Meta-Programação

Motivação:

- Manipular lógica em lógica.
- Formalização de *Provabilidade*
- Programas e dados como formulas lógicas (representação de programas e provas)
- Programação de provadores de teoremas,
- Útil para transformações de programas, adicionar explicações em sistemas periciais, meta-interpretadores, etc.

Meta-Programação: requer a capacidade de manipular o nome de expressões bem como o seu valor.

Meta-Interpretadores

Um *meta-interpretador* de uma linguagem é um interpretador para a linguagem escrito na própria linguagem. Assim um programa deste tipo pode analisar, transformar ou simular outros programas.

No caso do Prolog, um meta-interpretador é definido por uma nova relação.

Por exemplo, a relação solve(Query) é verdade se Query for verdade em relação ao programa que vai ser interpretado.

Pode-se escrever um meta-interpretador mais simples que faz uso da noção de meta-variável.

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

Podemos no entanto simular todo o modelo computacional do Prolog. Isto é, A é verdade se existir uma clausula A $\leftarrow$ B tal que B é verdade.

Meta-linguagem:

Lógica separa *linguagem objecto* (que fala à cerca do mundo) da *linguagem meta* (que fala à cerca da linguagem objecto).

O meta-interpretador "Vanilla":

$$\mathsf{solve(P)} \leftarrow \mathsf{clausula(P, Q)} \ \& \ \mathsf{solve(Q)}$$

$$\mathsf{solve(true)}$$

$$\mathsf{solve(P\ \&\ Q)} \leftarrow \mathsf{solve(P)} \ \& \ \mathsf{solve(Q)}$$

em que as clausulas a manipular são definidas por:

$$\mathsf{clausula(mortal(X), homem(X)\ \&\ true)}$$

$$\mathsf{clausula(homem(socrates), true)}$$

Raciocínio no nível *meta* pode simular o raciocínio no nível *objecto*.

Uma definição mais elaborada (com parametrização de teorias)

$$\mathsf{demo(T, P)} \leftarrow \mathsf{demo(T, P} \leftarrow \mathsf{Q)} \ \& \ \mathsf{demo(T, Q)}$$

$$\mathsf{demo(T, P\ \&\ Q)} \leftarrow \mathsf{demo(T, P)} \ \& \ \mathsf{demo(T, Q)}$$

$$\mathsf{demo(P\ \&\ Q, P)}$$

$$\mathsf{demo(R\ \&\ Q, P)} \leftarrow \mathsf{demo(Q, P)}$$

Podemos assim trabalhar com varias teorias ao mesmo tempo.

Programas como Dados

Formalizar um interpretador de Prolog em Prolog

```
solve([]).
solve([G|Goals]):- clausula(G,Mgs),
                   append(Mgs,Goals,Ts), solve(Ts).
```

Programa definido como:

```
clausula(pai(joao,rui),[]).
clausula(pai(rui,luis),[]).
clausula(antecessor(X,Y),[pai(X,Y)]).
clausula(antecessor(X,Y),[antecessor(X,Z),pai(Z,Y)]).
```

onde

$$\mathsf{clausula}(\mathsf{p}, [\mathsf{q}, \mathsf{r}])$$

na meta-linguagem significa

$$\mathsf{p} \leftarrow \mathsf{q} \;\&\; \mathsf{r}$$

na linguagem objecto.

Meta-Interpretador para provas com interacção do utilizador

```
solve([]).
solve([G|Goals]):- clausula(G,Mgs),
                   append(Mgs,Goals,Ts), solve(Ts).
solve([G|Goals]):- askable(G), ask(G,A),
                   resposta(A), solve(Goals).

ask(G,A):- write("? "), write(G), read(A).

resposta(sim).
resposta(nao):- fail.
```

Podemos adicionar mecanismos de explicação para descrever porque determinado "goal"é perguntado ao utilizador.

Outros interpretadores podem ser produzidos. Por exemplo um meta-interpretador de Prolog com mecanismos de tabulação com garantias de terminação. Outro exemplo são os interpretadores que simulam computação bottom-up.

Interpretação e Materialização de Provas

```
como(G):- solve([G],Prova), interpreta(Prova).

solve([],[]).
solve([G|Goals],[(G se Mgs)|Res]):- clausula(G,Mgs),
                append(Mgs,Goals,Ts), solve(Ts,Res).

interpreta([]).
interpreta([se(X,[])|Tail]):- nl, write(X),
                write(' e um facto '),
                interpreta(Tail), !.
interpreta([se(X,Corpo)|Tail]):- nl, write(X),
                write(' e provado pela regra '),
                write(X),  write(' se '),
                mostra(Corpo), interpreta(Tail).

mostra([X]):- write(X).
mostra([X|Xs]):- write(X), write(' & '), mostra(Xs).
```

Aplicação directa: Explicações em Sistemas Periciais ou outro tipo de sistemas de prova.

Um trace:

a interrogação

$$?\mathsf{como}(\mathsf{antecessor}(\mathsf{joao}, \mathsf{X}))$$

tem como "output":

```
antecessor(joao,rui)e provado pela regra
antecessor(joao,rui) se pai(joao,rui)

pai(joao,rui) e um facto

X = rui ? ;

antecessor(joao,luis) e provado pela regra
antecessor(joao,luis) se antecessor(joao,rui) & pai(rui,luis)

antecessor(joao,rui) e provado pela regra
antecessor(joao,rui) se pai(joao,rui)

pai(joao,rui) e um facto

pai(rui,luis) e um facto

X = luis ?
```

Podíamos obviamente embelezar o output, etc.

Meta-Interpretação *Blended In*

Uma implemetação alternativa é incluir informação meta ao nível objecto.

Assim, a construção do trace passa para um novo parâmetro no predicado original.

```
antecessor(X,Y,[antecessor(X,Y) se pai(X,Y)|M]):-
                          pai(X,Y,M).
antecessor(X,Y,[antecessor(X,Z) se pai(Z,Y)|M]):-
                          antecessor(X,Z,M1),
                          pai(Z,Y,M2), append(M1,M2,M).

pai(joao,rui,[pai(joao,rui)]).
pai(rui,luis,[pai(rui,luis)]).
```

?antecessor(joao, X, Prova)

tem como resultado:

X = rui

Prova = [antecessor(joao, rui) se pai(joao, rui), pai(joao, rui)]

Prova pode ser convenientemente tratado e imprimido na forma de uma explicação.

Especificar Unificação

Podemos, por exemplo, elaborar o nossa meta-interpretador por forma a definir unificação.

```
solve([],[]).
solve([G|Goals],[(G se Mgs)|Res]):-
                clausula(X,Mgs), unifica(X,G),
                append(Mgs,Goals,Ts), solve(Ts,Res).
```

Onde unifica é a nossa versão pessoal do algoritmo de unificação.

```
unifica(X,Y):- var(X), var(Y), X = Y.
unifica(X,Y):- var(X), nonvar(Y),
               nao_ocorre(X,Y), X = Y.
unifica(X,Y):- var(Y), nonvar(X),
               nao_ocorre(Y,X), Y = X.
unifica(X,Y):- nonvar(X), nonvar(Y),
               constant(X), constant(Y), X = Y.
unifica(X,Y):- nonvar(X), nonvar(Y), compound(X),
               compound(Y), term_unify(X,Y).

term_unify(X,Y):- functor(X,F,N), functor(Y,F,N),
               unify_args(N,X,Y).

unify_args(N,X,Y):- N > 0, unify_args(N,X,Y),
               N1 is N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).
unify_args(N,X,Y):- arg(N,X,ArgX), arg(N,Y,ArgY),
               unifica(ArgX,ArgY).
```

var(X) sucede se X é uma variável não instanciada. Isto é se X é efectivamente uma variável. nonvar é o oposto de var.

Definir "Occurs Check":

Evita-se assim a unificação de $\mathsf{X} = \mathsf{f}(..., \mathsf{X}, ...)$.

```
% Occurs Check
nao_ocorre(X,Y):- var(Y), X \== Y.
nao_ocorre(X,Y):- nonvar(Y), constant(Y).
nao_ocorre(X,Y):- nonvar(Y), compound(Y),
               functor(Y,F,N), nao_ocorre(N,X,Y).

nao_ocorre(N,X,Y):- N > 0, arg(N,Y,Arg),
               nao_ocorre(X,Arg), N1 is N - 1,
               nao_ocorre(N1,X,Y).
nao_ocorre(0,X,Y).
```

Os exemplos práticos mais vulgares onde *occurs-check* é necessário para se obter respostas correctas são os programas que usam diferença de listas.

Por exemplo a query

$$?\mathsf{qsort}([], [\mathsf{a} \mid \mathsf{X}] - \mathsf{X})$$

não obtém resposta porque o Prolog entra em ciclo com a tentativa de fazer a substituição $\mathsf{X}/[\mathsf{a} \mid \mathsf{X}]$.

Formalização do Procedimento SLD

demonstrate(Prog, [])

demonstrate(Prog, Goals) $\leftarrow$

select(Goals, Goal, Rest) &

member(Procedure, Prog) &

renamevars(Procedure, Goal, Proc2) &

parts(Proc2, Head, Body) &

match(Goal, Head, Subst) &

add(Body, Rest, Intgoals) &

apply(Intgoals, Subst, Newgoals) &

demonstrate(Prog, Newgoals)

Esta seria a formalização mais completa possível do mecanismo SLD no formato de meta-interpretador.