Seventh LACCEI Latin American and Caribbean Conference for Engineering and Technology (LACCEI'2009)
"Energy and Technology for the Americas: Education, Innovation, Technology and Practice"
June 2-5, 2009, San Cristóbal, Venezuela.

Mecanismos de traducción de Prolog a Java mediante ProgtoJav

Jhon Edgar Amaya
Universidad Nacional Exp. del Táchira, San Cristobal, Tachira, Venezuela, jedgar@unet.edu.ve

Jacinto Dávila
Universidad de Los Andes, Merida, Venezuela, jacinto@ula.ve

RESUMEN

Se plantea una estrategia de integración consistente en la traducción de los predicados de Prolog a una asociación de clases Java, donde reside un control individualizado declarativo. El control declarativo individualizado, consiste en la integración de los mecanismos propios de los lenguajes declarativos de Prolog como la resolución, en cada una de las clases Java vinculadas a los predicados correspondientes. Se desarrolló una aplicación de traducción automática de programas en Prolog a Java, utilizando la estrategia de integración diseñada.

Palabras claves: Java, Prolog, Integración, Control declarativo.

ABSTRACT

We describe an integration strategy for translation of Prolog predicates to a group of Java classes with a declarative control for class. Declarative control for class is the integration of the mechanisms of declarative languages like Prolog resolution in each Java class. We developed a program for automatic translation of Prolog into Java using the integration strategy designed.
.

Keywords: Java, Prolog, Integration, Declarative control.

1. INTRODUCTION

La Máquina Abstracta de Warren (WAM) suele ser la referencia básica al implementar los compiladores de Prolog. Consiste básicamente en la manipulación de los términos en Prolog que se convierten en registros, los cuales son agrupados en una secuencia de elementos almacenados en una *pila*. El concepto clave en la WAM es la definición de una metodología apropiada para manipular los términos de Prolog. Prolog está basado en la lógica clausal, que es a su vez un subconjunto de la lógica de primer orden. (Aït-Kaci, 1999) detalla el método seguido para la implementación de una WAM. La idea central de esta metodología consiste en definir progresivamente diferentes etapas de diseño para la consecución de una WAM funcional y efectiva. Bajo un esquema optimizado como la WAM, resulta complejo pensar en una arquitectura abierta y distribuida que implemente los mecanismos como resolución y unificación, propios de los lenguajes declarativos. Una descripción en forma de objetos podría resultar una alternativa viable para una futura implementación distribuida de los mecanismos de los lenguajes declarativos (Calejo, 2004).

La implementación de un lenguaje declarativo como Prolog en una estructura orientada a objetos pretende optimizar las búsquedas de los términos (Van Roy, 1994) (InterProlog, 2005) (Tarau, 1994). El principio de Warren, establece que "los registros deben ser localizados de tal forma que se evite el movimiento innecesario de datos, así como minimizar el tamaño del código”. La aproximación orientada a objeto brinda varias posibilidades para construir modelos de sistemas complejos, pero solamente los lenguajes orientados a objeto con una semántica declarativa serian adecuados para el análisis lógico de los sistemas, incluyendo la programación de los mismos (Kowalski, 2006)(Morozov, 1999).

La idea general de un traductor de instrucciones de Prolog a Java, involucra la definición de la estructura de las clases que representen los predicados correspondientes, así como, los mecanismos que permitan implementar los procesos de búsqueda de soluciones propios de los lenguajes declarativos; aprovechando además las potencialidades de los lenguajes orientados a objetos. Existen diferentes propuestas para la integración de lenguajes declarativos e imperativos como por ejemplo (JPL, 2004), (InterProlog, 2005), (JProlog, 1996), (Jinni, 2005) y (Prolog Café, 1997). Nuestra propuesta se decanta por una alternativa de traducción de Prolog a Java pero incluyendo una innovación, en lugar de definir un conjunto de clases que implementen un compilador Prolog y que “consuma” las cláusulas y los términos del programa; se procedió a definir el control sobre el espacio de búsqueda dentro de la misma clase que se asocia a un predicado. El objetivo de la incorporación de mecanismos de control en las clases permite brindar independencia a nivel de plataforma. Adicionalmente, permite reducir el consumo de recursos, ya que la clase está ajustada específicamente a la cantidad de cláusulas asociadas a un predicado particular.

En las próximas cuatro secciones se discute la información acerca de la estructura de las clases Java, justificación del formato propuesto, definición de las clases funcionales, así como el procedimiento para la generación automática de las clases Java a partir de un programa en Prolog. Al final se presentan los resultados correspondientes así como el trabajo de futuro del proyecto.

2. Estructura De Las Clases Java Asociadas A Predicados

Se definió una estructura particular para las clases de Java para soportar mecanismos y funciones de los lenguajes declarativos. La estructura de la clase diseñada se basa en la propuesta de (Tarau, 1994)(Jinni, 2005), en particular se define el nombre de la clase como el nombre del predicado más la aridad. Se realiza un análisis del programa en Prolog en búsqueda de todos predicados de la misma estructura (nombre y aridad), bien sea como hechos o consecuente de una regla. Supóngase el código en Prolog ***padre(a,d), hijo(a,f)*** y ***padre(X,G) :- hijo(X,H), hijo(G,H)*** (1). ***hijo(a,f)*** corresponde a una cláusula asociada al predicado ***hijo/2***. Del programa (1) se generarán dos clases ***Padre_2*** e ***Hijo_2*** asociado a los predicados correspondientes. Las cláusulas se dividen en dos grupos. El primer grupo corresponde a las cláusulas que en Prolog reciben el nombre de *hechos*. El segundo grupo corresponde a las cláusulas llamadas *reglas*. La estructura de las clases involucra por lo tanto, la agrupación de cláusulas cuyos consecuentes sean de la misma forma del predicado, así como los átomos asociados al predicado, asi se definen dos atributos denominados ***numberRules*** y ***numberAtoms***, respectivamente. La idea subyacente es la realización de una búsqueda expedita de las consultas realizadas en las clases Java. Se excluyen los términos no *grounded* de los atributos de la clase Java, ya que ellos pueden poseer una o varias variables, por lo cual, tienen una influencia particular en espacios de búsqueda y deberán ser tratados de forma diferente. Se incluye en la estructura de la clase, métodos que permitan obtener la información de los atributos propios de la clase particular.

2.1 IMPLEMENTACIÓN DE LA UNIFICACIÓN DE PROLOG EN LAS CLASES JAVA

Para la implementación de la unificación en Java se utilizó una adaptación del algoritmo de Robinson. Se creó la clase ***UnifTerm.java***, en la cual se incluye el algoritmo de unificación junto con la generación del *Unificador Más General*, que consiste de un vector de términos en el cual se asocia una variable libre con un término. Se definieron los mecanismos de construcción de términos a partir del *Unificador Más General*, para obtener átomos de otros átomos con variables libres —denominadas ***Hi***—. Esta acción de construcción corresponde al concepto conocido como sustitución (Hogger, 1990). Los métodos encargados de realizar el proceso de sustitución los d denominamos ***buildSus****. Se define un método de búsqueda de términos —se denomina ***searchTCV***—, cuya consulta abarque algún tipo de variable y tiene como parámetros de entrada una cantidad de términos asociados a la cantidad de subtérminos del predicado. Para llevar a cabo el proceso de unificación se definieron tres vectores: ***varbl***, ***argum*** *y* ***argUf***. El primero consiste en todos los términos asociados a las variables libres, como en el caso de un compilador de Prolog tradicional. Es una estructura temporal que permita almacenar el estado de una búsqueda en un momento particular. El segundo vector permite almacenar la información correspondiente a los términos de entrada, en este caso constituyen las variables que forman la consulta. El tercero constituye el compendio de los valores temporales del proceso de unificación y contiene las variables asociadas a los subtérminos del predicado sobre el cual se está ejecutando la unificación.

Para entender un poco mejor la utilización de los vectores ***varbl***, ***argum*** *y* ***argUf***, supóngase que se tiene un predicado de la forma ***a(aa, bb),*** y se realiza la consulta ***?a(aa,bb)*** entonces, los vectores contendrán los siguientes valores antes de la unificación ***argum={aa, bb}, argUf={H1,bb}*** y ***varbl={H1,H2}***. Luego del proceso de unificación los vectores contendrán ***argum={aa,bb}***, ***argUf={aa, bb}***, ***varbl={H1,H2}***, donde ***H1=aa*** y ***H2=bb***. Las variables ***Xi*** asociadas al vector ***argum*** permanecen invariables durante el proceso de unificación. Las variables ***Ti*** del vector ***argUf***, se inicializan de acuerdo a la definición del predicado excepto que las variables del predicado ahora se expresan como variables libres ***Hi***. Las variables ***Ti*** se modifican según el valor aportado por las variables libres asociadas a las variables ***Hi*** del vector ***varbl*** una vez concluido el proceso de unificación.

En el momento de llevar a cabo el proceso de generación de las clases se utiliza un método de *parsing*. Los átomos obtenidos deben sustituir sus variables por variables libres —términos del tipo ***Hi***—. Por ejemplo, sea la regla ***padre(X,G) :- hijo(X,H), hijo(G,H),*** entonces se generará ***padre(H1,H2)*** y se almacenará en las variables temporales con los valores de los términos ***H1*** y ***H2***. El nombre de las variables es no es relevante ya que para la WAM constituyen apuntadores específicos a registros en la memoria de la máquina.

Definidos los vectores ***argum, argUf y varbl***; se procede a realizar la unificación entre los vectores correspondientes a los términos ***Xi*** y ***Ti***. Si el algoritmo de unificación puede lograr su cometido, se procede a la construcción de los términos que serán almacenados en unas estructuras temporales denominadas ***termAi*** y que representan el resultado de la consulta. Por ejemplo, ***termA1=(UnifTerm).buildSus(T1, mGUnf),*** donde ***mGUnf***, es el unificador más general (Hogger, 1990). Se requiere definir un conjunto de atributos que permitan almacenar las respuestas a las consultas, por lo tanto se define ***n*** —equivalente a la aridad del predicado— términos como atributos. Las respuestas a las consultas se almacenan en los términos ***termAi*** y se utiliza un método definido en la clase ***Term*** que permite imprimir en pantalla el valor de dichos términos. El método de impresión de la clase se denomina ***String printAnswer(Term termIn, Term termExt),*** donde ***termIn*** es el término a imprimir y ***termExt*** retiene el valor inicial del término antes de cualquier procesamiento.

2.2 SIMULANDO LA ESTRATEGIA DE SELECCIÓN DE CLAUSULAS DE PROLOG

Dada la estructura de la clase en Java cuando se incluyen términos *grounded* y átomos en general, se debe abordar la estructura a implementarse para tratar con las reglas. El control asociado a la clase está determinado por el método ***searchT***. Se incluye un nuevo método denominado ***boolean searchRule(Term, Term, ..., Term, int),*** que tiene como parámetros los subtérminos del término a consultar. En el método ***searchRule*** residen agrupadas todas las reglas asociadas al predicado que se encuentran dentro del programa Prolog sujeto a análisis. Es importante definir la estructura interna de cada regla para que se lleve a cabo el proceso de inferencia como en el caso de los lenguajes declarativos tradicionales. En primer lugar el método utiliza los mismos tres vectores discutidos en la Sección 2.1. con lo cual en las reglas también se definen variables libres. Con cada una de las reglas se trata de unificar las variables de entrada —denominadas ***Xi***— con respecto a las variables temporales —términos ***Ti***—.

(Tarau, 1994)(Diaz, 1995)(Calejo, 2004) proponen una estructura similar a un "case" para llevar el control de Prolog en una lenguaje como C. Nuestra propuesta incluye una generalización, utilizando estructuras simples y encadenadas basadas en ***while-if***. La estructura ***while*** permite construir el espacio de búsqueda y definir la búsqueda bajo la estrategia primero en profundidad. En Prolog se conoce esta estrategia como encadenamiento hacia atrás o *backward chaining.* Se establece una estructura ***while-if*** por cada uno de los términos existentes para verificar la asertividad de una regla. Se utiliza la estructura ***while (j1=0 ó j1< predicado_regla.getNumberTerms()),*** donde ***j1*** es una variable entera y *predicado_regla* es una clase asocida al predicado que es una condición de la regla analizada. El método ***getNumberTerms*** devuelve la cantidad de términos *grounded* definidos en la clase *predicado_regla.* La variable ***j1***, se utiliza para controlar el acceso a las estructuras ***if***. Como puede verse, se tienen dos sentencias disjuntas. La segunda de las sentencias permite el recorrido del árbol de búsqueda, a través de los términos definidos en la clase *predicado_regla*. La primera de las sentencias garantiza que en el caso de la consulta se acceda al menos una vez a las sentencias ***if***. De no existir esta última condición, y se diese el caso de que la clase Java no tuviese términos *grounded*, la consulta nunca accedería a las sentencias ***if***, lo cual eliminaría uno o varios nodos en el árbol de búsqueda, lo cual es una condición indeseada. La instrucción ***if*** permite realizar búsquedas dentro de la clase o invocar predicados de otras clases, así podemos definir la búsqueda como ***if (predicado_regla.searchT (term, ..., term, int) = true),*** donde se especifica si un predicado de la regla puede ser resuelto por medio del método ***searchT*** —lo cual implica que la instrucción ***if*** es cierta—, y por lo tanto obtener la respuesta a la consulta solicitada.

Ahora bien, en el caso de una llamada a un predicado que es diferente al asociado a la clase actual, se requiere la instanciación de las clases Java asociadas a los predicados correspondientes. Por ejemplo, supóngase que se incluye en (1), una cuarta cláusula que corresponde a ***hermano(X,G) :- hijo(H,X), hijo(H,G)*** (2)***,*** a la cual se le aplican los criterios de traducción previamente discutidos. Entonces, el predicado ***hermano/2*** estará asociado a la clase ***Hermano_2.java***, que hace un llamado al predicado ***hijo/2***, por lo que se requiere instanciar un objeto del tipo ***Hijo_2***, de la forma ***Hijo_2 hijo_2i = new Hijo_2()***. Por cada llamada a un tipo predicado en la regla, se crea una estructura ***while*** y por cada una de las apariciones de un predicado se crea una estructura ***if.*** En el caso de la regla (2), tendríamos un ciclo *while*, debido al predicado ***hijo/2;*** y dos estructuras ***if,*** por los términos ***hijo(H,X)*** e ***hijo(H,G).***

Para entender un mejor el funcionamiento de las estructuras ***while-if***, veamos el programa con las siguientes cláusulas: ***hijo(edgar, leo), hijo(edgar, carlos), hijo(edgar, jhon), hijo(jhon, jhonjr)*** y ***abuelo(H,G) :- hijo(H,X), hijo(X,G)*** (3) y se desea consultar quien es el nieto de ***'edgar'*** —***?abuelo(edgar,C).***— En primer lugar, se crean dos archivos ***Abuelo_2.java*** e ***Hijo_2.java***. Los atributos de la clase ***Hijo_2.java*** serán ***numberAtoms=4, numberRules=0, arity=2, termA1=new Term()*** y ***termA2=new Term()***, así como cada uno de los cuatro terminos *grounded.* En este caso solo tenemos términos *grounded*, y por lo tanto no se crea ninguna estructura ***while-if***. La clase ***Abuelo_2.java***, tendrá los atributos siguientes: ***numberAtoms=0, numberRules=1, arity=2, termA1=new Term()*** y ***termA2=new Term()***. Adicionalmente, se crea el bloque donde se incluye la estructura ***while-if*** para el proceso de unificación. Una vez se realiza la consulta ***?abuelo(edgar,C)***, se lleva a cabo en primer lugar, un proceso de verificación para saber si existe un término que pueda ser unificado directamente con ***abuelo(edgar,C)***. En (3) no existe un término *grounded* asociado al predicado ***abuelo/2***, por lo cual, el programa procede a ejecutar la estructura asociada a la regla ***abuelo(H,G) :- hijo(H,X), hijo(X,G)***. Se inicializan las variables libres ***Hi***. Existen tres variables libres debido a la forma como se estructura la regla, es decir, ***abuelo(H1,H2):-hijo(H1,H3),hijo(H3,H2).*** El siguiente paso consiste en realizar el proceso de unificación entre ***abuelo(edgar,C)*** y ***abuelo(H1,H2).*** Lo anterior permite determinar si puede ser apliacada o no una regla. En el caso del ejemplo, se puede observar que la sustitución debe ser ***H1/edgar*** y ***H2/C***. Una vez verificado que se puede aplicar la regla, por medio de la instrucción ***if(unificación)*** el programa ingresa al ciclo ***while***, el cual permite "recorrer" todos los términos grounded de ***hijo/2***. Nuestra propuesta mantiene la misma estrategia de Prolog, ya que la primera cláusula que aparece en el programa, es la primera en ser probada. El primer término asociado a ***hijo(edgar,C)*** en (3) es ***hijo(edgar, leo)*** y este término se asocia a la instrucción ***if(hijo_2i.searchT(H1,H3,CBackT) == true)***, donde ***H1/edgar*** y ***H3/leo***. En la clase ***Hijo_2.java*** existe el término ***hijo(edgar,leo)***, por lo cual, la instrucción ***if*** tendrá un valor igual a ***true***. Luego, al ejecutar la siguiente instrucción ***if(hijo_2i.searchT(H3,H2,CBackT) == true)***, donde ***H3/leo*** y ***H2/C***. Al realizar la consulta a la clase ***Hijo_2.java***, el programa no encuentra términos que satisfagan la consulta, por lo que la segunda instrucción ***if*** tendrá un valor igual a ***false***. El programa ahora retorna el control al ciclo ***while***, y se intenta ahora con el siguiente término —en este caso ***hijo(edgar, carlos)*** — Se sigue el mismo procedimiento con cada termino pero no existe otro término que pueda ser unificado con ***hijo(carlos, C)***. Por último, se prueba con el término ***hijo(edgar,jhon)*** donde la segunda instrucción ***if*** será igual a ***true*** *ya* que se valida la existencia del término ***hijo(jhon,jhonjr)*** en la clase ***Hijo_2.java***. Ergo, asigna los términos ***termA1*** y ***termA2*** de la clase ***Abuelo_2.java***, con los valores que arrojan ***T1*** y ***T2***, luego de realizar la sustitución con los valores que tienen las variables ***H1*** y ***H2***. En la Figura 1.b se puede observar el grafico de búsqueda además del recorrido del programa en Java visualizando los métodos de control diseñados.

El último método se denomina ***searchRuleVar*** que permite devolver valores no *grounded* y se utiliza para garantizar un criterio de estandarización en la totalidad de la traducción. Posee la misma estructura básica del método ***searchTCV***, pero solo devuelve valores de las variables ***Hi***. En la Figura 2 muestra la estructura genérica de las clases en Java propuestas asociadas a predicados.

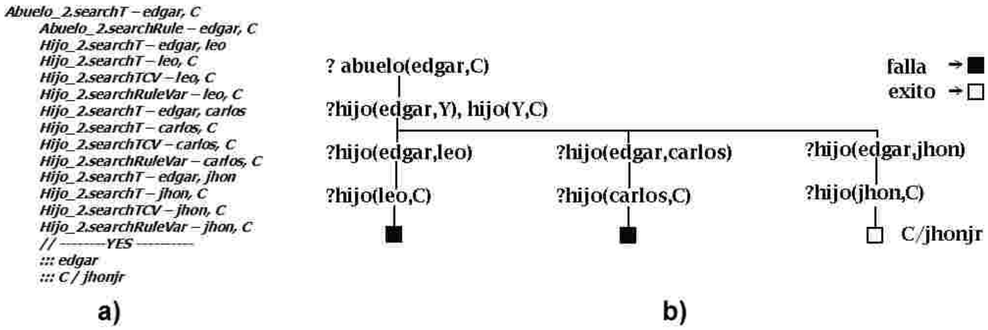

Figura 1. Métodos de control y árbol de búsqueda del programa (3).

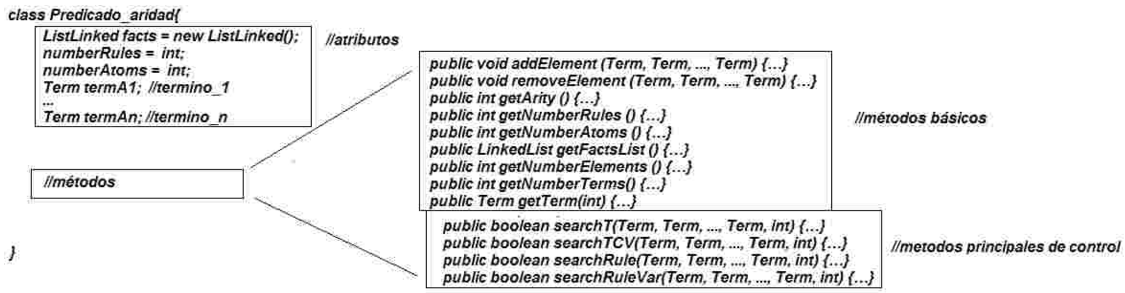

Figura 2. Esqueleto de las clases java asociadas a predicados

2.3 DEFINICIÓN DEL CONTROL EN LAS CLASES JAVA

El proceso de resolución en Prolog se lleva a cabo mediante el algoritmo de SLD (*Selection, Linear and Definite, en inglés) (Hogger, 1990)*. La estrategia de búsqueda estándar usada por un programa Prolog consiste en una estrategia primero en profundidad, secuencial, top-down con backtracking. Una estrategia similar es la que se implementó como algoritmo de control para procesar la búsqueda de soluciones en Java. Nuestra principal innovación consiste en la inclusión de un esquema de control declarativo en las clases Java asociadas a predicados de Prolog —que consiste en la implementación de los mecanismos propios de los lenguajes declarativos como la resolución en cada una de las clases de Java—.

Dado que una implementación de forma estricta e idéntica a Prolog en un lenguaje como Java daría lugar a estructuras que consumirían recursos muchos computacionales, debido en primer lugar a que no tendríamos una estructura de registros optimizada para la plataforma, y en segundo lugar seria ineficiente la existencia de una pila única o ***head*** en Java. En Prolog el programa principal siempre tiene el control, ya que todos los términos del programa están almacenados en la estructura definida por la WAM, pero en nuestra propuesta la clase que se instancia desde la clase de consulta es la que desencadena el control de la búsqueda. Una ventaja de este tipo de control es que se puede realizar de forma independiente, la modificación de cada uno de los predicados sin afectar la totalidad del programa, es decir, podemos realizar una compilación selectiva de predicados.

La inclusión de los mecanismos de control en la clase Java está determinada por el orden de las llamadas de cada uno de los métodos asociados a la búsqueda, a saber, ***searchT, searchTCV, searchRule*** y ***searchRuleVar***. En primer lugar, el inicio del ciclo de consulta verifica la existencia de términos *grounded*, por ello el primer método tratado en la consulta es ***searchT***. Si se comprueba la existencia del término, devuelve una respuesta afirmativa y se almacena la respuesta en los atributos de la clase correspondiente. En caso contrario, se verifica si hay o no reglas que tratar. Si en la clase no hay reglas que procesar, se debe verificar si hay términos con variables. En caso afirmativo se procede a llamar el método ***searchTCV***. En la Figura 3.a se muestra flujo de control cuando no se analizan reglas. En el caso de que existan reglas, se debe realizar un análisis de cada una de ellas. Se procede a realizar el llamado al método ***searchRule***. En el método ***searchRule*** residen todas las reglas asociadas al predicado sujeto a análisis. La selección de la ejecución de una regla está dado por la capacidad de unificar o no el término a consultar. Adicionalmente, se debe indicar que en el método ***searchRule***, se puede realizar un llamado al método ***searchTCV***, en caso de que existan términos con variables en la especificación de la clase. En la Figura 3.b. se resume el control cuando existen reglas en la clase.

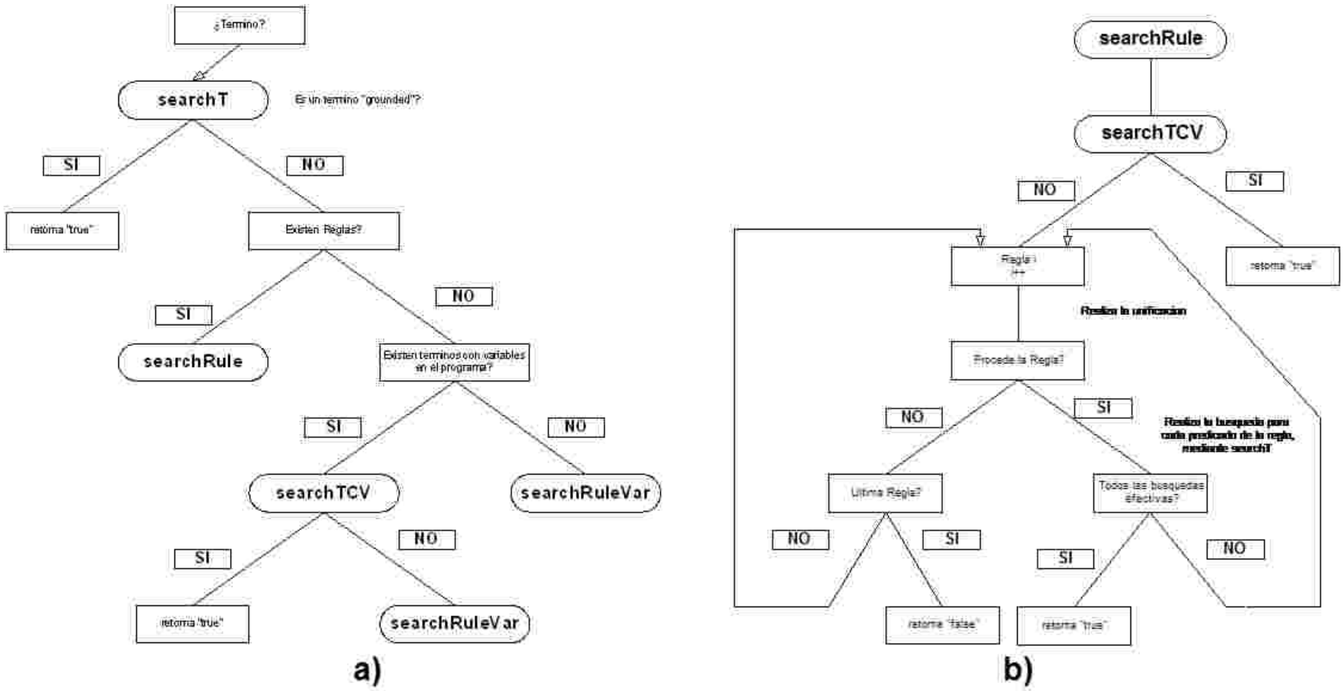

Figura 3. Control declarativo sin reglas y con reglas

Dado que la *Máquina de Virtual de Java* (JVM) controla la recursividad, es necesaria la definición de un variable de control —denominada ***CBackT***— que permita la generación automática de variables libres independientes para cada uno de los ciclos recursivos de búsqueda. Supónganse el programa Prolog mostrado en la Figura 4.a y se la realización de la consulta ***?actual(3)***. En la Figura 4.b. se muestras de las trazas a cada una de las clases de la forma: ***nombreClase.metodo - termino_1, …, termino_n, CBackT.*** Se puede observar el proceso de generación de consultas a la clase ***Tiempo_2.java*** y como es iniciada por el método ***searchT***, es decir, la generación de nuevas metas —de la forma ***?tiempo(Hi, 1)***, *en este caso*—, es lanzada únicamente por este método, luego de lo cual, el control declarativo a través del árbol de búsqueda sigue el flujo mostrado en la Figura 4.c. Todos los métodos ***search**** poseen la variable denominada ***CBackT.*** La variable ***CBackT***, va cambiando a media que descendemos en el árbol de búsqueda. Cada nueva hoja consultada en el árbol implica un incremento unitario de la variable ***CBackT***. El valor inicial de las variables está dado por ***Hi=concatenar ("Hi", CBackT),*** donde *concatenar* es un método que permite variar el valor de inicialización de las variables según el valor ***CBackT***. Si inicializamos las variables libre siempre con el mismo valor el backtracking perderá el rastro de las variables consultadas.

2.4 GENERACION DE LAS CLASES JAVA

La clase ***ScriptW*** permite generar los archivos ***.java*** con el nombre ***NombrePredicado_AridadAsociada.java,*** para cada uno de los átomos existentes en el vector ***head*** —sin multiplicidad—. Todos los predicados que no han sido definidos como un átomo o en una regla se manejan con una clase especial en Java denominada ***clase no definida***. Se evita de esta forma la generación de un error de la JVM, ya no si no pudiese encontrar la definición de la clase, se generaría una excepción ***classNotFound*** en Java y en Prolog una falla debido a que predicado no está definido. Para mantener la congruencia con Prolog, los métodos de las ***clases no definidas*** siempre "fallan" —cualquier consulta hecha a esos métodos retorna un valor ***false*** a través del método ***searchT*** —.

2.5 DEFINICIÓN DE LAS CLASES FUNCIONALES

Para la implementación se definieron cuatro clases básicas y que denominamos clases funcionales, ya que ellas cumplen la *función* de definición y manipulación de términos, en todos aquellos programas que se desarrollen bajo la presente propuesta. En las clases funcionales se incluyen la definición de operadores, proceso de resolución, definición de términos, entre otros. A continuación se discuten las clases funcionales.

2.5.1 CLASE TERM

La clase ***Term*** representa el núcleo del proceso de implementación, ya que todos los elementos propios de los lenguajes declarativos radican en la manipulación de predicados y consecuentemente en términos. Cada término está constituido por un conjunto de términos básicos, agrupados en un estructura tipo vector. El término puede ser una lista o un término individual. El término básico, es definido en la forma clásica, por ejemplo, sea el término ***hijo(hola, U),*** este se iniciaría como ***Term("hijo", 2, "hola", 0, "U", 0).***

2.5.2 CLASE BSCTERM

Son considerados los términos básicos con los cuales se estructura todo término tipo ***Term***. La clase ***BscTerm*** está estructurada como un objeto cuyos atributos son nombre, aridad y subtérminos. Se definieron los métodos para la inicialización de los términos básicos, consulta y modificación del nombre del término, aridad y subtérminos, entre otros.

2.5.3 CLASE UNIFTERM

El proceso de resolución se lleva a cabo mediante la definición de un algoritmo basado en SLD (Hogger, 1990), que se incluye como parte del método ***searchRule***, en cada una de las clases asociadas a los predicados. La clase ***UnifTerm*** tiene tres conjuntos básicos de métodos. El primero, consiste de los métodos encargados de calcular el unificador más general de un par de términos por medio de la implementación del algoritmo de Robinson y se denominan métodos ***unify***. El segundo grupo es el encargado de realizar el proceso de sustitución y se denominan métodos ***buildSus***. El último grupo corresponde a los métodos que permiten formar los vectores que llevan a cabo la manipulación del unificador más general. El formato de estos métodos es ***Vector formVector(Term, …, Term).*** La máxima cantidad de términos soportados por los métodos de ***formVector*** es de 12.

2.5.4 CLASE OPERATORS

Para agregar mayor funcionalidad a un programa que utilice una forma declarativa, se requiere la definición de las operaciones básicas, tales como las operaciones aritméticas. En algunos compiladores este tipo operaciones reciben el nombre ***Built-in*** ó constructores (JProlog, 1996). Para resolver consultas especiales se requiere que este tipo constructores sean definidos en Java. Se definió una clase donde se agrupan las operaciones aritméticas básicas de Prolog, como la suma, la asignación numérica (su nombre en inglés), entre otras. Se pretende brindar la suficiente flexibilidad para que sean incluidas nuevas funcionalidades como se discute a continuación.

En Prolog los operadores se definen de la siguiente forma, ***:-op(precedencia, tipo, nombre).*** Donde *Precedencia*, es el valor de asociado con la definición de precedencia de las operaciones, entre mayor sea el valor de precedencia más prontamente debe realizarse la operación. *Tipo*, indica el tipo de operación que será llevada a cabo, bien sea infija, postfija o prefija. *Nombre*, indica el símbolo o conjunto de símbolos que representan la operación. La estructura de los métodos de la clase, supone que las operaciones son declaradas de forma prefija y fue diseñada para permitir la sobrecarga de métodos. La clase ***Operators*** consta de 2 atributos, a saber: ***n*** *y* ***operator.*** El primero de ellos, indica la cantidad de operaciones existente en momento particular. El atributo ***operator*** es un vector de objetos tipo ***Signs***. La clase ***Signs***, posee tres atributos básicos: ***codeProlog, signHere*** y ***valuePre***. El atributo ***codeProlog*** es el campo que contiene el conjunto de símbolos asociados a la representación de una operación particular, por ejemplo, "+", "is", etc. El atributo ***signHere*** es la representación en la clase ***Signs*** de la operación particular pero con un nombre específico.

La mayoría de las operaciones básicas tienen al menos dos métodos que las definen. La primera formada por el nombre de la operación en Java (definida por el atributo ***signHere***), cuyos parámetros son términos. La respuesta de la operación es un valor de asertividad: ***true*** o ***false***. La segunda instancia, lo constituye el mismo nombre de la operación en Java, pero seguido por la letra “t”, y cuyo valor de retorno es un término. No todas las operaciones requieren esta doble definición.

Las operaciones definidas dentro de la clase ***Operators,*** hasta el momento abarcan una cantidad de 16. Las operaciones implementadas fueron: ***IS*** (asignación numérica a una variables), ***SUMA***, ***RESTA***, ***MAYOR, MENOR***, ***UNIV*** (unifica listas a funtores y viceversa), ***UNIF*** (*true* si puede unificar los términos), ***NUNIF*** (la contraparte de UNIF), ***OR*** (operación or en Prolog), ***IDENT*** (*true* si los términos son idénticos), ***NIDENT*** (la contraparte de IDENT) y ***NOT*** .

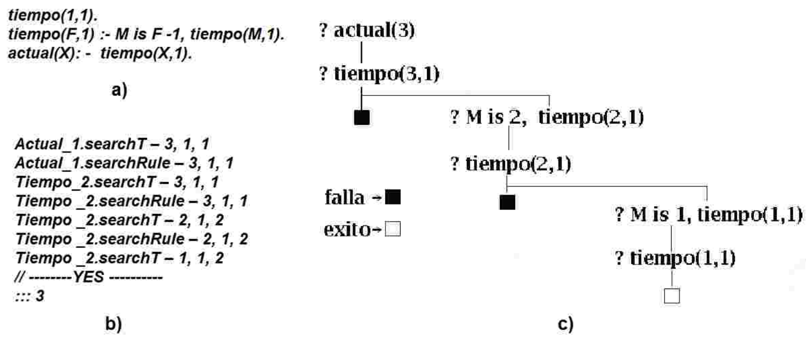

Figura 4. Árbol de búsqueda del control declarativo

2.6 GENERACION DE LA CLASE DE CONSULTA

Se desarrolló un programa que genere de forma automática la clase asociada a la pregunta. La entrada a la interfaz consiste en un predicado que representa la consulta. El proceso de generación de las consultas se realiza mediante mecanismos de *parsing* a través de una clase denominada ***ScriptQ***, que genera la clase de consulta que recibe por nombre ***Query.java***. La clase ***Query.java*** consiste en una instancia de la clase asociada al predicado que se consulta. Adicionalmente, se procede al llamado del método de búsqueda soluciones denominado ***searchT***.

3. PRUEBAS

Se llevaron a cabo tres conjuntos de banco de pruebas. El primero, verifica el funcionamiento de la resolución y la asertividad de las consultas. El segundo conjunto, permite comprobar el soporte de la multimodalidad y la negación por falla. Para el último banco de pruebas básicas, se diseño un programa para corroborar la correctitud de las operaciones aritméticas.

En primer lugar, para comprobar el funcionamiento de los mecanismos de traducción diseñados se procedió a realizar un programa típico en Prolog para la resolución de genealogías, en este caso *Gen.pro*, cuyo contenido puede verse en http://www.unet.edu.ve/ProgToJav. Para cada predicado se genera una clase Java, con el nombre del predicado seguido de un “underscore” y la aridad asociada al predicado. Se realizaron consultas sencillas, como por ejemplo, la consulta correspondiente a la pregunta ***?hijo(leo, G)***. Obteniéndose la siguiente respuesta a la consulta ***YES, leo, G/edgar***, ídem como se llevaría a cabo en un IDE de Prolog como Amzi!.

Se comprobó la multimodalidad de Prolog, es decir, que dos consultas con diferente estructura pueden generar una misma respuesta. Para comprobarlo se pueden generar consultas como ***?hermano(jhon, X)*** y ***?hermano(X, leo)*** para el programa mencionado en la Sección 3.1., además donde se puede observar la existencia el predicado ***not***, que el caso de Prolog se maneja como cualquier otro predicado para permitir la negación por falla. Se definió el predicado ***not*** como ***built-in***, es decir, como un elemento esencial para funcionamiento del programa. Todos los ***built-in*** se agrupan en la clase ***Operators***. Retomando el programa de la Sección 3.1., se utiliza el predicado ***not*** anidado con una operación de unificación “=”. Este predicado en el programa evita las respuestas incongruentes, tales como, que *“jhon”* es hermano de si mismo. Por ejemplo, si se ejecuta la consulta ***?hermano(jhon, jhon)***, la salida del programa seria ***NO***, es decir, que en todo el conjunto de reglas que componen el programa fue imposible encontrar o derivar el término ***hermano(jhon, jhon)***.

Para comprobar los mecanismos de manipulación aritmética, se diseñó un programa que calcula la sumatoria de números enteros. Sea el número entero ***i***, entonces la sumatoria seria, ***i + (i-1)+ (i-2) + ….+ 1.*** La solución para esta progresión será ***S=(i * (i+1))/2***. El programa en Prolog esta compuesto por dos cláusulas ***s(1,1)*** y ***s(F,G):- Z is F-1, s(Z,H) , G is H+F***. Si se genera una consulta del tipo ***s(10,G).*** Se obtiene la siguiente respuesta ***G/55***. El programa anterior permite demostrar el funcionamiento de las operaciones aritméticas de suma y resta.

4. RESULTADOS

(Van Roy, 1994)(Tarau and Neumerkel, 1994) disertan sobre los requerimientos de una cantidad de tiempo particular que debe ser consumido por el proceso de generación de las clases. El costo en cuanto a la generación de clases de Java, en nuestro caso es directamente proporcional a la cantidad de cláusulas, así como a la cantidad de clases existentes en el programa que se desea convertir. Las mediciones realizadas en cuanto a la generación de las clases en Java y su compilación respectiva, no supera los 75 segundos, en programas típicos de Prolog. Una vez generadas las clases Java, el tiempo requerido para la búsqueda de soluciones a partir de las clases-predicados, está determinado por la estructura de las clases y la máquina virtual sobre la que se ejecutan.

Para comparar el rendimiento de la propuesta con respecto a un IDE de Prolog. Se procedió a realizar mediciones sobre el tiempo de ejecución de los programas Prolog en una plataforma nativa como Amzi!. Convertimos luego los programas en Prolog, mediante el traductor de nuestra propuesta y realizamos la medición de tiempo de ejecución para los programas en Java y el programa nativo. Se tomó como espacio muestreal el programa descrito en la Sección 3.3., para una profundidad de búsqueda de ochocientos (800) niveles el retardo comparativo entre ambas soluciones no supera los 3.5 segundos. En el caso de un IDE de Prolog como Amzi! u otros, su código está optimizado a través de bibliotecas nativas de la plataforma donde se ejecuta y por lo tanto, es de esperar que la respuesta de un programa en Prolog nativo sea más rápida que nuestra propuesta, ya ésta es un código interpretado que se ejecuta sobre la JVM. Las pruebas se ejecutaron en una máquina Penthium MMX 200 MHz con 32 MB de memoria RAM.

5. DISCUSIÓN Y TRABAJO FUTURO

La estructura de la clase Java que permite la asociación de predicados de Prolog, presenta como innovación la inclusión de una nueva estrategia de control que denominamos "control declarativo" como parte de la clase, omitiendo de esta forma la generación de un único compilador de Prolog sobre el cual se realicen las llamadas a objetos asociados a Prolog, como en (InterProlog, 2005) (JProlog, 1996). El control declarativo está representado por los mecanismos como la resolución, la sustitución y la generación de los espacios de búsqueda. La definición del espacio de búsqueda se realiza mediante la utilización de cuatro (4) métodos específicos, que constituyen el sustento del control declarativo en las clases: ***searchT, searchRule, searchVarRule*** y ***searchTCV***.

Nuestro objetivo como el de la mayoría de las implementaciones que pretenden integrar Prolog y Java, consiste en la búsqueda de una arquitectura sencilla y práctica para implementar mecanismos propios de los lenguajes declarativos, como por ejemplo la resolución y la sustitución. Por esta razón, se puede observar que las clases Java presentan estructuras simples para implementar estos mecanismos. La simplicidad de las estructuras se puede corroborar en la implementación del control declarativo a través de los métodos ***search****. Un paso importante dado en el proyecto lo constituye la separación de las cláusulas en dos grupos: hechos y reglas. La separación permite al programador —una vez haya generado el código de las clases— pueda incluir o excluir dinámicamente un hecho en la clase generada.

La modularidad de las clases funcionales permite a futuras versiones intentar implementar conceptos como CLP (Constraint Logic Programming). La sustentación de tal afirmación, radica en que podría modificarse las clases ***Term, BscTerm, UnifTerm*** y ***Operators***, para incluir operaciones (los axiomas de precondición y poscondición) vinculadas con el espacio de búsqueda propio de los programas CLP, como se menciona en (Mesnard et al., 1996) y la estructuración de los operadores en la clase ***Operators***, brinda esta posibilidad.

La idea de mantener la estructura de las variables libres utilizadas en Prolog y no asociarlas directamente al tipo de dato —una valiosa sugerencia del Profesor Roussel, co-realizador del compilador de Prolog—, permite que el proceso de resolución se puede llevar a cabo en un menor tiempo de ejecución, ya que no requiere la diferenciación directa del tipo de dato. Por ejemplo, en el caso de los caracteres numéricos solo se reconocen como tal, al momento de llevar a cabo una operación aritmética o lógica. La asociación indirecta del tipo de dato, conllevó a la consecución de un mecanismo de optimización de la búsqueda de soluciones en el espacio de búsqueda correspondiente.

Se propone la incorporación de nuevas operaciones, realizando modificaciones o agregando nuevos métodos en la clase ***Operators***. Se requiere además, el establecimiento de estrategias para la medición del tiempo consumido por la propuesta, y compararlo con programas ya existentes y con suficiente soporte, como por ejemplo SWI, Jinni e Interprolog. A pesar que en los resultados se plasmó una medición de tiempo de ejecución de la propuesta con respecto a Amzi! —IDE de Prolog—, se requieren pruebas más exhaustivas para compararlo con otras programas. Se debe planificar un banco de pruebas más amplio en las cuales se pueda medir y analizar el rendimiento de la propuesta versus otras traducciones, como por ejemplo jProlog, Interprolog, entre otras. El conjunto de pruebas, debe incluir además diferentes plataformas de hardware así como diferentes sistemas operativos.

Referencias

Aït-Kaci, Hassan. (1999). “Warren's Abstract Machine A Tutorial Reconstruction”. MIT PRESS. 1999.

Calejo, Miguel. (2004). “InterProlog: Towards a Declarative Embedding of Logic Programming in Java”. *JELIA*, pp 714—717.

Van Roy, Peter. (1994). “Issues In Implementing Constraint Logic Languagues”. *DEC Paris Research Lab*.

Tarau, P. (1994). "Low-level Issues in Implementing a High-Performance Continuation Passing Binary Prolog Engine". *JFPLC*, pp 287— .

Morozov, Alexey A. (1999). “Actor Prolog: an object-oriented language with the classical declarative semantics”. Semantics, Proc. IDL’99 Workshop, pp 39—53.

Hogger, Christopher. (1990). “Essentials of Logic Programming”. Clarendon Press.

Kowalski, Robert. (2006). “Computational Logic in an Object-Oriented World”. Reasoning, Action and Interaction in AI Theories and Systems, pp 59-82. Springer

Mesnard, Frédéric, Hoarau, Sébastien and Maillard, Alexandra. (1996). “CLP(X) for Proving Program Properties”. Frontiers of Combining Systems (FroCos), pp 321—338.

Tarau, Paul and Neumerkel, Ulrich. (1994). “A Novel Term Compression Scheme and Data Representation in the BinWAM”, PLILP, pp 73—87.

Diaz, Daniel. (1995). “WAMCC: Compiling Prolog to C”, 12th International Conference on Logic Programming}, pp 317—331, MIT Press.

InterProlog. (2005). Declarativa - Serviços de Informática, Lda. Parque de Ciência e Tecnologia da Universidade do Porto. http://www.declarativa.com/InterProlog/default.htm

JProlog. (1996). Katholieke Universiteit Leuven and the Universite de Moncton. http://www.cs.kuleuven.be/~bmd/PrologInJava/

Jinni. (2005). BinNet Corporation. http://www.binnetcorp.com/Jinni/

Prolog Café. (1997). Kobe University. Japan. http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/

JPL (2004). SWI Prolog. http://www.swi-prolog.org/packages/jpl/objectives.html

Authorization and Disclaimer

Authors authorize LACCEI to publish the paper in the conference proceedings. Neither LACCEI nor the editors are responsible either for the content or for the implications of what is expressed in the paper.