TECHNISCHE UNIVERSITÄT DRESDEN

Computational Logic Group

Hannes Strass (based on slides by Michael Thielscher)

Faculty of Computer Science, Institute of Artificial Intelligence, Computational Logic Group

Pure Prolog

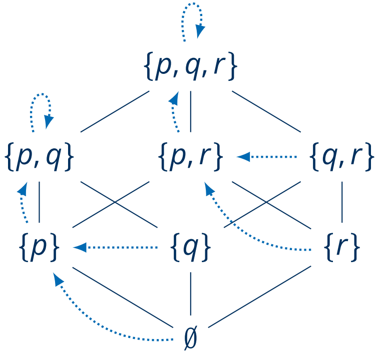
Lecture 6, 13th Nov 2023 // Foundations of Logic Programming, WS 2023/24

Previously . . .

- Definite Horn clauses possess the **model intersection property**.
- Thus each definite logic program has a **unique least Herbrand model**.
- The least fixpoint of a program's **one-step consequence operator** T_P coincides with its least Herbrand model.
- First-order clauses in combination with SLD resolution constitute a **Turing-complete** computation mechanism.

Consider the program
$P = \{p \leftarrow, \quad q \leftarrow p, \quad r \leftarrow r\}$.

The operator T_P maps as follows: $I \dashrightarrow T_P(I)$



Overview

Pure Prolog vs. Logic Programming

Lists in Prolog

Adding Arithmetics to Pure Prolog

Adding the Cut to Prolog

Pure Prolog vs. Logic Programming

Syntax of Pure Prolog

- clause $p(a) \leftarrow$ (fact) expressed in Prolog as

```
p(a).
```

- clause $p(x, a) \leftarrow q(x), r(x, y_i)$ expressed as

```
p(X,a) :- q(X), r(X,Yi).
```

- `% Comment`
- ambivalent syntax:

```
p(p(a,b), [c,p(a)]).
```

  predicate `p/2`, functions `p/1`, `p/2`
- anonymous variables:

```
p(X,a) :- q(X), r(X,_).
```

Specifics of Prolog

- leftmost selection rule $\rightsquigarrow$ **LD** resolution, **LD** resolvent, ...
- a program is a **sequence** of clauses
- unification without occur check
- **depth-first search** (with **backtracking**)

LD Trees and Prolog Trees

Finitely branching trees of queries, possibly marked with “success” or “failure”, produced as follows:

Definition

Let P be a program and Q_0 be a query.

- Start with tree $\mathcal{T}_{Q_0}$, which contains Q_0 as unique node
- **LD tree** for $P \cup \{Q_0\}$:
  repeatedly apply to current tree $\mathcal{T}$ and **every** unmarked leaf Q in $\mathcal{T}$ the operation $\mathit{expand}(\mathcal{T}, Q)$
  ($\rightsquigarrow$ LD tree obeys leftmost selection rule)
- **Prolog tree** for $P \cup \{Q_0\}$:
  repeatedly apply to current tree $\mathcal{T}$ and **leftmost** unmarked leaf Q in $\mathcal{T}$ the operation $\mathit{expand}(\mathcal{T}, Q)$
  ($\rightsquigarrow$ Prolog tree additionally obeys order of clauses and depth-first search)

Operation Expand

Definition

Operation *expand*($\mathcal{T}, Q$) is defined by:

- if $Q = \Box$, then mark Q with “success”;
- if Q has no LD-resolvents, then mark Q with “failure”;
- else add for each clause that is applicable to the **leftmost** atom of Q an LD-resolvent as descendant of Q. Respect the order in which the clauses appear in the program if a Prolog tree is constructed.

($\rightsquigarrow$ implements leftmost selection rule)

Outcomes of Prolog Computations (1)

Definition

Let P be a program and Q_0 be a query.

- Q_0 **universally terminates** $:\Longleftrightarrow$ LD tree for $P \cup \{Q_0\}$ is finite
- Q_0 **diverges** $:\Longleftrightarrow$ LD tree for $P \cup \{Q_0\}$ contains an infinite branch to the left of any success node
- Q_0 **potentially diverges**
  $:\Longleftrightarrow$ LD tree for $P \cup \{Q_0\}$ contains a success node, all branches to its left are finite, an infinite branch exists to its right
- Q_0 **produces infinitely many answers**
  $:\Longleftrightarrow$ LD tree for $P \cup \{Q_0\}$ has infinitely many success nodes, all infinite branches lie to the right of them
- Q_0 **fails** $:\Longleftrightarrow$ LD tree for $P \cup \{Q_0\}$ is finitely failed

Note: We assume here that also in LD trees the order of clauses in the program is respected.

Lists in Prolog

Some List Processing Predicates (1)

```
% app(Xs,Ys,Zs) :- Zs is the concatenation of lists Xs and Ys
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

% rev1(Xs,Ys) :- Ys is the reversal of list Xs
rev1([],[]).
rev1([X|Xs],Ys) :- rev1(Xs,Zs), app(Zs,[X],Ys).

% rev2(Xs,Ys) :- Ys is the reversal of list Xs
rev2(Xs,Ys) :- rev(Xs,[],Ys).
rev([],Ys,Ys).
rev([X|Xs],Ys,Zs) :- rev(Xs,[X|Ys],Zs).

% sub(Xs,Ys) :- Xs is a sublist of list Ys
sub(Xs,Ys) :- app(Xs,_,Zs), app(_,Zs,Ys).
```

Some List Processing Predicates (2)

```
% perm(Xs,Ys) :- Ys is a permutation of list Xs
perm([],[]).
perm(Xs,[X|Ys]) :- app(X1s,[X|X2s],Xs), app(X1s,X2s,Zs), perm(Zs,Ys).

% quick(Xs,Ys) :- Ys is obtained by sorting Xs using quicksort
quick([],[]).
quick([X|Xs],Ys) :- small(Xs,X,Ss), quick(Ss,X1s), great(Xs,X,Gs),
quick(Gs,X2s), app(X1s,[X|X2s],Ys).

small([],_,[]).
small([Y|Ys],X,[Y|Zs]) :- Y<X, small(Ys,X,Zs).
small([Y|Ys],X,Zs) :- Y>=X, small(Ys,X,Zs).

great([],_,[]).
great([Y|Ys],X,[Y|Zs]) :- Y>=X, great(Ys,X,Zs).
great([Y|Ys],X,Zs) :- Y<X, great(Ys,X,Zs).
```

Adding Arithmetics to Pure Prolog

Arithmetic Expressions

Definition

An **arithmetic expression** is a term over variables and the following function symbols:

`0`, `1`, `-1`, `2`, `-2`, ...	(nullary)
`−`, `abs`	(unary)
`+`, `−`, `∗`, `//`, `mod`	(binary)

A **ground arithmetic expression** (**gae**) is a variable-free arithmetic expression.

Comparison Relations and GAEs (1)

Comparison relations are defined only for gaes.

```
| ?- 5*2 > 3+4.
yes

| ?- [] < 5.
DOMAIN ERROR: []<5 - arg 1: expected expression, found []

| ?- X < 5.
INSTANTIATION ERROR: _33<5 - arg 1
```

Comparison Relations and GAEs (2)

Comparison relations are defined only for gaes.

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.

| ?- max(2, 3, Z).
Z = 3

| ?- max(Z, 7, 7).
INSTANTIATION ERROR: _33=<7 - arg 1

| ?- max(Z, 7, 8).
Z = 8
```

Evaluation of GAEs

The evaluation of gaes is triggered by the sub-query

s `is` t

- ▷ t is a gae with value $val(t)$
  ⇝ case distinction on s:
  - ⊳⊳ s is a gae syntactically identical to $val(t)$
    ⇝ sub-query succeeds with cas ε
  - ⊳⊳ s is a variable
    ⇝ sub-query succeeds with cas $\{s/val(t)\}$
- ▷ t is not a gae
  ⇝ runtime error

Evaluation of GAEs – Examples

```
| ?- 7 is 3+4.
yes

| ?- X is 3+4.
X = 7

| ?- 8 is 3+4.
no

| ?- 3+4 is 3+4.
no

| ?- X is Y+1.
INSTANTIATION ERROR: _36 is _33+1 - arg 2
```

Quiz: Lists and GAEs

Quiz

Consider the following Prolog program: . . .

Adding the Cut to Prolog

The Cut – Motivation (1)

Consider the following program:

```
countdown(X, Y, []) :- X < Y.
countdown(X, Y, [X|L]) :- X1 is X-1, countdown(X1, Y, L).
```

Upon query `countdown(`m`,`n`,L)` for $m, n \in \mathbb{N}$ with $m \geq n$, it produces the list $[m, m-1, \ldots, n]$.

However, it also produces $[m, m-1, \ldots, n, n-1]$, $[m, m-1, \ldots, n, n-1, n-2]$, ...

In this concrete case, we could replace the program by

```
countdown(X, Y, []) :- X < Y.
countdown(X, Y, [X|L]) :- X >= Y, X1 is X-1, countdown(X1, Y, L).
```

Problem solved?

The Cut – Motivation (2)

Imagine a similar program:

```
count_until(X, Y, []) :- complex_computation(X, Y).
count_until(X, Y, [X|L]) :- X1 is X-1, count_until(X1, Y, L).
```

What now?

- There might not be a negation of `complex_computation`.
- Even if there was, we would like to avoid repeating (parts of) the `complex_computation` in every clause body.

Solution: Remove unwanted answers by disallowing backtracking from certain points on – the cut.

```
count_until(X, Y, []) :- complex_computation(X, Y), !.
count_until(X, Y, [X|L]) :- X1 is X-1, count_until(X1, Y, L).
```

The Cut – Advantages and Disadvantages

The **cut** operator is a nullary predicate symbol, denoted by "!", which can prune off subtrees of Prolog trees.

Advantages:

▷ Efficiency gain, since search space is reduced.

▷ Simplification of programs (e.g. of programs dealing with sets).

Disadvantages:

▷ Major source of errors in Prolog programs (e.g. if successful branches are pruned off or wrong answers are delivered).

▷ Harder verification of programs, since procedural interpretation must be used (declarative interpretation cannot be used, since semantics of cut depends on leftmost selection rule and clause ordering).

Informal Semantics of Cut

Let P be a Prolog program containing exactly the following k clauses for a predicate p:

$$
\begin{array}{lcl}
p(t_{1,1}, \dots, t_{1,n}) & \leftarrow & \vec{A}_1 \\
\vdots & & \\
p(t_{i,1}, \dots, t_{i,n}) & \leftarrow & \vec{B}, !, \vec{C} \\
\vdots & & \\
p(t_{k,1}, \dots, t_{k,n}) & \leftarrow & \vec{A}_k
\end{array}
$$

Let some atom $p(t_1, \dots, t_n)$ in a query be resolved using the i-th clause for p and let later on the cut atom thus introduced become the leftmost atom. Then:

- The indicated occurrence of ! succeeds immediately.
- All other ways of resolving atoms of $\vec{B}$ are discarded.
- All derivations of $p(t_1, \dots, t_n)$ using the $(i + 1)$-st to k-th clause for p are discarded.

Formal Semantics of Cut

Definition

Let Q be a node in an initial fragment of a Prolog tree $\mathcal{T}$ with cut as leftmost atom in Q. The **origin** of this cut occurrence is the youngest ancestor of Q in $\mathcal{T}$ that contains less cut atoms than Q.

Roughly: origin $\triangleq$ query whose selected (leftmost) atom introduced the cut.

Definition

Prolog trees with cuts are constructed by extending the operation *expand*$(\mathcal{T}, Q)$:

- if $Q = \ !, \vec{A}$ and Q' is the origin of this cut occurrence, then add $\vec{A}$ as only direct descendant of Q and remove from $\mathcal{T}$ all the nodes that are descendants of Q' and lie to the right of the path connecting Q' and Q.

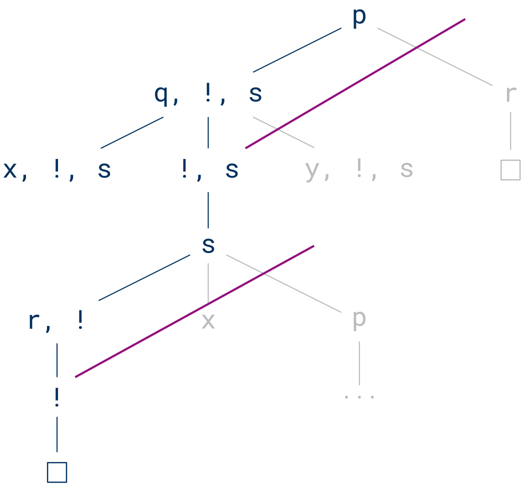
Cut: Example and Visualization (1)

```
p :- q, !, s.
p :- r.

q :- x.
q.
q :- y.

s :- r, !.
s :- x.
s :- p.

r.
```



Cut: Example and Visualization (2)

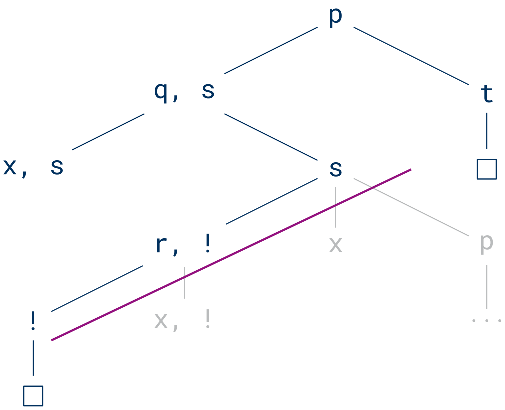
```
p :- q, s.
p :- t.

q :- x.
q.

s :- r, !.
s :- x.
s :- p.

r.
r :- x.

t.
```



Sets in Prolog (1)

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

set([],[]).
set([X|Xs],Ys) :- member(X,Xs), !, set(Xs,Ys).
set([X|Xs],[X|Ys]) :- set(Xs,Ys).

| ?- set([1,2,1],Us).
Us = [2,1] ? ;
no

| ?- set([1,2,1],[2,1]).
yes

| ?- set([1,2,1],[1,2]).
no
```

Sets in Prolog (2)

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

```
union([],Ys,Ys).
union([X|Xs],Ys,Zs) :- member(X,Ys), !, union(Xs,Ys,Zs).
union([X|Xs],Ys,[X|Zs]) :- union(Xs,Ys,Zs).
```

```
| ?- union([1,2],[1,3],Us).
Us = [2,1,3] ? ;
no
```

Incorrect Use of Cut (1)

Pruning Successful Branches

```
only_b(a) :- !, test(a).
only_b(b) :- !, test(b).
test(b).

| ?- only_b(a).
no

| ?- only_b(b).
yes

| ?- only_b(X).
no
```

Incorrect Use of Cut (2)

Allowing Wrong Answers

```
% max(X,Y,Z) :- Z is the maximum of X and Y
max(X,Y,Y) :- X =< Y, !.
max(X,_,X).

| ?- max(2,5,Z).
Z = 5

| ?- max(2,1,Z).
Z = 2

| ?- max(2,5,2).
yes
```

Meta-Variables

A **meta-variable** is a variable in the position of an atom.

Meta-variables must become instantiated before they are selected!

```
p(a).
a.

| ?- p(X), X.
X = a

| ?- p(X), Y.
INSTANTIATION ERROR: call(user:_34) - arg 1

| ?- X, p(X).
INSTANTIATION ERROR: call(user:_34) - arg 1
```

Using Meta-Variables

```
or(X, _) :- X.
or(_, Y) :- Y.
% or is also predefined in Prolog: :- op(1100, xfy, ;).
% or(X,Y) is written as:  X ; Y

if_then_else(P, Q, _) :- P, !, Q.
if_then_else(_, _, R) :- R.
% if_then_else is also predefined in Prolog: :- op(1050, xfy, ->).
% if_then_else(P,Q,R) is written as:  P -> Q ; R

not(X) :- X, !, fail.
not(_).
% atom fail always fails
% not is also predefined in Prolog: :- op(900, fy, \+).
% not(X) is written as:  \+ X
```

Conclusion

Summary

- Prolog employs SLD resolution with the **leftmost** selection rule ($\rightsquigarrow$ **LD** resolution), traverses the search space using **depth-first search** (with backtracking), and regards a program as a **sequence** of clauses.
- Prolog also offers list processing and arithmetics.
- The **cut** prunes certain branches of Prolog trees, and can lead to more efficient programs, but also to programming errors.

Suggested action points:

- Repair the incorrect uses of cut on slides 30 and 31 (using the cut correctly).
- Define a unary predicate `is_set` that succeeds iff its argument is a list of terms without duplicates. Write versions with(out) `!`, and with(out) `\+`.