

Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016



Outline

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

Full Prolog

semantics (revisited), cuts, correctness proofs, meta-logical predicates, nondeterministic programming, efficient programs, complexity

Summary of Last Lecture

Answer Set Programming

- novel approach to modelling and solving search and optimisation problems
- \neg programming, but a specification language
- \neg Turing complete
- purely declarative
- restricted to finite models

Example ((part of) 8-queens problem)

```
:- not (1 = count(Y : queen(X,Y))), row(X)
```

- expresses that exactly one queen appears in every row and column
- is read as a rule: “if X is a row, $1 = \text{count}(Y : \text{queen}(X,Y))$ holds”

Negative Definitions

Definition

negative definitions define a relation with the help of negation

Example

```
land(X) :- not sea(X).
```

Fact

negative definitions are dangerous as their scope is usually larger than expected and they are difficult to maintain, if underlying definitions get refined

Example

```
:- land(27).
:- land(kar_rinne).
:- land(milka_kuh).
```

Semantics (revisited)

Definitions

- SLD-derivation of **monotone logic program** P and goal clause G consists of
 - 1 maximal sequence G_0, G_1, G_2, \dots of goal clauses
 - 2 sequence C_0, C_1, C_2, \dots of variants of rules in P
 - 3 sequence $\theta_0, \theta_1, \theta_2, \dots$ of substitutions
 such that
 - $G_0 = G$
 - G_{i+1} is resolvent of G_i and C_i with mgu θ_i
 - C_i has no variables in common with G, C_0, \dots, C_{i-1}
- SLD refutation is finite SLD derivation ending in \square
- computed answer substitution of SLD refutation of P and G with substitutions $\theta_0, \theta_1, \dots, \theta_m$ is restriction of $\theta_0\theta_1 \dots \theta_m$ to variables in G

Definition (search tree)

a search tree (aka SLD tree) of a goal G is a tree T such that

- the root of T is labelled with G ; the nodes of T are labelled with conjunctions of goals, where one goal is selected (**wrt a selection function**)
- \exists edge from node N for each clause, whose head unifies with the selected goal; edges are labelled with (partial) answer substitutions
- leaves are success nodes, if \square has been reached or failure nodes otherwise

Definition (proof tree)

a proof tree for a program P and a goal G is a tree, whose nodes are goals and whose edges represent reduction of goals such that

- the root is the query G
- the edges are labelled with (partial) answer substitutions
- a proof tree for G_1, \dots, G_n is set of proof trees for G_i

Monotone Logic Programs and Herbrand Models

(yet another connection between proofs and programs)

Definitions

- the **Herbrand universe** for a program P is the set of all closed terms built from constants and function symbols appearing in the program
- the **Herbrand base** is the set of all ground goals formed from predicates in P and terms in the Herbrand universe
- an **interpretation** is a subset of the Herbrand base
- an interpretation I is a **model** if it is closed under rules:

$$\forall \text{ rules } A : -B_1, \dots, B_n : \text{ if } B_1, \dots, B_n \in I, \text{ then } A \in I$$
- the **minimal** model of P is the intersection of all models

Theorem

the minimal model is unique

Declarative, Operational, and Denotational Semantics

Definition

- the **declarative** semantics of P (aka its **meaning**) is the minimal model of P
- we also say that the **meaning** of a logic program P , is the set of (ground unit) goals deducible from P

Definitions

the **operational** semantics describes the meaning of a program procedurally

Definition

the **denotational** semantics assign meanings to programs based on associating with the program a function over the domain computed by the program

Rule Order

Fact

The rule order determines the order in which solutions are found

Example

```
parent(terach,abraham).      parent(abraham,isaac).
parent(isaac,jakob).         parent(jakob,benjamin).

ancestor1(X,Y) :- parent(X,Y).
ancestor1(X,Z) :- parent(X,Y), ancestor1(Y,Z).
```

Example

```
append1([X|Xs],Ys,[X|Zs]) :-      append2([],Ys,Ys).
      append1(Xs,Ys,Zs).           append2([X|Xs],Ys,[X|Zs]) :-
append1([],Ys,Ys).                 append2(Xs,Ys,Zs).
```

Goal Order

Fact

Goal order determines the SLD tree

Example

```
grandparent1(X,Z) :- parent(X,Y), parent(Y,Z).
grandparent2(X,Z) :- parent(Y,Z), parent(X,Y).
```

Example

```
reverse1([X|Xs],Zs) :- reverse1(Xs,Ys), append1(Ys,[X],Zs).
reverse1([],[]).
```

```
reverse2([X|Xs],Zs) :- append1(Ys,[X],Zs), reverse2(Xs,Ys).
reverse2([],[]).
```

```
:- reverse1([a,b,c,d],Xs), Xs=[d,c,b,a].
:- reverse2([a,b,c,d],Xs), Xs=[d,c,b,a].
```

Redundant Solutions

Example

```
minimum(N1,N2,N1) :- N1 <= N2.
minimum(N1,N2,N2) :- N2 <= N1.

:- minium(2,2,M)
```

Example

```
minimum(N1,N2,N1) :- N1 <= N2.
minimum(N1,N2,N2) :- N2 < N1.
```

Observation

similar care is necessary with the definition of **partition**, etc.

Redundant Solutions (part II)

Example

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

```
?- member(X,[a,b,a]).
```

```
X ↦ a ;
```

```
X ↦ b ;
```

```
X ↦ a ;
```

```
false
```

Example

```
member_check(X,[X|Xs]).
member_check(X,[Y|Ys]) :- X ≠ Y, member_check(X,Ys).
```

Fact

some care is necessary in pruning the search tree, as this may change the meaning of a program

Example

```
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

Example

```
select_fst(X, [X|Xs], Xs).
select_fst(X, [Y|Ys], [Y|Zs]) :- dif(X, Y), select_fst(X, Ys, Zs).
```

Observation

`select(a, [a,b,a,c], [a,b,c])` is in the meaning of the 1st program;
`select_fst(a, [a,b,a,c], [a,b,c])` is **not** in the meaning of the 2nd

Removal of Duplicates

```
no_doubles([], []).
no_doubles([X|Xs], Ys) :-
    member(X, Xs), !,                cut
    no_doubles(Xs, Ys).
no_doubles([X|Xs], [X|Ys]) :-
    no_doubles(Xs, Ys).

:- no_doubles([a,b,a,c,b], X).
X ↦ [a,c,b] ;
false
```

Effect of Cut

- ! succeeds
- ! fixes all choices between (and including) moment of matching rule's head with parent goal and cut
- if backtracking reaches !, the cut fails and the search continues from the last choice made before the clause containing ! was chosen

Examples of Cuts

Example (With Cuts)

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X < Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
    X = Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y, !, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.
```

Example

```
minimum(X, Y, X) :- X ≤ Y, !.
minimum(X, Y, Y) :- X > Y, !.
```

Fact

cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) :- X ≤ Y, ordered([Y|Xs]).

bubblesort(Xs, Ys) :-
    append(As, [X,Y|Bs], Xs),
    X > Y, !,
    append(As, [Y,X|Bs], Xs1),
    bubblesort(Xs1, Ys).
bubblesort(Xs, Xs) :-
    ordered(Xs), !.

:- bubblesort([3,2,1], Xs)
Xs ↦ [1,2,3]
```

Definition (Negation as Failure)

- negation $\backslash +$ is implemented using cut
- the principle of negation is limited and known as **negation as failure**

Example

```
not X :- X, !, fail.
not X.
```

Observation

if G does not terminate, $\text{not}(G)$ may or may not terminate

Example

```
married(abraham,sarah).
married(X,Y) :- married(Y,X)
:- not married(abraham,sarah).
```

Cut-Fail Combinations

Example (Implementing \neq)

```
X  $\neq$  X  $\rightarrow$  !, fail.
X  $\neq$  Y.
```

Example (Implementing if_then_else)

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

Example (Implementing same_vars)

```
same_vars(foo,Y) :- var(Y), !, fail.
same_vars(X,Y) :- var(X), var(Y).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) :- nl.

row(A,B,_ ) :- wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) :- E, !, wr(true), nl.
row(_,_,_) :- wr(false), nl.

wr(true) :- write('T').
wr(false) :- write('F').

:- table(A,B,or(A,implies(B,or(B,and(A,B))))).
:- table(A,B,false).
```

Cut and Generate and Test

Example (integer division with cut)

```
is_integer(0).
is_integer(N) :-
    is_integer(N1),
    N is N1 + 1.

divide(N1,N2,Result) :-
    is_integer(Result),
    Product1 is Result * N2,
    Product2 is (Result+1)*N2,
    Product1 <= N1,
    Product2 > N1,
    !. /* what happens if removed? */

:- divide(27,6,Res), Res=4.
```