University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 10.1

Inference Engines

- Backward Chaining
- Meta Interpretation
- Tracing

Department of Computer Science and AI
1998/9

Interpreting Rule Formalisms

Earlier we considered some of the qualitative differences between forward and backward chaining algorithms. Next we consider how to go about implementing such algorithms. Any strategy has to involve two stages.

- **Language Definition.** The language provides a means of expressing rules and facts. Typically this will involve the definition of special operators.
- **Language Interpretation.** This provides a *semantics* for the language. Below we consider two broad styles of interpretation: forward and backward chaining.

Backward Chaining
A Trivial Example

Backward chaining is the strategy employed by the Prolog interpreter itelf. Thus one way of implementing a backward chaining rule interpreter is to employ the language of Prolog itself, e.g.

Rule

```
grandfather(X,Z) :-
  father(X,Y), father(Y,Z).
```

Facts

```
father(daniel,mike).
father(mike,maurice).
```

Implementing the Interpreter

Obviously, we could just use the Prolog query mechanism to execute a query in this language. However, our aim is to express the interpreter as a separate predicate such as **`meta1/1`**: kdoub rmizz amusc

```
meta1(Q) :- call(Q)
```

This is not very revealing since all the work is is carried out by the Prolog built-in **`call`** which simply invokes its argument as a normal query.

`meta2` is slightly more revealing

meta-2

```
meta2(true) :- !.

meta2((G1,G2)) :-
 !,
 meta2(G1),
 meta2(G2).

meta2(G) :-
 clause(G,Subgoals),
 meta2(Subgoals).
```

N.B. `clause(Head,Body)` retrieves database items with head Head and subgoals Body.

Operation of `clause(Head,Body)`

- For a *fact*, Body is the special atom **`true`**

```
?- clause(father(X,Y),Z).
X=daniel, Y=mike, Z=true;
X=mike, Y=maurice, Z=true
```

- For a *rule*, Body is a comma-separated list of subgoals, i.e. (FirstGoal, RestGoals).

```
?- clause(grandfather(X,Y),Z).
Z= (father(X,Y),father(Y,Z));
```

Illustration

```
?- meta2(gf(dan,X)).
   Call: 7) meta2(gf(dan,A))
 ^ Call: 8) clause(gf(dan,A), _L140)
 ^ Exit: 8) clause(gf(dan,A),(f(dan,B), f(B,A)))
   Call: 8) meta2( (f(dan,B), f(B,A)))
   Call: 9) meta2(f(dan,B))
 ^ Call:10) clause(f(dan,B ),C)
 ^ Exit:10) clause(f(dan,mike),true)
   Call:10) meta2(true)
   Exit:10) meta2(true)
   Exit: 9) meta2(f(dan, mike))
   Call: 9) meta2(f(mike,A))
 ^ Call:10) clause(f(mike,A),C)
 ^ Exit:10) clause(f(mike,maurice), true)
   Call:10) meta2(true)
   Exit:10) meta2(true)
   Exit: 9) meta2(f(mike, maurice))
   Exit: 8) meta2( (f(dan, mike), f(mike, maurice))
   Exit: 7) meta2(gf(dan, maurice))
```

meta2: Summary

- **Language Syntax.** same as Prolog - not ideal for user not familiar with Prolog
- **Functionality.** same as Prolog - nothing gained.
- Knowledge base not syntactically distinguishable from the rest of the program.

For these reasons we will modify things in stages.

- Change Syntax
- Modify Interpreter

Designing a Simple Rule Formalism

The following operator declaration

```
:- op(1100,xfx,['=>']).
:- op(500,fx,[=>]).
```

allows us to write rules like this:

```
father(X,Y), father(Y,Z) => father(X,Z).
```

and facts like this:

```
=> father(daniel,mike).
=> father(mike,maurice).
```

The definition of the interpreter is supplied by means of the predicate **meta3**.

Definition of `meta3`

The interpreter looks like this:

```
meta3((G1,G2)) :-
 !,
 meta3(G1),
 meta3(G2).

meta3(G) :- => G, !.

meta3(G) :-
 (Subgoals => G),
 meta3(Subgoals).
```

meta3: Demonstration

```
meta3(gf(dan,X)).
Call:  7) meta3(gf(dan,A))
Call:  8) =>gf(dan,A)
Fail:  8) =>gf(dan,A)
Redo:  7) meta3(gf(dan,A))
Call:  8) _L140=>gf(dan,A)
Exit:  8) f(dan,B), f(B,A)=>gf(dan,A)
Call:  8) meta3( (f(dan,B), f(B,A)))
Call:  9) meta3(f(dan,B))
Call: 10) =>f(dan,B)
Exit: 10) =>f(dan, mike)
Exit:  9) meta3(f(dan, mike))
Call:  9) meta3(f(mike,A))
Call: 10) =>f(mike,A)
Exit: 10) =>f(mike, maurice)
Exit:  9) meta3(f(mike, maurice))
Exit:  8) meta3( (f(dan, mike), f(mike, maurice)))
Exit:  7) meta3(gf(dan, maurice))
```

Pre-Declared Operators

```
1   fx  $      500 yfx \/    700 xfx is
200 xfy ^      600 xfy :     700 xfx \=
200 xfy ^      700 xfx <     700 xfx \==
300 xfx mod    700 xfx =     700 xfx =@=
400 yfx *      700 xfx =..   900 fy   not
400 yfx /      700 xfx =:=   900 fy   \+
400 yfx //     700 xfx <     1000 xfy ,
400 yfx <<     700 xfx ==    1050 xfy ->
400 yfx >>     700 xfx =@=   1050 xfy *->
400 yfx xor    700 xfx =\=   1100 xfy ;
500 fx   +     700 xfx >     1100 xfy |
500 fx   -     700 xfx >=    1200 fx  :-
500 fx   ?     700 xfx @<    1200 fx  ?-
500 fx   \     700 xfx @=<   1200 xfx -->
500 yfx +      700 xfx @>    1200 xfx :-
500 yfx -      700 xfx @>=   500 yfx /\
```

Adding Functionality I

1. Names for Rules and Facts
2. Proof Trees

```
r1 :: father(X,Y), father(Y,Z)
      => grandfather(X,Z).

r2 :: parent(X,Y), male(X)
      => father(X,Y).

f1 :: => male(daniel).
f2 :: => male(mike).
f3 :: => male(maurice).
f3 :: => parent(daniel,mike).
f4 :: => parent(mike,maurice).

?- meta4(grandfather(daniel,X),P).

X = maurice
P = r1(r2(f3, f1), r2(f4, f2))
```

Implementation: `meta4`

```
:- op(1100,xfx,[=>]).
:- op(500,fx,[=>]).
:- op(1150,xfx,[::]).


meta4((G1,G2),(P1,LP)) :-
  !,
 meta4(G1,P1),
 meta4(G2,LP).

meta4(G,F) :- (F :: => G), !.

meta4(G,PTree) :-
  (R::Subgoals => G),
 meta4(Subgoals,PS),
 mktree(R,PS,PTree).
```

Making a Proof Tree: mktree/3

```
% Root is an atom
% Args is a comma-list
% Tree is a tree with functor Root
% and arglist Args

mktree(Root,Args,Tree) :-
 clist2plist(Args,Largs),
 Tree=..[Root|Largs].

% convert a comma list to a Prolog list
clist2plist((E1,(E2,E3)),[E1|Rest]) :-
 !, clist2plist((E2,E3),Rest).

clist2plist((E1,E2), [E1|[E2]]).
```

Adding Functionality II
Tracing Information

```
?- meta5(grandfather(daniel,X),Z).
call> grandfather(daniel, _G280)
call> father(daniel, _G393)
exit> father(daniel, mike)
call> father(mike, _G280)
exit> father(mike, maurice)
exit> grandfather(daniel, maurice)

X = maurice
Z = r1(r2(f3, f1), r2(f4, f2)) ;
redo> grandfather(daniel, maurice)
redo> father(mike, maurice)
fail> father(mike, _G280)
redo> father(daniel, mike)
fail> father(daniel, _G393)
fail> grandfather(daniel, _G280)
No
```

meta5

```
meta5((G1,G2),(P1,LP)) :-
 [as before]

meta5(G,F) :- (F :: => G), !.

meta5(G,PTree) :-
 write('call> '), write(G), nl,
 (R::Subgoals => G),
 meta5(Subgoals,PS),
 write('exit> '), write(G), nl,
 mktree(R,PS,PTree),
 meta5_redo(G,Ptree).

meta5(G,PTree) :-
 write('fail> '), write(G),nl,
 fail.

meta5_redo(G,Ptree) :- true.

meta5_redo(G,Ptree) :-
  write('redo> '), write(G),nl,
  fail.
```

Other Additions

- Explanation Based Generalisation. A technique for adapting old proofs to new problems by generalisation. Based on generation of proof trees.
- Intepreter which asks when it does not have the required facts.
- Breadth First Interpreter.
- Best First Interpreater.