analiza problemas de comparación de reglas y de encadenamiento de reglas rebatibles..

[Nut92] NUTE, D. Basic defeasible logic. In *Intensional Logics for Programming*, L. F. del Cerro, Ed. Claredon Press, Oxford, 1992.

En este trabajo presenta una propuesta específica de una familia de lógicas rebatibles que ilustran las decisiones básicas de diseño que uno debe tomar cuando se formula un sistema de estas características . El lenguaje formal es el mismo que para LDR, pero lo que cambia sustancialmente es la teoría de prueba. Los conflictos entre las reglas rebatibles con consecuentes incompatibles son resueltos de diferentes maneras: usando una relación explícita de superiordad sobre las reglas (oráculo), o un método para determinar cuál de las dos reglas tiene un antecedente más específico. Las pruebas son árboles cuyos nodos están etiquetados por: teorías rebatibles, fbf de la lógica, y un $+$ o un $-$ para indicar que la fbf es derivable, o demostrablemente no derivable. Las lógicas rebatibles se definen como conjuntos de condiciones sobre los nodos de los árboles de prueba. Todo el desarrollo es hecho para fórmulas totalmente instanciadas y luego al final del trabajo introduce una forma de generalizarlo para fórmulas con variables. Por último en una sección sobre la decibilidad del los sistemas Nute argumenta que la profundidad de un árbol de prueba no es mayor a $4.N.(M+1)$ donde N es el número de fórmulas atómicas de la teoría y M el número de reglas rebatibles..

[Nut94] NUTE, D. Defeasible logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol 3*, D. Gabbay, C. Hogger, and J.A.Robinson, Eds. Oxford University Press, 1994, pp. 355–395.

En el artículo se postula que los trabajos de Delgrande, Pollock, Geffner y Pearl, y Loui, si representan propuestas de lógicas rebatibles para el razonamiento no monótono. Dichas propuestas son descriptas y comparadas. Sin embargo según Nute, la lógica no monótona de McDermott y Doyle, circunscripción de McCarthy, default logic de Reiter, y la lógica autoepistémica de Moore, no son lógicas rebatibles, y explica por qué..

[Pol70] POLLOCK, J. The Structure of Epistemic Justification. *American Philosophical Quaterly, monographs series 4* (1970), 62–78.

[Pol74] POLLOCK, J. *Knowledge and Justification.* Princeton, 1974.

[Pol87] POLLOCK, J. Defeasible Reasoning. *Cognitive Science 11* (1987), 481–518.

[Pol91a] POLLOCK, J. Self-defeating arguments. *Minds and Machines (Special issue: defeasible reasoning) 1*, 4 (November 1991).

[Pol91b] POLLOCK, J. A theory of defeasible reasoning. In *International Journal of Intelligent Systems* (1991), vol. 6, John Wiley & Sons, Inc., pp. 33–54.

[Pol92a] POLLOCK, J. How to reason defeasible. *Artificial Intelligence 57* (1992), 1–42.

[Pol92b] POLLOCK, J. New foundations for practical reasoning. *Minds and Machines 2* (1992), 113–144.

[Pol95] POLLOCK, J. *Cognitive Carpentry: A Blueprint for How to Build a Person.* MIT Press, 1995.

[Pol96a] POLLOCK, J. Implementing defeasible reasoing. *workshop on Computation Dialectics* (1996).

Este trabajo muestra como el sistema OSCAR resuelve los siguientes problemas que surgen de la implementación del razonamiento rebatible: (1) como construir argumentos, (2) como computar la fuerza conclusiva de un argumento, (3) como decidir cual argumento está derrotado y cual no..

[Pol96b] POLLOCK, J. Oscar - A general purpose defeasible reasoner. *Journal of Applied Non-Classical Logics 6* (1996), 89–113.

[Poo85] POOLE, D. L. On the Comparison of Theories: Preferring the Most Specific Explanation. In *Proc. 9th IJCAI* (1985), IJCAI, pp. 144–147.

[Pra93] PRAKKEN, H. *Logical Tools for Modelling Legal Arguments*. PhD thesis, Vrije University, Amsterdam (Holanda), January 1993.

[Pra97] PRAKKEN, H. *Logical Tools for Modelling Legal Argument. A Study of Defeasible Reasoning in Law*. Kluwer Law and Philosophy Library, 1997.

[PS96] PRAKKEN, H., AND SARTOR, G. A system for defeasible argumentation, with defeasible priorities. In *Proc. of the International Conference on Formal Aspects of Practical Reasoning, Bonn, Germany* (1996), Springer Verlag.

[PS97] PRAKKEN, H., AND SARTOR, G. Argument-based logic programming with defeasible priorities. *J. of Applied Non-classical Logics 7*, 25-75 (1997).

[PV00] PRAKKEN, H., AND VREESWIJK, G. Logical systems for defeasible argumentation. In *Handbook of Philosophical Logic, 2nd ed.*, D.Gabbay, Ed. Kluwer Academic Pub., 2000.

Presenta un excelente *survey* sobre el estado del arte de la argumentación rebatible. Introduce inicialmente los conceptos fundamentales de la argumentación rebatible, comunes a la mayoría de los formalismos: argumentos, ataque, comparcion y derrota. Siguiendo la notación de [Dun95] define los conceptos involucrados en el análisis entre argumentos y derrotadores, clasificándolo en dos propuestas: "*unique status-assigments approach* ", y "*multiple status-assigments approach* ". Luego, resume a los principales sistemas existentes: [BDKT97, Pol95, Vre93, SL92, Pra97, Nut94], entre otros.

[Rei78] REITER, R. On Closed-world Data Bases. In *Logic and Data Bases*, Gallaire and Minker, Eds. Plenum Press, New York, 1978, pp. 55–76.

[Rob65] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. In *Journal ACM 12* (January 1965), pp. 23–41.

Este trabajo presenta por primera vez un algoritmo de resolución.

[Rob00a] ROBOCUP. Related information, papers and references. In *the Official Web Page of RoboCup* (http://www.robocup.org/, 2000).

[Rob00b] RoboLog. Related information, papers and references. In *the Official Web Page of RoboLog* (http://www.uni-koblenz.de/ag-ki/ROBOCUP/SIMULATION/simulator.html, 2000).

[SCG94] Simari, G. R., Chesñevar, C. I., and García, A. J. The role of dialectics in defeasible argumentation. In *XIV International Conference of the Chilenean Computer Science Society* (November 1994).

En este trabajo se trata por primera vez el problema de líneas de argumentación falaces. Se introduce el concepto de árbol de dialéctica aceptable, y marcado de un árbol de dialéctica. Este trabajo dió origen al estudio de las falacias que luego fue extendido en muchos aspectos en trabajos posteriores, e incluso en esta tesis..

[SG95] Simari, G. R., and García, A. J. A knowledge representación language for defeasible argumentation. In *CLEI'95, Canela, Brasil* (August 1995).

En este trabajo se introduce por primera vez un lenguaje de representación de conocimiento que evolucionaría luego al lenguaje de la programación en lógica rebatible (DeLP) actual..

[Sim89] Simari, G. R. *A Mathematical Treatment of Defeasible Reasoning and its Implementation*. PhD thesis, Washington University, Department of Computer Science (Saint Louis, Missouri, EE.UU.), December 1989.

[SL92] Simari, G. R., and Loui, R. P. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence 53* (1992), 125–157.

[SOMB99a] Stolzenburg, F., Obst, O., Murray, J., and Bremer, B. RoboLog Koblenz: Spatial agents implemented in a logical expressible language. In *Team Descriptions — Simulation League* (1999), S. Coradeschi, T. Balch, G. Kraetzschmar, and P. Stone, Eds., Linköping University Electronic Press, pp. 116–120.

[SOMB99b] Stolzenburg, F., Obst, O., Murray, J., and Bremer, B. Spatial agents implemented in a logical expressible language. In *Proceedings of the 3rd International Workshop on RoboCup in Conjunction with 16th Joint International Conference on Artificial Intelligence* (Stockholm, 1999), M. M. Veloso, Ed., IJCAI press, pp. 205–210.

[SOMB00] Stolzenburg, F., Obst, O., Murray, J., and Bremer, B. Spatial agents implemented in a logical expressible language. In *RoboCup-99: Robot Soccer WorldCup III*, M. Veloso, E. Pagello, and H. Kitano, Eds., LNAI. Springer, 2000. To appear.

[Tar99a] Tarau, P. Inference and Computation Mobility with Jinni. In *The Logic Programming Paradigm: a 25 Year Perspective* (1999), K. Apt, V. Marek, and M. Truszczynski, Eds., Springer, pp. 33–48. ISBN 3-540-65463-1.

[Tar99b] Tarau, P. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents* (London, U.K., 1999), pp. 109–123.

[Tea95] Team, R. Robocup. In *in Proc. of IJCAI-95 Workshop on Entertainment and AI/Alife* (Montreal, 1995).

[Ten91] Tennet, R. *Semantics of Programming Languages*. Prentice Hall, 1991.

[TK95] Toni, F., and Kakas, A. C. Computing the acceptability semantics. In *Proceedings of the 3rd. International Workshop on Logic Programming and Non-monotonic reasoning* (Lexington,USA, June 1995), Springer Verlag, pp. 401–415.

Presentan una teoría de prueba y un procedimiento de prueba para la semántica de aceptabilidad definida en [KMD94]..

[Ver96] Verheij, B. *Rules, Reasons, Arguments: formal studies of argumentation and defeat*. PhD thesis, Maastricht University, Holland, December 1996.

[Ver98] Verheij, B. Argue! an implemented system for computer-mediated argumentation. In *Proc. of the 10th Netherlands/Belgium Conference on Artificial Intelligence* (1998), CWI, Amsterdam, pp. 57–66.

[Vre93] Vreeswijk, G. A. *Studies in Defeasible Argumentation*. PhD thesis, Vrije University, Holland, 1993.

[Vre97] Vreeswijk, G. A. Abstract argumentation systems. *Artificial Intelligence 90* (1997), 225–279.

[WYY97] Wang, X., You, J., and Yuan, L. Logic programming without default negation revisited. In *Proceedings of IEEE International Conference on Intelligent Processing Systems* (1997), IEEE, pp. 1169–1174.

Presentan una reformulación de la programación en lógica sin negación por falla de [KMD94]. El formalismo resultante se llama "Priority Logic Programming". Un programa es un conjunto de reglas y una relación de orden entre las reglas. Una regla puede ser usada si no existe otra regla de mayor prioridad que la bloquee. Introduce además una transformación de programas normales a programas con prioridades, que lamentablemente presenta problemas..