University of BRISTOL

Artificial Intelligence: Logic Programming III

Oliver Ray

From pure to practical logic programming

- The last lecture began to build upon the Datalog paradigm by introducing **recursive definitions** and **structured terms** as core features of Prolog and exploring their relation to **classical logic** and **functional programming**
- Now we understand more about the Prolog language and its denotational semantics, this lecture begins to explore Prolog's operational semantics in order to explain how Prolog queries are actually answered (and controlled!)
- The computational basis of Prolog is formalised through the concepts of **unification**, **resolution**, **proof trees** and **search trees** – which allow us to visualise the search space explored by Prolog (under its default strategies)
- We also take a closer look at some of Prolog's operators (and especially some different notions of "**equality**" (arising from the fact Prolog variables more like unknowns than aliases) and the "**cut**" operator (for pruning the search space)

An overview of Prolog’s core operators

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	**dynamic**, **discontiguous**, **initialization**, **meta_predicate**, **module_transparent**, **multifile**, **public**, **thread_local**, **thread_initialization**, **volatile**
1105	xfy	|
1100	xfy	;
1050	xfy	->, *->
1000	xfy	,
990	xfx	:=
900	fy	\+
700	xfx	<, =, =.., =@=, \=@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, **as**, **is**, >:<, :<
600	xfy	:
500	yfx	+, -, /\, \/, **xor**
500	fx	?
400	yfx	*, /, //, **div**, **rdiv**, <<, >>, **mod**, **rem**
200	xfx	**
200	xfy	^
200	fy	+, -, \
100	yfx	.
1	fx	$

A note on some Prolog “equality” operators

Numeric evaluation: X is 1+2 X/3 3 is 1+2 yes

https://www.swi-prolog.org/pldoc/doc_for?object=(is)/2

note: rhs is evaluated numerically and the result is unified with the lhs

Term unification: X = 1+2 X/+(1,2) 3 = 1+2 no

https://www.swi-prolog.org/pldoc/man?predicate=%3D/2

note: the mgu is computed and applied to both terms (if it exists)

On this unit you will only *need* these two!

Term equivalence: X == 1+2 no 3 == 1+2 no

https://www.swi-prolog.org/pldoc/man?predicate=%3D%3D/2

note: a variable X is only identical to itself (or *notionally* to a ‘sharing’ variable Y)

Numeric equality: X =:= 1+2 error 3 =:= 1+2 yes

https://www.swi-prolog.org/search?for=%3D%3A%3D

note: both sides are evaluated numerically and the results are unified

Term variant: X =@= 1+2 no 3 =@= 1+2 no

https://www.swi-prolog.org/pldoc/doc_for?object=(%3D@%3D)/2

note: terms are variable renamings of each other (equivalent → variant → unifiable)

Equality constraint: X #= 1+2 X/3 3#=Y+2 Y/1 3 #= 1+2 yes

https://www.swi-prolog.org/pldoc/man?section=clpfd

note: requires “:- use_module(library(clpfd)).” and subsumes both “is” and “=:=“

Computing Answers: Intuition

- Prolog returns **computed answer substitutions** by repeatedly **resolving** query literals with (user-defined) clauses until there are no literals left to prove
- A query literal is chosen by a **selection rule** (which, by default, returns the **leftmost** literal)
- A clause is chosen by a **search rule** (which, by default, returns clauses from **top-to-bottom**)
- A fresh **variant** of the database clause is created by renaming all of the variables to new ones and alternative clauses may be subsequently considered in a process called **backtracking**
- A **most general unifier (mgu)** is found for the selected literal and head of the chosen clause
- The **resolvent** of the query and the clause (on the selected literal) is formed by applying the mgu and replacing the selected literal by the body of clause to leave a new query
- This is repeated until the **empty clause** (with no literals) written □ is obtained
- The composition of all the mgus is taken and applied to the original query to yield an **answer**
- If there are no resolvents, the branch is a failed dead end denoted by # or an underlined goal

Substitutions and MGUs: Formalised

- A **substitution** is a **set** of **bindings** of (distinct) **variables** to **terms** (distinct from themselves):

  $\theta=\{X_1/t_1, \ldots X_n/t_n\}$ where $\boldsymbol{\varepsilon}$ is used to denote the empty substitution $\theta=$**{}**
- The **application** of a substitution θ to an expression E is denoted $E\theta$ and obtained by replacing any (free) variable X_i in E by the corresponding term t_i from θ, if one exists
- The **composition** of two substitutions θ_1 and θ_2 denoted $\theta_1\theta_2$ is defined as follows:

  $\theta_1\theta_2 = \{X/(t\theta_2) \mid X/t\in\theta_1 \wedge X\neq t\theta_2\} \cup \{Y/s\in\theta_2 \mid Y\neq X \text{ for all } X/t\in\theta_1\}$ so $E(\theta_1\theta_2)=(E\theta_1)\theta_2$
- A substitution θ_1 is (as or) **more general** than a θ_2 iff there exists some θ_3 such that $\theta_1\theta_3 =\theta_2$
- A substitution θ is a **unifier** of two expressions E_1 and E_2 iff $E_1\theta=E_2\theta$
- A substitution θ is a **most general unifier (mgu)** of two expressions E_1 and E_2 iff θ is a unifier of E_1 and E_2 that is more general than all other unifiers of E_1 and E_2 (and so is unique up to renaming)
- Given two expressions E_1,E_2 and an (initially empty) substitution θ, we can **compute** an mgu as follows: $mgu(E_1,E_2,\theta)=mgu(E_1\{X/t\}, E_2 \{X/t\}, \theta\{X/t\})$ where X is a variable from one expression at the first syntactic position where the two expressions differ and t is corresponding term in the other (nb. if neither is a variable then there is no mgu; if both are variables then we can bind either to the other; strictly we should fail if t mentions X but this '**occurs check**' is usually omitted in Prolog)

Example: substitutions

if $t_1 = p(W, f(W, X))$ $\theta_1 = \{W/X, X/W\}$ $\theta_3 = \{W/a, Y/a, X/Z\}$

$t_2 = p(Y, f(a, Z))$ $\theta_2 = \{W/a, Y/a, Z/X\}$ $\theta_4 = \{W/a, Y/a, Z/V, X/V\}$

then $t_1\theta_1 = p(W, f(W,X))\{W/X, X/W\} = p(X, f(X,W))$

$t_2\theta_1 = p(Y, f(a,Z))\{W/X, X/W\} = p(Y, f(a,Z))$

Thus θ_1 is NOT a unifier of t_1 and t_2

But θ_2 and θ_3 and θ_4 all ARE unifiers of t_1 and t_2 (EASY EXERCISE!)

θ_2 is more general than θ_3 as $\{W/a, Y/a, Z/X\}\{X/Z\} = \{W/a, Y/a, X/Z\}$ – as Z/Z excluded

θ_3 is more general than θ_4 as $\{W/a, Y/a, X/Z\}\{Z/V\} = \{W/a, Y/a, X/V, Z/V\}$

θ_4 is NOT more general than θ_3 as $\{W/a, Y/a, Z/V, X/V\}\,\theta = \{W/a, Y/a, X/Z\}$ would imply $V/Z \in \theta$ in order to exclude Z/. from the composition, but then V/Z would have to be in the composition, which is a contradiction

Example: MGU

given	plus(	X	,	Y	,	s(Y)	)
and	plus(	s(V)	,	W	,	s(s(V))	)

X/s(V)

plus(	s(V)	,	Y	,	s(Y)	)
plus(	s(V)	,	W	,	s(s(V))	)

Y/W

plus(	s(V)	,	W	,	s(W)	)
plus(	s(V)	,	W	,	s(s(V))	)

W/s(V)

plus(	s(V)	,	s(V)	,	s(s(V))	)
plus(	s(V)	,	s(V)	,	s(s(V))	)

{ X/s(V) , Y/s(V) , W/s(V) }

Unification & Resolution: Example

Selected query literal

?- length([a,b,c], N).

?- length([a,b,c],N).

for convenience: use . for '[|]'

length(.(a, .(b, .(c, []))) , N)
length(.(H1, T1) , N1)

H1/a

length(.(a, .(b, .(c, []))) , N)
length(.(a, T1) , N1)

H1/a, T1/[b,c]

length(.(a, .(b, .(c, []))) , N)
length(.(a, .(b, .(c, []))) , N1)

H1/a, T1/[b,c], N1/N

most general unifier (mgu)

`{H1/a, T1/[b,c], N1/N}`

Selected clause variant

```
length([H1|T1], N1) :-
    length(T1, M1),
    N1 is 1+M1.
```

```
length([a,b,c], N) :-
    length([b,c],M1),
    N is 1+M1.
```

knowledge base

```
length([], 0).
length([_|T], N) :- length(T, M), N is 1+M.
```

resolvent

?- length([b,c], M1), N is 1+M1.

Proof Tree: Example

```
?- length([a,b,c],N).                                  length([H1|T1],N1) :- length(T1,M1), N1 is 1+M1.
        {H1/a, T1/[b,c], N1/N}
?- length([b,c],M1), N is 1+M1.                        length([H2|T2],N2) :- length(T2,M2), N2 is 1+M2.
        {H2/b, T2/[c], N2/M1}
?- length([c],M2), M1 is 1+M2, N is 1+M1.              length([H3|T3],N3) :- length(T3,M3), N3 is 1+M3.
        {H3/c, T3/[], N3/M2}
?- length([],M3), M2 is 1+M3, M1 is 1+M2, N is 1+M1.   length([],0).
        {M3/0}
?- M2 is 1+0, M1 is 1+M2, N is 1+M1.
        {M2/1}
?- M1 is 1+1, N is 1+M1.
        {M1/2}
?- N is 1+2.
        {N/3}   computed answer substitution
□               empty clause
```

length([], 0).
length([_|T], N) :- length(T, M), N is 1+M.

Accumulators and Tail Recursion

- You may have noticed the definition of length/2 on the previous slide results in an unnecessarily inefficient memory footprint (which is linear with respect to the length of the list) by gradually collecting together all of the “is” literals before actually starting to evaluate them (at least under Prolog’s default left-most selection strategy)

```
% length(+List, -Len)
% Len is length of List
length([], 0).
length([_|T], N) :- length(T, M), N is 1+M.
```

- A deterministic tail recursive definition (like the original Haskell) will often be more memory efficient - but that possibility was ruled out here due to the use of a moded arithmetic operator (which requires all its input arguments to be ground at the time of a call)

- But, it is often relatively easy to obtain an efficient tail recursive Prolog definition using an auxiliary argument called an accumulator (which stores the intermediate result of the computation up until this point, starting from some initial given value)

```
% length(+List, +Acc, -Len)
% length(List, 0, Len) ↔ length(List, Len)
length([], A, A).
length([_|T], A, N) :- M is 1+A, length(T, M, N).
```

Proof Tree: Revisited

?- length([a,b,c],0,N).

length([H1|T1], A1, N1) :- M1 is 1+A1, length(T1, M1, N1).

{H1/a, T1/[b,c], A1/0, N1/N}

?- M1 is 1+0, length([b,c],M1,N).

{M1/1}

?- length([b,c],1,N).

length([H2|T2], A2, N2) :- M2 is 1+A2, length(T2, M2, N2).

{H2/b, T2/[c], A2/1, N2/N}

M2 is 1+1, length([c],M2,N).

{M2/2}

?- length([c],2,N).

length([H3|T3], A3, N3) :- M3 is 1+A3, length(T3, M3, N3).

{H3/c, T3/[], A3/2, N3/N}

M3 is 1+2, length([],M3,N).

{M3/3}

?- length([],3,N).

length([], A4, A4).

{A4/3, N/3}

□

```
length([], A, A).
length([_|T], A, N) :- M is 1+A, length(T, M, N).
```

Memory Usage: Implications

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% len(+List, -Len)
% Len is length of List
%
len([],0).
len([_|T],N) :- len(T,M), N is 1+M.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/**<examples>
?- biglist(_Xs),len(_Xs,L).
?- biglist(_Xs),len(_Xs,0,L).
*/
biglist(Xs) :- findall(X,between(1,5000000,X),Xs).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% len(+List, +Acc, -Len)
% len(List, 0, Len) <=> len(List, Len)
%
len([],A,A).
len([_|T],A,N) :- M is 1+A, len(T,M,N).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
biglist(_Xs),len(_Xs,L).

Stack limit (0.2Gb) exceeded
  Stack sizes: local: 64.0Mb, global: 0.1Gb, trail: 1Kb
  Stack depth: 698,836, last-call: 0%, Choice points: 12
  Possible non-terminating recursion:
    [698,836] len([length:4,301,197], _1624)
    [698,835] len([length:4,301,198], _1656)
```

```
biglist(_Xs),len(_Xs,0,L).

L = 5000000

1.775 seconds cpu time
```

Behind the scenes peek: length(?List,?Len)

```
length(List, Length) :-
    var(Length),
    !,
    '$skip_list'(Length0, List, Tail),
    (   Tail == []
    ->  Length = Length0                    % +,-
    ;   var(Tail)
    ->  Tail \== Length,                    % avoid length(L,L)
        '$length3'(Tail, Length, Length0)   % -,-
    ;   throw(error(type_error(list, List),
                    context(length/2, _)))
    ).
length(List, Length) :-
    integer(Length),
    Length >= 0,
    !,
    '$skip_list'(Length0, List, Tail),
    (   Tail == []                          % proper list
    ->  Length = Length0
    ;   var(Tail)
    ->  Extra is Length-Length0,
        '$length'(Tail, Extra)
    ;   throw(error(type_error(list, List),
                    context(length/2, _)))
    ).
```

```
length(_, Length) :-
    integer(Length),
    !,
    throw(error(domain_error(not_less_than_zero, Length),
                context(length/2, _))).
length(_, Length) :-
    throw(error(type_error(integer, Length),
                context(length/2, _))).

'$length3'([], N, N).
'$length3'([_|List], N, N0) :-
    N1 is N0+1,
    '$length3'(List, N, N1).
```

case handling

type checking

accumulator

fast list access (“swiss army knife”)

error handling

Search (SLD) Tree

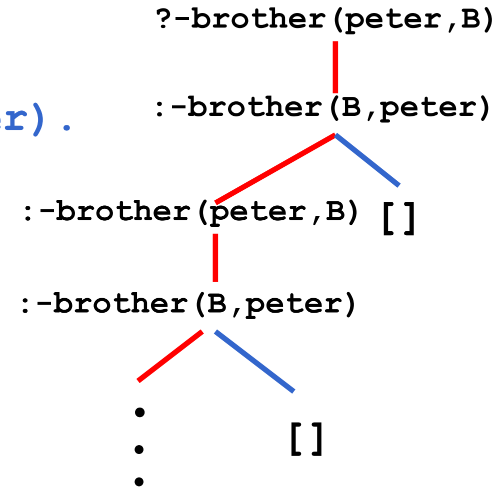
- Shows search space reachable through backtracking
- Nodes are queries: the root is the initial query and children are the resolvents of the parent on its first literal
- Leaves represent success branches (empty clause []) or failure branches with no resolvents (underlined)
- A proof tree can be obtained for each success branch by reconstructing the resolved clauses and mgus

```
student_of(X,T):-
    follows(X,C),teaches(T,C).
follows(paul,computer_science).
follows(paul,expert_systems).
follows(maria,ai_techniques).
teaches(adrian,expert_systems).
teaches(peter,ai_techniques).
teaches(peter,computer_science).
```

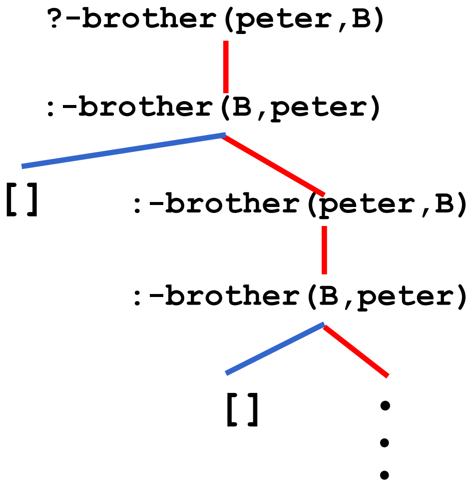
```
?-student_of(S,peter)

:-follows(S,C),teaches(peter,C)

:-teaches(peter,computer_science)
:-teaches(peter,expert_systems)
:-teaches(peter,ai_techniques)

[]
[]
```

Infinite Search Trees

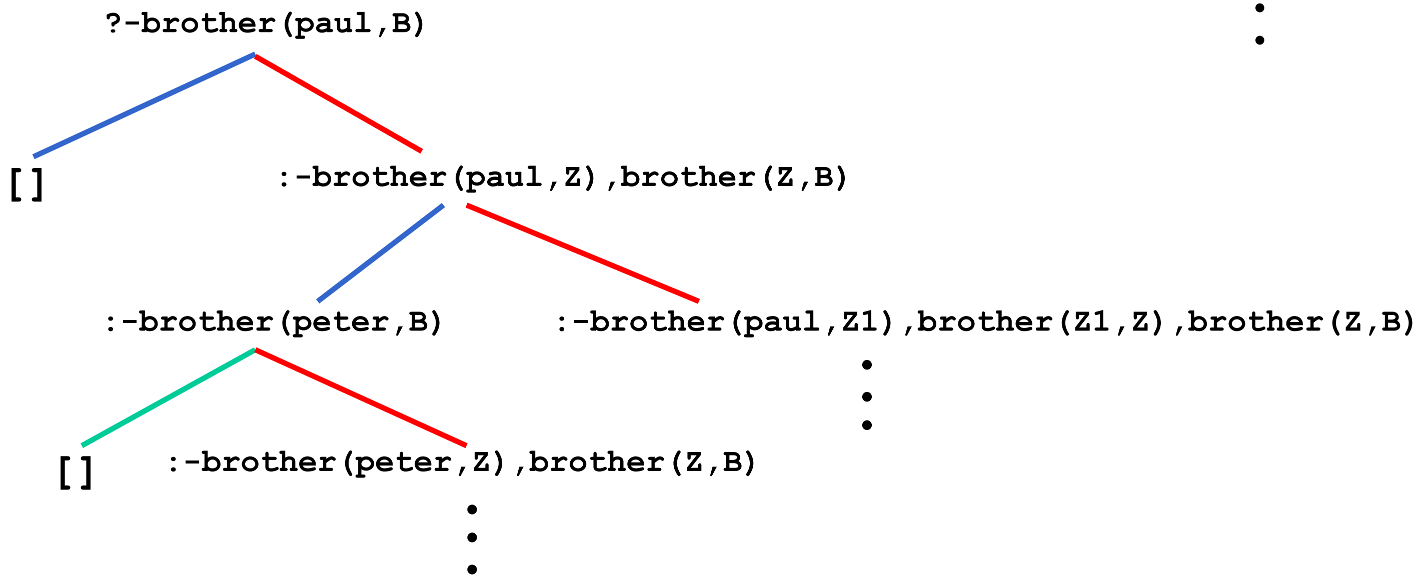
```
brother_of(X,Y) :-
  brother_of(Y,X).
brother_of(paul,peter).
```




```
brother_of(paul,peter).
brother_of(X,Y):-
      brother_of(Y,X).
```




```
brother(paul,peter).
brother(peter,adrian).
brother(X,Y):-
      brother(X,Z),
      brother(Z,Y).
```



Thank you