Diseño e implementación de un generador de evaluadores concurrentes para gramáticas de atributos NC(1)*

Marcelo Arroyo, Jorge Aguirre, Nicolás Florio
{marroyo,jaguirre,nflorio}@dc.exa.unrc.edu.ar

Dpto. de Computación - FCEFQyN- Univ. Nac. de Río Cuarto

Resumen

En este trabajo se presenta el diseño e implementación de *NCEval*: un generador estático de evaluadores concurrentes de gramáticas de atributos para la familia *NC(1)*. Los evaluadores generados son del tipo multiplan multivisita y utilizan información computada estáticamente para la selección de planes de evaluación y *segmentos* o conjuntos de atributos independientes en el árbol sintáctico de entrada. Los segmentos o regiones se evalúan concurrentemente y su independencia hace que no se requiera ningún mecanismo de sincronización ni comunicación entre procesos. La clasificación de las gramáticas de atributos *NC* (non-circular) - propuesta por Wuu Yang en [Yan991]- cubre el conjunto de las gramáticas de atributos bien definidas en una jerarquía denominada *NC(n)* (non-circular with n lookahead symbols). Estas gramáticas pueden ser evaluadas por métodos estáticos multivisita. La familia *NC(0)* se corresponde con la familia *ANCAG* (Absolutely Non-Circular Attribute Grammars), la familia NC(1) contiene a la familia *ANCAG* (que hasta ahora se consideraba como la mayor familia que permitía generar evaluadores estáticos eficientemente) y la familia $NC(\infty)$ se corresponde con las *WDAG* (Well Defined Attribute Grammars). El particionado de las instancias de atributos en un árbol atribuido en regiones disjuntas se basa en el algoritmo propuesto en [Yan992], el cual produce la partición más fina posible de los atributos de la gramática en base a sus dependencias. El generador de evaluadores produce estáticamente los planes de evaluación proyectados en función de las posibles particiones. El diseño de *NCEval* y el modelo de evaluación es orientado a objetos y ha sido implementado en Java.

Palabras clave: Gramáticas de Atributos, Compiladores, Concurrencia, Lenguajes, Objetos.

*El presente trabajo se desarrolló en el marco de proyectos subsidiados por la Agencia Córdoba Ciencia y la Secretaría de Ciencia y Técnica de la UNRC.

1. Introducción

Desde que D. Knuth en 1968 [Knu68] introdujo la idea de las Gramáticas de Atributos (GA), éstas han sido de gran interés en las ciencias de la computación y en ingeniería de software, ya que permiten describir la semántica de los lenguajes de programación y se han utilizado ampliamente para el desarrollo de herramientas de generación de procesadores de lenguajes basados en especificaciones, conocidos generalmente como *compiladores-compiladores* o *sistemas de generación de compiladores o traductores.*

Entre las principales ventajas de este formalismo se pueden mencionar su modularidad, y su declarativid. Una GA es modular porque cada regla o producción es independiente de las demás y es declarativa porque sus atributos no tienen un orden de evaluación explícito. Las implementaciones deben determinar el orden de evaluación en base a las dependencias implícitas en las declaraciones de las ecuaciones (atribución) asociadas a cada regla o producción.

El formalismo ha evolucionado gradualmente desde un concepto teórico a un paradigma computacional avanzado. En principio las investigaciones pusieron especial énfasis en su caracterización y en el desarrollo de los algoritmos básicos para su implementación. Posteriormente se fueron definiendo subclases como las GA *s-atribuidas*, *l-atribuidas*, *absolutamente no circulares*, *AG ordenadas*, etc. Cada clase está asociada con estrategias de evaluación eficientes. Posteriormente se han propuesto nuevas clases (como la jerarquía *NC(n)*, que se describen en la sección 2) y variantes del formalismo en cuanto a sus lenguajes de especificación para su adaptación a otras áreas de aplicación como las *GA Condicionales*, las *GA orientadas a objetos*, las *GA de alto orden*, etc. Otras líneas de investigación se han concentrado en la propuesta de modelos de evaluación concurrente de gramáticas de atributos.

Se han desarrollado una gran variedad de sistemas basados en gramáticas de atributos, como *OLGA*, *LIDO*, *FCN-2* y *YACC* (ésta última probablemente ha sido la herramienta más utilizada soportando *GA l-atribuidas*).

Este trabajo describe una herramienta orientada a objetos para la generación de evaluadores concurrentes para la familia de gramáticas de atributos *NC(1)*. Los procesos de evaluación se ejecutan concurrentemente y cada uno de ellos evalúa un segmento (conjunto de atributos) del árbol sintáctico atribuido de entrada. Los segmentos se seleccionan en tiempo de ejecución en base a información computada estáticamente y los atributos en un segmento son independientes de los atributos en otro segmento, por lo tanto los procesos evaluadores no requieren ninguna sincronización ni comunicación. En las primeras secciones se presenta una introducción a las GA *NC(n)* y a los métodos de evaluación. Posteriormente se presenta el esquema del funcionamiento general del sistema, su diseño y su modelo de implementación.

2. Gramáticas de Atributos

Una gramática de atributos es una tupla $GA =< G, A, R >$ donde G es una *gramática libre de contexto*, A es un conjunto finito de *atributos* y R es un conjunto finito de *reglas semánticas*. $G =< N, T, P, S >$ donde N es un conjunto de *símbolos no terminales*, T es el conjunto de *símbolos terminales*, $V = N \bigcup T$, P es un conjunto de pares de la forma $X \rightarrow \alpha$ denominadas *producciones*, donde $X \in V$ y $\alpha \in V^{*1}$, $S \in N$ es el *símbolo se comienzo*.

En una GA se asocia un conjunto de atributos $A(X) = H(X) \bigcup S(X)$ con $(H(X) \bigcap S(X) = \emptyset)$, con cada símbolo $X \in V$. El conjunto *H(X)* es el conjunto de atributos heredados de X y *S(X)* es el conjunto de atributos sintetizados de X, $H(S) = \emptyset$ y $S(X) = \emptyset$, $\forall X \in T$.

Una producción $p \in P$, de la forma $X_0 \rightarrow \alpha_0 X_1 \ldots \alpha_{n-1} X_n \alpha_n$, $(n \geq 0)$, tiene una ocurrencia $X_i.a$ si $a \in A(X_i)$, $0 \leq i \leq n$, $X \in N, \alpha \in T^*$. Un conjunto de *reglas semánticas* R_p de la forma $X_i.a = f(y_1 \ldots y_k)$ se asocia con una producción p con las siguientes restricciones:

1. $i = 0$ y $a \in S(X_i)$, o $1 \leq i \leq n$ y $a \in H(X_i)$.
2. cada y_j, $1 \leq i \leq k$, es un atributo que ocurre en p.
3. f es una función (denominada *función semántica*) que mapea valores de $y_1, \ldots, y_k$ al valor de $X_i.a$.

En una regla de la forma $X_i.a = f(y_1 \ldots y_k)$, se dice que la ocurrencia $X_i.a$ depende de cada ocurrencia y_i, $1 \leq i \leq k$. El conjunto de reglas semánticas $R = \bigcup_p R_p$.

Un árbol de derivación para una cadena (o programa) ω, generado por $G =< N, T, P, S >$, es un árbol donde:

1. Cada nodo tiene rótulo X o ε.
2. El rótulo de la raíz es S.
3. Si un nodo está rotulado por X tiene sus hijos rotulados $X_1, \ldots, X_n$ (de izquierda a derecha), entonces $X \rightarrow X_1 \ldots X_n$ es una producción en P.
4. Los rótulos de las hojas, concatenados desde izquierda a derecha forman ω.

Un *árbol atribuido* para un programa ω es un árbol de derivación donde cada nodo n, rotulado X, contiene instancias que corresponden a los atributos de X. Por cada atributo $a \in A(X)$ la instancia correspondiente contenida en n se denota $n.a$.

[1] La cadena vacía se denota ε.

3. Evaluación de gramáticas de atributos

La evaluación de atributos de un árbol atribuido T es un proceso que computa los valores de las instancias de atributos de T de acuerdo con las reglas semánticas R, esto es, el valor la instancia n.a correspondiente a un símbolo X se computa usando la regla $X.a = f(y_1, \ldots, y_k) \in R_p$[2] y

1. $a \in H(X)$ y p es la producción $Y \rightarrow \ldots X \ldots$ aplicada en la generación de n en T, o
2. $a \in S(X)$ y p es la producción $X \rightarrow \ldots$ aplicada en la generación de n en T.

La regla semántica se ejecuta invocando a la función f con los valores de las instancias de $y_1, \ldots, y_k$ como argumentos y asignado el valor retornado por f a la instancia $n.a$[3]. Un proceso de evaluación debe ser consistente con las dependencias entre las instancias de los atributos, es decir, no se puede evaluar una regla que define a una instancia si no están definidos los valores de las instancias de las que depende.

El *significado de un programa* ω es el conjunto de valores de las instancias de los atributos del nodo raíz del árbol atribuido T correspondiente a ω.

Una *GA* G es *no circular* o *bien definida* si es posible encontrar un orden de evaluación para todas las instancias de cualquier árbol atribuido T, generado a partir de G. Se puede establecer si una GA es bien definida si el grafo de dependencias entre atributos para T, inducido por R, es acíclico para cualquier T generado a partir de G.

A continuación se describen los principales métodos o estrategias utilizadas para implementar evaluadores de GA.

3.1. Evaluadores estáticos

El problema de decisión si una AG es bien definida es intrínsecamente exponencial[Kaz75], lo que ha llevado a definir subclases de GA que imponen restricciones en las dependencias entre los atributos y permiten bajar la complejidad de los algoritmos de evaluación.

Algunas clases como las GA *s-atribuidas* (sólo contienen atributos sintetizados) y las *l-atribuidas,* permiten implementar evaluadores en una sola pasada por el árbol atribuido T. Otras clases utilizadas ampliamente en sistemas basados en GA son las *ordered AG* y las *absolutamente no circulares (ANCAG)*, las cuales permiten que el costo de la decisión sea lineal y permiten implementar evaluadores multivisita[4].

[2] Se dice que la regla define a la instancia $X.a$.

[3] Esto implica que los valore de las instancias $y_1, \ldots, y_k$ deben estar disponibles para evaluar $n.a$.

[4] El evaluador puede visitar varias veces a cada nodo de T.

Para éstas clases de AG se pueden generar estáticamente planes de evaluación para cada producción basados en el orden parcial[5] impuesto por las dependencias.

Kastens propuso un método de evaluación para las *ordered AG*[Kas80] basado en secuencias de visita[Kas91] que ese puede aplicar a las clases mencionadas arriba. Las secuencias de visita son secuencias de operaciones *visit(n,i)*, *eval(r)* y *leave()* y se pueden generar a partir de los ordenes parciales (planes) computados. La operación *visit(n,i)* realiza la *i-ésima* visita al nodo hijo *n*. La operación *eval(r)* ejecuta la regla semántica *r*. La operación *leave()* retorna al nodo padre.

En 1999 Wuu Yang propuso una nueva clasificación basada en el análisis de todos posibles contextos inferiores posibles de cada símbolo no terminal. La clasificación define una jeraquía de clases denominadas *NC(0)*, *NC(1)*, ..., *NC(∞)*. La clase *NC(0)* se corresponde con la $ANCAG$ y la clase *NC(1)* contiene a las $ANCAG$.

DEFINICIÓN: Sea q una producción de la forma $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$ y sea p_i una producción cuya parte izquierda sea X_i, $1 \leq i \leq k$. El *Grafo de Dependencias Aumentado* de q con subárboles derivados por las producciones $p_1, p_2, \ldots, p_k$, se define como:

$$ADP(q|p_1 p_2 \ldots p_k) = DP(q) \cup DCG_{X_1}(p_1) \ldots DCG_{X_k}(p_k)$$

donde

$$DCG_X(q) = \cup\{\Pi(ADP(q|p_1 p_2 \ldots p_k), A(X))$$

y $DP(q)$ es el *grafo de dependencias de la producción q*.

En [Yan991] se presenta un algorimo para computar $DCG_X(q)$ basado en $DP(p)$ y *el grafo de dependencias transitivas del símbolo* X, *Down(X)*[6].

DEFINICIÓN: Una gramática de atributos G es *NC(1)* si y sólo si, para toda producción q de G de la forma $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$, cada grafo en $SADP(q) = \{ADP(q|p_1 p_2 \ldots p_k) | p_i = X_i \rightarrow \ldots, 1 \leq i \leq k\}$ es acíclico.

Para cada producción q de G y cada posible contexto inferior $p_1 p_2 \ldots p_k$ de q, el orden (o plan) de evaluación de los atributos que ocurren en q puede generarse a partir de un orden topológico de $ADP(q|p_1 p_2 \ldots p_k)$. Un evaluador sólo tendría que seleccionar el plan correspondiente en cada nodo del árbol atribuido dado como entrada, y evaluar las instancias en el orden determinado por el plan, aplicando la regla semántica correspondiente. *AGEval*

[5] Generado por un orden topológico del grafo de dependencias de la producción.

[6] El grafo *Down(X)* denota las dependencias (transitivas) entre los atributos de X que puedan ocurrir en algún subárbol derivado a partir de X.

utiliza un algoritmo de evaluación basado en la generación de secuencias de visita para cada plan.

En la figura 1 se muestra una GA *NC(1)* y en la figura 2 se muestran dos de sus instancias de árboles atribuidos (las flechas indican las dependencias entre los atributos).

p_0: $S \rightarrow X$
attribution
$S.j{:=}X.b$
$S.k{:=}X.d$
$X.a{:=}0$
$X.c{:=}1$
end
p_1: $X \rightarrow 1Y$
attribution
$Y.e{:=}X.a$
$X.b{:=}Y.f$
$Y.g{:=}X.c$
$X.d{:=}Y.h$
end
p_2: $X \rightarrow 2Y$
attribution
$Y.e{:=}X.a$
$X.b{:=}Y.f$
$Y.g{:=}X.c$
$X.d{:=}plus(Y.h,Y.f)$
end
p_3: $Y \rightarrow 3$
attribution
$Y.f{:=}Y.e$
$Y.h{:=}Y.g$
end
p_4: $Y \rightarrow 4$
attribution
$Y.h{:=}Y.e$
$Y.f{:=}Y.g$
end

Figura 1: GA1: Un ejemplo de una GA NC(1)

3.2. Evaluadores dinámicos

Otra estrategia de evaluación es realizar el chequeo de circularidad generando el grafo de dependencias (GD), a partir del árbol atribuido T dado como entrada, en tiempo de ejecución. Si el grafo es acíclico se pueden evaluar los atributos según un orden topológico. En caso contrario el evaluador notificará que la gramática de atributos es circular.

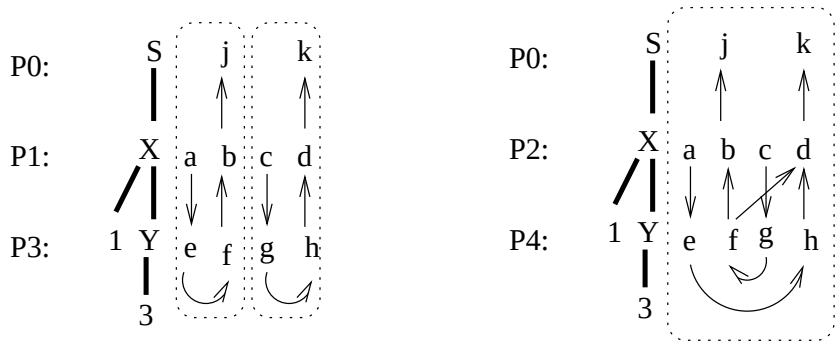

Figura 2: Dos árboles derivados de la GA de la fig. 1

La ventaja de este enfoque es que es posible evaluar algunas instancias de árboles atribuidos en donde las instancias de atributos no contengan dependencias cíclicas aún cuando la gramática de atributos sea circular.

El mayor inconveniente es que la creación y mantención del grafo en memoria puede consumir mucho tiempo y espacio.

3.3. Evaluación concurrente

Una técnica simple para evaluar concurrentemente los atributos de un árbol atribuido, como la utilizada en [Ag98], es que cada atributo es un proceso que computa su valor cuando las instancias de los atributos de los que depende hayan sido evaluados. Cada objeto (instancia) contiene un flag que indica si ya ha sido evaluado o no. El evaluador dispara cada instancia de atributos como un proceso[7]. Una instancia de un atributo debe bloquearse hasta que todas las instancias de las que depende se hayan evaluado. Esta estrategia se conoce como *evaluación bajo demanda*. El problema de este enfoque es que se pueden disparar un gran número de procesos y un gran número de ellos estarán bloqueados esperando por valores de otras instancias dependientes.

Otras técnicas de evaluación concurrente más eficientes se basan en estrategias de división o particionado de instancias de atributos (splitter strategy). Los principales objetivos de una estrategia de particionado son los siguientes:

- permitir que instancias de atributos independientes sean evaluadas concurrentemente.
- minimizar la comunicación y/o sincronización entre los procesos evaluadores.

Kuiper [kui89] propone tres métodos estáticos para realizar el particionado.

[7] En [Ag98] se implementan como Java threads.

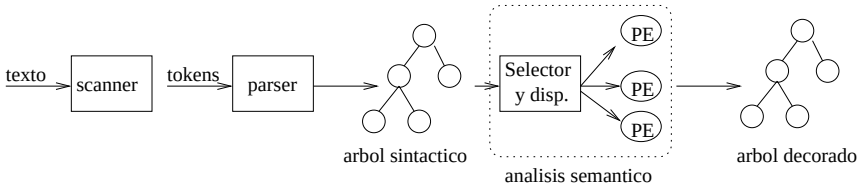

Figura 3: Estructura conceptual de un compilador concurrente

- basados en el árbol atribuido: divide el árbol en partes conexas (denominadas regiones). Cada región (subárbol) generalmente se define en base a un nodo raíz (seleccionado por producciones o por símbolos no terminales). Un proceso evaluará las instancias de cada región.
- por atributos: todos las instancias de un atributo se evalúan por el mismo proceso.
- combinaciones de los métodos anteriores.

NCEval utiliza una *estrategia basada en las dependencias* entre atributos como la propuesta por Yang en [Yan992]. La estrategia consiste en detectar estáticamente las dependencias entre los atributos y generar (estáticamente) todas las posibles particiones de atributos que puedan ocurrir en cada producción, según su contexto (superior e inferior). El evaluador, dado un árbol atribuido T, y basándose en dicha información, selecciona una partición de las instancias de atributos en cada nodo de T. Las combinaciones de las particiones seleccionadas en cada nodo forman *regiones* independientes en T. El evaluador asigna un proceso de evaluación por cada región. Los procesos evaluadores, al ser independientes, no requieren ninguna comunicación ni sincronización. En [Arr00] se dan detalles del algoritmo de particionado y de la combinación con un evaluador basado en secuencias de visita.

4. Esquema general del sistema

En la figura 3 se muestra el esquema de un compilador concurrente que utilizaría un analizador semántico generado por *NCEval*.

4.1. Etapa de generación del evaluador

NCEval toma como entrada una especificación de una gramática de atributos y luego realiza los siguientes pasos:

1. Generar los planes de evaluación para cada producción en base al cómputo de los grafos $LDP(p)$, para cada producción p.
2. Computar π_q: las particiones viables de las ocurrencias de atributos de cada producción q.

3. Generar secuencias de visita para cada producción a partir de la proyección de las particiones sobre los planes correspondientes.

Para la gramática de atributos de la figura 1 NCEval genera los siguientes planes de evaluación para cada producción:

P0: {[a,c,b,d,j,k]}
P1: {[a,e,c,f,g,h,b,d],[c,a,g,e,f,h,b,d]}
P2: {[a,c,e,g,f,h,b,d]}
P3: {[e,f,g,h]}
P4: {[e,g,f,h]}

Las particiones generadas para cada producción:

P0:{{[k,d,c],[b,a,j]},{[k,c,d,b,j,a]}}
P1:{{[c,d,h,g],[b,e,f,b]},{[c,d,f,g],[d,h,e,a]},{[d,e,c,g,h,f,b,a]}}
P2:{{[e,a,c,d,g,h,b,f]},{[f,a,d,e,c,h,b,c]}}
P3:{{[g,f],[h,e]},{[e,h,f,h]}}
P4:{{[h,g],[e,f]},{[h,f,e,g]}}

Los planes proyectados en base a las particiones que correspondan a los mismos contextos, resultan:

P0:{{[c,d,k],[a,b,j]},{[a,c,b,d,j,k]}}
P1:{{[c,g,h,d],[a,e,f,b]},{[c,g,f,b],[a,e,h,d]},{[a,c,e,g,f,h,b,d]}}
P2:{{[a,e,c,f,g,h,b,d]},{[c,a,g,e,f,h,b,d]}}
P3:{{[g,f],[e,h]},{[e,g,f,h]}}
P4:{{[g,h],[e,f]},{[e,g,f,h]}}

En la figura 2 se muestran las regiones inducidas por las particiones seleccionadas en cada nodo por *NCEval*. Cada región puede ser evaluada independientemente.[8]

4.2. El evaluador

En tiempo de ejecución, el evaluador recibe como entrada un árbol atribuido T y realiza los siguientes pasos:

1. Seleccionar las particiones de instancias de atributos y el plan de evaluación para cada nodo de T.

2. Se dispara un proceso de evaluación por cada región del árbol, inducida por las particiones.

5. Modelo de implementación

En la figura 4 se muestra el esquema de funcionamiento de un evaluador generado por NCEval. El evaluador toma como entrada un árbol atribuido. Con la información generada estáticamente por NCEval, el evaluador selecciona las particiones y los planes proyectados[9] para cada producción y para cada región del árbol - inducida por la partición seleccionada para los

[8]En el árbol de la derecha queda determinada una única región.

[9]En realidad ya selecciona la proyección correspondiente de la secuencia de visitas computada estáticamente.

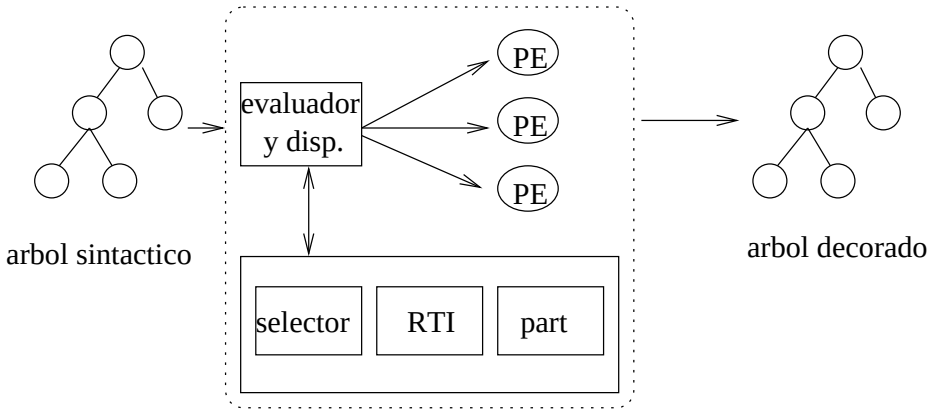

Figura 4: Esquema de NCEval en tiempo de ejecución

atributos del nodo raíz - dispara un proceso responsable de la evaluación de dicha región. Cada nodo del árbol tiene asociada la secuencia de visitas de cada partición seleccionada. Cada proceso va evaluando las instancias de los atributos en cada nodo ejecutando las operaciones de la secuencia de visita que le corresponde. Cada secuencia de visita es un objeto que además de contener la secuencia de operaciones correspondientes tiene un atributo de control que indica la próxima operación a computar.

6. Diseño

La estrategia de diseño se basa en los conceptos de reutilización y en la generación de especializaciones de clases que representan un evaluador abstracto.

El diseño se divide en dos partes: el generador de evaluadores y el evaluador a ser generado. El generador de evaluadores se implementó utilizando las herramientas JLex y JavaCup. El generador realiza el análisis sintáctico de la especificación de una grámatica de atributos y construye una representación interna de dicha grámatica (representado por la clase Grammar). Luego se generan los planes de evaluación, las particiones viables y funciones de selección de planes y selecciones según el contexto (contenidas en las clases Partition y RTI). A partir de ésta información se generan los conjuntos de planes de visita correspondientes para cada producción y cada partición. Finalmente se genera el evaluador concreto (subclase de AbstractEval) que representa un evaluador específico para la GA dada. El evaluador concreto se apoya en una instancia de la clase *Selector*, la cual es responsable de la selección de las particiones asociadas a cada producción según su contexto en tiempo de ejecución.

Para aislar el comportamiento general de los aspectos variantes se utilizaron los patrones de diseño *strategy* y *facade*. La clase *AbstractEvaluator*

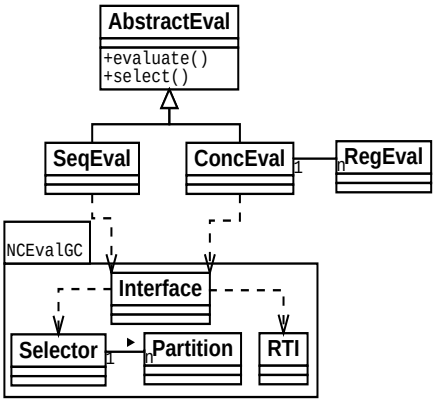

Figura 5: Diagrama de clases del evaluador

abstrae el comportamiento general de las posibles estrategias de evaluación soportadas: secuencial o concurrente. Para lograr esta abstracción se utilizó el patrón *strategy*. Las subclases de *AbstractEvaluator* generadas en cada caso implementan los algoritmos de evaluación mencionados.

La interface de los procesos evaluadores se diseñó aplicando el patrón *facade,* abstrayendo la interface con el grupo de clases que mantiene la información necesaria para el método de evaluación correpondiente.

El usuario puede especificar (mediante los flags -c o -s) si desea generar un evaluador NC(1) concurrente o secuencial.

7. Conclusiones y trabajo futuro

NCEval se implementó en Java y puede formar parte de un back-end o evaluador semántico de una herramienta de procesamiento de lenguajes o compilador-compilador. Si bien es hasta ahora un prototipo, se han realizado las pruebas sufucientes como para analizar la calidad (y legibilidad) del código generado y se encuentran en desarrollo varias optimizaciones. Entre las extensiones planificadas se encuentra el desarrollo de una interface para tomar como entrada árboles atribuidos en formato xml. Otra extensión en desarrollo es la implementación de un evaluador distribuido para utilizarse y experimentar en clusters.

Referencias

[Knu68] D. Knuth. 1968. *Semantics of context free languages*. Math Systems Theory 2, June 2.Pag: 127-145.

[Yan991] Wuu Yang. 1999. *A Classification of Non-Circular Attribute Grammars Based on the Look-Ahead Behavior*. Internal report, National Chiao-Tung University.

[Yan992] Wuu Yang. 1999. *A Finest Partitioning Algorithm for Attribute Grammars*. Second Workshop on Attribute Grammars and their Applications - WAGA 99.

[Yan98] Wuu Yang. 1998. *Multi-Plan Attribute Grammars*. Internal report, National Chiao-Tung University.

[kui89] M. F. Kuiper. 1989. Parallel Attribute Evaluation. PhD thesis, Utrech University.

[Sar93] J. Saraiva, P. Henriques. 1993. *Concurrent Attribute Evaluation*. Internal Report, Universidad do Minho, Braga, Portugal.

[Jou91] Martin Jourdan. 1991. *Survey of Parallel Attribute Evaluation Methods*. In Attribute Grammars, Applications and Systems. Prague, pp. 234-255, vol. 545 of Lecture Notes in Comp. Science, Springer Verlag.

[Kas80] Uwe Kastens. 1980. *Ordered Attribute Grammars*. Acta Informatica 13, pp. 229-256.

[Kas91] Uwe Kastens. 1991. *Implementation of visit-oriented attribute evaluators*. Lecture Notes in Computer Science 545, pp. 114-139. Springer Verlag.

[Kaz75] Kazayeri, Ogden, Rounds. 1975. *The intrinsecally exponential complexity of the circularity problems for attribute grammars*. Comm. ACM 18.

[Gam95] Gamma, Helm, Johnson, Vlissides. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Ed. Adsison Wesley. ISBN: 0-201-63361-2.

[Ag98] J. Aguirre, V. Grinspan, M. Arroyo. 1998. *Diseño e Implementación de un entorno de Ejecución Concurrente y Orientado a Objetos, generador por un Compilador de Compiladores*. Anales ASOO de 98 JAIIO, pp. 187-195. Buenos Aires, Argentina.

[Arr00] M. Arroyo, J. Aguirre, N. Florio. 2000. *Un Generador de Evaluadores de Gramáticas de Atributos NC(1) de Máximo Parelelismo sin Sincronización entre Procesos*. Anales de CACIC 2000, pp. 523-536, Ushuaia, Argentina.

[Gos96] J. Gosling , Joy, Steele. 1996. *The Java Language Specification*. Addison Wesley.