ARTICULO

Intérprete de reglas para aumentar la eficiencia de un programa lógico

Antonio Menchén Peñuela

Dpto. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Facultad de Informática

E-mail: atenea.lsi.us.es

Resumen

En el presente trabajo se propone un intérprete de reglas que elimina ciertas ineficiencias que aparecen cuando el programador escribe sus programas lógicos con toda naturalidad.

Palabras clave: programación lógica, compilador de reglas, intérprete de reglas, procesamiento del lenguaje natural.

Planteamiento del problema

Un programa lógico lo constituyen una serie de cláusulas que se suelen clasificar en tres grupos: **hechos, reglas** y **cláusulas negativas** u **objetivos**. Mientras que en otros estilos de programación como el funcional, podemos partir de una distinción clara entre **datos** y **programas**, en programación lógica no hay ninguna diferencia, ya que son casos particulares de cláusulas. Así es muy corriente que un dato en el estilo funcional sea tratado como un programa en el lógico. Un ejemplo de ello lo podemos encontrar en un **compilador**; si nos centramos tan solo en el **analizador sintáctico**, las reglas de reescritura pueden ser representadas en un lenguaje funcional como una estructura del tipo de las listas, que serán tratadas posteriormente por el algoritmo de análisis (**parser**). En lenguaje lógico, sin embargo, se escriben en forma de cláusulas (las más conocidas son las gramáticas de cláusulas definidas **DCGs**), por lo que constituyen un programa lógico.

A pesar de parecer más cómodo el diseño de analizadores en la programación lógica que en la funcional, resulta ser el método más inflexible de los dos. Así, en el estilo funcional podemos plantearnos distintos analizadores que tengan en cuenta las distintas ineficiencias que puedan darse debido a las reglas de reescritura. Por ejemplo si hay dos reglas que consisten en:

A --> B C
A --> B D

puede observarse que al reescribir **A** si hay una secuencia de palabras que estamos analizando que se reescribe como **B**, y es imposible reescribir como **C** alguna

secuencia de las restantes, un analizador descendente volvería a reescribir **B** para luego intentar la reescritura de **D**. Esta ineficiencia se evita con los llamados analizadores basados en la estructura **chart**, que es un registro de los símbolos reescritos y de las reglas aplicadas parcial o totalmente.

Por el contrario, en un lenguaje lógico la gramática anterior es un programa lógico con dos cláusulas:

a(...) <- b(...) & c(...)

a(...) <- b(...) & d(...)

donde es evidente la ineficiencia debida a la aparición del literal **b(...)** al comienzo de las cláusulas para **a(...)**. Esto puede convertirse en un serio problema cuando tratamos con gramáticas extensas en las que los símbolos son reescritos una y otra vez.

En lo que sigue me propongo eliminar estas ineficiencias de forma automática sin perjuicio de la "naturalidad" en la expresividad del diseñador de la gramática. Para ello voy a hacer primeramente una pequeña introducción a las técnicas de los llamados **compiladores de reglas** e **intérpretes de reglas**, después se tratará la forma de utilizarlas para conseguir nuestro objetivo, y por último veremos la aplicabilidad del método desarrollado a un campo de interés como es el del procesamiento del Lenguaje Natural.

Compiladores e intérpretes de reglas

Los primeros son programas lógicos que manipulan reglas escritas de forma muy variada, para obtener un programa lógico. Los intérpretes de reglas, sin embargo, ejecutan directamente las reglas sin convertirlas previamente en programas lógicos.

Un ejemplo muy sencillo es el siguiente:

compilador de reglas

translate_rule((H <- verdadero),H).

intérprete

interpreta(verdadero).

interpreta(Regla) <-
clause((Regla <- Cuerpo)) &
interpreta(Cuerpo).

en el primero se incorpora en memoria el hecho **H**.

en el segundo al llamar al intérprete de reglas, se ejecuta el hecho **H**. (Ver [Shieber,87] para más información sobre los intérpretes de reglas, y [Abramson,89] para los compiladores de reglas).

Cuando se trata de reglas de reescritura, el compilador de reglas tiene como primera cláusula:

translate_rule((Simbolo –> Reescritura), (Regla <- Cuerpo)) <-
t_sim(Simbolo,S,SR,Regla) &
t_ree(Reescritura,S,SR,Cuerpo).

donde **t_sim** y **t_ree** son las cláusulas que nos permiten compilar la parte izquierda y derecha de cada regla de reescritura.

Con todo ello dadas las dos reglas para el símbolo **A**, se tendría el programa lógico:

(1)

a(S,SR) <- b(S,SR1) & c(SR1,SR).

a(S,SR) <- b(S,SR1) & d(SR1,SR).

donde los argumentos son listas de diferencias, es decir, **SR1** es la secuencia de entrada (**S**) que resta una vez reescrito el símbolo **B**, y **SR** lo que resta de **SR1** una vez reescrito el símbolo **C** (si tomamos la 1ª cláusula).

Solucionando el problema

Dado el ejemplo de reglas de reescritura que estamos tratando, podemos plantearnos obtener unas reglas equivalentes en la que eliminásemos la aparición del símbolo (o símbolos) redundantes, al comienzo de dos reglas con el mismo símbolo en la parte izquierda:

A –> B E
E –> C
E –> D

con lo que si partimos de esta gramática equivalente, al aplicarle el compilador de reglas obtendríamos el programa lógico:

(2)

a(S,SR) <- b(S,SR1) & e(SR1,SR).

e(S,SR) <- c(S,SR).

e(S,SR) <- d(S,SR).

que no presenta ineficiencias. Ahora bien, si hacemos esto cometeremos un grave error en el camino: estamos pretendiendo que el diseñador de la gramática se plantee inventarse símbolos (como **E**), que posiblemente no signifiquen nada para él.

La solución que vamos a plantear es partir de un programa lógico como (1) y obtener de forma automática un programa lógico como (2). Esto lo podremos realizar combinando las dos técnicas que hemos introducido.

Solución propuesta

Se propone un intérprete que tiene las siguientes reglas:

```
?- assert(simbolos(49)).

interpreta(Regla) <-
  clause(Regla,true).

interpreta(Regla) <-
  clause(Regla,(Cuerpo1 & RCuerpo1)) &
  clause(Regla,(Cuerpo1 & RCuerpo2)) &
  RCuerpo1 \== RCuerpo2 &
  retract(Regla,(Cuerpo1 & RCuerpo1)) &
  retract(Regla,(Cuerpo1 & RCuerpo2)) &
  retract(simbolos(N1)) &
  atom_chars(Atomo,[97,N1]) &
  N2 is N1 + 1 &
  assert(simbolos(N2)) &
  Regla =.. [_,S,SR] &
  Cuerpo1 =.. [_,S,SR1] &
  RCuerpo3 =.. [Atomo,SR1,SR] &
  assert(Regla, (Cuerpo1 & RCuerpo3)) &
  assert(RCuerpo3,Rcuerpo1) &
  assert(RCuerpo3,RCuerpo2) &
  fail.

interpreta(Regla).
```

para entender este intérprete retomemos el ejemplo dado en las reglas (1):

a(S,SR) <- b(S,SR1) & c(SR1,SR).

a(S,SR) <- b(S,SR1) & d(SR1,SR).

La primera cláusula es negativa y se traduce en guardar en un término, **simbolos**, el ASCII del número **1**.

La segunda es la regla para los hechos obtenidos por el compilador de reglas.

En la tercera nos aseguramos primero que hay dos reglas que tienen la misma cabeza, la misma primera cláusula en el cuerpo y que difieren en el resto (**&** es asociativo a la izquierda). A continuación eliminamos estas dos reglas, obtenemos el valor ASCII del número guardado y construimos un átomo a partir del átomo **a** y del número (esto se traduce en nuestro ejemplo en que el símbolo **E** es en realidad **A1**; en general va a ser **An**), guardamos el número siguiente y construimos la regla para **a** y las dos reglas para **a1**; por lo que se obtendrían 3 reglas como en (2). El fallo y la última regla están para que esto se pruebe con todos los casos posibles.

Aplicación al procesamiento del Lenguaje Natural (PLN)

En uno de los libros que mejor tratan el PLN [Allen,94], se introducen las gramáticas lógicas al final de los primeros capítulos dedicados a los analizadores del Lenguaje Natural, y puede comprobarse que no salen muy bien paradas, dando la impresión de que se tratan como una curiosidad más. A mi entender nuestro buen amigo J. Allen siente en su fuero interno (como buen americano) que la programación lógica no tiene entre sus aplicaciones más indicadas la de utilizarse para algoritmos de análisis. Yo comparto su parecer, es evidente que un analizador descendente basado en la estructura **chart** es mucho más eficiente que una **DCG**, pero si utilizamos la técnica que hemos desarrollado aquí podremos eliminar la ineficiencia que este buen señor encuentra en un analizador descendente puro, como el que se obtiene con una **DCG**.

Tomemos el ejemplo propuesto en la página 54 del citado libro. Sea la gramática del inglés:

S –> NP VP
NP –> ART ADJ N
NP –> ART N
NP –> ADJ N
VP –> AUX VP
VP –> V NP

Si admitimos que la palabra **can** pertenece a tres clases de símbolos léxicos, a saber: **AUX**, **V**, y **N**, el análisis de la frase *The can holds the water* tiene una ineficiencia en un analizador descendente puro al reescribir 4 veces como **ART** la palabra **the** (dos para la primera y dos para la segunda). Por otro lado, un analizador ascendente (incluso si está basado en la estructura chart) tiene una ineficiencia al considerar secuencias de símbolos que no pueden producirse (todas en las que **can** no se toma de la clase N); proponiéndose como solución un analizador descendente basado en la estructura chart.

En cuanto a la aplicación de nuestra técnica, podemos, sin desdeñar la programación lógica obtener una eficiencia del mismo orden de la que se consigue con el analizador desarrollado por Allen. En efecto las reglas:

np(S,SR) <- art(S,SR1) & adj(SR1,SR2) & n(SR2,SR).

np(S,SR) <- art(S,SR1) & n(SR1,SR).

que se obtienen por el compilador de reglas, se convierten después de ser tratadas por el intérprete de reglas en:

np(S,SR) <- art(S,SR1) & a1(SR1,SR).

a1(S,SR) <- adj(S,SR1) & n(SR1,SR).

a1(S,SR) <- n(S,SR).

quedando por tanto eliminada las ineficiencias.

Para reglas más complicadas como:

VP –> V
VP –> V NP
VP –> V PP

donde son más de dos reglas las que tienen en común el primer símbolo, se obtendría por la aplicación sucesiva del intérprete (por el fallo):

vp(S,SR) <- v(S,SR1) & a2(SR1,SR).

a1(S,S).

a1(S,SR) <- np(S,SR).

a2(S,SR) <- a1(S,SR).

a2(S,SR) <- pp(S,SR).

donde, de nuevo, respetando el diseño del experto en sintaxis se eliminan las ineficiencias que aparecerían en un analizador descendente puro.

Bilbiografía

[Allen,94] **James Allen.** *Natural language understanding*. Second Edition. **The Benjamin/Cummings Publishing Company.** 1994

[Abramson,89] **Harvey Abramson & Veronica Dahl.** *Logic Grammars*. Springer-Verlag. 1989

[Shieber,87] **Stuart M. Shieber & Fernando C. N. Pereira.** *Prolog and Natural-Language Analysis*. **CSLI.** 1987