Tema 7: Meta Intérprete *Vanilla*

Meta intérprete “Vanilla”

Shell para sistema basado en reglas

Contenido

- Meta intérpretes
- *Vanilla* básico
- *Vanilla* traza
- *Vanilla* pruebas

Meta intérprete

- Intérprete de un lenguaje escrito en el propio lenguaje
- Interés
  - Acceso al proceso de computo del lenguaje
  - Desarrollo de entornos de programación integrados

Meta intérprete más sencillo

```
solve(A):- call(A).
```

O bien:

```
solve(A):-A.
```

- Sin interés, pues no permite acceder a los elementos del proceso de computo.
- Más interesante: hacer explícita la regla de cómputo y la regla de búsqueda

Meta intérprete “Vanilla” (I)

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

- Lectura Declarativa
  - La meta vacía es cierta.
  - la meta conjuntiva (A, B) es cierta si A es cierta y B es cierta.
  - La meta A es cierta si existe una cláusula A:-B y B es cierta.

Meta intérprete “Vanilla” (II)

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

- Lectura Operacional
  - La meta vacía está resuelta.
  - Para resolver la meta (A, B) resolver primero A y después B (Regla de computo).
  - Para resolver la meta A, seleccionar una cláusula cuya cabeza unifique con A y resolver el cuerpo, usando la regla de búsqueda de Prolog.

Meta intérprete “Vanilla” (III)

Mejor:

```
solve(true):-!.
solve((A,B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

¿Por qué?

Meta intérprete “Vanilla” (IV)

```
solve(true):-!.
solve((A,B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

- Limitado a Prolog “puro”:
  - Sin modificación de la reevaluación: corte, fail, repeat...
  - Sin negación por fallo (programas definidos)
  - Sin asociación de procedimientos: predicados predefinidos

Extensión “Vanilla” con predicados predefinidos

```
builtin(A is B).  builtin(A > B).   builtin(A < B).
builtin(A = B).   builtin(A =:= B). builtin(A =< B).
builtin(A >= B).  builtin(functor(T, F, N)).
builtin(read(X)). builtin(write(X)).

solve(true):- !.
solve((A,B)) :-!, solve(A), solve(B).
solve(A):- builtin(A), !, A.
solve(A) :- clause(A, B), solve(B).
```

Extensión “Vanilla” traza

```
builtin(A is B). builtin(A > B).  builtin(A < B).
builtin(A = B).  builtin(A =:= B).builtin(A =< B).
builtin(A >= B). builtin(functor(T, F, N)).
builtin(read(X)).builtin(write(X)).

solve_traza(Meta)  :- solve_traza(Meta, 0).
solve_traza(true, Prf) :- !.
solve_traza((A,B), Prf) :-
   !, solve_traza(A, Prf), solve_traza(B, Prf).
solve_traza(A, Prf):-
   builtin(A), !, A, display(A, Prf), nl.
solve_traza(A, Prf) :-
   clause(A,B), display(A, Prf), nl,
   Prf1 is Prf +1, solve_traza(B, Prf1).

display(A, Prf):-
   Espacios is 3*Prf, tab(Espacios), write(A).
```

Traza conexión unidireccional

```
valor(w1, 1).
conectado(w1, w2).
conectado(w2, w3).
valor(W,X):-conectado(V,W), valor(V,X).

4 ?- solve_traza(valor(w3,X)).
valor(w3, _G332)
   conectado(w2, w3)
   valor(w2, _G332)
      conectado(w1, w2)
      valor(w1, 1)
X = 1 .
```

Extensión “vanilla” pruebas

```
builtin(A is B).    builtin(A > B).     builtin(A < B).
builtin(A = B).     builtin(A =:= B).   builtin(A =<
   B).
builtin(A >= B).    builtin(functor(T, F, N)).
builtin(read(X)).   builtin(write(X)).


solve(true,true) :- !.
solve((A, B), (ProofA, ProofB)) :-
   !, solve(A, ProofA), solve(B, ProofB).
solve(A, (A:-builtin)):- builtin(A), !, A.
solve(A, (A:-Proof)) :- clause(A, B), solve(B, Proof).
```

Prueba conexión unidireccional

```
2 ?- solve(valor(W,X),Prueba).
W = w1,
X = 1,
Prueba = (valor(w1, 1):-true) ;
W = w2,
X = 1,
Prueba = (valor(w2, 1):- (conectado(w1, w2):-true), (valor(w1,
    1):-true)) ;
W = w3,
X = 1,
Prueba = (valor(w3, 1):- (conectado(w2, w3):-true), (valor(w2,
    1):- (conectado(w1, w2):-true), (valor(w1, 1):-true))) ;
false.
```