Chapter 5

Pure PROLOG

Outline

- Pure PROLOG vs. logic programming
- Lists in Pure PROLOG
- Adding Arithmetics to Pure PROLOG
- Adding the Cut to Pure PROLOG

Syntax of Pure Prolog

- `p(X,a) :- q(X), r(X,Yi).` $\triangleq p(x, a) \leftarrow q(x), r(x,y_i)$
- `% Comment`
- Ambivalent syntax:

  `p(p(a,b), [c,p(a)])` $\triangleq p_1(p_2(a, b), [c, p_3(a)]) \leftarrow$

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

  `p(X,a) :- q(X), r(X,_)` $\triangleq p(x, a) \leftarrow q(x), r(x, y)$

Specifics of Prolog

- Leftmost selection rule
  LD-resolution, LD-resolvent, ...
- A program is a sequence of clauses
- Unification without occur check
- Depth-first search, backtracking

LD-Trees and Prolog Trees

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

- P program and Q_0 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 $expand(\mathcal{T}, Q)$
  (⋪| 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 $expand(\mathcal{T}, Q)$
  - (⋪| Prolog Tree additionally obeys order of clauses and depth-first search)

The Expand Operation

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. If a Prolog tree is constructed, respect the order in which the clauses appear in the program.

Outcomes of Prolog Computations (I)

Assume here that also in LD-trees the order in which the clauses appear in the program is respected:

- Q_0 universally terminates

  :⇔ LD-tree for $P \cup \{Q_0\}$ is finite

- Q_0 diverges

  :⇔ LD-tree for $P \cup \{Q_0\}$ contains an infinite branch to the left of any success node

- Q_0 potentially diverges

  :⇔ 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

Outcomes of Prolog Computations (II)

- Q_0 produces infinitely many answers

  :⇔ LD-tree for $P \cup \{Q_0\}$ has infinitely many success nodes, all infinite branches lie to the right of them

- Q_0 fails

  :⇔ LD-tree for $P \cup \{Q_0\}$ is finitely failed

Recap: The List Datastructure

[$\boldsymbol{a}_1$,..., $\boldsymbol{a}_n$]

[*head* | *tail*]

[**apples**,**pears**,**plums**]

= [**apples** | [**pears**,**plums**]]

```
member(X, [X | List]).
member(X, [Y | List]) :- member(X, List).
```

Some List Processing Predicates (I)

```
% 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 (II)

```
% 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) :- smaller(Xs,X,Ss), quick(Ss,X1s),
                    greater(Xs,X,Gs), quick(Gs,X2s),
                    app(X1s,[X|X2s],Ys).

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

Arithmetic Expressions

arithmetic expression

:⇔

term over variables and the following function symbols:

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

ground arithmetic expression (GAE)

:⇔ variable free arithmetic expression

Comparison Relations and GAEs (I)

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 (II)

```
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

$$s \texttt{ is } t$$

- t is a GAE with value $val(t) \Longrightarrow$
  - s is a GAE syntactically identical to $val(t)$
    $\Longrightarrow$ sub-query succeeds with CAS ϵ
  - s is a variable
    $\Longrightarrow$ sub-query succeeds with CAS $\{s/val(t)\}$
  - else $\Longrightarrow$ sub-query fails
- t is not a GAE $\Longrightarrow$ 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}
```

The Cut – Advantages and Disadvantages

Cut operator is 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:

- Main 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 the semantics of the 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:

$$p(t_{1,1}, \ldots, t_{1,n}) \leftarrow \underline{\underline{A}}_1$$

$$\ldots$$

$$p(t_{i,1}, \ldots, t_{i,n}) \leftarrow \underline{\underline{B}}, !, \underline{\underline{C}}$$

$$\ldots$$

$$p(t_{k,1}, \ldots, t_{k,n}) \leftarrow \underline{\underline{A}}_k$$

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

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

Formal Semantics of Cut

Let Q be a node in an initial fragment of a Prolog tree $\mathcal{T}$ with the cut as leftmost atom.
Origin of this cut-occurrence :⇔

youngest ancestor of Q in $\mathcal{T}$ that contains less cut atoms than Q

Construction of Prolog trees with cuts by extending the operation $expand(\mathcal{T}, Q)$ (cf. Slide 6):

- if $Q = !, \underline{\underline{A}}$ and Q' is origin of this cut-occurrence, then add $\underline{\underline{A}}$ 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.

Using the Cut: Sets in Prolog (I)

```
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
```

Using the Cut: Sets in Prolog (II)

```
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: Successful Branches Pruned off

```
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: 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
```

Objectives

- Pure PROLOG vs. logic programming
- Lists in Pure PROLOG
- Adding Arithmetics to Pure PROLOG
- Adding the Cut to Pure PROLOG