Metaintérpretes

Objetivos

- Introducir las facilidades de Prolog para construir metaprogramas y metaintérpretes
- Mostrar ejemplos de metaintérpretes útiles

Metaintérpretes

Metaprogramas en Prolog

- Tratan a otros programas como datos
- En Prolog hay equivalencia entre programas y datos: ambos son términos.
- Es muy simple escribir programas para interpretar otros programas en Prolog, o para interpretar autómatas de distinto tipo.

Intérpretes AFND

Un autómata finito se define por la tupla:

(Q,Σ,δ,I,F)

Q - Conjunto finito de estados

Σ - Alfabeto finito

δ – Mapeo Q x Σ → Q

I – Estado inicial

F – Conjunto de estados finales

Intérpretes AFND

- Los autómatas finitos son dispositivos reconocedores de lenguajes, de amplio uso en informática.
- Los lenguajes reconocidos por AFNDs son la clase de lenguajes regulares
- Ejemplos:
  - (a*ab)*
  - {x Є (0|1)+, x mod 2 = 0}

Intérpretes AFND

- Es muy sencillo escribir un intérprete para un AFND:

```
acepta(A,Xs) ← el autómata A acepta la tira
contenida en la lista Xs

acepta(A,Xs):-inicial(A,Q),acepta(A,Q,Xs).
acepta(A,Q,[]):-final(A,Q).
acepta(A,Q,[X|Xs]):-
   delta(A,Q,X,Q1),
   acepta(A,Q1,Xs).
```

El no determinismo surge naturalmente de la ejecución Prolog.

Intérpretes APND

- Los autómatas con pushdown reconocen la clase de lenguajes Independientes de Contexto.
- Se agrega a los autómatas finitos un Stack.
- Esto permite reconocimiento sintáctico de todos los lenguajes de programación.

Intérpretes APND

Un autómata con pushdown se define por la tupla:

(Q,Σ,G,δ,I,Z,F)

Q – Conjunto finito de estados

Σ – Alfabeto finito de las tiras

G – Alfabeto finito del stack

δ – Mapeo Q x Σ x G* → Q x G*

I – Estado inicial

Z – Símbolo inicial del stack

F – Conjunto de estados finales

Intérpretes APND

Ejemplos:

1- Lenguaje con tiras de la forma $a^n b^n$

2- Palíndromos

Intérpretes APND

```
acepta(A,Xs) ← el APND A acepta la tira
contenida en la lista Xs

acepta(A,Xs):-
   inicial(A,Q),
   acepta(A,Q,Xs,[]).

acepta(A,Q,[],[]):-
   final(A,Q).
acepta(A,Q,[X|Xs],S):-
   delta(A,Q,X,S,Q1,S1),
   acepta(A,Q1,Xs,S1).
```

El no determinismo surge naturalmente de la ejecución Prolog.

Metaintérpretes

Un metaintérprete de un lenguaje es un intérprete para el lenguaje escrito en el mismo lenguaje

- Es un caso particular de metaprograma
- Las facilidades de Prolog hacen que resulte sencillo interpretarse a sí mismo

Metaintérpretes

Prolog tiene facilidades especiales para tratar como datos sus propios programas o para tratar datos como programas.

- Facilidad de metavariable
  - call(A)

  … o simplemente A
- Inspección de términos
  - functor
  - arg
  - univ
- Inspección de programas
  - clause(Cabeza,Cuerpo)
- Modificación del programa
  - assert(Clausula), retract(Clausula), los veremos la próxima clase

Metaintérpretes

- Un metaintérprete es un programa Prolog cuyo objetivo es “interpretar” programas Prolog, o sea, hacer que estos ejecuten.
- Un metaintérprete funciona en combinación con el intérprete Prolog que lo está interpretando a él.
- Esto permite distintas granularidades en el mecanismo de ejecución de Prolog.

Metaintérpretes

- Metaintérprete elemental, granularidad máxima:

```
solve(G):-G.
```

No tiene sentido, lo único que hace es agregar un predicado en la invocación.

Predicado clause

Premite acceder a las cláusulas de un programa

```
clause(+Head, ?Body)
```

Ejemplo:

Si tengo el predicado member:

```
member(X, [X|Xs]).
member(X, [Y|Ys]) :- member(X, Ys).
```

Sucede lo siguiente:

```
?- clause(member(X,Ys),Body).
Ys = [X|Xs],
Body = true;
Ys = [Y|Xs1],
Body = member(X,Ys1).
```

Metaintérpretes

- Granularidad intermedia

solve(Obj) ← se verifica el objetivo Obj dado el programa Prolog puro indicado por clause/2

```
solve(true).
solve((A,Gs)) :- solve(A), solve(Gs).
solve(G) :- clause(G,Body), solve(Body).
```

Este metaintérprete también se comporta igual que el intérprete de Prolog puro, pero tiene un nivel de agregación que permite insertar cambios.

Metaintérpretes

- Granularidad intermedia

Se suele usar cut

```
solve(true) :-!.
solve((A,Gs)) :- !, solve(A), solve(Gs).
solve(G) :- clause(G,Body), solve(Body).
```

Metaintérpretes

- Problema: El predicado clause no se puede utilizar con muchos predicados del sistema
  - Por ejemplo: <, is, var
  - Son predicados built in
- Hay que permitir que esos casos los maneje Prolog directamente
  - Utilizamos predicate_property para saber si es un predicado built in

```
predicate_property(G,built_in)
```

Metaintérpretes

- Resolviendo predicados built in

```
solve(true) :-!.
solve((A,Gs)) :- !, solve(A), solve(Gs).
solve(G) :-
   predicate_property(G,built_in),
   !,G.
solve(G) :-
   \+ predicate_property(G,built_in),
   clause(G,Body), solve(Body).
```

Traza

Ejemplo:

Un metaintérprete que imprima la traza de ejecución

```
largo([],0).
largo([_|Xs],N1):-
   largo(Xs,N),
   N1 is N + 1.
```

```
solve_trace(largo[1,2,3,4,5],X)).

largo([1,2,3,4,5],_G1)
|  largo([2,3,4,5],_G2)
|  |  largo([3,4,5],_G3)
|  |  |  largo([4,5],_G4)
|  |  |  |  largo([5],_G5)
|  |  |  |  |  largo([],0)
|  |  |  |  |  _G5 is 0+1
|  |  |  |  _G4 is 1+1
|  |  |  _G3 is 2+1
|  |  _G2 is 3+1
|  _G1 is 4+1
X = 5
```

Metaintérpretes

- Granularidad más fina

En el metaintérprete anterior la unificación es implícita.

Prolog realiza el backtracking por nosotros.

Podríamos manejar explícitamente la unificación y el backtracking

  – Es mucho más complejo
  – En general no se requiere llegar a este nivel de detalle

Metaintérpretes

- Granularidad más fina

Se podrían hacer cosas más poderosas

  - Implementar el cut
  - Cambiar el orden en que se seleccionan las cláusulas
  - Modificar el orden en que se recorre el árbol