Un compilador para la programación en lógica 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 *programación en lógica rebatible* es una extensión de la programación en lógica que captura aspectos del razonamiento del sentido común que son difíciles de expresar en la programación en lógica tradicional. Los *programas lógicos rebatibles* (PLR) permiten representar información incompleta y potencialmente inconsistente, y utilizan los conceptos de la *argumentación rebatible* a fin de poder decidir entre metas contradictorias.

El objetivo de este trabajo es presentar un compilador que se desarrolló para los programas lógicos rebatibles. Dicho compilador toma como entrada un PLR, y produce como salida un programa en instrucciones de una máquina abstracta para la programación rebatible llamada JAM.

Para realizar este trabajo, se combinaron técnicas convencionales de compilación, como el análisis recursivo descendente, con técnicas especiales para la programación en lógica. De esta forma, cada cláusula de programa puede compilarse por separado e integrarse a un único código ejecutable. Esto permite que el programa este dividido en varios archivos, o agregar nuevas cláusulas al programa sin necesidad de recompilar el programa entero. El compilador además permite la generación de código para predicados predefinidos del lenguaje.

El compilador construído puede utilizarse también para traducir programas en Prolog a códigoWAM (Warren's Abstract Machine), y podría modificarse fácilmente para producir compiladores para otras extensiones de la programación en lógica.

[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 programación en lógica se ha convertido en uno de los principales exponentes de la programación declarativa. Sin embargo, aunque se la ha utilizado como herramienta de representación de conocimiento, presenta limitaciones para manejar ciertas características del razonamiento del sentido común, como utilizar información incompleta, o potencialmente inconsistente. Los trabajos de Gelfond y Lifschitz [11], Inoue [12], Alferes y Pereira [2], Dung [3, 4], y Fillottrani [5], establecen nuevos criterios para extender la programación en lógica introduciendo la negación clásica, y manteniendo la negación por falla, sin embargo, presentan ciertos problemas cuando tienen que decidir entre metas contradictorias.

La *programación en lógica rebatible* [16, 9, 8] es una extensión de la programación en lógica convencional, que utiliza los conceptos de la *argumentación rebatible* a fin de capturar aspectos del razonamiento rebatible. Los *programas lógicos rebatibles* (PLR), estarán formados por dos tipos de cláusulas: las *cláusulas de programa extendido*, y las *cláusulas de programa rebatible*. De esta forma, un programa estará formado por dos conjuntos disjuntos de cláusulas: uno que representa conocimiento estricto (seguro), y otro que contiene información rebatible (tentativa). Los PLR permiten la representación de información incompleta y potencialmente inconsistente, y pueden decidir entre metas contradictorias, utilizando un criterio de preferencia.

La Maquina Abstracta de Warren o WAM (Warren's Abstract Machine) [18, 19] se ha convertido en el estándar *de facto* para la implementación del lenguaje PROLOG. Por lo tanto, a fin de implementar la programación en lógica rebatible, se diseñó una máquina abstracta llamada JAM (Justification Abstract Machine), como una extensión de la WAM. Al igual que la WAM, la JAM constituye sólo una parte de la implementación de la programación en lógica rebatible. Además de la máquina abstracta, se necesita de un compilador que tome como entrada un PLR, y produzca como salida un programa en instrucciones JAM.

El objetivo de este trabajo es describir un compilador que se desarrolló para los PLR. Para esto, se combinaron técnicas convencionales de compilación, como el análisis recursivo descendente, con técnicas especiales para la programación en lógica. De esta forma, cada cláusula de programa puede compilarse por separado e integrarse a un único código ejecutable. Esto permite que el programa este dividido en varios archivos, o agregar nuevas cláusulas al programa sin necesidad de recompilar el programa entero. El compilador además permite la generación de código para predicados predefinidos del lenguaje.

El compilador construído puede utilizarse también para traducir programas en Prolog a código WAM, y podría modificarse fácilmente para producir compiladores para otras extensiones de la programación en lógica.

2. Programas Lógicos Rebatibles

Se mostrará en esta sección el lenguaje de la programación en lógica rebatible, y como está organizado el compilador. Mas detalles sobre los PLR pueden encontrarse en [16, 6, 9, 8]. En el libro *Warren's abstract machine, a tutorial reconstruction*, de Hassan Aït-Kaci [1], puede encontrarse una descripción muy clara de la WAM, sin embargo, no se muestra como compilar un programa. La definición completa de la JAM se encuentra en [10].

En el lenguaje de la programación en lógica rebatible (al igual que en PROLOG) un *término* es una *variable* (denotada con una mayúscula inicial), o bien una *constante* (denotada con una minúscula inicial), o bien una *estructura* de la forma $f(t_1, \ldots, t_n)$ donde f se llama símbolo de functor (denotado igual que una constante) y los t_i son términos (también se los suele llamar *subtérminos*). El número de subtérminos para un functor dado, es fijo y se llama su *aridad*. El mismo símbolo de un functor puede utilizarse con diferentes aridades, pero serán considerados functores diferentes. Es por esto, que se usará la notación f/n para referirse a un functor de símbolo f y aridad n. Una constante c puede verse como un caso especial de estructura de aridad 0, y por lo tanto puede denotarse como el functor $c/0$.

Un *predicado atómico* es una estructura de la forma $p(t_1, \ldots, t_k)$, $k \geq 0$, donde p es el nombre del predicado y cada t_i es un término, que se llamará *subtérmino* o *parámetro* del predicado p. Se llama *aridad* del predicado p a la cantidad de subtérminos que posee p, y se utiliza la notación p/k para indicar que el predicado p tiene aridad k. Obsérvese que las variables se diferencian de los demás elementos de un programa, porque estan denotadas con un identificador que comienza con una letra mayúscula. Un *literal* "l" es un predicado atómico "a" o un predicado atómico negado "$\sim a$". El símbolo "$\sim$" representa 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 una CPR o CPE, el antecedente "$p_1, \ldots, p_n$" representa la conjunción de los elementos "p_i". Todas las variables de una cláusula de programa están implicitamente clausuradas universalmente. Las cláusulas de programa no son implicaciones de la lógica clásica; deben tomarse como reglas de inferencia, y no poseen un valor de verdad. Su semántica informal [13] es: *si el antecedente es verdadero, entonces el consecuente es verdadero.*

En una cláusula de PLR, habitualmente se llama *cabeza* al consecuente "l", y *cuerpo* al antecedente "$p_1, \ldots, p_n$". Siguiendo esta terminología, los símbolos " $\prec$ " y " $\leftarrow$ " habitualmente se los llama *cuello* de la cláusula. De esta forma, un *hecho* será un cláusula sin cuerpo, y una *consulta*, una cláusula sin cabeza.

Por lo tanto, una cláusula de programa puede considerarse como un conjunto de consultas que al resolverse implican un nuevo hecho. Esta forma de ver las reglas permite que todo el desarrollo de la WAM, y por ende el de la JAM, se haga sólo sobre consultas y hechos. La ejecución del código JAM comienza siempre por una consulta que se hace al programa, y el modelo de ejecución 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.

Ejemplo 2.1 : A continuación se muestra un PLR donde se utilizan CPE, y CPRs. En la figura 1, se muestra el código JAM, de las dos primeras cláusulas.

```
aterrizar(pista(X)) —≺ libre(pista(X)).
∼aterrizar(pista(X)) ← ocupada(pista(X)).
∼aterrizar(pista(X)) —≺ reparación(pista(X)).
ocupada(pista(dos)) ← true.

aterrizar(pista(dos) —≺ true.
libre(pista(uno)) —≺ true.
aterrizar(pista(tres)) —≺ true .
reparacion(pista(uno)) ← true.
```

```
aterrizar/1   defeasible_try_me_else E
              allocate 1
              get_structure pista/1 A1
              unify_variable V1
              put_structure pista/1 A1
              set_value V1
              call libre/1
              deallocate
              proceed

∼aterrizar/1: try_me_else E1
              allocate 1
              get_structure pista/1 A1
              unify_variable V1
              put_structure pista/1 A1
              set_value V1
              call ocupada/1
              deallocate
              proceed
```

Figura 1: Código JAM de las dos primeras cláusulas del ejemplo 2.1

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

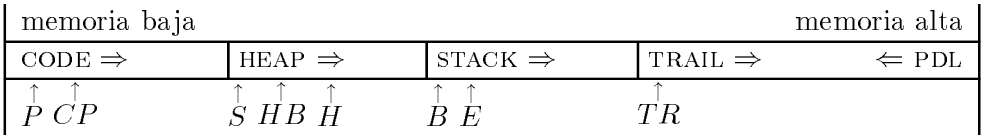

Figura 2: Disposición de la memoria y registros.

Cada cláusula de un PLR puede compilarse por separado, si cierta información es guardada en una tabla de símbolos que perdure hasta que se haya compilado todo el programa. A fin de generar el código JAM de un programa, se realizó un analizador sintáctico recursivo descendente de una pasada, que recorre las cláusulas de izquierda a derecha, y almacena la información procesada en dos tablas de símbolos: una *tabla de símbolos temporaria*, cuyo tiempo de vida es la compilación de una cláusula, y otra que perdura incluso durante la ejecución, que se llamará *tabla de símbolos permanente* o *tabla de identificadores* (IDENT). Como se vió en los ejemplos anteriores, casi todos los elementos de una cláusula tienen asociado una instrucción JAM. Por lo tanto, las instrucciones se van generando y guardando en la tabla temporaria, a medida que se va

compilando la cláusula. Una vez que se ha completado la compilación de una cláusula, se carga en memoria el código JAM, la tabla temporaria es vaciada, y se continúa con la compilación de la siguiente cláusula del programa. La tabla de símbolos temporaria posee la siguiente información:

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

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

Ejemplo 2.2 : La figura 3 muestra la información que se tiene en las tablas de símbolos temporaria y permanente, al finalizar la compilación de la cláusula "`p(X,Y)` ← `q(f(X),a),r(g(X,Y))`"

At.	Id.	Ari	Categoría	A_i	X_j	Instrucción
	p	2	functor			just_me
						allocate 2
1	X		variable	A_1	Y_1	get_variable $A_1 Y_1$
1	Y		variable	A_2	Y_2	get_variable $A_2 Y_2$
	f	1	functor	A_1	X_1	put_structure A_1
1	X		variable		Y_1	set_value Y_1
	a	0	constante	A_2		put_constant A_2
	q	2	functor			call q/2
	g	2	functor	A_1	X_1	put_structure A_1
2	X		variable		Y_1	set_value Y_1
2	Y		variable		Y_2	set_value Y_2
	r	1	functor			call r/1
						deallocate
						proceed

Id.	Ari	Clase	Dir
p	2	predicado	1
f	1	functor	-
a	0	constante	-
q	2	functor	-
g	2	functor	-
r	1	functor	-

Figura 3: Tablas de símbolos después de compilar `p(X,Y) :- q(f(X),a),r(g(X,Y))`

La compilación de una cláusula está dividida en la compilación de la cabeza, y luego, si existen, la compilación de cada una de las consultas del cuerpo. Al detectar el nombre del predicado de la cláusula, este se guarda en la tabla temporaria, se generan las instrucciones `just_me` y `allocate` y luego se siguen compilando (recursivamente) los subtérminos del predicado (ver ejemplo 2.2). Al terminar de compilar los parámetros, se conoce la aridad del predicado, la cuál es guardada en la tabla de símbolos. La compilación de cada una de las consultas del cuerpo de la cláusula es similiar a la de la cabeza: se guarda el identificador de la consulta en la tabla temporaria, se genera el código de los subtérminos, y luego con el identificador de la consulta que está en la tabla, se genera la instrucción `call`. Una vez que se ha terminado de compilar el cuerpo de la cláusula, se generan las instrucciones `deallocate` y `proceed`.

3. Compilación de términos

Como se vió en las secciones anteriores, los términos pueden estar en la cabeza o en el cuerpo de cualquier cláusula de programa. La JAM es una extensión de la WAM, y en el caso de los términos, se utilizan las mismas instrucciones que en la WAM. En esta sección se mostrará como se compila un término, y como se obtiene el código WAM (JAM) asociado.

La estructura de memoria que se utiliza para almacenar todos los términos del programa, es el HEAP. Cada elemento o celda del HEAP será un par *(etiqueta,pos)*, donde *pos* es una dirección de memoria, y *etiqueta* podrá ser: *referencia*, *constante*, *estructura*, *functor*, o *lista*. Una variable puede tener ligado un término, o estar sin ligar. Por lo tanto una variable será representada en el HEAP por el par *(referencia,pos)* donde *pos* es la dirección al término a la que esté ligada, o la dirección de ella misma en el caso que esté sin ligar. Una constante estará representada en el HEAP por el par *(constante,pos)* donde *pos* será lo posición del identificador de la constante en la tabla IDENT. 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. Una estructura $f(t_1, \ldots, t_n)$, se representará con $n+1$ celdas del HEAP, la primera con el par *(functor,pos)* donde *pos* es la posición del símbolo de functor en IDENT, y en las n celdas siguientes, se almacenarán referencias a la representación de los n sub-términos de f/n.

Ejemplo 3.1 : La figura 4 muestra la representación del término `f(Z,h(Z,W),a)` a partir de la dirección 54 del HEAP, y la tabla IDENT con los identificadores del término.

51	functor	1
52	referencia	52
53	referencia	53
54	functor	2
55	referencia	52
56	estructura	51
57	constante	3

	IDENT
1	h/2
2	f/3
3	a/0

Figura 4: HEAP con la estructura `f(Z,h(Z,W),a)`

Los términos serán almacenados en el HEAP, y la unificación de terminos en consultas con téminos de los hechos es realizada leyendo y escribiendo en el HEAP. Las instrucciones WAM generadas para los términos, serán las encargadas de realizar este trabajo, por lo tanto hay instrucciones especializadas para cada tipo de término y serán diferentes las que se generen en una consulta que las que se generen en un hecho. A continuación se mostrará como el compilador genera la instrucción adecuada para variables, constantes y estructuras.

Al encontrar una variable, la instrucción WAM que debe generarse depende de varios factores: si está en una consulta, o en un hecho, si es la primera aparición, si está dentro de una estructura, etc.. Hay ocho instrucciones especializadas que corresponden al código generado para una variable. En el momento de encontrar una variable, el compilador genera la instrucción correcta siguiendo el algoritmo indicado en el cuadro de la tabla 1.

El tratamiento de las constantes es bastante más sencillo, hay cuatro instruciones especializadas, las cuales se obtienen siguiendo lo indicado por la tabla 2.

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

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

Tabla 1: Instrucciones especializadas para variables

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

Tabla 2: Instrucciones especializadas para constantes

Hassan Aït-Kaci [1]. Se mostrará en forma separada, como compilar términos que pertenecen a consultas, o que pertenecen a hechos.

El código de una consulta además de invocar a las cláusulas correspondientes, debe preparar los registros y la memoria para que la ejecución del código de la cabeza de una cláusula encuentre todas las estructuras que tiene que unificar. Como se vió en la sección anterior una estructura con functor f/n estará formada por $n+1$ celdas contiguas de memoria, más lo necesario para representar los n sub-términos. De esta forma, para alojar en la dirección d del HEAP la estructura $f(t_1, \ldots, t_n)$, en d estará la posición del símbolo del functor en IDENT, y en las n celdas siguientes, la dirección de la representación de los n subtérminos de f/n, *que tienen que heber sido colocados en el* HEAP *antes*. Esto es, al generar la representación en el HEAP de una estructura con subestructuras, primero hay que almacenar las subestructuras, y luego la estructura podrá referenciar a los subtérminos. A fin de almacenar temporariamente apuntadores a términos y subtérminos durante la compilación, se utilizarán registros de términos. Se asociará un registro a cada elemento del término (functor, variable o constante). Por ejemplo, en el término `p(Z,h(Z,W),f(W))`, los registros son asignados de la siguiente forma: X_1= `p(`X_2`,`X_3`,`X_4`)` ; X_2= `Z` ; X_3= `h(`X_2`,`X_5`)` ; X_4= `f(`X_5`)` ; X_5= `W` .

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

Como antes de construir el término, deberán haberse construído los subtérminos, para poder hacer una referencia a ellos, entonces es necesario ordenar la secuencia de registros de la forma plana de tal forma que *ningún registro aparezca en la parte derecha, si no fue usado previamente en la parte izquierda.* Esta forma será llamada **forma plana de términos que pertenecen a una consulta** (FPC). La FPC del término del ejemplo anterior es: X_2= `Z` ; X_5= `W` ; X_3= `h(`X_2`,`X_5`)` ; X_4= `f(`X_5`)` ; X_1= `p(`X_2`,`X_3`,`X_4`)` . A partir de esta representación, es posible "olvidar" los nombres externos de las variables, y referirse a ellas por su registro X asociado.

De esta forma la FPC queda sólo con: X_3= `h(`X_2`,`X_5`)` ; X_4= `f(`X_5`)` ; X_1= `p(`X_2`,`X_3`,`X_4`)` ; donde X_2 es la variable `Z`, y X_5 la variable `W`.

El compilador recorre el término de izquierda a derecha, siguiendo su estructura recursiva. Los elementos encontrados son volcados en ese orden, a la tabla de símbolos temporaria. Cada vez que un subtérmino se termina de reconocer sintácticamente, entones la tabla de símbolos temporaria se modifica para que el subtérmino almacenado tenga la forma FPC. Una vez que se tiene la FPC de una estructura en la tabla de símbolos, entonces un recorrido secuencial sobre la tabla permitirá generar el código de un término. Para esto, se utiliza lo mostrado antes para constantes y variables, y además, si se encuentra un registro asociado a un functor, como es el caso de X_3, se generará la instrucción `put_structure`.

La compilación de un término que pertenece a un hecho será muy similar a la de las consultas. Sin embargo, previamente a la ejecución del código WAM de un hecho, necesariamente debió ejecutarse el código de una consulta. Por lo tanto, la consulta previamente ejecutada puede haber dejado: (1) una estructura sin variables, que el código del hecho deberá intentar unificar simplemente **leyendo** del HEAP, o (2) variables sin ligar, que el código del hecho deberá unificar **escribiendo** en el HEAP la nueva estructura. Esto implica que al ejecutarse un hecho se podrán tener dos modos: uno de lectura (read) y otro de escritura (write).

Los términos de consultas necesitaban ordenarse en FPC, a fin de que los subtérminos sean apilados en el HEAP primero. En el caso de los términos que pertenecen a hechos, la situación es distinta, porque la información a unificar ya está disponible en el HEAP. Por lo tanto la **forma plana de un término que pertenece a un hecho** (FPH) estará ordenada de afuera hacia adentro del término (top-down). Por ejemplo la FPH del siguiente término `p(f(X),h(Y,f(a)),Y)` es: X_1 = p(X_2,X_3,X_4); X_2 = f(X_5) ; X_3 = h(X_4,X_6) ; X_6 = f(a) . Al igual que en un término de consulta, la forma plana nos permite obtener directamente las instrucciones WAM, utilizando lo visto para constantes y variables, y además, en el caso de encontrar un registro asociado a un functor, se genera la instrucción `get_estructure f/n` X_j.

4. Compilación de cláusulas

Un *programa lógico rebatible* (PLR), es un conjunto finito de *cláusulas de programa extendido* (CPE) y *cláusulas de programa rebatible* (CPR). Un predicado "`p/n`" del PLR, puede estar definido por una sóla cláusula, o puede existir un conjunto de cláusulas que lo definan (todas con cabeza "`p/n`"). En el caso que un conjunto de cláusulas defina a "`p/n`", este conjunto es tomado como una disyunción, esto es, una consulta "`p/n`" tendrá éxito, si al menos una de las cláusulas que definen a "`p/n`" tiene éxito. Si al intentar probar "`p/n`" con una de las cláusulas, esta falla, entonces se debe intentar probar con otra, y asi sucesivamente. A este mecanismo se lo denomina *backtraking*.

En el conjunto de instrucciones de la WAM [1], hay tres instruciones especializadas en manejar el backtraking: "`try_me_else E`", "`retry_me_else E`" y "`trust_me`", que siempre están ubicadas como la primera instrucción del código WAM de una cláusula. Cuando un predicado está definido por más de una cláusula, se utiliza la instrucción `try_me_else E` para la primera, `retry_me_else E` a partir de la segunda, y `trust_me` para la última cláusula. El parámetro `E` en las dos primeras instrucciones, es la dirección de código de la próxima cláusula que define al predicado. En el caso que el predicado esté definido por una sóla clausula se utiliza la instrucción `just_me`. Esta

última instrucción no forma parte del conjunto original de la WAM, se la incluyó en la JAM para facilitar la compilación de los PLR.

En los PLR, hay dos tipos de cláusulas, CPE y CPR, y en ejecución debe poder diferenciarse que tipo de cláusula se está usando. Por lo tanto, se utilizan las cuatro instrucciones anteriores para las CPE, y se agregan para las CPR las instrucciones: "`defeasible_just_me`", "`defeasible_try_me_else E`", "`defeasible_retry_me_else E`" y "`defeasible_trust_me`".

En un PLR un predicado puede estar definido por varias cláusulas, e incluso de diferente tipo (ver ejemplo 2.1). Las cláusulas que definen un predicado pueden estar intercaladas con cláusulas que definen a otro predicado, e incluso estar en diferentes archivos. Por lo tanto cada vez que encuentre una cláusula que pertenece a la defincón de un predicado, el compilador deberá modificar el código compilado para colocar la instrucción de backtracking que corresponda. Para esto se utiliza información almacenada en la tabla de símbolos permanente (IDENT).

Cada vez que se compila una cláusula, que es la primera que define a un predicado `p/n`, el código JAM de dicha cláusula es almacenado en la memoria encabezado por la instrucción `just_me` o `defeasible_just_me` (según sea CPE o CPR), indicando de esta forma, que `p/n` tiene una sola definición. En la tabla IDENT, se almacena el identificador "`p`" del predicado, y la dirección del área de código donde quedó almacenado el código JAM de la primer cláusula.

El proceso de compilación utiliza la siguiente estrategia para generar la instrucción que corresponda al encabezado de cada cláusula. En la tabla de símbolos temporaria, toda cláusula tiene como primera instrucción `just_me` o `defeasible_jut_me`. Al generar el código WAM, si es la primera aparición de una cláusula para definir el predicado, entonces se deja la instrucción generada, y se actualiza la información de la tabla IDENT. Pero si ya había una definición anterior de este predicado (esto es, ya figuraba en IDENT como predicado definido), entonces se debe corregir su código, para que concuerde con la situación actual. En la tabla IDENT, se indica para cada predicado `p/n` la dirección del código de la primera cláusula que define que lo define. Pero también se guarda la dirección de la última cláusula, para permitir modificar el código de la definición de `p/n`, cada vez que se encuentra durante la compilación otra cláusula con cabeza `p/n`.

Cada vez que el compilador encuentra una cláusula que es parte de la defincón de un predicado `p/n` que ya está presente en IDENT, la primera instrucción del código JAM de la última cláusula de la definición de `p/n` debe reemplazare como indica la tabla siguiente.

La instrucción:	Es reemplazada por:
`just_me`	`try_me_else`
`defeasible_just_me`	`defeasible_try_me_else`
`trust_me`	`retry_me_else`
`defeasible_trust_me`	`defeasible_retry_me_else`

No obstante, en la nueva cláusula que se esta compilando, la primera instrucción del código JAM será: `trust_me` en el caso de un CPE, o `defeasible_trus_me` si es un CPR. La figura 5 muestra las primeras instruciones de cada cláusula del PLR del ejemplo 2.1.

En la programación en lógica rebatible, una consulta a un PLR, involucra construir un argumento que sea una justificación para la consulta [16, 10]. En la JAM, para construir un argumento para una consulta `q/n`, se debe construir una derivación rebatible de `q/n`, y verificar que sea

```
 aterrizar/1:   defeasible_try_me_else 197
                ⋮
          103:  proceed

~aterrizar/1:   try_me_else 143
                ⋮
          142:  proceed

          143:  defeasible_trust_me
                ⋮
          182:  proceed

   ocupada/1:   just_me
                ⋮
          196:  proceed

          197:  defeasible_retry_me_else 267
                ⋮
          231:  proceed

     libre/1:   defeasible_just_me
                ⋮
          266:  proceed

          267:  defeasible_trust_me
                ⋮
          301:  proceed

reparacion/1:   just_me
                ⋮
          315:  proceed
```

Figura 5: Código JAM

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.

Durante la compilación se debe generar el código necesario para que se realice la verificación de la consistencia durante la construcción de un argumento. Por ejemplo, una vez probado el cuerpo de una cláusula como `p(X) :- q1(X),q2(X)` se debe verificar la consistencia de la cabeza `p(X)` (con la instanciación que haya sufrido `X`). Por lo tanto, la máquina abstracta necesita dos modos de prueba, uno que genera derivaciones rebatibles (para la construcción del argumento), y otro que genera derivaciones fuertes (para la verificación de la consistencia).

Probar la consistencia de una cláusula con cabeza `p/n`, es equivalente a probar la no existencia de una derivación fuerte de una consulta `p/n`, lo cual es equivalente a tener éxito en una consulta

`naf p`, que sólo utilice las CPE del PLR, los hechos temporarios que se tenga en ese momento, y alguna CPE invertida. El operador `naf` es la negación por falla de PROLOG, y está incluído en la JAM como un operador predefinido (ver sección 6).

Por lo tanto, el compilador generará al final de cada cláusula una consulta adicional que al compilarse generará el código necesario para la verificación de la consistencia. Esto es, se agregará el complemento de la cabeza de la cláusula, como una consulta adicional al final del cuerpo, precedida con el símbolo "`#`". Por ejemplo, la cláusula "`p(X) :- q1(X),q2(X)`" se transformará en "`p(X) :- q1(X),q2(X) # ~ p(X)`"; y la cláusula "`~r(X) :- q(X)`", se transformará en "`~r(X) :- q(X) # r(X)`". El símbolo "`#`" debe interpretarse como "*y no puede probarse en forma fuerte*", esto es, "`# ~vuela(charo)`" debe leerse como "y no puede probarse en forma fuerte `~vuela(charo)`". El símbolo `#` le indicará al compilador que lo que sigue es la verificación de consistencia de la cláusula que está compilando, y producirá que se genere la instrucción `prove_consistency_else`. Luego se compilará al submeta que sigue al `#` como una submeta estándar del PLR, y al terminar con la compilación de la submeta, se agregará la instrucción `end_consistency_proof`.

Dos nuevas instrucciones `prove_consistency_else` y `end_consistency_proof` serán las encargadas de marcar el inicio y el fin de una prueba de consistencia, De esta forma, toda secuencia de código que esté entre estas dos instrucciones se ejecutará en modo "*derivación fuerte*", *i.e.*, sólo podrán usarse CPE, hechos temporarios, y, quizás, alguna CPE invertidas. Tambíen el compilador debe agregar las instrucciones `save_subterms` y `save_fact`, al principio y al final del código de cada cláusula, a fin de que se generen los hechos temporarios [10].

Definición 4.1 : [10] 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.

Las CPE no son contrapositivas, entonces las CPE invertidas no son cláusulas del programa. Su utilización está limitada a la aceleración de la búsqueda de inconsistencias. Es por esto, que las CPE invertidas sólo pueden utilizarse con ciertas restricciones: sólo al principio de la búsqueda de inconsistencia, y una vez que se utilizó una cláusula que no es invertida, debe continuarse la derivación con CPE normales y hechos temporarios. Además, a excepción de la consulta que era la cabeza en la CPE sin invertir, el cuerpo de una CPE invertida debe probarse utilizando únicamente CPE y hechos temporarios (no pueden utilizarse CPR, o CPE invertidas).

Las CPE invertidas deben estar en el programa compilado, antes de que se efectúe una consulta. Para resolver esto, al compilar un PLR, se generarán todas las CPE invertidas a partir de las CPE del programa, y luego se compilará el programa normalmente. Las CPE invertidas pueden ir en cualquier parte de un programa, porque no importa en que orden son encontradas. Por lo tanto se las ubicará al final del PLR a fin de usar primero las cláusulas normales, y como último recurso las CPE invertidas. El código JAM de una CPE invertida será como el de una CPE común, salvo la instrucción de encabezado.

El ejemplo 2.1 tiene una única CPE, que genera una sóla CPE invertida: "`~ocupada(pista(X)) @- aterrizar(pista(X))`". La compilación de una CPE invertida se realiza igual que una CPE, con la única diferencia que la primera instrucción de su código JAM, será una de las cuatro instrucciones siguientes:

```
just_me_checking_consistency
try_me_checking_consistency_else
retry_me_checking_consistency_else
trust_me_checking_consistency
```

La forma de generar esta instrucción es análoga a lo explicado antes para CPE, y CPR.

El siguente ejemplo muestra el programa del ejemplo 2.1 con lo agregado por el compilador antes de generar el código JAM. Es importante destacar que esto nunca es visto por el programador, y es realizado por un preprocesador, justo antes de compilar el programa.

Ejemplo 4.1 :

```
aterrizar(pista(X)) —≺ libre(pista(X)) # ∼aterrizar(pista(X)).
∼aterrizar(pista(X)) ← ocupada(pista(X)) # aterrizar(pista(X)).
∼aterrizar(pista(X)) —≺ reparación(pista(X)) # aterrizar(pista(X)).
ocupada(pista(dos)) ← true.
aterrizar(pista(dos) —≺ # ∼aterrizar(pista(dos)).
libre(pista(uno)) —≺ # ∼libre(pista(uno)).
aterrizar(pista(tres)) —≺ # ∼aterrizar(pista(tres)) .
reparacion(pista(uno)) ← true.
∼ocupada(pista(X)) @- aterrizar(pista(X)).
```

En la figura 6, se muestra el código JAM, de las dos primeras cláusulas del ejemplo 2.1 con el código JAM para la verificación de la consistencia incluído.

```
aterrizar/1:  defeasible_try_me_else E
              allocate 1
              save_subterms aterrizar/1 cpr
              forbid_inv
              get_structure pista/1 A1
              unify_variable V1
              put_structure pista/1 A1
              set_value V1
              call libre/1
              set_defeasible
              prove_consistency_else L
              put_structure pista/1 X3
              set_value V1
              put_structure aterrizar/1 A1
              set_value V3
              call naf/1
              end_consistency_proof
L:            save_fact aterrizar/1
              deallocate
              proceed

∼aterrizar/1: try_me_else E1
              allocate 1
              save_subterms aterrizar/1 1
              forbid_inv
              get_structure pista/1 A1
              unify_variable V1
              put_structure pista/1 A1
              set_value V1
              call ocupada/1
              prove_consistency_else L1
              put_structure pista/1 X3
              set_value V1
              put_structure aterrizar/1 A1
              set_value V3
              call naf/1
              end_consistency_proof
L1:           save_fact aterrizar/1
              deallocate
              proceed
```

Figura 6: Código JAM para las dos primeras cláusulas del ejemplo 4.1

Para que las CPE invertidas se usen apropiadamente, el compilador debe agregar la instrucción `forbid_inv` en las CPR, y CPE. Además las CPR, deben llevar la instrucción `set_defeasible`, que indica que la derivación que se está haciendo, es rebatible.

5. Listas

Las listas son un tipo de dato muy utilizado en la programación en lógica. En los PLR, se utilizarán las listas de la misma forma que se utilizan en PROLOG. Una lista tiene la siguiente definición recursiva: una lista es vacía, o es la concatenación de un elemento con una lista. Una lista vacía se nota con "`[ ]`", y una lista no vacía con "`[elemento|<lista>]`". Por ejemplo la lista que tiene los elementos `a,b,c` se nota `[a | [b | [c | [ ] ]]]`. Pero también es posible utilizar una notación abreviada que se indica de la siguente forma: `[a,b,c]`. En general se puede escribir cualquier combinación de ambas notaciones, como por ejemplo `[a,b|[c]]`. Los elementos de una lista pueden ser constantes, variables, estructuras, y también listas. Por ejemplo la siguiente lista contiene elementos de todos estos tipos: `[a, f(a) , f(g(b),X), [1,2,3] , f([],[a,b]) ]` .

A continuación se mostrará como almacenar una lista en el HEAP. La lista vacía "`[ ]`" es una constante, y por lo tanto se la representará como tal. Siguiendo la definición recursiva, una lista no vacía puede pensarse como un functor que tiene dos subtérminos, un elemento y una lista. Un functor de aridad 2 se representaba en el HEAP utilizando 3 celdas, la primera para el nombre del functor, y las dos siguientes para los dos subtérminos del functor. En el caso de las listas no hay un nombre de functor, y por lo tanto es posible ahorrar una celda en la representación. De esta forma, una lista se representará en el HEAP con dos celdas consecutivas. La primera para el primer elemento, y la segunda para indicar la posición del HEAP donde se representa el resto de la lista. En el caso de los elementos que no son listas, una celda del HEAP tendrá la forma que ya fue descripta. Para las listas se agrega una nueva etiqueta `lista` para indicar que la celda del HEAP contiene la dirección del comienzo de una lista. La figura 7 muestra la representación en el HEAP de la lista `[a,b,c]` (equivalente a `[a|[b|[c|[]]]]`) que se encuentra a partir de la dirección 102.

	(clase)	(dirección)
⋮	⋮	⋮
102	constante	a
103	lista	104
104	constante	b
105	lista	106
106	constante	c
107	constante	nil
⋮	⋮	⋮

Figura 7: Representación de la lista `[a,b,c]` en el Heap

La compilación de una lista que está en notación recursiva, es muy parecida a la de una estructura, ya que una lista puede pensarse como un functor de aridad 2, donde el primer subtérmino es un elemento y el segundo una lista. En el caso de una lista vacía, el compilador genera el código correspondiente a la constante`[]`. La compilación de una lista en notación abreviada es un poco diferente. Debido a que hay dos notaciones para listas, y que además pueden estar mezcladas, la idea es que dos listas iguales, independientemente de su notación,

generen la misma secuencia de instrucciones. Para esto, el compilador se encargará de absorver las diferencias y generar el mismo código.

A continuación se muestra un algoritmo que realiza el reconocimiento sintáctico y la generación de código para una lista con la notación recursiva. Obsérvese que simplemente se genera código para `put_list` o `get_list`, y luego se analiza sintácticamente la estructura.

Algoritmo: CompilarLista
1. Generar la instrucción `get_list` o `put_list`
2. Compilar el primer elemento de la lista
3. GenerarFormaPlana
4. Saltear la barra
5. Compilar el resto de la lista utilizando este mismo algoritmo
6. GenerarFormaPlana

Dos listas iguales, deben generar la misma secuencia de instrucciones, independientemente de su notación. Para esto, el compilador siempre generará código como si se hubiera encontrado una lista en notación recursiva. En el caso de la notación abreviada que recorrerá los elementos de la lista, pero generará una secuencia de instrucciones idéntica a la de la notación recursiva. En la notación recursiva, siempre hay una lista vacía al final, pero en el caso de la notación abreviada, la lista vacía del final está implícita, y por lo tanto hay que incluirla explícitamente. Como la notación puede ser combinada, entonces puede ocurrir que la lista vacía esté o no al final. Para resolver esto, se realiza lo siguiente: si no se encontro una barra, y no hubo una lista vacía inmediatamente antes, entonces hay que generar código para lista vacía explícitamente. En caso contrario, el código se genera al encotrar la lista vacía.

A continuación se muestra el algoritmo completo para compilar una lista, independientemente de la notación usada. Como la lista vacía se considera como una constante, el algoritmo no necesita considerar el caso de lista vacías.

Algoritmo: ProcesarLista
1. Generar la instrucción `get_list` o `put_list`
2. Compilar el primer elemento de la lista
3. Mientras haya una coma: compilar un elemento de la lista
4. Si hay una barra: compilar el resto de la lista con este mismo algoritmo
5. Si no hubo lista vacía: incluir lista vacía al final.

Ejemplo 5.1 : Las figura 8 muestra el código WAM del siguiente programa:

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

A la derecha de las instrucciones se muestra a modo de comentario, el elemento del programa que las generó.

member/2:	try_me_else M1	member
	allocate	(
	get_var $X_1 A_1$	X,
	get_list A_2	[
	unify_value X_1	X|
	unify_void	_]
	deallocate	)
	proceed	.
M1:	trust_me	member
	allocate	(
	get_var $X_1 A_1$	X,
	get_list A_2	[
	unify_void	_|
	unify_var X_3	Y])
	put_value $X_1 A_1$	:-member(X,
	put_value $X_3 A_2$	Y
	call member/2	
	deallocate	)
	proceed	.

Figura 8: Código de `member/2`

6. Predicados y operadores predefinidos

En Prolog, existen algunos predicados y operadores predefinidos, que no están incluidos en el reporte original de la WAM, y tampoco en el trabajo de Hassan Aït-Kaci. Tal es el caso del operador de negación por falla finita, el predicado `call/1`, y los predicados `true/0` y `fail/0`. En esta sección se mostrará como incorporar a los elementos citados como predefinidos en el lenguaje.

6.1. El predicado `true/0`

Los PLR permiten el uso de *presuposiciones* (*i.e.*, CPRs sin cuerpo). Para distiguir presuposiciones de hechos, se utiliza la sintaxis: "`p -< true`". El predicado `true/0` siempre da éxito, por lo tanto, es equivalente a tener el hecho "`true`" en el programa . Una implementación posible sería incluir en todo PLR, a `true/0` como un hecho más. Sin embargo, esto quedaría en manos del programador, y su olvido significaría no poder utilizar ninguna presuposición. Es por esta razón, que se prefirió tratar a `true/0` como un predicado predefinido, incluyendo en todo programa el siguiente código WAM:

```
true/0:  proceed
```

Como el código del predicado `true` es nulo, otra alternativa sería que el compilador directamente no genere ninguna instrucción al encontrarse con una llamada al predicado `true/0`.

6.2. El predicado `fail/0`

El predicado `fail/0`, en contraposición con `true/0`, falla siempre. Es equivalente a un hecho que no está presente en un programa. Por la misma razón que antes, se prefirió incluirlo como un predicado predefinido para el programa, utilizando el siguiente código:

```
fail/0:  neck_cut
         fail_and_backtrack
```

Donde la nueva instrucción `fail_and_backtrack`, lo único que hace es fallar y llamar a backtracking, y la instrucción `neck_cut` previene de que se redefina `fail/0` y pueda dar éxito. También podría hacerse que el compilador genere directamente la instrucción `fail_and_backtrack` en el código de la cláusula que llama a `fail/0`.

6.3. El predicado `call/1`

El predicado extra lógico `call/1` recibe por parámetro un término `P` que corresponde a un predicado, y efectúa la consulta `P`. Una llamada a `call(P)` puede hacerse sólo si `P` es: una constante (predicado de aridad 0), un término `p/n`, o una variable que esta instanciada en alguno de los dos elementos anteriores. En el caso de la variable, si no está unificada a un predicado como en los dos primeros casos, `call(X)` fallará.

El código del predicado `call/1` será el siguiente:

```
call/1:  prepare_call A1
         execute ??
```

Estará siempre en memoria, y debe cargarse antes que cualquier programa. El parámetro de la instrucción `execute` estará indefinido hasta el momento en que se ejecute. La instrucción `prepare_call` será la encargada de colocar el parámetro correcto para `execute`, y haber cargado todos los registros de argumentos que utilizará el predicado a llamar. La instrucción `prepare_call` utiliza la dirección que tiene en el registro A_1 para obtener el nombre del predicado a llamar y su aridad. Si el predicado a llamar no está definido, entonces falla. Una vez obtenida la aridad se cargan los registros de argumento, y luego se modifica el código del programa para colocar el parámetro correcto a la instrucción `execute` (que es la siguiente a `prepare_call`).

6.4. El operador de negación por falla de PROLOG

Como los PLR tienen definido su propio `not`, que tiene una semántica diferente a la negación por falla de Prolog, se utilizará "`naf`" para referirse al operador de negación por falla de PROLOG, y "`not`" para el correspondiente a los PLR.

Obsérvese en primer lugar que es posible definir al operador `naf` como un predicado dentro de un programa, si se dispone de los predicados extra lógicos `call/1`, `!` (cut) y `fail/0`.

Ejemplo 6.1 :

```
naf(X) :- call(X),!,fail
naf(_)
```

```
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
```

Figura 9: Código del operador `naf`/1

Se implementará al `naf` como un predicado predefinido, incorporando el código al igual que el predicado `call/1`, al principio del programa. Esto permitirá además algunas optimizaciones en el código, el cual se muestra en la figura 9.

Cuando el compilador encuentra una consulta "`naf q/n`", compila a "`q/n`" como un término, y luego coloca la instrucción "`call naf/1`". De esta forma, al ejecutarse el código, el predicado `q/n` queda en referenciado por el registro A_1, y el predicado `naf` lo puede utilizar como parámetro.

7. Compilación de una consulta

Al efectuar una consulta a un PLR, se debe construir una justificación para dicha consulta [16, 10]. La compilación de una consulta para un PLR, contiene algunos elementos adicionales que no poseen las cláusulas del programa. La figura 10 muestra el código JAM que se generará para contruir una justificación para la consulta `q(X,Y)` (el símbolo ";" indica que el resto de la linea es un comentario).

```
?:  allocate 2                  ; comienzo de la consulta
    save_query_var X Y1         ;
    put_var_perm A1 Y1          ;
    save_query_var Y Y2         ;
    put_var_perm A2 Y2          ;
    call q/2                    ; intenta construir un argumento para q/2
    prepare_to_defeat 1         ; prepara para buscar derrotadores
    put_constant defeat1 A1     ; intenta probar que ...
    call naf                    ; ... no existen derrotadores
    show_answer Y1              ; muestra la respuesta de la variable X
    show_answer Y2              ; muestra la respuesta de la variable Y
    deallocate                  ;
    end_query                   ; fin consulta
```

Figura 10: 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 un derrotador será aceptable si no está a su vez derrotado, el éxito o falla del predicado `defeat1/0` dependerá del árbol de dialéctica que se forme a partir de él (más detalles sobre este proceso pueden encontrarse en [10]).

Como se vió en la sección 3 al compilar un programa, o una consulta, las variables pasan a ser simplemente una dirección de memoria y su identificador desaparece como tal. Sin embargo, el identificador de las variables de una consulta debe recordarse a fin de mostrar la respuesta encontrada. Para que la máquina virtual se comporte de la manera antes indicada, esto es, que responda con las sustituciones de las variables de la consulta, se extendió la WAM con dos nuevas instrucciones: `save_query_var` que guarda la posición de la variable para mostrar la respuesta al terminar la consulta, y `show_answer` que muestra la respuesta correspondiente.

Al generar el código JAM de una consulta, cuando se detecta la primera aparición de una variable `X`, se guarda el identificador `X` en la tabla de identificadores, y se genera la instrucción "`save_query_var P` Y_n" donde Y_n es el lugar dentro del registro de entorno de la consulta, donde se ubicará la variable `X`, y `P` la posición del `X` en la tabla de identificadores. Cuando se termina de generar el código de la consulta, y antes de la instrucción `deallocate`, se genera una instrucción "`show_answer P`" para cada variable encontrada en la consulta.

La instrucción `show_answer Var` se ejecuta después que terminó la consulta en forma exitosa. El parámetro `Var` indica la posición de la tabla de identificadores donde se encuentran: el identificador de la variable y la posición de memoria donde comienza la estructura a mostrar. Si la estructura a mostrar es una variable, muestra la dirección de memoria de la misma precedida del símbolo "`_`". Si es una constante, muestra el identificador de la misma, y en el caso de listas o estructuras complejas, recorrerá el HEAP, mostrando el término.

8. Conclusiones

El compilador desarrollado permite traducir un programa lógico rebatible a instrucciones de la nueva máquina abstracta para la argumentación rebatible llamada JAM. El compilador genera código para predicados predefinidos del lenguaje, y permite que el programa este dividido en varios archivos. El compilador construído puede utilizarse también para traducir programas en Prolog a código WAM, y podría modificarse fácilmente para producir compiladores para otras extensiones de la programación en lógica.

Referencias

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

[2] Alferes José J. Pereira Luis M. *Contradiction: when avoidance equals removal (part I and II).* Proc. of Ext. of Logic Programming, 4th International Workshop ELP'93 St. Andrews U.K. March 1993.

[3] Dung Phan M. *Negations as Hypotheses: an Abductive Foundation for Logic Programming.* Proc. of 8th. Int. Conf. on Logic Programming (ICLP) 1991.

[4] 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.

[5] Fillottrani, Pablo R. *Sistemas de razonamiento no monótono y su relación con la semántica de las bases de datos deductivas.* Tesis de Magister en Ciencias de la Computación, Bahía Blanca, Universidad Nacional del Sur, Octubre de 1995.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] García A. J. y Simari G. R. *Una extensión de la máquina abstracta de Warren para la argumentación rebatible.* Cuarto Workshop Argentino sobre Aspectos Teóricos de la Inteligencia Artificial (ATIA'97), Universidad Nacional de la Plata, 1997.

[11] Gelfond M. and Lifschitz V. *Logic Programs with Classical Negation.* Proc. of 7th. Int. Conf. on Logic Programming (ICLP) 1990

[12] Inoue K. *Extended Logic Programming with Default Assumptions.* Proc. of 8th. Int. Conf. on Logic Programming (ICLP) 1991.

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

[14] 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.

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

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

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

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

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