Una arquitectura para la ejecución de Programas Lógicos Rebatibles

A. G. Stankevicius[1] A. J. García G. R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca – Buenos Aires – ARGENTINA
e-mail: {astank, ccgarcia, grs}@criba.edu.ar

Resumen

La *Programación en Lógica Rebatible* es una extensión a la programación en lógica convencional, surgida de la investigación llevada a cabo en el campo de la Inteligencia Artificial, específicamente en el área de razonamiento no monótono.

Actualmente, las arquitecturas definidas para la ejecución de programas lógicos (*e.g.*, Prolog) utilizan como estándar de facto para su implementación una *máquina abstracta*. La programación en lógica rebatible cuenta con una máquina abstracta definida a tal efecto, la *Justification Abstract Machine* (JAM).

El objetivo de este trabajo consiste en definir una arquitectura apropiada para la ejecución de programas lógicos rebatibles, basada en la JAM.

1. Introducción

La programación en lógica rebatible [Gar97a] es una extensión a la programación en lógica convencional surgida de la investigación llevada a cabo en el campo de la Inteligencia Artificial, específicamente en el área de razonamiento no monótono. En reiteradas oportunidades [GL90, Ino91, GS95] se ha comprobado que la programación en lógica convencional no posee la expresividad requerida para capturar ciertos aspectos del razonamiento de sentido común, principalmente por su incapacidad de representar información potencialmente inconsistente o incompleta (información parcial).

Actualmente, las arquitecturas definidas para la ejecución de programas lógicos se basan en máquinas abstractas. Luego de que David Warren diseñó una máquina

[1] Becario de Iniciación a la Investigación, Secretaría General de Ciencia y Tecnología, UNS.

abstracta para la ejecución de Prolog [War83, War88], hoy en día denominada *Warren's Abstract Machine* (WAM), ésta se convirtió en el estándar *de facto* para la implementación de este lenguaje [Aït91]. La máquina abstracta puede implementarse como una máquina ser real (hardware) o virtual (implementada sobre alguna otra plataforma). La principal ventaja de contar con una máquina abstracta radica en que permite separar claramente la compilación de la ejecución, esto es, se puede modificar el compilador o la máquina abstracta en forma independiente. De esta manera, los esfuerzos para mejorar la eficiencia pueden centrarse en nuevas arquitecturas para la máquina abstracta, y las extensiones al lenguaje fuente pueden resolverse dentro del compilador.

Los programas lógicos rebatibles son una extensión a los programas lógicos convencionales (ver sección 2). Como tales, admiten ser ejecutados dentro de una arquitectura basada en una máquina abstracta con relativa naturalidad. En [GS97a] se define una máquina abstracta a tal efecto, denominada Justification Abstract Machine (JAM). El objetivo de este trabajo consiste en definir una arquitectura apropiada para la ejecución de programas lógicos rebatibles, basada en la JAM.

Este trabajo esta organizado de la siguiente manera: en la siguiente sección se introducen los conceptos básicos relacionados con la programación en lógica rebatible. En la sección 3 se bosqueja un primer modelo de arquitectura, analizando sus ventajas y desventajas, e identificando los puntos débiles de la misma. Posteriormente, se presenta la arquitectura propuesta, que incorpora las mejoras consideradas y soluciona los problemas identificados sobre el modelo preliminar. Luego, en la sección 4 se analiza una implementación particular de la arquitectura especificada. Finalmente, en la sección 5 se presentan las conclusiones obtenidas.

2. La programación en lógica rebatible

En esta sección se introducen los conceptos básicos de la programación en lógica rebatible, siguiendo a [GS97a]; para una descripción más detallada se puede consultar [Gar97] o [GS99]. La programación en lógica rebatible surge como una extensión de la programación en lógica convencional, empleando conceptos de la argumentación rebatible *(defeasible argumentation)*, con el objeto de capturar aspectos del razonamiento de sentido común difíciles de expresar a través de la programación en lógica convencional. Los programas lógicos rebatibles permiten expresar información potencialmente inconsistente o incompleta, pudiendo decidir entre metas contradictorias mediante un criterio de preferencia.

En el lenguaje de la programación en lógica rebatible, un predicado atómico es una estructura de la forma $p(t_1,\ldots,t_k)$, con $k \geq 0$ donde p es el nombre del predicado y los t_i son términos (en el sentido usual de Prolog). Un literal l es, o bien un predicado atómico a, o bien un predicado atómico negado $\sim a$ (siguiendo el espíritu de [Llo87]). Notar que el símbolo '~' denota la negación clásica; no se debe confundir con el operador *not*, que denota la negación por falla. El complemento de un literal será considerado en base a la negación clásica, esto es, '$comp(a) = \sim a$' y '$comp(\sim a) = a$'.

Un **programa lógico rebatible** (PLR), es un conjunto finito de **cláusulas de programa extendidas** (CPE) y **cláusulas de programa rebatibles** (CPR). Una CPE es una cláusula de la forma '$l \leftarrow p_1,\ldots,p_n.$', con $n \geq 0$, donde l es un literal y cada p_i es un literal o un literal precedido por el operador *not* de la negación por falla. Si particularmente $n = 0$, entonces se notará '$l \leftarrow$ ***true***.', y en ese caso l se denomina *hecho* (donde **true** recibe la interpretación Prolog usual). Por otra parte, una CPR es una

cláusula de programa de la forma *'l –< p_1,…,p_n.'*, con las misma consideraciones sobre *l* y los p_i que para las CPE. Nuevamente, en el caso particular de *n = 0*, se notará *'l –< **true**.'*, y en esta oportunidad *l* se denomina *presuposición*. El símbolo '–<' se utiliza para distinguir una CPE de una CPR. Esta distinción es importante, ya que las CPE representan información concreta (indisputable), y por el contrario las CPR representan información tentativa. 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 resta, dado un determinado PLR, el conjunto de sus CPE se denotará ***S***, y el conjunto de sus CPR se denotará ***D***. Una **meta** (o consulta) es simplemente un literal. Una **derivación rebatible** para una meta *m* es el conjunto de cláusulas de programa instanciadas (CPE y CPR), que permiten derivar *m*. Un conjunto de cláusulas es **consistente** si no es posible derivar rebatiblemente un par de literales complementarios (recordemos, complementarios con respecto a la negación clásica). Análogamente, un conjunto de cláusulas es **inconsistente** si es posible derivar rebatiblemente un par de literales complementarios. En un PLR, el conjunto ***S*** de CPE debe ser consistente, mientras que el conjunto ***D*** de CPR e incluso el propio PLR pueden ser inconsistentes.

Ejemplo 2.1: En el siguiente PLR, es posible apreciar la expresividad del lenguaje.

vuela(X) –< pájaro(X).	pingüino(pengo) ← **true**.
~vuela(X) –< enfermo(X).	pájaro(tweety) ← **true**.
~vuela(X) ← pingüino(X).	pájaro(fred) –< **true**.
pájaro(X) ← pingüino(X).	enfermo(fred) –< **true**.

Notar que, entre otras metas, es posible derivar rebatiblemente tanto *'vuela(fred)'* como *'~vuela(fred)'*.

La noción de derivación rebatible no prohibe la derivación rebatible de literales complementarios. Por esta razón, resulta necesario definir un criterio de inferencia que acepte a lo sumo una de las metas contradictorias entre sí; es aquí donde la argumentación rebatible entra en juego.

Definición 2.1: [GS97a] Dado un PLR, formado por los conjuntos ***S*** y ***D***, decimos que ***A*** es un argumento para la meta *m*, si ***A*** es un conjunto de CPR instanciadas, tomadas del conjunto ***D***, tal que:

- Existe una derivación rebatible de *m* a partir de ***S*** ∪ ***A***.
- ***S*** ∪ ***A*** es consistente.
- ***A*** es menor conjunto (con respecto a la inclusión de conjuntos) que cumple con las condiciones anteriores.

Si ***A*** es un argumento para *m*, se dice que ⟨***A***, *m*⟩ es una estructura de argumento. Un argumento ***B*** para *q* se denomina subargumento de la estructura de argumento ⟨***A***, *m*⟩ sí, y sólo sí, ***B*** ⊆ ***A***.

De acuerdo al programa presentado en el ejemplo 2.1, existen derivaciones rebatibles para *'vuela(tweety)'* y *'vuela(pengo)'*. Sin embargo, sólo posee argumento *'vuela(tweety)'*, ya que las cláusulas (instanciadas) empleadas en la derivación de la meta *'vuela(pengo)'* son inconsistentes con ***S*** y no constituyen un argumento.

Utilizando el PLR del ejemplo 2.1 es posible construir argumentos para los literales complementarios *'vuela(fred)'* y *'~vuela(fred)'*. Para decidir entre estas metas en conflicto resta especificar las condiciones que determinan cuando un argumento para un literal es lo suficientemente fuerte como para considerarse una justificación. El proceso de justificación para un literal involucra la construcción de un argumento para ese literal que no esté derrotado.

Definición 2.2: **[GS97a]** Sea S el conjunto de CPE de un PLR. Se dice que $\langle \boldsymbol{A}_1, h_1 \rangle$ contraargumenta a $\langle \boldsymbol{A}_2, h_2 \rangle$ en el literal h sí, y sólo sí, existe un subargumento $\langle \boldsymbol{A}, h \rangle$ de $\langle \boldsymbol{A}_2, h_2 \rangle$ tal que $\boldsymbol{S} \cup \{h_1, h\}$ sea inconsistente. El argumento $\langle \boldsymbol{A}, h \rangle$ se denomina subargumento de desacuerdo.

Definición 2.3: **[GS97a]** Un argumento $\langle \boldsymbol{A}_1, h_1 \rangle$ derrota a otro argumento $\langle \boldsymbol{A}_2, h_2 \rangle$ en un literal h sí, y sólo sí, existe un subargumento $\langle \boldsymbol{A}, h \rangle$ de $\langle \boldsymbol{A}_2, h_2 \rangle$ tal que $\langle \boldsymbol{A}_1, h_1 \rangle$ contraargumenta a $\langle \boldsymbol{A}_2, h_2 \rangle$ en el literal h y se cumple alguna de estas condiciones:

- $\langle \boldsymbol{A}_1, h_1 \rangle$ es *mejor*[2] que $\langle \boldsymbol{A}, h \rangle$ (derrotador propio).
- $\langle \boldsymbol{A}_1, h_1 \rangle$ no puede compararse con $\langle \boldsymbol{A}, h \rangle$ (derrotador de bloqueo).

Por último, no debe existir un subargumento propio $\langle \boldsymbol{B}, q \rangle$ de $\langle \boldsymbol{A}_1, h_1 \rangle$ (i.e., $\boldsymbol{B} \subset \boldsymbol{A}_1$) tal que $\boldsymbol{S} \cup \{q, h_2\}$ sea inconsistente.

Ejemplo 2.2: El argumento para *'~vuela(fred)'* es un contraargumento para el argumento *'~vuela(fred)'*.

La siguiente definición captura la esencia del mecanismo empleado para determinar cuando un argumento constituye una justificación. Como se mencionó con anterioridad, un argumento para un literal será una justificación cuando no esté derrotado. Un argumento estará derrotado cuando posea derrotadores que a su vez no estén derrotados, y así sucesivamente.

Definición 2.4: **[GS97a]** Un árbol de dialéctica para la estructura de argumento $\langle \boldsymbol{A}, h \rangle$, denotado $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$, se define inductivamente de la siguiente manera:

1.- Si $\langle \boldsymbol{A}, h \rangle$ no posee derrotadores (propios o de bloqueo), entonces el árbol formado por sólo un nodo, conteniendo la estructura de argumento $\langle \boldsymbol{A}, h \rangle$, es un árbol de dialéctica.

2.- Si $\langle \boldsymbol{A}, h \rangle$ posee derrotadores (propios o de bloqueo) $\langle \boldsymbol{A}_1, h_1 \rangle, \ldots, \langle \boldsymbol{A}_n, h_n \rangle$, entonces el árbol formado por un nodo raíz conteniendo la estructura de argumento $\langle \boldsymbol{A}, h \rangle$, padre de las raíces de los árboles de dialéctica asociados a cada uno de sus derrotadores, será un árbol de dialéctica.

Sea $\boldsymbol{T}_{\langle \boldsymbol{A}_0, h_0 \rangle}$ el árbol de dialéctica para la estructura de argumento $\langle \boldsymbol{A}_0, h_0 \rangle$. Todo camino λ en $\boldsymbol{T}_{\langle \boldsymbol{A}_0, h_0 \rangle}$ desde la raíz (conteniendo $\langle \boldsymbol{A}_0, h_0 \rangle$) hasta alguna hoja (conteniendo $\langle \boldsymbol{A}_n, h_n \rangle$), denotado $\lambda = [\langle \boldsymbol{A}_0, h_0 \rangle, \langle \boldsymbol{A}_1, h_1 \rangle, \ldots, \langle \boldsymbol{A}_n, h_n \rangle]$, constituye una **línea de argumentación** para $\langle \boldsymbol{A}_0, h_0 \rangle$. En λ, $\langle \boldsymbol{A}_0, h_0 \rangle$ será un argumento de soporte; para el resto, si $\langle \boldsymbol{A}_i, h_i \rangle$ es un argumento de soporte (resp. interferencia), $\langle \boldsymbol{A}_{i+1}, h_{i+1} \rangle$ será un argumento de interferencia (resp. soporte) en λ. El conjunto de argumentos de soporte en λ se denota $\lambda_{\mathbf{S}}$ y el conjunto de argumentos de interferencia se denota $\lambda_{\mathbf{I}}$.

[2] Cualquier orden parcial puede ser utilizado; por ejemplo, el criterio de especificidad [Poo85, GS96].

Un árbol de dialéctica se dice **aceptable** si todas sus líneas de argumentación son aceptables. Una línea argumentativa λ se dice aceptable si satisface las siguientes condiciones:

- λ_S y λ_I son consistentes (*i.e.*, no es el caso que en base a $S \cup \lambda_S$ ó $S \cup \lambda_I$ se puedan derivar literales complementarios).
- Un argumento $\langle \boldsymbol{A}_n, h_n \rangle$ no debe ser un subargumento de otro argumento $\langle \boldsymbol{A}_i, h_i \rangle$ para todo $i < n$.

Definición 2.5: [GS97a] Los nodos de un árbol de dialéctica $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$ se etiquetan como *nodo derrotado* (nodo-**D**) o *nodo no-derrotado* (nodo-**U**) de acuerdo a la siguiente especificación recursiva:

1.- Toda hoja de $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$ se etiqueta como nodo-**U**.

2.- Sea $\langle \boldsymbol{B}, q \rangle$ un nodo interno de $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$:

a) $\langle \boldsymbol{B}, q \rangle$ se etiqueta como nodo-**U** sí, y sólo sí, todo hijo de $\langle \boldsymbol{B}, q \rangle$ está etiquetado como nodo-**D**.

b) $\langle \boldsymbol{B}, q \rangle$ se etiqueta como nodo-**D** sí, y sólo sí, tiene al menos un hijo etiquetado como nodo-**U**.

Definición 2.6: [GS97a] Un argumento $\langle \boldsymbol{A}, h \rangle$ se considera una justificación para el literal *h* sí, y sólo sí, la raíz del árbol de dialéctica aceptable $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$ es un nodo-**U**.

Definición 2.7: [GS97a] Sea ***P*** un PLR:

- El conjunto de respuestas positivas de ***P*** está formado por los literales *h* para los cuales existe un argumento ***A***, tal que $\langle \boldsymbol{A}, h \rangle$ es una justificación para *h*.
- El conjunto de respuestas negativas de ***P*** está formado por los literales *h* para los cuales toda estructura de argumento $\langle \boldsymbol{A}, h \rangle$ posee al menos un derrotador propio etiquetado como nodo-**U** en $\boldsymbol{T}_{\langle \boldsymbol{A}, h \rangle}$.
- El conjunto de respuestas indecisas de ***P*** está formado por los literales *h* para los cuales existe al menos un argumento para ellos, y además, toda estructura de argumento $\langle \boldsymbol{A}, h \rangle$ posee el nodo raíz de su árbol de dialéctica etiquetado como nodo-**D**, pero no posee derrotadores propios o bien, si lo tuviera, todos ellos están etiquetados como nodos-**D** (en otras palabras, posee al menos un derrotador de bloqueo etiquetado como nodo-**U**).
- El conjunto de respuestas desconocidas de ***P*** está formado por los literales *h*, posiblemente una cantidad infinita de ellos, tales que no pertenecen a ninguno de los conjuntos anteriores.

Dado un PLR ***P*** y una meta *m*, un intérprete para programas lógicos rebatibles deberá responder: SI, NO, INDECISO o DESCONOCIDO, en el caso de que *m* pertenezca al conjunto de respuestas positivas, negativas, indecisas o desconocidas, respectivamente.

3. Una arquitectura para la ejecución de PLR

Si bien existen diferentes formas de diseñar una arquitectura capaz de ejecutar PLR (o cualquier otra extensión de la programación en lógica convencional), el enfoque basado una máquina abstracta posee varias propiedades que lo hacen atractivo. Ya hemos mencionado una de ellas, la clara distinción que establece entre la compilación y la ejecución. Otro aspecto a destacar es la alta portabilidad: el sistema puede

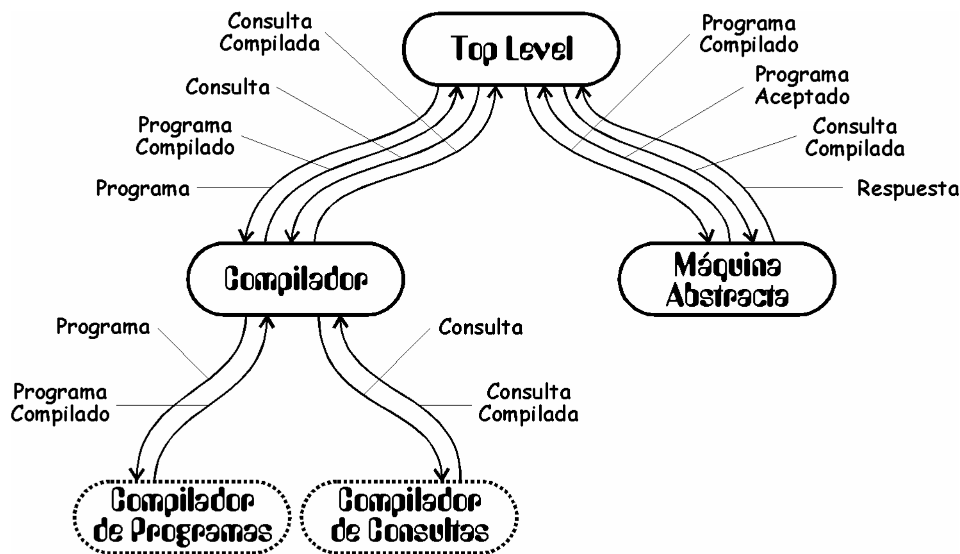

Figura 1. Un bosquejo de la arquitectura para ejecutar PLR.

cambiar de plataforma simplemente portando la máquina abstracta; el compilador no requiere mayores modificaciones, pues su máquina destino no ha sido modificada. Estas son, entre otras, las razones que fundamentan nuestra preferencia por este tipo de arquitectura (ver sección 4).

A continuación se presentará en primer lugar un modelo preliminar de arquitectura sobre el cual se analizarán sus ventajas y desventajas, identificando los puntos débiles del mismo. Posteriormente, en la sección 3.3 se presentará la arquitectura definitiva

3.1. Modelo preliminar

La figura 1 presenta el modelo preliminar de arquitectura. Este modelo cuenta con tres componentes: un compilador, una máquina abstracta (JAM) y un 'top-level' que gestiona la interacción entre ellos. El compilador a su vez supervisa dos componentes de carácter optativo (con contorno punteado) que compilan programas (PLR) o consultas (metas), respectivamente. Notar que, con el objeto de no complicar excesivamente el diagrama de la arquitectura, se han omitido los flujos de información relacionados con las condiciones excepcionales (*e.g.*, errores).

A continuación, se detallan las principales funciones de cada uno de estos componentes:

Top-Level

Este componente es el encargado de administrar el flujo de la información dentro de todo sistema basado en esta arquitectura. Esencialmente, la salida generada por el compilador luego de procesar un PLR o una consulta debe ser suministrada como entrada a la máquina abstracta (JAM).

En el caso de estar diseñando un intérprete (ver sección 4), como es usual en las implementaciones de Prolog, el 'top-level' es el encargado de interactuar con el usuario. Aún cuando la arquitectura se utilice en la construcción de un compilador, el 'top-level' sigue siendo el responsable de suministrar la entrada que corresponda a los otros componentes.

Compilador

El compilador es el encargado de realizar dos operaciones fundamentales: compilar PLRs y compilar consultas. Si bien ambas tareas son similares, existen razones para abordarlas por separado (lo cual se sugiere en la figura 1 mediante componentes opcionales). Esto se debe a que la compilación de consultas puede prescindir de la generación de cláusulas invertidas, código de verificación de consistencia, etc. (para un análisis más profundo de estos aspectos se puede consultar [GS97b] o [SGS99]).

Máquina Abstracta

Dentro de este componente se lleva a cabo la ejecución de los PLR. Como se mencionó con anterioridad, la máquina abstracta puede implementarse como una máquina real o virtual. Desde nuestra óptica, esta diferencia es inapreciable. La idea básica es que la JAM recibe el código compilado del programa que se desea ejecutar (*i.e.*, el código asociado a cada cláusula contenida en el programa actual). Posteriormente, recibe el código asociado a la consulta que se desea resolver y genera las respuestas correspondientes, así como información suplementaria sobre la última computación efectuada (*e.g.*, las sustituciones realizadas durante la resolución de la consulta o los argumentos construidos durante la misma).

La arquitectura de la JAM es conceptualmente sencilla [GS97a]. Cuenta con un conjunto de instrucciones y una estructura de memoria donde almacena diferentes tipos de elementos (código, pila de ejecución, línea de argumentación actual, etc.); también cuenta con una tabla de símbolos donde almacena la posición de comienzo en el área de código de la definición de cada cláusula cargada en la misma. La definición completa de las instrucciones de la JAM puede consultarse tanto en [Gar97] (su definición original), como en [Sta99] (la variante cuya implementación se discute en la sección 4).

3.2. Análisis del modelo preliminar

Analizando la arquitectura esquematizada en la figura 1, se puede apreciar la clara división entre la compilación y la ejecución (mencionada como característica de las arquitecturas basadas en máquinas abstractas); de más está decir la importancia que se le atribuye a la modularidad en la Ingeniería de Software, como principio válido para la obtención de productos de calidad.

Otra característica a destacar en el caso de construir un intérprete, si se implementa la interacción con el usuario dentro del componente denominado ‘top-level’, es que la división conceptual entre compilación, ejecución e interacción con el usuario se preserva (nuevamente, esta separación fomenta la modularidad).

Una propiedad quizás no tan evidente como las anteriores, es la flexibilidad que provee este esquema de distribución de tareas. Por caso, la implementación referida en la sección 4 consiste en una modificación de la definición original de la JAM para generar más información suplementaria, con el objeto de facilitar la visualización y la interpretación de PLR; la arquitectura fue lo suficientemente flexible como para adaptarse a esta variante de la JAM requiriendo sólo sencillas modificaciones.

Esta arquitectura permite implementaciones altamente eficaces si la división de tareas se explota mediante multithreading. La existencia de interfaces entre componentes conceptualmente claras (*e.g.*: ‘programa’, ‘programa compilado’, etc.) facilita

notablemente la definición de un thread por cada componente de la arquitectura. Este punto será abordado nuevamente en la sección 4.

Finalmente, una vez enumeradas las virtudes de esta arquitectura debemos considerar sus puntos débiles. La compilación incremental, propiedad presente en los programas lógicos convencionales y mantenida por los PLR, es una singular característica que en general no está presente en otros paradigmas de programación.

El siguiente ejemplo, si bien sencillo, captura la esencia de la compilación incremental:

Ejemplo 3.1: Sean P_1 y P_2 los siguientes PLR:

P_1 p(X) ← q(X).
q(a).

P_2 q(b).

Una vez compilado P_1, la consulta *p(X)* se satisface con la sustitución $X = a$. Si a continuación, compilamos P_2 (sin compilación incremental) el código generado para P_1 se pierde y la consulta *p(X)* no posee solución alguna (de acuerdo a la definición 2.7 la respuesta correspondiente será UNKNOWN). Si por el contrario, luego de compilar P_1 compilamos incrementalmente P_2, la consulta *p(X)* posee en esta oportunidad dos soluciones, las sustituciones $X = a$ y $X = b$.

Desafortunadamente, el modelo preliminar no admite compilación incremental. Para entender las razones que provocan esta situación, debemos recordar que al definir un entorno de ejecución para PLR, sus cláusulas (conceptualmente independientes entre sí) estarán relacionadas por la regla de selección que se adopte al implementar el lenguaje. Esta situación es análoga a la que se presenta al comparar las cláusulas de un programa lógico convencional (independientes entre sí por definición) y las cláusulas de un programa Prolog (relacionadas por la regla de selección que se utilice) [Llo97].

En el caso particular que nos concierne, la definición cargada en la memoria de la JAM, asociada a una determinada cláusula depende de la existencia de otras cláusulas para el mismo predicado. El código generado por el compilador denota el tipo de definición de la cláusula: puede ser la única definición, o bien la primer definición, pero existen otras, o bien no es la primer definición, ni la última, pero restan más alternativas, o bien es la última definición. Esta distinción fue introducida por Warren como una de las alternativas posibles para implementar la noción de *backtracking* (característica distintiva presente en todo tipo de programa lógico). Cada una de estas alternativas se distingue a través de diversas instrucciones de la JAM. En otras palabras, la misma cláusula debe generar distintas secuencias de instrucciones de acuerdo a su ubicación en el programa fuente y a la existencia de otras cláusulas para el mismo predicado. Un análisis más a fondo de esta situación escapa al alcance del presente trabajo; para mayor información se puede consultar [GS97b] o [SGS99].

La compilación incremental se pierde debido a que el compilador no puede adivinar si existen o existirán más definiciones de la cláusula que está compilando. Mediante un preprocesador se puede conocer, a lo sumo, si existen otras definiciones en el mismo programa fuente; lamentablemente, la compilación incremental se torna interesante cuando permite combinar diversos programas fuentes.

3.3. Arquitectura propuesta

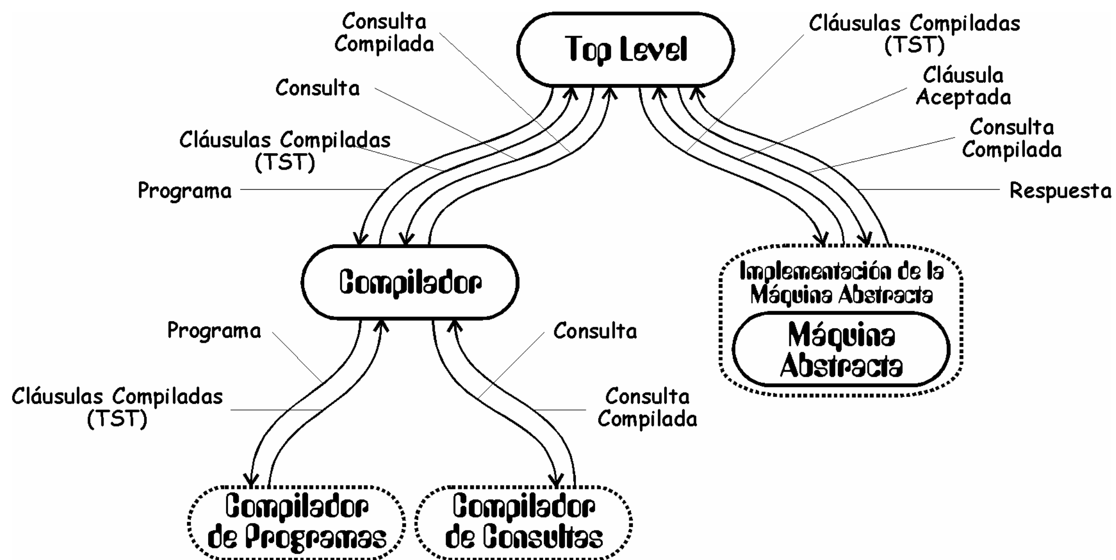

Figura 2. Arquitectura definitiva, incorpora las mejoras propuestas.

La figura 2 presenta la arquitectura definitiva, que ataca al problema de recuperar la compilación incremental. Los componentes principales son los mismos, simplemente debemos extender las funciones realizadas por uno de sus componentes (el que contiene a la máquina abstracta).

Para comprender la solución a este problema, debemos reconocer el principal inconveniente que enfrenta el compilador para realizar una compilación incremental: la falta de información acerca de la existencia de otras definiciones para el predicado actual. La máquina abstracta no está en esa situación: la JAM eventualmente tendrá acceso a la totalidad de las definiciones para cada predicado. Por esta razón, la idea detrás de la arquitectura refinada consiste en compilar cada definición, de cada predicado, como si fuera la única definición; posteriormente, la implementación de la máquina abstracta al recibir la cláusula compilada será la responsable de realizar las modificaciones necesarias. Esencialmente, se deben corregir la cláusula compilada que está siendo cargada y la última definición almacenada en la memoria. La posición del código de la última definición para un determinado predicado se puede recuperar de la tabla de símbolos (una de la estructuras de datos presentes en la JAM). Las modificaciones específicas, en términos de instrucciones JAM, pueden ser consultadas en [Sta99].

Debemos notar que la definición original de la máquina abstracta no contempla realizar estas modificaciones; las mismas serán responsabilidad de la implementación (sea esta virtual o física). Para resaltar este aspecto, en la figura 2 la implementación de la máquina abstracta contiene un subcomponente que representa la máquina abstracta original. La nueva funcionalidad requerida (*i.e.*, realizar las modificaciones apropiadas al cargar nuevas definiciones de cláusulas) estará soportada dentro del componente denominado ‘Implementación de la Máquina Abstracta’ pero fuera del subcomponente denominado ‘Máquina Abstracta’.

El flujo de información que transporta las cláusulas compiladas desde el compilador hasta la implementación de la máquina abstracta está etiquetado ‘Cláusulas Compiladas (TST)’; la TST, Tabla de Símbolos Temporaria, es una estructura de datos interna al compilador [GS97b, SGS99], que se utiliza durante la compilación de una cláusula y su tiempo de vida coincide con la compilación de la cláusula a la cual

está asociada (*i.e.*, se descarta al terminar la compilación). Ya que, entre otras cosas, la TST contiene el código generado por el compilador, se ha adoptado esta estructura como medio por el cual el 'top-level' alimenta a la JAM. Notar que la secuencia de instrucciones JAM por sí misma no es suficiente, pues la máquina abstracta desconoce a que predicado asociar esta secuencia; más aún, la estructura que se adopte también debe permitir la conversión de las referencias relativas en absolutas de una manera sencilla. Si bien la TST satisface estos requerimientos, debemos admitir que contiene información extra que no sería necesario acarrear. La utilización de esta estructura de datos debe considerarse como una sugerencia, más que un requerimiento.

Es posible argumentar que no es una decisión afortunada realizar las correcciones necesarias para obtener compilación incremental en la implementación de la máquina abstracta, ya que es durante la compilación cuando se debe realizar la mayor cantidad de tareas posibles, para aliviar la carga durante la ejecución (de por sí bastante costosa en recursos). El lector observador habrá notado que esta objeción es improcedente; el trabajo extra encomendado a la máquina abstracta se realiza durante la carga de las cláusulas y no durante la ejecución propiamente dicha (i.e., no existe sobrecarga durante la resolución de consultas).

Esta arquitectura refinada preserva las virtudes identificadas en la discusión previa, solucionando el principal inconveniente detectado. En la siguiente sección se comenta una implementación específica, basada en esta última arquitectura.

4. Implementación de la arquitectura propuesta

En esta sección, se discute la implementación llevada a cabo como parte del trabajo reportado en [Sta99]. El objetivo en ese trabajo consistió en extender la definición original de la JAM para generar más información suplementaria, con el objeto de facilitar la visualización y la interpretación de PLR. Como suele suceder durante el desarrollo de proyectos de software, se debió optar entre modificar alguna implementación preexistente o construir una nueva. Por diversas razones se decidió optar por la segunda alternativa; la arquitectura discutida en la sección 3 surgió durante la etapa de diseño del nuevo sistema.

El entorno elegido para implementar el nuevo sistema fue JDK 2.0 (conocido anteriormente como JDK 1.2), ya que el mismo está disponible en una multitud de plataformas. La elección del lenguaje de programación Java resultó sencilla luego de considerar sus atractivas características. Tal vez se pueda cuestionar que Java, al ser un lenguaje interpretado y altamente dinámico, puede no resultar 'eficiente'. Como la eficiencia no era un objetivo perseguido durante la construcción del nuevo sistema, esta observación fue sencillamente ignorada.

Cada componente principal de la arquitectura fue implementado mediante un objeto (recordemos, Java es un lenguaje orientado a objetos). Se diseño un compilador específicamente para el sistema (ver [GS99]). La arquitectura de la máquina abstracta fue implementada de la manera menos restrictiva posible, esto es, la mayor parte de sus estructuras internas son dinámicas, para permitir la utilización de todo recurso que esté disponible durante la ejecución de los PLR.

Con el objeto de no construir estructuras intermedias que actúen como *buffers* entre los componentes, se decidió implementar cada componente como un thread independiente (lo cual fue factible en virtud de la alta independencia entre los mismos). Fue una grata sorpresa descubrir que de esta manera la arquitectura empleada re-

cupera la eficiencia, el sistema implementado puede automáticamente (sin modificación alguna) emplear más de un procesador (si los mismos están disponibles). En general los threads (de Java) suelen competir para ganar el control del procesador en el cual ejecuta la JVM *(Java Virtual Machine)*, el entorno de interpretación del lenguaje Java, pero las nuevas implementaciones de ésta permiten (ante la presencia de más de un procesador) que cada thread ejecute en distintos procesadores.[3]

5. Conclusiones

Se definió una arquitectura que permite la ejecución de programas lógicos rebatibles y que soluciona el principal inconveniente identificado sobre el modelo preliminar. Particularmente, se resuelve el problema que plantea la compilación incremental, singular característica no presente en la mayor parte los lenguajes de programación, preservando las virtudes del modelo en el cual se basa.

Su viabilidad como marco para guiar la construcción de un entorno de ejecución para PLR quedó establecida luego de llevar a cabo una satisfactoria implementación basada en ella. El citado sistema está disponible (contactando al autor) para quienes deseen experimentar con el mismo.

6. Referencias

[Aït91] Aït-Kaci, H., *"Warren's Abstract Machine, a tutorial reconstruction"*, MIT Press, 1991.

[Gar97] García, A. J., *"La Programación en Lógica Rebatible: su definición teórica y computacional"*, Tesis de Magister, Universidad Nacional del Sur, Bahía Blanca, 1997.

[GL90] Gelfond, M., y Lifschitz, V., *"Logic Programs with Classical Negation"*, en proceedings, Seventh International Conference on Logic Programming (ICLP), 579–597, 1990.

[GS95] García, A. J., y Simari, G. R., *"Una aproximación a la programación en lógica rebatible"*, Segundo Workshop en Aspectos Teóricos de la Inteligencia Artificial (ATIA'95), 515–526, Universidad Nacional del Sur, Bahía Blanca, 1995.

[GS96] García, A. J., y Simari, G. R., *"El criterio de especificidad en la programación en lógica rebatible"*, Tercer Workshop en Aspectos Teóricos de la Inteligencia Artificial (ATIA'97), 603–615, Universidad Nacional de San Luis, San Luis, 1996.

[GS97a] García, A. J., y Simari, G. R., *"Una extensión de la máquina abstracta de Warren para la argumentación rebatible"*, Cuarto Workshop en Aspectos Teóricos de la Inteligencia Artificial (ATIA'97), Universidad Nacional de la Plata, La Plata, 1997.

[GS97b] García, A. J., y Simari, G. R., *"Un compilador para la programación en lógica rebatible"*, Cuarto Workshop en Aspectos Teóricos de la Inteligencia Artificial (ATIA'97), Universidad Nacional de la Plata, La Plata, 1997.

[GS99] García, A. J., y Simari, G. R., *"Defeasible Logic Programming"*, a ser publicado, 1999.

[3] Actualmente sólo disponible para la plataforma Solaris sobre SPARC de Sun.

[Ino91] Inoue, K., *“Extended Logic Programming with Default Assumptions”*, en proceedings, Eight International Conference on Logic Programming (ICLP), 1991.

[Llo87] Lloyd, J. W., *“Foundations of Logic Programming”*, second edition, Springer-Verlag, 1987.

[Poo85] Poole, D. L., *“On the Comparison of Theories: Preferring the Most Specific Explanation”*, en proceedings, Ninth International Joint Conference on Artificial Intelligence (IJCAI), 144–147, 1985.

[SL92] Simari, G. R., y Loui, R. P., *“A Mathematical Treatment of Defeasible Reasoning and its Implementation”*, Artificial Intelligence, 53:125–157, 1992.

[SGS99] Stankevicius, A. G., García, A. J. y Simari, G. R., *“Compilation Techniques for Defeasible Logic Programs”*, a ser publicado, 1999.

[Sta99] Stankevicius, A. G., *“Visualización e Interpretación de Programas Lógicos Rebatibles”*, Tesis de Licenciatura, Bahía Blanca, Universidad Nacional del Sur, 1999.

[War83] Warren, D. H. D., *“An Abstract Prolog Instruction Set”*, Technical Note 309, SRI Interational, Menlo Park, California, 1983.

[War88] Warren, D. H. D., *“Implementation of Prolog. Lectures Notes Tutorial Number 3”*, Fifth International Conference and Symposium on Logic Programming, Washington, 1988.