Una extensión de la máquina abstracta de Warren para la argumentación rebatible. [1]

Alejandro J. García[2] Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Departamento de Ciencias de la Computación,
Universidad Nacional del Sur
Av.Alem 1253 – (8000) Bahía Blanca, ARGENTINA
FAX: (54) (91) 563401
e-mail: ccgarcia@criba.edu.ar grs@criba.edu.ar

Resumen

La *máquina abstracta de Warren*, o Warren's Abstract Machine (WAM) es una máquina virtual que consiste de una arquitectura de memoria y un conjunto de instrucciones diseñadas especialmente para la ejecución de Prolog. Actualmente la WAM ha convertido en el estándar *de facto* para la implementación de ese lenguaje.

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

La *argumentación rebatible* es una formalización del *razonamiento rebatible*, en el cual las conclusiones obtenidas pueden ser rechazadas ante la aparición de nueva evidencia. Un *argumento* para una conclusión c constituye una pieza de razonamiento tentativa que un agente inteligente está dispuesto a aceptar para explicar c. La *programación en lógica rebatible* es una extensión de la programación en lógica que contiene todas las características de la *argumentación rebatible*.

En este trabajo se presenta la extensión de la WAM, para obtener una máquina abstracta para la programación en lógica rebatible. Para el diseño de la nueva máquina abstracta se construyó una nueva arquitectura de memoria y un nuevo conjunto de instrucciones con características especiales para la argumentación rebatible.

[1] Financiado parcialmente por la Secretaría de Ciencia y Técnica, Universidad Nacional del Sur.

[2] Becario del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), República Argentina

1. Introducción

La *argumentación rebatible* [16, 15, 3] es una formalización del razonamiento rebatible, donde se pone especial énfasis en la noción de *argumento*. Un argumento para una conclusión C constituye una pieza de razonamiento tentativa que un agente inteligente está dispuesto a aceptar para explicar C. Si el agente adquiriese luego nueva información, la conclusión C junto con el razonamiento que la produjo podrían quedar invalidados.

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

La programación en lógica [9] se ha convertido en uno de los principales exponentes de la programación declarativa. Sin embargo, aunque se la ha utilizado como herramienta de representación de conocimiento, presenta limitaciones para adaptarse al razonamiento rebatible. Como los agentes inteligentes tienden a razonar en forma rebatible, sería sumamente interesante disponer de un paradigma de programación donde conclusiones previas puedan ser refutadas ante la presencia de mayor información.

La *programación en lógica rebatible* [17, 7, 6] es una una extensión de la programación en lógica convencional, que utiliza los conceptos de la argumentación rebatible a fin de capturar aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Los programas lógicos rebatibles (PLR) permiten la representación de información incompleta y potencialmente inconsistente, y pueden decidir entre metas contradictorias.

En 1977 David H. D. Warren, diseñó una *máquina abstracta* para la ejecución de Prolog, que consiste de una arquitectura de memoria y un conjunto de instrucciones [20, 21]. Este diseño se denominó posteriormente Maquina Abstracta de Warren o WAM (Warren's Abstract Machine) y se ha convertido en el estándar *de facto* para la implementación de compiladores para el lenguaje Prolog [1]. La ventaja de tener una máquina abstracta (MA) es que dado un programa P en un lenguaje $\mathcal{L}$, el programa P puede ser traducido a instrucciones de la M, y luego ejecutar sobre la arquitectura de la máquina virtual dichas instrucciones. La MA puede implementarse como una máquina virtual, o directamente sobre una máquina real, o diseñarse un nuevo procesador con la arquitectura de la MA diseñada. En el caso de la WAM, estas tres alternativas han sido desarrolladas [18].

El objetivo de este trabajo es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. La nueva máquina abstracta (que se llamará JAM), estará diseñada como una extensión de la WAM. La arquitectura de la JAM estará formada al igual que en la WAM por un conjunto de instrucciones, una estructura de memoria, y un conjunto de registros, que permitirán construir argumentos, contraargumentos y generar el árbol de dialéctica necesario para obtener una justificación. Disponer de una máquina abstracta para la argumentación rebatible permitirá desarrollar este tipo de sistemas de una manera eficiente.

2. Programación en lógica rebatible

En esta sección se describirá brevemente el lenguaje de programación en lógica rebatible, más detalles pueden encontrarse en [17, 4, 7, 6, 8]. La programación en lógica rebatible es una una extensión de la programación en lógica convencional, que utiliza los conceptos de la argumentación rebatible a fin de capturar aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Los programas lógicos rebatibles permiten la representación de información incompleta y potencialmente inconsistente, y pueden decidir entre metas contradictorias, utilizando un criterio de preferencia.

En el lenguaje de la programación en lógica rebatible, un *literal* "l" es un átomo "a" o un átomo negado "$\sim a$", siguiendo la definición de Lloyd [9]. El símbolo "$\sim$" representará la negación clásica, y "`not`" la negación por falla. El símbolo " $-$ " en cambio, se utilizará para indicar el complemento de un literal con respecto a la negación clásica, *i.e.*, $\overline{a} = \sim a$, y $\overline{\sim a} = a$.

Un *programa lógico rebatible* (PLR), es un conjunto finito de *cláusulas de programa extendido* (CPE) y *cláusulas de programa rebatible* (CPR). Una CPE es una cláusula de la forma "$l \leftarrow p_1, \ldots, p_n$", $(n \geq 0)$ donde l es un literal, y cada p_i es un literal o un literal precedido por el símbolo `not` de la negación por falla. Si n=0, entonces se denotará "$l \leftarrow$ `true`", y se dirá que l es un *hecho* (donde "`true`" tiene la interpretación usual). Una CPR en cambio, es una cláusula de la forma "$l \prec p_1, \ldots, p_n$", con las mismas consideraciones para los p_i que en las CPE. El símbolo " $\prec$ " se utiliza para distinguir una CPR de una CPE, porque una CPR se utilizará para representar conocimiento rebatible, *i.e.*, información tentativa que puede ser usada en la medida que no sea contradecida. La cláusula "$l \prec A$" debe leerse como: "razones para creer en el antecedente A son buenas razones para creer en el consecuente l". En una CPR, si n=0 se denotará "$l \prec$ `true`", y se dira que l es una *presuposición*. De esta forma, un programa estará formado por dos conjuntos disjuntos de cláusulas: uno que representa conocimiento estricto (seguro), y otro que contiene información rebatible (tentativa).

En lo que sigue se denotará con $\mathcal{S}$, al conjunto de CPE de un PLR, y con $\mathcal{D}$, al conjunto de CPR. Una *meta* (o consulta) es simplemente un literal m. Una *derivación rebatible* para una meta m es el conjunto de cláusulas de programa instanciadas (CPE y CPR), que permiten derivar m. El símbolo "$|\sim$ " será una abreviatura de "deriva rebatiblemente a", por lo tanto, si existe una derivación rebatible de m a partir de $\mathcal{P}$, se notará $\mathcal{P} |\sim m$, Un conjunto de cláusulas es *consistente* si no es posible derivar rebatiblemente un par de literales complementarios. Analogamente, un conjunto de cláusulas es *inconsistente* si es posible derivar rebatiblemente un par de literales complementarios. En un PLR, el conjunto $\mathcal{S}$ de CPE debe ser consistente, mientras que el conjunto $\mathcal{D}$ de CPR y el propio PLR pueden ser inconsistentes.

El objetivo de incluir la negación clásica es poder representar información potencialmente inconsistente, ya que tanto una meta de la forma "$\sim a$", como una de la forma "a" pueden ser derivadas rebatiblemente a partir de un PLR. Esto permite escribir cláusulas como "$\sim$`peligroso(X)` $\prec$ `niño(X)`" o "`peligroso(X)` $\prec \sim$`conocido(X)`". La negación por falla, en cambio, se incluye para poder trabajar con información incompleta. Es por ello que la utilización de la negación por falla se restringe sólo al cuerpo de una cláusula, por ejemplo, "$\sim$`cruzar-la-via(V)` $\prec$ `not` $\sim$`viene-tren-por(V)`", o "`buscar(X)` $\prec$ `perdido(X),not muerto(X)`". De esta forma, el lenguaje permite la representación de información incompleta y potencialmente inconsistente.

Ejemplo 2.1 : A continuación se muestra un programa donde puede verse la expresividad de los PLR:

```
vuela(X) —≺ ave(X).                          ave(X) ← gallina(X).
ave(X) ← pingüino(X).                        ∼vuela(X) —≺ gallina(X).
∼vuela(X) ← pingüino(X).                     vuela(X) —≺ gallina(X),asustado(X).
pingüino(petete) ← true.                     gallina(coco) —≺ true.
anida-arbol(X) —≺ vuela(X).                  asustado(coco) —≺ true.
anida-suelo(X) —≺ not anida-arbol(X).
```

Obsérvese que a partir del PLR anterior, pueden derivarse rebatiblemente las metas"`vuela(coco)`" y "`vuela(petete)`", pero también pueden derivarse rebatiblemente "∼`vuela(coco)`" y "∼`vuela(petete)`".

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

Definición 2.1 : Dado un PLR, formado por el conjunto $\mathcal{S}$ de CPE, y el conjunto $\mathcal{D}$ de CPR, un *argumento* $\mathcal{A}$ para una meta m, es un subconjunto de CPR instanciadas de $\mathcal{D}$, tal que:
(1) Existe una derivación rebatible de m a partir de $\mathcal{S} \cup \mathcal{A}$ (*i.e.*, $\mathcal{S} \cup \mathcal{A} |\sim\ m$),
(2) $\mathcal{S} \cup \mathcal{A}$ es consistente, y
(3) $\mathcal{A}$ es el menor subconjunto (con respecto a la inclusión de conjuntos) que cumple las dos condiciones anteriores.
Si $\mathcal{A}$ es un argumento para m, también se dirá que $\langle\ \mathcal{A}, m\ \rangle$ es una *estructura de argumento*.
Un argumento $\langle \mathcal{B}, q \rangle$ es un *subargumento* de $\langle \mathcal{A}, h \rangle$ si $\mathcal{B} \subseteq \mathcal{A}$.

En el ejemplo 2.1 existen derivaciones rebatibles para "`vuela(coco)`", y "`vuela(petete)`", pero sin embargo, sólo "`vuela(coco)`" tiene un argumento, ya que la derivación de "`vuela(petete)`" es inconsistente con $\mathcal{S}$. A continuación se muestra un argumento $\mathcal{A}_1$ para "∼`vuela(coco)`", y dos argumentos $\mathcal{A}_2$ y $\mathcal{A}_3$ para "`vuela(coco)`".

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

En la programación en lógica convencional hay dos respuesta posibles a una consulta: "SI" cuando se puede derivar con éxito la consulta, y "NO" en caso contrario. Por lo tanto, si se usara el método de inferencia de la programación en lógica tradicional, ante la consulta " —≺ `vuela(coco)`" el programa respondería SI, pero ante la consulta " —≺ ∼`vuela(coco)`" también respondería SI. Lo que se propone hacer en este lenguaje, es utilizar el mecanismo de inferencia de la argumentación rebatible, el cuál puede decidir entre conclusiones contradictorias utilizando un criterio de comparación. Utilizando este método de inferencia, habrá cuatro respuestas posibles: SI, NO, DESCONOCIDO, e INDECISO.

El proceso de obtención de una justificación para m, involucra la construcción de un argumento $\mathcal{A}$ que no esté derrotado. Para verificar si $\mathcal{A}$ está derrotado, se construyen a partir del PLR, *contraargumentos* que son posibles *derrotadores* de $\mathcal{A}$. Tanto la argumentación rebatible, como la programación en lógica rebatible, pueden definirse independientemente del criterio de preferencia entre argumentos, no obstante, en este trabajo se asumirá que se utiliza el criterio de *especificidad* [10, 13, 7].

Definición 2.2 : Sea $\mathcal{S}$ el conjunto de CPE del PLR. Se dirá que $\langle \mathcal{A}_2, h_2 \rangle$ *contraargumenta a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si, existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que $\mathcal{S} \cup \{h, h_2\}$ es inconsistente. El argumento $\langle \mathcal{A}, h \rangle$ se llamará subargumento de desacuerdo, y al literal h se lo llamará punto de contraargumentación.

Definición 2.3 : Un argumento $\langle \mathcal{A}_2, h_2 \rangle$ *derrota a* $\langle \mathcal{A}_1, h_1 \rangle$ en un literal h, si y sólo si existe un subargumento $\langle \mathcal{A}, h \rangle$ de $\langle \mathcal{A}_1, h_1 \rangle$ tal que: $\langle \mathcal{A}_2, h_2 \rangle$ contraargumenta a $\langle \mathcal{A}_1, h_1 \rangle$ en el literal h y se cumple una de estas dos opciones:
(1) $\langle \mathcal{A}_2, h_2 \rangle$ es estrictamente más específico que $\langle \mathcal{A}, h \rangle$ (derrotador propio), o
(2) $\langle \mathcal{A}_2, h_2 \rangle$ no puede compararse con $\langle \mathcal{A}, h \rangle$ (derrotador de bloqueo).

A continuación se mostrará cómo es el proceso por el cuál se decide que un argumento es una justificación para una meta m. Como se explicó antes, un argumento será una *justificación* de m, cuando no esté derrotado. Un argumento estará derrotado cuando poseea derrotadores que a su vez no estén derrotados. Por lo tanto, para decidir si un argumento está derrotado o no, hay que construir un árbol de argumentos, donde los nodos son derrotadores de sus padres. La definición de árbol de dialéctica es la que caracteriza a este proceso.

Definición 2.4 : Un *árbol de dialéctica* para $\langle \mathcal{A}, h \rangle$, denotado $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, se define como sigue:

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

Sea $\langle \mathcal{A}_0, h_0 \rangle$ una estructura de argumento, y $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$ su árbol de dialéctica asociado. Todo camino λ en $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$ desde la raíz $\langle \mathcal{A}_0, h_0 \rangle$ hasta una hoja $\langle \mathcal{A}_n, h_n \rangle$, denotado $\lambda = [\ \langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$, constituye una *línea de argumentación* para $\langle \mathcal{A}_0, h_0 \rangle$. En λ, $\langle \mathcal{A}_0, h_0 \rangle$ será un argumento de soporte, y si $\langle \mathcal{A}_i, h_i \rangle$ es un argumento de soporte (resp. interferencia) en λ, entonces $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ es un argumento de interferencia (resp. soporte) en λ. Se denota con λ_S al conjunto de argumentos de soporte pertenecientes a λ, y con λ_I al conjunto de argumentos de interferencia de λ.

Un árbol de dialéctica se dirá aceptable, si todas sus líneas de argumentación λ son aceptables, esto es, cumplen las siguientes condiciones:
(1) Todo derrotador $\langle \mathcal{A}_i, h_i \rangle$ de la línea cumple que para todo subargumento propio $\langle \mathcal{B}, q \rangle$ de $\langle \mathcal{A}_i, h_i \rangle$, no sea el caso que $\langle \mathcal{B}, q \rangle$ y $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$ estén en desacuerdo.
(2) λ_S y λ_I, deben contener argumentos concordantes, esto es, $\mathcal{S} \cup \lambda_S \not\sim \bot$, y $\mathcal{S} \cup \lambda_I \not\sim \bot$.
(3) Un argumento $\langle \mathcal{A}_n, h_n \rangle$ no debe ser un subargumento de otro $\langle \mathcal{A}_i, h_i \rangle$ con $i < n$.

Definición 2.5 : Los nodos de un árbol de dialéctica $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ se etiquetan recursivamente como *nodo no-derrotado* (nodo-U) o *nodo derrotado* (nodo-D) de la siguiente forma:

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

Definición 2.6 : Un argumento $\langle \mathcal{A}, h \rangle$ es una *justificación* para h sssi la raíz del árbol de dialéctica aceptable $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un nodo-U.

Al efectuar la consulta "∼`vuela(coco)`" al PLR del ejemplo 2.1, se puede construir el argumento $\mathcal{A}_1$ mostrado antes. Pero el argumento $\mathcal{A}_2$ es un derrotador para $\mathcal{A}_1$, y como no existe otro argumento que derrote a $\mathcal{A}_2$, entonces $\mathcal{A}_2$, es un nodo-U en el árbol de dialéctica. Por lo tanto $\mathcal{A}_1$ no es una justificación. Sin embargo, si se efectúa la consulta "`vuela(coco)`", por el orden en que están dispuestas las cláusulas, primero se obtiene el argumento $\mathcal{A}_3$. En este caso el argumento $\mathcal{A}_1$ es un derrotador de $\mathcal{A}_3$, pero como $\mathcal{A}_2$ es a su vez un derrotador de $\mathcal{A}_1$, entonces $\mathcal{A}_1$ no es un derrotador aceptable y por lo tanto $\mathcal{A}_3$, es una justificación para "`vuela(coco)`".

Ahora es posible definir los dos tipos de negación del lenguaje. El significado de "∼`a`" es "*existe una justificación para* ∼`a`", mientras que el de "`not a`" es "*no existe una justificación para* `a`". Por lo tanto, la negación por falla en los PLR tiene un significado distinto a la negación por falla en PROLOG.

Definición 2.7 : Sea $\mathcal{P}$ un PLR:

1. El conjunto de respuestas positivas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, existe un argumento $\mathcal{A}$ que es una justificación para h.
2. El conjunto de respuestas negativas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, se cumple que para todo argumento $\mathcal{A}$ de h, existe un derrotador propio de $\mathcal{A}$, rotulado como *nodo-U*.
3. El conjunto de respuestas indecisas de $\mathcal{P}$ es un conjunto de literales L tal que para todo $h \in L$, se cumple que para todo argumento $\mathcal{A}$ de h, $\mathcal{A}$ no tiene derrotadores propios (o son nodos-D), pero si tiene al menos un derrotador de bloqueo, rotulado como *nodo-U*.
4. El conjunto de respuestas desconocidas de $\mathcal{P}$ es un conjunto posiblemente infinito de literales L tal que para todo $h \in L$, h no pertenece a los conjuntos de respuestas anteriores.

Dado un PLR $\mathcal{P}$ y una meta definida m, un intérprete de programas lógicos rebatibles, responderá: SI, NO, INDECISO, o DESCONOCIDO, en el caso que m pertenezca al conjunto de respuestas positivas, negativas, indecisas, o desconocidas, respectivamente. Si la consulta está precedida por `not`, entonces la respuesta sólo podrá ser SI, o DESCONOCIDO.

3. La Máquina Abstracta de Warren

La Maquina Abstracta de Warren o WAM (Warren's Abstract Machine) [20, 21] se ha convertido en el estándar *de facto* para la implementación del lenguaje PROLOG. El objetivo de este trabajo es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible, utilizando la WAM como punto de partida. Por lo tanto, en esta sección se describirá brevemente a la WAM. Más detalles sobre la WAM pueden encontrarse en [21] y [1] .

La WAM es una máquina abstracta que consiste de una arquitectura de memoria y un conjunto de instrucciones diseñadas especialmente para la ejecución de PROLOG. Los programas son traducidos a estas instrucciones y luego se ejecutan sobre la máquina abstracta. En general, cada símbolo de PROLOG corresponde a una instrucción WAM. La máquina abstracta puede implementarse como una máquina virtual, o directamente sobre una máquina real, o diseñarse un nuevo procesador con esta arquitectura. En el caso de la WAM, estas tres alternativas han sido desarrolladas [18].

Ejemplo 3.1 : A continuación se muestra un programa escrito en PROLOG, cuyo correspondiente código WAM está en la figura 1. En el código WAM cada etiqueta de la forma "`p/n:`" o "`E:`" marca el comienzo del código de cada cláusula de programa, mientras que la etiqueta "`?:`" marca el comienzo del código de la consulta.

```
p(X) :- q(X).
p(X) :- r(X).
p(X) :- s(X).
s(a).
:- p(X).
```

```
p/1:  try_me_else E1
      allocate 1
      get_variable X1 A1
      put_value X1 A1
      call q/1
      deallocate
      proceed
E1:   retry_me_else E2
      allocate 1
      get_variable X1A1
      put_value X1A1
      call r/1
      deallocate
      proceed
```

```
E2:   trust_me
      allocate 1
      get_variable X1 A1
      put_value X1 A1
      call s/1
      deallocate
      proceed
s/1:  just_me
      allocate
      get_constant a A1
      deallocate
      proceed
```

```
?:    allocate 1
      put_variable X1 A1
      call p/1
      deallocate
      proceed
```

Figura 1: Código WAM del programa del ejemplo 3.1

La ejecución del código WAM comienza siempre en la etiqueta "`?:`". Obsérvese que el modelo de ejecución de la WAM consiste simplemente en una consulta que llama y un hecho (o cabeza de una cláusula) que espera ser llamado. Como las cláusulas a su vez tienen consultas, esto produce el encadenamiento hacia atrás (backward chaining) de cláusulas. Para ejecutar el programa, la WAM posee un registro llamado P que almacena la dirección de la instrucción en curso. Después de ejecutar cada instrucción, el registro P es incrementado a fin de disponer de la dirección de la siguiente instrucción, salvo cuando se ejecuta la instrucción `call`, que modifica al registro P asignándole la dirección del código del predicado a ejecutar. Además de P, existe un registro CP que almacena el punto de retorno de una llamada, esto es, la dirección siguiente a la instrucción `call`. Las instrucciones `allocate` y `deallocate` son las encargadas de mantener en el STACK el el punto de retorno y el valor de las variables que deben preservarse cada vez que se llama a una nueva cláusula (ver figura 3). Una particularidad de los programas lógicos, es que puede haber varias cláusulas que definan un mismo predicado. Esta característica requiere de un mecanismo de *backtracking*, el cuál está implementado en la WAM utilizando las instrucciones `try_me_else`, `retry_me_else`, y `trust_me`, las cuales crean *puntos de elección* que son guardados en el STACK (ver figura 3), y en ellos se almacena la información que debe restaurarse en caso que una cláusula falle, y deba buscarse otra forma de probar la consulta. En el ejemplo 3.1 la consulta tiene éxito gracias a la tercera cláusula que define a `p/1`.

La organización de la memoria de la máquina abstracta puede verse en la figura 2, donde se muestra cada área de la memoria y la dirección hacia donde crece. También están indicados en la misma figura, los registros que tienen punteros a cada una de las áreas de memoria mencionadas. En primer lugar se encuentra el código del programa que ha sido compilado y traducido a instrucciones WAM. El HEAP es una pila que contiene todas las estructuras creadas por unificación. El STACK también es una pila, que contiene dos clases de objetos: los *entornos* (environments), y los *puntos de elección* (choice points). El TRAIL contiene referencias a variables que han sido ligadas durante la unificación y que deben ser desligadas al hacer backtracking. Por último, se

encuentra una pila llamada PDL (push-down list), que se utiliza para realizar la unificación de términos

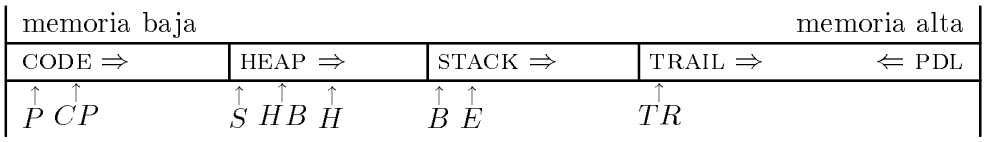

Figura 2: Disposición de la memoria y registros.

Además de los registros anteriores, existe un conjunto de registros llamados *registros de términos* y *registros de parámetros*. Los registros de términos se notarán X_j y se utilizarán durante la ejecución del programa lógico para almacenar temporariamente apuntadores a términos en el HEAP. Los registros de parámetros se notarán A_i y se utilizarán para implementar el pasaje de parámetros entre una consulta que invoca a otra. Todo identificador del programa lógico, (esto es, símbolos de constantes, símbolos de los functores, nombres de predicados, etc.) se almacenará en una estructura separada del HEAP, llamada *tabla de identificadores* o IDENT.

La estrucutura de memoria que se utiliza para almacenar todos los términos del programa, es el HEAP. Una variable puede tener ligado un término, o estar sin ligar. Por lo tanto una variable será representada en el HEAP como una dirección al término a la que esté ligada, o como una referencia a sí misma en el caso que esté sin ligar. Una constante estará representada en el HEAP por una referencia a su identificador. Mientras que una estructura de la forma $f(t_1, \ldots, t_n)$ estará representada como la secuencia $f/n, @_1, @_2, \ldots, @_n$ donde $@_i$ es la dirección de memoria de los subtérminos de f/n.

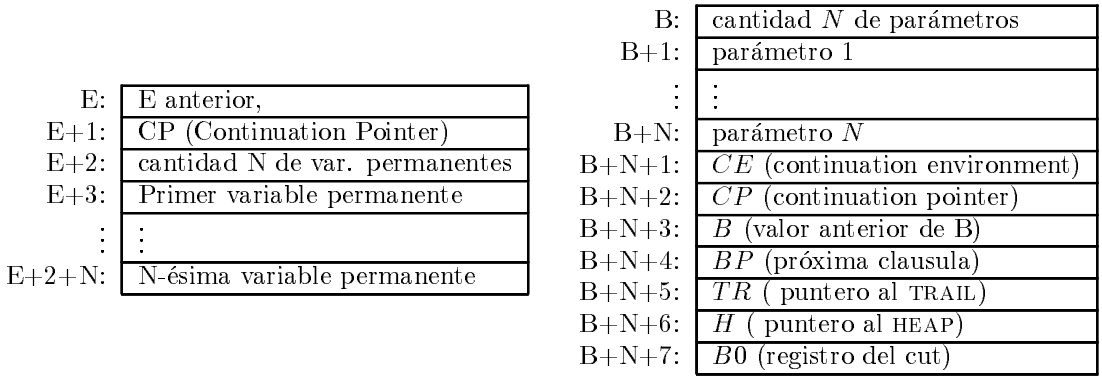

Figura 3: Registro de entorno y registro del punto de elección.

En el STACK son almacenados los registros de entorno, y los puntos de elección. De esta forma, los puntos de elección sirven de protección a los entornos que están bajo ellos, impidiendo que sean sobreescritos durante la ejecución, y puedan ser recuperados durante el backtracking. En el caso que no sea necesario utilizar un punto de elección, este se desapila liberando además la memoria de los entornos que estaban protegidos bajo él. Por lo tanto, la pila "AND" de entornos y la pila "OR" de puntos de elección se mantendrán en la misma pila. Como el STACK contiene dos tipos de registros con información diferente, tendrá un puntero para el tope de cada tipo de registro. En la figura 3 se muestra la información que mantienen los entornos y los puntos de elección.

4. Una máquina abstracta para la programación en lógica rebatible

El objetivo de este trabajo es la definición de una máquina abstracta para la implementación de la programación en lógica rebatible. En esta sección se mostrará el desarrollo de una nueva máquina abstracta llamada JAM (Justification Abstract Machine), la cuál está diseñada como una extensión de la WAM.

La arquitectura de la JAM está formada al igual que en la WAM por un conjunto de intrucciones, una estructura de memoria, y un conjunto de registros. Además de las estructuras de memoria de la WAM, la JAM posee: una pila llamada LINEA que mantiene la linea de argumentación en curso (*i.e.*, una rama del árbol de dialéctica); una tabla de hechos temporarios (THT) donde estarán representados los argumentos; y un nuevo HEAP llamado T-HEAP, para almacenar las estructuras internas de los hechos temporarios.

4.1. Construcción de argumentos

Para construir un argumento para una consulta `q`, se debe construir una derivación rebatible de `q`, y verificar que sea consistente. La derivación rebatible se realizará utilizando el mismo mecanismo que utiliza la WAM para derivar una meta en PROLOG, con la diferencia que se utilizarán tanto CPEs, como CPRs. A continuación se mostrará cómo la JAM verifica la consistencia de las derivaciones rebatibles.

Dado que el conjunto $\mathcal{S}$ es consistente, la proposición siguiente indica cuando un conjunto de nuevos literales puede asumirse consistentemente con $\mathcal{S}$. Se llamará $Co(\mathcal{A})$ al conjunto de literales instanciados que son consecuentes de las CPRs de un conjunto de cláusulas $\mathcal{A}$.

Proposición 4.1 : [8] *Sea $\mathcal{S}$ el conjunto de* CPEs *de un* PLR*, y $\mathcal{A}$ un argumento para una meta m. El conjunto $\mathcal{S} \cup \mathcal{A}$ será consistente si y sólo si el conjunto $\mathcal{S} \cup Co(\mathcal{A})$ es consistente.*

La verificación de la consistencia se llevará a cabo simultáneamente con la construcción del argumento, y utilizando únicamente encadenamiento hacia atrás. Para esto, cada vez que se utiliza una CPR "$c \prec L$" en la derivación rebatible de una meta h, se verificará si $\mathcal{S} \cup \{c\}$ es consistente, es decir, si es posible asumir el consecuente c consistentemente con $\mathcal{S}$. De esta forma, una CPR instanciada (grounded) "$c \prec L$" se dirá *aceptada* y por lo tanto podrá ser utilizada en la construcción de un argumento $\mathcal{A}$, sólo si $\mathcal{S} \cup Co(\mathcal{A}') \cup \{c\}$ es consistente. Donde $\mathcal{A}'$ es el conjunto de CPR aceptadas previamente en la construcción de un argumento $\mathcal{A}$, y por lo tanto $\mathcal{S} \cup Co(\mathcal{A}')$ es consistente. Cada elemento h de $Co(\mathcal{A}')$ se llamará *hecho temporario*, ya que, una vez que h fue aceptado, se comportará igual que un hecho del PLR. Será temporario porque sólo existirá como un hecho durante la construcción del argumento.

Cuando se quiere verificar que un conjunto $\mathcal{S} \cup \{h\}$ es consistente se debe verificar que $\mathcal{S} \cup \{h\}$ no derive un par de literales complementarios. Si el par de literales fuera h y $\overline{h}$, entonces sería sencillo de implementar, pero esto no siempre es así. Por ejemplo, si el conjunto $\mathcal{S}$ es { $\sim$`p` $\leftarrow$ `q` ; `p` $\leftarrow$ `h` ; `q` $\leftarrow$ `true` }, $\mathcal{S} \cup \{h\}$ es inconsistente, y el par de literales es `p` y $\sim$`p`. Si se pudiera asumir la contrapositiva de las CPE, en el ejemplo anterior se podría haber generado la siguiente derivación para $\sim$`h`: { $\sim$`h` $\leftarrow$ $\sim$`p` ; $\sim$`p` $\leftarrow$ `q` ; `q` $\leftarrow$ `true` }. El problema es que las CPE no son contrapositivas. Esto es, si se tiene la CPE "`p` $\leftarrow$ `h`" no se puede asumir que exista la cláusula "$\sim$`h` $\leftarrow$ $\sim$`p`". Aunque la contrapositiva no pueda utilizarse en las derivaciones rebatibles, se usarán CPE *invertidas* sólo para verificar la existencia del par de literales complementarios, y no para producir una derivación.

Definición 4.1 : Invertir una CPE "$p \leftarrow a_1, \ldots, a_i, \ldots, a_n$" significa reemplazar el consecuente con el complemento de un antecedente, y dicho antecedente con el complemento del consecuente, obteniendo por ejemplo la siguiente CPE invertida: "$\overline{a_i}$ @— $a_1, \ldots, \overline{p}, \ldots, a_n$". Se utilizará el símbolo " @— " para distinguir una CPE invertida de una CPE común. Esto también se conoce como estrategia de Loveland [11].

Observación importante: Las CPE invertidas no son cláusulas de programa, y por lo tanto no pueden formar parte de ningún argumento. Estas cláusulas serán utilizadas solamente para acelarar la búsqueda de un par de literales complementarios durante la verificación de la consistencia, y la construcción de contraargumentos. Aunque el proceso de invertir una CPE se asemeje a la contraposición de la implicación no debe pensarse de ningún modo que las CPE sean contrapositivas. Las siguientes restricciones garantizan que su utilización sea la correcta: (1) Las CPE invertidas sólo pueden utilizarse al principio de una derivación, ya que no formarán parte de ella. (2) Una vez que una cláusula de programa es usada en la derivación, ya no se podrán utilizar más CPE invertidas. (3) Si la CPE "$p \leftarrow a_1, \ldots, a_i, \ldots, a_n$" es invertida para ser usada como "$\overline{a_i}$ @— $a_1, \ldots, \overline{p}, \ldots, a_n$". entonces los literales $a_1, \ldots, a_n$ deberán derivarse únicamente utilizando CPE o hechos temporarios.

Lema 4.1 : [8] *Dado un conjunto consistente de* CPE $\mathcal{S}$ *y un literal* h. *El conjunto* $\mathcal{S} \cup \{h\}$ *es inconsistente si y solo si existe una derivación de* $\overline{h}$ *a partir de* $\mathcal{S} \cup \{h\}$, *utilizando (si fuera necesario, y con las restricciones dadas) alguna* CPE *invertida.*

Teorema 4.1 : *Una* CPR "$h \prec L$" *puede ser utilizada en la construcción de* $\mathcal{A}$, *si y sólo si no es posible derivar* $\overline{h}$ *a partir de* $\mathcal{S} \cup Co(\mathcal{A}') \cup \{h\}$.
Demostración: una CPR "$h \prec L$" puede ser utilizada en la construcción de $\mathcal{A}$ si y solo si es aceptada, esto es, $\mathcal{S} \cup Co(\mathcal{A}') \cup \{h\}$ es consistente, lo cuál por el lema 4.1 es cierto si y sólo si no es posible derivar $\overline{h}$ a partir de $\mathcal{S} \cup Co(\mathcal{A}') \cup \{h\}$.

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

Se llamará *derivación fuerte* (en contraposición a una derivación rebatible), a una derivación que utilice únicamente las CPE del PLR, los hechos temporarios que se tenga en ese momento, y alguna CPE invertida. Por el teorema 4.1, probar la consistencia de una cláusula con cabeza "`p/n`", es equivalente a probar la no existencia de una derivación fuerte de una consulta "`p/n`", lo cual es equivalente a tener éxito en una derivación fuerte de "`naf p`", donde "`naf`" es la negación por falla de PROLOG. Para esto, el compilador generará al final de cada cláusula una consulta adicional que al compilarse generará el código necesario para la verificación de la consistencia. Esto es, se agregará el complemento de la cabeza de la cláusula, como una consulta adicional al final del cuerpo, precedida con el símbolo "`#`". Por ejemplo, la cláusula "`vuela(X) -< ave(X)`" se transformará en "`vuela(X) -< ave(X) # ~ vuela(X)`". El símbolo "`#`" debe interpretarse como "*y no puede probarse en forma fuerte*", esto es, "`# ~vuela(charo)`" debe leerse como "y no puede probarse en forma fuerte `~vuela(charo)`". Un preprocesador generará todas las CPE invertidas a partir de las CPE del programa, por lo tanto al compilar el programa, se generará código para las CPE, las CPR, la verificación de la consistencia, y las CPE invertidas.

Dos nuevas instrucciones "`prove_consistency_else`" y "`end_consistency_proof`" (ver figura 4) serán las encargadas de marcar el inicio y el fin de una prueba de consistencia. De esta forma, toda secuencia de código que esté entre estas dos instrucciones se ejecutará en modo "*derivación fuerte*". El símbolo `#` le indicará al compilador que lo que sigue es la verificación de consistencia de la cláusula que está compilando, y se generará entonces la instrucción `prove_consistency_else`. Luego se compilará la submeta que sigue al `#` como una submeta estandar del PLR, y al terminar con la compilación de la submeta, se agregará la instrucción `end_consistency_proof`. La instrucción `prove_consistency_else` tiene un paramentro `E` que indica la dirección siguiente a `end_consistency_proof`, a fin de poder saltar allí, cuando no es necesario verificar la consistencia.

```
PROCEDURE prove_consistency_else(E);
    IF (NOT Derivacion_rebatible)
        OR Prueba_consistencia
    THEN  PC:=E {salto a donde indica E}
    ELSE  Prueba_consistencia:=TRUE;
          Uso_invertidas:=TRUE;
END prove_consistency_else;

PROCEDURE set_defeasible;
    Derivacion_rebatible:=TRUE;
END set_defeasible;
```

```
PROCEDURE end_consistency_proof;
   Prueba_consistencia:=FALSE;
   Uso_invertidas:=FALSE;
END end_consistency_proof;

PROCEDURE forbid_inv;
   Uso_cpe_invertidas:=FALSE;
END forbid_inv;
```

Figura 4: Implementación de las instrucciones para verificación de consistencia

El registro `prueba_consistencia` será el encargado de bloquear el uso de CPRs, mientras que el registro `uso_cpe_invertidas` indicará que pueden utilizarse CPE invertidas. Al principio de la prueba de consistencia se podrán usar CPE invertidas, pero una vez que se utilice alguna CPE, o hecho temporario, ya no se podrán utilizar más. Por lo tanto, las CPE y los hechos temporarios, deben prohibir el uso posterior de una CPE invertida. Una nueva instrucción "`forbid_inv`" (ver figura 4) presente en el código de las CPE, pondrá al registro `uso_cpe_invertidas` en FALSE. Pero además el resto de una CPE invertida debe probarse sólo con CPE, o hechos temporarios, por lo tanto la misma CPE invertida, debe usar `forbid_inv`, luego de ejecutar la primer consulta de su cuerpo. Las instrucciones "`only_strong`" y " `stop_only_strong`" en el código de las CPE invertidas (ver figura 5) son las encargadas de garantizar que la prueba del cuerpo de una CPE invertida, se realice sólo con CPE y hechos temporarios, cuando no esté activa la verificación de la consistencia y se las utilice para construir contraargumentos (ver sección 4.2).

Para que efectivamente las CPR fallen cuando el registro `prueba_consistencia` está en TRUE, y que las CPE invertidas puedan ser utilizadas sólo cuando el registro `uso_cpe_invertidas` esté en TRUE, se crearon las siguientes instrucciones.

Para CPRs:	Para CPE invertidas:
`defeasible_just_me`	`just_me_checking_consistency`
`defeasible_try_me_else`	`try_me_checking_consistency_else`
`defeasible_retry_me_else`	`retry_me_checking_consistency_else`
`defeasible_trust_me`	`trust_me_checking_consistency`

Al compilar las cláusulas de programa, el código de las CPE mantendrá la primer instrucción igual que las cláusulas de PROLOG en la WAM. Sin embargo, el código de las CPR y las CPE invertidas tendrá como primera instrucción alguna de las que se indica en la tabla anterior. De esta forma se podrá diferenciar en tiempo de ejecución, que tipo de cláusula se está utilizando. El uso y la forma de generar estas nuevas instrucciones será el mismo que se utilizaba para las cuatro instrucciones en la WAM. En cuanto a la implementación de estas instrucciones, será prácticamente la misma que sus cuatro equivalentes en la WAM salvo que se les agrega al final del su implementación la sentencia:
"`IF (prueba_consistencia OR solo_cpe) THEN BackTrack`" en el caso de las CPR, y
"`IF (NOT uso_cpe_invertidas OR solo_cpe) THEN BackTrack`" a las CPE invertidas.
En la figura 5 se muestra el código JAM de la CPR "`vuela(X) -< ave(X)`" y de la CPE invertida "∼ `pingüino(X) @- vuela(X)`", ambas del PLR del ejemplo 2.1.

```
vuela/1:   defeasible_just_me
           allocate 1
           save_subterms vuela/1 CPR
           forbid_inv
           get_variable V1 A1
           put_value V1 A1
           call ave/1
           set_defeasible
           prove_consistency_else L
           put_structure ∼vuela/1 A1
           set_value V1
           call naf/1
           end_consistency_proof
      L:   save_fact vuela/1
           deallocate
           proceed
```

```
∼pingüino/1:   trust_me_checking_consistency
               allocate 1
               get_variable V1 A1
               put_value V1 A1
               call vuela/1
               forbid_inv
               only_strong
               stop_only_strong
               deallocate
               proceed
```

Figura 5: Código JAM de "`vuela(X) -< ave(X)`" y de "∼`pingüino(X) @- vuela(X)`".

Al comenzar la ejecución de cada consulta, un registro llamado `derivación_rebatible` es puesto en FALSE. De esta forma, mientras no sea usada ninguna CPR en la prueba de una consulta, el registro `derivación_rebatible` permanecerá en FALSE, indicando que no es necesaria la verificación de la consistencia de las reglas usadas en la derivación. Si una CPR es usada, la instrucción `set_defeasible` pondrá el registro en TRUE, y a partir de allí, se verificará la consistencia de toda submeta (ver figura 5). Dicho registro debe guardarse en los puntos de elección del STACK, a fin de que funcione adecuadamente con el backtracking.

Como se explicó antes, durante la construcción una derivación rebatible, cada vez que una CPR es aceptada, la cabeza de la cláusula, con la instanciación que tienen sus variables en ese momento, deben guardarse como un hecho más en el programa. Este hecho tiene su vida limitada a la derivación en curso, y es por esto que se lo denomina *hecho temporario*. Las instrucciones `save_subterms` y `save_fact`, son las encargadas de generar los hechos temporarios (ver figura 5). Como los hechos, las CPE invertidas, y por su puesto los hechos temporarios ya generados antes, no deben generar hechos temporarios, entonces no tendrán en su código JAM estas dos nuevas instrucciones.

Para generar los hechos temporarios se incluyeron en la arquitectura, dos nuevas áreas de memoria: una tabla de hechos temporarios (THT), que mantiene la información de los hechos tem-

porarios de cada argumento; y un "Heap Temporario " (llamado T-HEAP) donde se almacenarán los subtérminos de los hechos temporarios, en realidad referencias a las estructuras creadas durante la unificación y que corresponden a los hechos temporarios. Esto es, si "`p(f(a),b,[1,2])`" es un hecho temporario, la THT tendrá la información del predicado `p/3`, y en el T-HEAP habrá tres celdas de memoria consecutivas (una para cada subtérmino de `p`) con las referencias a las estructuras "`f(a)`", "`b`", y "`[1,2]`" que estarán almacenadas en el HEAP. El T-HEAP es una estructura de memoria igual al HEAP común, pero debe estar en memoria superior al STACK a fin de evitar problemas con las ligaduras entre variables. Tanto la THT como el T-HEAP funcionan como una pila que mantiene el recorrido en profundidad del árbol de derivación rebatible (esto facilitará la eliminación de elementos al hacer backtraking). Un registro global llamado T indicará el tope del T-HEAP (en realidad, la primer celda disponible). Otro registro global llamado F apuntará al tope de la THT. Ambos registros serán guardados en los puntos de elección, lo cuál permitirá que el backtracking deseche los hechos temporarios que no pueden usarse más.

La THT mantiene la siguiente información: (1) Posición del identificador del hecho temporario en la Tabla de Identificadores. (2) Posición de los subtérminos del hecho temporario en el T-HEAP. (3) Dirección de la primera instrucción del código del hecho temporario. (4) Posición del hecho temporario padre en árbol de derivación del argumento. (5) Tipo de cláusula que generó el hecho temporario: CPE o CPR. La información de la THT se utiliza para la verificación de la consistencia, la construcción del argumento, la construcción de contraargumentos, y la comparación por especificidad, por lo tanto algunos de sus elementos serán explicados más adelante.

```
PROCEDURE save_subterms(PosIdent,C)
  IF NOT Prueba_consistencia
  THEN
   Aridad:=IDENT[PosIdent].aridad;
   THT[F].PosIdent:=PosIdent;
   THT[F].punt_a_THeap:=T;
   THT[F].Tipo_clausula:=C;
   STACK[E+4]:=F;
   FOR i:=1 TO Aridad DO
          T-HEAP[T]:=A[i];
          T:=T+1;
   END for;
   F:=F+1;
  END if;
END save_subterms;
```

```
PROCEDURE unify_subterms(Ht)
  IF HechoTemporarioActivo(Ht)
  THEN
   i:=1;
   N:=IDENT[THT[Ht].PosIdent].Aridad;
   PosT:=THT[Ht].punt_a_THeap;
   WHILE NOT Falla AND (i <= N)
     DO  unify(Ai,PosT+i-1);
         i:=i+1;
     END while;
  ELSE Falla:=TRUE;
  END if;
  IF Falla THEN backtrack;
END; unify_subterms
```

Figura 6: Implementación de las instrucciones `save_subterms` y `unify_subterms`

Mientras un hecho temporario está activo, éste debe comportarse como un hecho más del programa, y por lo tanto, debe existir una secuencia de código JAM que lo represente. De esta forma, cuando se intente probar una submeta, los hechos temporarios estarán en el código del programa como una cláusula más, permitiendo el normal desenvolvimiento del flujo de control del programa, y fundamentalmente no interfiriendo con el backtracking. Por lo tanto, una vez que un hecho temporario es creado, debe generarse el código JAM correspondiente, y agregarse al área de código del programa. La instrucción `save_fact` realizará esta tarea.

Cuando se ejecuta la instrucción `save_subterms p/n C`, todavía no se sabe si `p/n` será un hecho temporario o no, por lo tanto sólo se reserva en la THT un lugar para el potencial hecho temporario. Esta instrucción tiene dos parámetros: el predicado `p/n` a guardar, y el tipo de cláusula que representa (CPE o CPR). Al ejecutarse (ver figura 6) realiza lo siguiente: (1) Almacena en la THT el lugar del T-HEAP donde quedarán los subtérminos. (2) Almacena en el registro de entorno del STACK, la dirección de la THT donde quedó el potencial hecho temporario, este valor será utilizado luego por la instrucción `save_fact p/n` para saber exactamente cuál es la entrada de la THT que debe actualizar. (3) Carga en el T-HEAP el contenido de los registros de argumento del predicado `p/n`.

Cuando se ejecuta la instrucción `save_fact p/n` es porque dicho hecho temporario ya ha sido probado y debe agregarse (temporariamente) al PLR. Por lo tanto, la instrucción `save_fact p/n` generará el código JAM para el hecho temporario que se acaba de crear, y modificará la THT y la tabla de identificadores para indicar la existencia de un nuevo hecho (temporario).

El código de los hechos temporarios es extremadamente simple, pues como son hechos, lo único que deben hacer es intentar unificarse con la consulta que llama. Además, como los hechos temporarios se formaron a partir de metas ya probadas, sus estructuras internas ya fueron escritas en el HEAP, y están referenciadas por los elementos del T-HEAP. Por lo tanto, un hecho temporario consta únicamente de tres instrucciones, la primera maneja el backtracking, la segunda unifica los subtérminos del predicado, y la tercera simplemente retorna el control en caso de una unificación exitosa.

```
C:  try_me_else Q
    unify_subterms Ht
    proceed
```

La instrucción "`unify_subterms Ht`" (figura 6) es la encargada de realizar el trabajo de unificación. Tiene un parámetro `Ht` que indica la posición en la THT del hecho temporario que se está ejecutando, con el cuál obtendrá la información de los subtérminos.

Ejemplo 4.1 : Utilizando el PLR del ejemplo 2.1, una vez que se termina de construir el argumento para "`huye-volando(coco)`", se tiene la siguiente información en la THT:

Identificador	punt_a_T-HEAP	Dir_código	cláusula	Padre
1. `huye-volando`	53	390	CPR	0
2. `vuela`	54	381	CPR	1
3. `ave`	55	372	CPE	2
4. `gallina`	56	363	CPR	3

y los siguientes hechos temporarios en el área de código:

```
363:  try_me_else L1
      unify_subterms 4
      proceed
372:  try_me_else L2
      unify_subterms 3
      proceed
```

```
381:  try_me_else L3
      unify_subterms 2
      proceed
390:  try_me_else L4
      unify_subterms 1
      proceed
```

4.2. Obtención de una justificación

El proceso de generación del árbol de dialéctica se podría especificar en términos de PROLOG, de la siguente manera:

```
justificar(Q) :- argumento(Q,A), naf derrotado(A)
derrotado(A) :- hallar_derrotador(A,D), naf derrotado(D)
```

Es decir, existe una justificación `A` para la consulta `Q`, si es posible construir un argumento `A`, y no es posible probar que `A` esté derrotado. Por su parte, un argumento `A` estará derrotado, si existe un derrotador `D` para `A`, tal que `D` no esté derrotado. Como "`naf derrotado(A)`" es una llamada utilizando la negación por falla de PROLOG, tendrá éxito si no hay derrotadores, y fallará en el caso que los haya. De esta forma, si hay derrotadores para el argumento de la consulta `Q`, la consulta `Q` fallará, y si no hay derrotadores, la consulta `Q` tendrá éxito. Es importante tener en cuenta que se está utilizando la negación por falla de PROLOG, y no la negación por falla de un PLR, ya que esta última tiene una definición diferente (ver sección 4.3).

El esquema de justificación anterior, da una pista muy interesante sobre cómo diseñar la máquina abstracta para que decida si el argumento hallado para una consulta es o no una justificación. Primero se intenta construir un argumento $\mathcal{A}$ para una consulta `q/n`. Si la construcción del argumento tiene éxito, entonces se debe probar que no existan derrotadores de nivel 1, luego de nivel 2, y así siguiendo. La figura 7 muestra el código JAM que se generará para contruir una justificación para la consulta `q/n` (el símbolo ";" indica que el resto de la linea es un comentario).

```
?:   allocate                    ; comienzo de la consulta
     ⋮                           ; código JAM de la consulta
     call q/n                    ; intenta construir el argumento
     prepare_to_defeat 1         ; prepara para buscar derrotadores
     put_constant defeat1 A1     ; intenta probar que ...
     call naf                    ; ... no existen derrotadores
     show_answer ...             ; si tiene éxito muestra las respuestas
     deallocate                  ;
     end_query                   ; fin consulta
```

Figura 7: Código JAM generado para una consulta

La instrucción `prepare_to_defeat 1` es la encargada de generar en tiempo de ejecución, el código JAM del predicado `defeat1/0` que será el encargado de buscar los derrotadores del argumento hallado para la consulta original `q/n`. Luego de generar el código de `defeat1/0`, se realiza la llamada `naf defeat1`, que como utiliza la negación por falla de PROLOG, tendrá éxito si no hay derrotadores, y fallará en el caso que los haya. De esta forma, si hay derrotadores para el argumento de la consulta `q/n`, la consulta `q/n` fallará, y si no hay derrotadores, la consulta `q/n` tendrá éxito. Cómo hay que verificar que un derrotador no esté a su vez derrotado, entonces el éxito o falla del predicado `defeat1/0` dependerá del árbol de dialéctica que se forme a partir de él.

Una nueva estructura de memoria llamada LINEA será la encargada de representar la linea de argumentación en curso. Como la THT trabaja como una pila, entonces se podrá tener en ella, todos los argumentos de la linea de argumentación en curso. Por lo tanto, la LINEA

solamente contendrá el inicio en la THT de cada argumento de la linea, y el inicio en la THT de los subargumentos de desacuerdo de cada argumento. La LINEA mantendrá una rama del árbol de dialéctica, y por ende funcionará como una pila. Un registro global `tope_linea` contendrá el tope de LINEA. Dicho registro será guardado en los puntos de elección a fin de actualizarlo durante el backtracking.

defeatK/0:	try_me_else L2	; intenta con $\overline{h_1}$, si falla sigue en L2
	allocate	;
	allow_inv	; permite CPE invertidas
	load_subterms N,T	; carga los registros de parámetros
	call $\overline{h_1}$	; intenta construir contraargumento
	set_disagree Hecho	; guarda el punto de desacuerdo
	not_circular	; verifica la circularidad en la linea
	not_less_specific	; verifica que sea derrotador
	prepare_to_defeat K+1	; se prepara para un nuevo nivel
	put_constant defeatK+1 A_1	;
	call naf	; continúa la linea de argumentación
	proper_defeat	; solamente en el caso K=1 ...
	deallocate	; ... verifica que sea derrotador propio
	proceed	;
L2:	retry_me_else L3	; intenta con $\overline{h_2}$ ...
	⋮	; ... si falla sigue con L3
	proceed	;
L3:	retry_me_else L4	;
⋮	⋮	;
LN:	trust_me	; intenta con $\overline{h_n}$ ...
	⋮	; ... si falla no hay derrotador
	proceed	;

Figura 8: Código JAM del predicado `defeatK/0` generado por `prepare_to_defeat K`

Sea $\mathcal{A}$ un argumento para el literal instanciado h, el *argumento completado* de $\mathcal{A}$, denotado $\mathcal{A}^c$, es un conjunto de CPE y CPR formado por las CPR de $\mathcal{A}$, más el subconjunto de CPE de $\mathcal{S}$ que no son hechos, y que fueron utilizadas para obtener la derivación de h a partir de $\mathcal{S} \cup \mathcal{A}$. Dado un argumento completado $\mathcal{A}^c$, se llama $Coc(\mathcal{A}^c)$ al conjunto de literales instanciados que son consecuentes de las cláusulas de $\mathcal{A}^c$.

Proposición 4.2 : [8] *Sea $\mathcal{S}$ el conjunto de* CPEs *de un* PLR*, y $\langle \mathcal{A}_1, h_1 \rangle$ un argumento. Existe un contraargumento $\langle \mathcal{A}_2, h_2 \rangle$, en un punto de contraargumentación $p \notin Coc(\mathcal{A}_1^c)$, si y sólo si existe una derivación $\mathcal{A}_2{}'$ para $\overline{h}$, en desacuerdo con un subargumento de $\langle \mathcal{A}_1, h_1 \rangle$ en un punto $h \in Coc(\mathcal{A}_1^c)$, que se obtiene utilizando algunas* CPE *invertidas con las condiciones establecidas.*

Una vez que se ha construído un argumento $\mathcal{A}$ en cualquier nivel en la linea de argumentación, su conjunto $Coc(\mathcal{A}^c)$ estará guardado como hechos temporarios en THT. Por lo tanto, para generar los contraargumentos de un argumento dado, se debe ir tomando uno a uno los elementos de $Coc(\mathcal{A}^c)$, y tratar de construir un argumento para el complemento de dicho elemento. Si h es un elemento de $Coc(\mathcal{A}^c)$, se debe construir un argumento para $\overline{h}$. Para buscar un contraargumento es posible utilizar todo el PLR más las CPE invertidas. Sin embargo, para probar los antecedentes de una CPE invertida, se debe usar solamente las CPE y los elementos de $Coc(\mathcal{A}^c)$.

La figura 8 muestra el código JAM del predicado `defeatK/0` generado en ejecución por la instrucción `prepare_to_defeat K`. Supóngase que $\overline{h_i}$ son los complementos de los elementos de $Coc(\mathcal{A}^c)$, para los cuales existe al menos una regla en el PLR, que tenga por cabeza un literal que unifique con $\overline{h_i}$, entoces cuando se ejecute `defeatK/0` realizará lo siguiente:

1. Intentará construir un contraargumento para uno de los posibles puntos de contraargumentación h_i, para lo cuál habilita el uso de CPE invertidas, utilizando la instrucción `allow_inv` que pone el registro `uso_cpe_invertidas` en TRUE.
2. La instrucción `load_subterms` tiene como parámetro la aridad `N` de la meta $\overline{h_i}$, y carga los primeros `N` registros de parámetro A_i con lo que está almacenado en el T-HEAP a partir de la posición `PosT` (ver figura 9).
3. La instrucción `call` efectúa la consulta que intentará construir un argumento para $\overline{h_i}$.
4. Si se obtuvo el contraargumento, guarda la posición del subargumento de desacuerdo en la linea de argumentación utilizando la instrucción `set_disagree` (ver figura 9).
5. La instrucción `not_circular` (ver figura 9) verifica que el contraargumento generado no esté presente en la linea, produciendo circularidad en la argumentación.
6. El contraargumento hallado debe ser un derrotador propio o de bloqueo, (*i.e.*, no debe ser menos específico que el subargumento de desacuerdo), esto lo verifica la instrucción `not_less_specific`.
7. La instrucción `prepare_to_defeat K+1` prepara el código necesario para buscar un derrotador en un nuevo nivel del árbol de dialéctica, y de esta forma se sigue avanzando en la linea de argumentación.
8. Una vez armado el código para la búsqueda de derrotadores de nivel `K+1` se llama a `naf defeatK+1/0`
9. Si el derrotador hallado para el nivel `K=1` no está derrotado, se debe verificar que tipo de derrotador es, ya que si es de bloqueo, igual se debe seguir buscando otro derrotador propio. Si en cambio es un derrotador propio, entonces ya no hay mas nada que buscar, pues la consulta no tiene justificación. La instrucción `proper_defeat` es la encargada de realizar esto, pero sólo se genera en el caso de K=1.

```
PROCEDURE not_circular;
 iniA:=LINEA[TopeLinea-1].PosTHT;
 finA:=F-1;
 Arg:=1;      Ultimo:=TopeLinea-1;
 WHILE (Arg < Ultimo) AND NOT Falla DO
  IniB:=LINEA[Arg].PosTHT;
  finB:=(LINEA[Arg].PosTHT+1) - 1;
  Falla:=Contenido(iniA,finA,iniB,finB);
  Arg:=Arg+1;
 END while;
 IF Falla THEN BackTrack;
END not_circular;
```

```
PROCEDURE load_subterms(N,PosT);
 FOR i:=1 TO N
     DO A[i]:=A-HEAP[PosT+i-1];
END load_subterms;


PROCEDURE set_disagree (PosF);
 LINEA[TopeLinea-2].Desacuerdo:=PosF;
END set_disagree;
```

Figura 9: Implementación de instrucciones usadas por `defeatK/0`

Obsérvese que si cualquiera de las instrucciones de `defeatK/0` falla, el mecanismo de backtracking, tratará de buscar otra forma de probar las cosas. Lo atractivo de esta estrategia es

usar siempre el mismo mecanismo de backtracking, mezclando los dos niveles de árboles: el de dialéctica y el de derivación rebatible.

La verificación de concordancia es una generalización de la verificación de la consistencia. Por lo tanto, para lograr que los argumentos de soporte (resp. interferencia) sean concordantes, al verificar la consistencia de una cláusula, se utilizarán todos lo hechos temporarios de todos los argumentos de soporte (resp. interferencia) de la linea de argumentación.

4.3. La negación por falla en los PLR

La negación por falla en los PLR no es la misma que en PROLOG. Según la definición de un PLR, la meta "`not p`" tiene éxito cuando no es posible justificar `p`, y falla cuando existe una justificación para `p`. Dada una submeta `not p`, la máquina abstracta debe buscar una justificación para `p`, lo cual involucrará construir un argumento $\mathcal{A}$ para `p`, y si existe, construir el correspondiente árbol de dialéctica a partir de $\mathcal{A}$. Pero nunca deberá buscar derrotadores para `not p`.

El operador `naf` de negación por falla de PROLOG, puede definirse en términos de PROLOG, utilizando los predicados extra lógicos `call` y cut (`!`). El operador `not` de los PLR puede definirse también con un PLR, como se muestra a continuación, donde `justify(X)` significa generar una justificación para `X`.

```
naf(X) :- call(X),!,fail.        not(X) :- justify(X),!,fail.
naf(_).                          not(_).
```

Ambos operadores fueron incorporados a la JAM como código predefinido. La figura 10 muestra el código de `naf` a la izquierda, y el de `not` a la derecha.

```
naf/1:  try_me_else L
        allocate 2
        get_level Y2
        get_variable V1 A1
        put_value V1 A1
        call call/1
        cut Y2
        fail_and_backtrack
    L:  trust_me
        proceed
```

```
not/1:  try_me_else L
        allocate 2
        get_level Y2
        get_variable V1 A1
        put_value V1 A1
        call call/1
        prepare_to_defeat K
        put_const defeatK
        call naf/1
        cut Y2
        fail_and_backtrack
    L:  trust_me
        allocate 1
        save_subterms not/1 1
        get_variable V1 A1
        save_fact not/1
        deallocate
        proceed
```

Figura 10: Código JAM para los operadores predifinidos `naf` y `not`

4.4. Obtención de la respuesta ante una consulta

Una consulta en PROLOG tiene dos respuestas posibles, SI o NO. Sin embargo, una consulta en un PLR puede tener cuatro respuestas posibles: SI, NO, INDECISO, y DESCONOCIDO. En la JAM, si una consulta tiene éxito, el registro `falla` queda en FALSE, mientras que si falla, el registro `falla` queda en TRUE. El registro `falla` alcanzaba en la WAM para dar las dos respuestas de PROLOG, pero no es suficiente para identificar las cuatro respuestas de un PLR. Por lo tanto se agregaron a la máquina abstracta los siguientes registros que tendrán un valor TRUE, o FALSE: `hubo_argumento` (indica si hubo al menos un argumento), `derrotador_propio` (indica si el derrotador hallado para el argumento de la consulta es propio o no), `hubo_bloqueo` (indica si hubo un derrotador de bloqueo), y `hubo_inconsistencia` (indica si no existe argumento por una inconsistencia). Ninguno de estos registros se guarda en el STACK, e inicialmente los cuatro están en FALSE.

```
IF NOT falla
   THEN WRITELN(' SI.')
   ELSE IF NOT hubo_argumento AND NOT Hubo_inconsistencia
           THEN WRITELN(' DESCONOCIDO.')
           ELSE IF derrotador_propio OR Hubo_inconsistencia
                 THEN WRITELN(' NO. ')
                 ELSE IF hubo_bloqueo
                      THEN  WRITELN(' INDECISO.')
                      ELSE  WRITELN(' DESCONOCIDO.');
```

Figura 11: Obtención de las respuestas en un PLR

La instrucción `prepare_to_defeat K`, cuado `K` vale 1, pone el registro `hubo_argumento` en TRUE. El registro `derrotador_propio` es puesto en TRUE por la instrucción `proper_defeat`. El registro `hubo_inconsistencia` se pone en TRUE cada vez que se ejecuta la instrucción `prove_consistency_else`, y se pone en FALSE cuando se ejecuta la instrucción `end_consistency_proof`. De esta forma si hay una falla mientras se está verificando la consistencia, el registro `hubo_inconsistencia` quedará en TRUE. El registro `hubo_bloqueo`, es puesto en FALSE cada vez que se ejecuta la instrucción `prepare_to_defeat K`, y `K` vale 1, y la instrucción `proper_defeat`, lo pone en TRUE sólo si el derrotador hallado es un derrotador de bloqueo del argumento de la consulta. La figura 11 muestra como se obtiene la respuesta de una consulta, en función de los valores que tengan estos registros.

5. Conclusiones

La nueva máquina abstracta permite la implementación de un sistema de argumentación rebatible, y la implementación de la programación en lógica rebatible. Al estar diseñada como una extensión de la WAM, hereda todo su desarrollo. Junto con el diseño de la JAM, fue implementado un compilador que traduce los PLR a instrucciones JAM. Luego se implementó la JAM como una máquina virtual, sobre la cuál se construyó un intérprete para programas lógicos rebatibles. Actualmente se está trabajando en varias optimizaciones para la JAM.

Referencias

[1] Aït-Kaci Hassan *Warren's abstract machine, a tutorial reconstruction.* MIT Press, 1991.

[2] Dung Phan M. *On the Acceptability of Argumets and its Fudamental Role in Nonmonotonic Reasoning and Logic Programming.* Proc. of Int. Join Conf. on Artificial Intelligence (IJCAI) 1993.

[3] García A. J., Chesñevar C. I., and Simari G. R. *Making Argument Systems Computationally Attractive.* Proc. XIII Int. Conf. of the Chilean Society for Computer Science, October 1993.

[4] García A. J. *Una aproximación a la programación en lógica rebatible.* 2do. Workshop en Aspectos Teoricos de la Inteligencia Artificial (ATIA'95). Bahía Blanca, Octubre de 1995.

[5] García A. J. y Simari G. R. *Compilación de programas lógicos que utilizan la negación por falla. Una extensión de la máquina abstracta de Warren* 1er. Congreso Argentino de Ciencias de la Computación. Bahía Blanca, Noviembre de 1995.

[6] García A. J. y Simari G. R. *Diferentes formas de hipótesis de mundo cerrado en la programación en lógica rebatible.* Segundo Congreso Argentino de Ciencias de la Computación (CACiC'96), Universidad Nacional de San Luis, Noviembre de 1996.

[7] García A. J. y Simari G. R. *El criterio de especificidad en la programación en lógica rebatible.* Tercer Workshop Argentino sobre Aspectos Teóricos de la Inteligencia Artificial (ATIA'96), Universidad Nacional de San Luis, Noviembre de 1996.

[8] García A. J. *La Programación en Lógica Rebatible, su definición teórica y computacional.* Tesis de Magister en Ciencias de la Computación, Universidad Nacional del Sur, Agosto de 1997, Bahía Blanca, Argentina.

[9] Lloyd J. W. *Foundations of Logical Programming.* 2nd. edition, Springer-Verlag 1987

[10] Loui R. P. *Defeats among arguments: a system of defeasible inference* Comp. Intell. 3, 1987.

[11] Loveland D. *Automated Theorem Proving: A Logical Basis .* North Holland, New York, 1978.

[12] Nute Donald, *Basic defeasible logic*, in *Intensional Logics for Programming*, Ed by Luis Fariñas del Cerro, Claredon Press – Oxford (c) 1992.

[13] Poole D. *On the Comparison of Theories: Preferring the most Specific Explanation* Proc. of the 9th Int. Joint Conf. on Artificial Intelligence.

[14] Prakken H. *Logical Tools for Modelling Legal Arguments (Ph.D. Thesis).* Vrije University, Amsterdam, Holanda. January 1993.

[15] Simari G. R. , Chesñevar C. I., and García A. J. *The Role of Dialectics in Defeasible Argumentation.* XIV Int. Conf. of Chilean Computer Science Society, November 1994.

[16] Simari G. R. and Loui R. P. *A Mathematical Treatment of Defeasible Reasoning and its Implementation.* Artificial Intelligence, 53: 125–157,1992.

[17] Simari G. R. y García A. J. *A Knowledge Representation Language for Defeasible Argumentation.* XXI Conferencia Latinoamericana de Informática. Canela, Brasil, Agosto 1995.

[18] Van Roy Peter L. *Can Logic Programmign Execute as Far as Imperative Programming?* Ph.D Thesis 1990, Berkeley University.

[19] Vreeswijk G. *Studies in Defeasible Argumentation* (Ph.D. Thesis). Vrije University, Amsterdam, Holanda. March 1993.

[20] Warren David H. D. *Compiling Predicate Logic Programs, Volumenes 1 y 2,* D.A.I. Research Reports Nro. 39. Mayo de 1977.

[21] Warren David H. D. *An Abstract Prolog Instruction Set.* Thechnical Note 309, SRI International, Menlo Park, CA, October 1983.

[22] Warren David H. D. *Implementation of Prolog. Lectures Notes Tutorial Nro 3* 5th International Comference and Symposium on Logic Programming, Seattle, WA 1988.