Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Definitions

- SLD-derivation of logic program P and goal clause G consists of
  1. maximal sequence $G_0, G_1, G_2, \ldots$ of goal clauses
  2. sequence $C_0, C_1, C_2, \ldots$ of variants of rules in P
  3. sequence $\theta_0, \theta_1, \theta_2, \ldots$ 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 $G, C_0, \ldots, C_{i-1}$
- SLD-refutation is finite SLD-derivation ending in $\square$
- computed answer substitution of SLD-refutation of P and G with substitutions $\theta_0, \theta_1, \ldots, \theta_m$ is restriction of $\theta_0\theta_1 \cdots \theta_m$ to variables in G

Definitions

- an interpretation is a subset of the Herbrand base
- an interpretation I is a model if it is closed under rules:

$$\forall A \leftarrow B_1, \ldots, B_n \quad A \in I, \text{ if } B_1, \ldots, B_n \in I$$

- the minimal model of P is the intersection of all models; the minimal model is unique

Definition

the declarative semantics of P (aka its meaning) is the minimal model of P

Definitions

- an interpretation is a subset of the Herbrand base
- an interpretation I is a model if it is closed under rules:

$$\forall A \leftarrow B_1, \ldots, B_n \quad A \in I, \text{ if } B_1, \ldots, B_n \in I$$

- the minimal model of P is the intersection of all models; the minimal model is unique

Definition

the declarative semantics of P (aka its meaning) is the minimal model of P

Definitions

the operational semantics describes the meaning of a program procedurally

Definitions

- an interpretation is a subset of the Herbrand base
- an interpretation I is a model if it is closed under rules:

$$\forall A \leftarrow B_1, \ldots, B_n \quad A \in I, \text{ if } B_1, \ldots, B_n \in I$$

- the minimal model of P is the intersection of all models; the minimal model is unique

Definition

the declarative semantics of P (aka its meaning) is the minimal model of P

Definition

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

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure Prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure Prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming

The Execution Model of Prolog

One Choice

goal in sequence of goals – any choice will do

2 rule in logic program

substitution – avoid choice by always taking mgu

The Execution Model of Prolog

One Choice

	goal in sequence of goals	–	any choice will do
2	rule in logic program		
	substitution	–	avoid choice by always taking mgu

Execution

- Prolog programs are executed using SLD resolution
  - leftmost and topdown selection
  - depth-first search with backtracking
- unification without occur check

Prolog Mode for Emacs

Bruda's Prolog Mode

1. goto `http://bruda.ca/emacs/prolog_mode_for_emacs`
2. download prolog.el, compile and put into sub-directory site-lisp
3. put the following into .emacs:

```
(autoload 'run-prolog "prolog"
          "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog"
           "Major mode for editing Prolog programs." t)
(setq prolog-system 'swi)
(setq auto-mode-alist
      (cons (cons "\\.pl" 'prolog-mode) auto-mode-alist))
```

Comparison to Conventional Programming Languages

Fact

a programming language is characterised by its control and data manipulation mechanisms

Comparison to Conventional Programming Languages

Fact

a programming language is characterised by its control and data manipulation mechanisms

Control

$$A \leftarrow B_1, \dots, B_n$$

procedure A
    call B_1
    call B_2
    $\vdots$
    call B_n
end

Comparison to Conventional Programming Languages

Fact

a programming language is characterised by its control and data manipulation mechanisms

Control

$$A \leftarrow B_1, \dots, B_n$$

```
procedure A
      call B_1
      call B_2
        ⋮
      call B_n
end
```

Observations

1. goal invocation corresponds to procedure invocation
2. differences show when backtracking occurs

Data Structures

1. data structures manipulated by logic programs (= terms) correspond to general record structures
2. like LISP, Prolog is a declaration free, typeless language
3. Prolog does not support destructive assignment where the content of the initialised variable can change

Data Structures

1. data structures manipulated by logic programs (= terms) correspond to general record structures
2. like LISP, Prolog is a declaration free, typeless language
3. Prolog does not support destructive assignment where the content of the initialised variable can change

Data Manipulation

1. data manipulation is achieved via unification
2. unification subsumes
   - single assignment
   - parameter passing
   - record allocation
   - read/write-once field access in records

Rule Order

Observation

The rule order determines the order in which solutions are found

Rule Order

Observation

The rule order determines the order in which solutions are found

Example

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

ancestor(X,Y) ← parent(X,Y).
ancestor(X,Z) ← parent(X,Y), ancestor(Y,Z).
```

Example

```
append([X|Xs],Ys,[X|Zs]) ←       append([],Ys,Ys).
    append(Xs,Ys,Zs).            append([X|Xs],Ys,[X|Zs]) ←
append([],Ys,Ys).                    append(Xs,Ys,Zs).
```

Example

```
is_list([]).   is_list([X|Xs]) ← is_list(Xs).
```

Definitions

- a list is complete if every instances satisfies the above type for lists
- otherwise it is incomplete

Example

`is_list([]).   is_list([X|Xs]) ← is_list(Xs).`

Definitions

- a list is complete if every instances satisfies the above type for lists
- otherwise it is incomplete

Example

- the lists `[a,b,c]` and `[a,X,c]` are complete
- the list `[a,b|Xs]` is not

Example

`is_list([]).   is_list([X|Xs]) ← is_list(Xs).`

Definitions

- a list is complete if every instances satisfies the above type for lists
- otherwise it is incomplete

Example

- the lists [a,b,c] and [a,X,c] are complete
- the list [a,b|Xs] is not

Definition

a domain is a set of goals closed under the instance relation

Termination

Observation

Prolog may fail to find a solution to a goal, even though the goal has a finite computation

Termination

Observation

Prolog may fail to find a solution to a goal, even though the goal has a finite computation

Definition

a termination domain of a program P is a domain D such that P terminates on all goals in D

Termination

Observation

Prolog may fail to find a solution to a goal, even though the goal has a finite computation

Definition

a termination domain of a program P is a domain D such that P terminates on all goals in D

Example

consider adding *married/2* to the family database, and the following "obvious" closure under commutativity:

```
married(X,Y) ← married(Y,X).
```

Termination

Observation

Prolog may fail to find a solution to a goal, even though the goal has a finite computation

Definition

a termination domain of a program P is a domain D such that P terminates on all goals in D

Example

consider adding *married/2* to the family database, and the following "obvious" closure under commutativity:

```
married(X,Y) ← married(Y,X).
```

NB: recursive rules which have the recursive goal as the first goal in the body are called left recursive

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

consider *append/3*, where the fact comes after the rule

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

consider *append/3*, where the fact comes after the rule

1. *append* terminates if the first argument is a complete list

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

consider *append/3*, where the fact comes after the rule

1. *append* terminates if the first argument is a complete list
2. *append* terminates if the third argument is complete

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

consider *append/3*, where the fact comes after the rule

1. *append* terminates if the first argument is a complete list
2. *append* terminates if the third argument is complete
3. *append* terminates iff the first or third argument is complete

Example

```
are_married(X,Y) ← married(X,Y).
are_married(X,Y) ← married(Y,X).
```

Example

consider *append/3*, where the fact comes after the rule

1. *append* terminates if the first argument is a complete list
2. *append* terminates if the third argument is complete
3. *append* terminates iff the first or third argument is complete

Proof of the First Fact.

- consider generic call: ← `append(Xs,Ys,Zs)`,
  where `Xs` is complete list; define $\|\leftarrow \texttt{append(Xs,Ys,Zs)}\| = \|\texttt{Xs}\|$
- $\|G\|$ decreases in every successor node of goal G in the SLD tree

∎

Goal Order

Observation

Goal order determines the SLD tree

Goal Order

Observation

Goal order determines the SLD tree

Example

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
grandparent2(X,Z) ← parent(Y,Z), parent(X,Y).
```

Goal Order

Observation

Goal order determines the SLD tree

Example

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
grandparent2(X,Z) ← parent(Y,Z), parent(X,Y).
```

Example

```
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
reverse([],[]).
```

Goal Order

Observation

Goal order determines the SLD tree

Example

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
grandparent2(X,Z) ← parent(Y,Z), parent(X,Y).
```

Example

```
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
reverse([],[]).
```

Example

```
sublist(Xs,AsXsBs) ←
    append(AsXs,Bs,AsXsBs), append(As,Xs,AsXs).
```

Goal Order

Observation

Goal order determines the SLD tree

Example

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
grandparent2(X,Z) ← parent(Y,Z), parent(X,Y).
```

Example

```
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
reverse([],[]).
```

Example

```
sublist(Xs,AsXsBs) ←
    append(As,Xs,AsXs), append(AsXs,Bs,AsXsBs).
```

Redundant Solutions

Example

$\texttt{minimum}(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$
$\texttt{minimum}(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

$\leftarrow \texttt{minium}(2,2,M)$

Redundant Solutions

Example

$\texttt{minimum}(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$
$\texttt{minimum}(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

$\leftarrow \texttt{minium}(2,2,M)$

Example

$\texttt{minimum}(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$
$\texttt{minimum}(N_1,N_2,N_2) \leftarrow N_2 < N_1.$

Redundant Solutions

Example

$\texttt{minimum}(\texttt{N}_1,\texttt{N}_2,\texttt{N}_1) \leftarrow \texttt{N}_1 \leqslant \texttt{N}_2.$
$\texttt{minimum}(\texttt{N}_1,\texttt{N}_2,\texttt{N}_2) \leftarrow \texttt{N}_2 \leqslant \texttt{N}_1.$

$\leftarrow \texttt{minium(2,2,M)}$

Example

$\texttt{minimum}(\texttt{N}_1,\texttt{N}_2,\texttt{N}_1) \leftarrow \texttt{N}_1 \leqslant \texttt{N}_2.$
$\texttt{minimum}(\texttt{N}_1,\texttt{N}_2,\texttt{N}_2) \leftarrow \texttt{N}_2 < \texttt{N}_1.$

Observation

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

Example

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

Example

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

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

X ↦ a

Example

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

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

Example

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

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

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(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).
```

Recursive Programming in Pure Prolog

Fact

some care is necessary in pruning the search tree

Recursive Programming in Pure Prolog

Fact

some care is necessary in pruning the search tree

Example

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

Recursive Programming in Pure Prolog

Fact

some care is necessary in pruning the search tree

Example

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

Example

```
select_first(X,[X|Xs],Xs).
select_first(X,[Y|Ys],[Y|Zs]) ← X ≠ Y, select_first(X,Ys,Zs).
```

Recursive Programming in Pure Prolog

Fact

some care is necessary in pruning the search tree

Example

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

Example

```
select_first(X,[X|Xs],Xs).
select_first(X,[Y|Ys],[Y|Zs]) ← X ≠ Y, select_first(X,Ys,Zs).
```

Observation

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

Example

```
members([X|Xs],Ys) ← member(X,Ys), members(Xs,Ys).
members([],Ys).
```

Example

```
members([X|Xs],Ys) ← member(X,Ys), members(Xs,Ys).
members([],Ys).
```

Example

```
selects([X|Xs],Ys) ← select(X,Ys,Ys1), selects(Xs,Ys1).
selects([],Ys).
```

Example

```
members([X|Xs],Ys) ← member(X,Ys), members(Xs,Ys).
members([],Ys).
```

Example

```
selects([X|Xs],Ys) ← select(X,Ys,Ys1), selects(Xs,Ys1).
selects([],Ys).
```

Observations

1. *members/2* ignores the multiplicity of elements
2. *members/2* terminates iff 1st argument is complete
3. the first restriction is lifted, the second altered with *selects/2*
4. *selects/2* terminates iff 2nd argument is complete

Example

```
%       no_doubles(Xs,Ys) <--
%           Ys is the list obtained by removing duplicate
%           elements from the list Xs
```

Example

```
%       no_doubles(Xs,Ys) <--
%           Ys is the list obtained by removing duplicate
%           elements from the list Xs
```

Example

```
non_member(X,[Y|Ys]) ← X ≠ Y, non_member(X,Ys).
non_member(X,[]).
```

Example

```
%      no_doubles(Xs,Ys) <--
%          Ys is the list obtained by removing duplicate
%          elements from the list Xs
```

Example

```
non_member(X,[Y|Ys]) ← X ≠ Y, non_member(X,Ys).
non_member(X,[]).

no_doubles([X|Xs],Ys) ←
    member(X,Xs), no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    non_member(X,Xs), no_doubles(Xs,Ys).
no_doubles([],[]).
```

Built-in Predicates for List Manipulation

- `append/3`
- `member/2`

Built-in Predicates for List Manipulation

- `append/3`
- `member/2`
- `last/2`

```
?- last([a,b,c,d],X).
X = d
```

Built-in Predicates for List Manipulation

- `append/3`
- `member/2`
- `last/2`

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

- reverse/2

```
?- reverse([a,b,c,d],X).
X = [d,c,b,a]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

- reverse/2

```
?- reverse([a,b,c,d],X).
X = [d,c,b,a]
```

- select/3

```
?- select(b,[a,b,c,d],X).
X = [a,c,d]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

- reverse/2

```
?- reverse([a,b,c,d],X).
X = [d,c,b,a]
```

- select/3

```
?- select(b,[a,b,c,d],X).
X = [a,c,d]
```

```
?- select(b,[a,b,c,b,d],X).
X = [a,c,b,d]
```

Built-in Predicates for List Manipulation

- append/3
- member/2
- last/2

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

- reverse/2

```
?- reverse([a,b,c,d],X).
X = [d,c,b,a]
```

- select/3

```
?- select(b,[a,b,c,d],X).
X = [a,c,d]
```

```
?- select(b,[a,b,c,b,d],X).
X = [a,c,b,d]
```

- length/2

```
?- length([a,b,c,d],X).
X = 4
```