Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016

Happy New Year!

Summary of Last Lecture

Definition

the intended meaning of a Prolog program is a set of ground facts G

Definition

a program P is called

- correct with respect to the intended meaning M, if the meaning of P is a subset of M
- complete if the intended meaning M is a subset of the meaning of P

Definition

let P be a Prolog program and Q be a query; the search tree visit and construction algorithm A generates a search tree (T, N, U) as follows:

1. initially the root becomes current node N, labelled with Q and ϵ
2. if the current sequence of goals Q is `true` backtrack to the first node in U (U is always updated by using a depth-first, leftmost strategy)
3. otherwise, let T be the first goal in Q
4. if $T = \texttt{true}$, delete T and goto Step 2
5. if T is user-defined, either expand the tree by n successor nodes, where n is the number of clauses $H_i :- B_i$ such that H_i unifies with T or backtrack; in the former case the successors are labelled by $Q \setminus \{T\} \cup B_i$, the leftmost child becomes the current node, update U
6. if T is built-in, perform the specific side effects of the predicate and goto Step 2

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 (revisted), cuts, correctness proofs, meta-logical predicates, nondeterministic programming, pragmatics, efficient programs, meta programming

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 (revisted), cuts, correctness proofs, meta-logical predicates, nondeterministic programming, pragmatics, efficient programs, meta programming

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

1. variables in system predicates do not behave as intended

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

1. variables in system predicates do not behave as intended
2. (logical) variables can be accidentally instantiated

Meta-logical Type Predicates

Definition

- `var`(*Term*) is true if *Term* is at present an uninstantiated variable
- `nonvar`(*Term*) is true if *Term* is at present not a variable
- `ground`(*Term*) is true if *Term* does not contain variables
- `compound`(*Term*) is true if *Term* is compound

Meta-logical Type Predicates

Definition

- `var`(*Term*) is true if *Term* is at present an uninstantiated variable
- `nonvar`(*Term*) is true if *Term* is at present not a variable
- `ground`(*Term*) is true if *Term* does not contain variables
- `compound`(*Term*) is true if *Term* is compound

Example

```
plus(X,Y,Z) :-
    nonvar(X), nonvar(Y), Z is X + Y.
plus(X,Y,Z) :-
    nonvar(X), nonvar(Z), Y is Z - X.
plus(X,Y,Z) :-
    nonvar(Y), nonvar(Z), X is Z - Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).

term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).
term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
unify_args(N,X,Y) :-
    N > 0, unify_arg(N,X,Y), N1 is N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).

term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :-
    N > 0, unify_arg(N,X,Y), N1 is N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).

unify_arg(N,X,Y) :-
    arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Remark

alternative sto the above (and below) implementation of unify:

- Term1 = Term2
- unify_with_occurs_check (Term1,Term2)

Remark

alternative sto the above (and below) implementation of `unify`:

- `Term1 = Term2`
- `unify_with_occurs_check (Term1,Term2)`

Definition (Comparing nonground terms)

- X == Y is true if X and Y are identical constants, variables, or compound terms
- X \== Y is true if X and Y are not identical

Remark

alternative sto the above (and below) implementation of unify:

- Term1 = Term2
- unify_with_occurs_check (Term1,Term2)

Definition (Comparing nonground terms)

- X == Y is true if X and Y are identical constants, variables, or compound terms
- X \== Y is true if X and Y are not identical

Example

```
:- X == 5
false
```

Unification with Occurs Check

Example

```
not_occurs_in(X,Y) :-
    var(Y), X \== Y.
not_occurs_in(X,Y) :-
    nonvar(Y), constant(Y).
not_occurs_in(X,Y) :-
    nonvar(Y), compound(Y),
    functor(Y,F,N), not_occurs_in(N,X,Y).
not_occurs_in(N,X,Y) :-
    N > 0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N - 1,
    not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).
```

Unification with Occurs Check

Example

```
not_occurs_in(X,Y) :-
    var(Y), X \== Y.
not_occurs_in(X,Y) :-
    nonvar(Y), constant(Y).
not_occurs_in(X,Y) :-
    nonvar(Y), compound(Y),
    functor(Y,F,N), not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :-
    N > 0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N - 1,
    not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).

unify(X,Y) :- var(X), nonvar(Y), not_occurs_in(X,Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), not_occurs_in(Y,X), Y = X.
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Other Control Predicates

- *fail*/0 *false*/0

```
:- fail.        :- false.
false           false
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Other Control Predicates

- *fail*/0 *false*/0

```
:- fail.        :- false.
false           false
```

- *true*/0

```
:- true.
true
```

Clause Database Operations

- *assert/1*

  $\leftarrow$ `assert(`C`).`

  `true`

Clause Database Operations

- *assert/1*

  $\leftarrow$ `assert(`C`).`

  `true`

- side effect: add rule C to program

Clause Database Operations

- *assert/1*

  $\leftarrow$ `assert(`C`).`
  `true`

- side effect: add rule C to program
- *asserta/1* or *assertz/1*

  $\leftarrow$ `asserta(`C`).`
  `true`

  add C first (last) to the database

Clause Database Operations

- *assert/1*

  ```
  ← assert(C).
  true
  ```

- side effect: add rule C to program
- *asserta/1* or *assertz/1*

  ```
  ← asserta(C).
  true
  ```

  add C first (last) to the database

- *retract/1* or *retractall/1*

  ```
  ← retract(C).
  false
  ```

- side effect: remove first rule (all rules) from program that unifies with C

Example (Fibonacci Numbers Revisited)

```
:- dynamic(fibonacci/2).
```

Example (Fibonacci Numbers Revisited)

```
:- dynamic(fibonacci/2).

fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X is Y+Z,
    assert(fibonacci(N,X)),
    !.
```

Example (Fibonacci Numbers Revisited)

```
:- dynamic(fibonacci/2).

fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X is Y+Z,
    asserta(fibonacci(N,X)),
    !.
```

Second-Order Programming

Definitions

- the predicate *bagof(Template,Goal,Bag)* unifies *Bag* with the alternatives of *Template* that meet *Goal*

Second-Order Programming

Definitions

- the predicate *bagof(Template,Goal,Bag)* unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack

Second-Order Programming

Definitions

- the predicate *bagof(Template,Goal,Bag)* unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions

Second-Order Programming

Definitions

- the predicate *bagof(Template,Goal,Bag)* unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var*^*Goal* tells *bagof* to existentially quantify *Var*

Second-Order Programming

Definitions

- the predicate *bagof(Template,Goal,Bag)* unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var^Goal* tells *bagof* to existentially quantify *Var*
- the predicate *setof(Template,Goal,Bag)* is similar to *bagof* but sorts the obtained multi-set (bag) and removed duplicates

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*,*Goal*,*Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var*^*Goal* tells *bagof* to existentially quantify *Var*
- the predicate *setof*(*Template*,*Goal*,*Bag*) is similar to *bagof* but sorts the obtained multi-set (bag) and removed duplicates

Definition

the predicate *findall*(*Template*,*Goal*,*Bag*) works as *bagof* if all excessive variables are existentially quantified

Applications of Set Predicates

Example

```
no_doubles(Xs,Ys) :- setof(X,member(X,Xs),Ys).

:- no_doubles([1,2,3,3],[1,2,3]).
```

Applications of Set Predicates

Example

```
no_doubles(Xs,Ys) :- setof(X,member(X,Xs),Ys).

:- no_doubles([1,2,3,3],[1,2,3]).
```

Example

```
no_doubles_wrong(Xs,Ys) :- bagof(X,member(X,Xs),Ys).

:- no_doubles_wrong([1,2,3,3],[1,2,3,3]).
```

Example (Facts)

```
father(andreas,boris).
father(andreas,christian).
father(andreas,doris).
father(boris,eva).
father(franz,georg).

female(doris).
female(eva).

mother(doris,franz).
mother(eva,georg).

male(andreas).
male(boris).
male(christian).
male(franz).
male(georg).
```

Example (Facts)

```
father(andreas,boris).          female(doris).          male(andreas).
father(andreas,christian).      female(eva).            male(boris).
father(andreas,doris).                                  male(christian).
father(boris,eva).              mother(doris,franz).    male(franz).
father(franz,georg).            mother(eva,georg).      male(georg).
```

Example

```
children(X,Cs) :- children(X,[],Cs).

children(X,A,Cs) :-
    father(X,C), children(X,[C|A],Cs).
children(X,Cs,Cs).
```

Example (Facts)

```
father(andreas,boris).
father(andreas,christian).
father(andreas,doris).
father(boris,eva).
father(franz,georg).

female(doris).
female(eva).

mother(doris,franz).
mother(eva,georg).

male(andreas).
male(boris).
male(christian).
male(franz).
male(georg).
```

Example

```
children(X,Cs) :- children(X,[],Cs).

children(X,A,Cs) :-
    father(X,C), children(X,[C|A],Cs).
children(X,Cs,Cs).
```

Example (cont'd)

```
children(X,Kids) :- setof(C,father(X,C),Kids).
children(AllKids) :- setof(C,X^father(X,C),AllKids).
children2(AllKids) :- setof(C,father(_X,C),AllKids).
```

Recall Propositional Tableaux

Example

consider the tableau proof of $P \to (Q \to R)) \to (P \lor S \to (Q \to R) \lor S)$

$$\neg((P \to (Q \to R)) \to (P \lor S \to (Q \to R) \lor S))$$

$$P \to (Q \to R)$$

$$\neg(P \lor S \to (Q \to R) \lor S)$$

$$P \lor S$$

$$\neg((Q \to R) \lor S)$$

$$\neg(Q \to R)$$

$$\neg S$$

$$\begin{array}{ccc} & \neg P & & Q \to R \\ P & & S & \end{array}$$

Free-Variable Semantic Tableaux

Definition (expansion rules)

$$\frac{\gamma}{\gamma(x)} \quad x \text{ a free variable} \qquad\qquad \frac{\delta}{\delta(f(x_1, \ldots, x_n))} \quad f \text{ a Skolem function}$$

- $x_1, \ldots, x_n$ denote all free variables of the formula δ
- Skolem function f must be new on the branch

Free-Variable Semantic Tableaux

Definition (expansion rules)

$$\frac{\gamma}{\gamma(x)} \quad x \text{ a free variable} \qquad \frac{\delta}{\delta(f(x_1, \dots, x_n))} \quad f \text{ a Skolem function}$$

- $x_1, \dots, x_n$ denote all free variables of the formula δ
- Skolem function f must be new on the branch

Definition (atomic closure rule)

1. $\exists$ branch in tableau T that contains two literals A and $\neg B$
2. $\exists$ mgu σ of A and B
3. then $T\sigma$ is also a tableau

Example

consider the tableau proof of

$$\exists x \forall y R(x, y) \rightarrow \forall y \exists x R(x, y)$$

and

$$\forall x \forall y (P(x) \land P(y)) \rightarrow \forall x \forall y (P(x) \lor P(y))$$

on the blackboard

Definition

a strategy S details:

1. which expansion rule is supposed to be applied
2. or that no expansion rule can be applied

a strategy may use extra information which is updated

Definition

a strategy S details:

1. which expansion rule is supposed to be applied
2. or that no expansion rule can be applied

a strategy may use extra information which is updated

Definition

a strategy S is fair if for sequence of tableaux $T_1, T_2, \ldots$ following S :

Definition

a strategy S details:

1. which expansion rule is supposed to be applied
2. or that no expansion rule can be applied

a strategy may use extra information which is updated

Definition

a strategy S is fair if for sequence of tableaux $T_1, T_2, \ldots$ following S :

1. any non-literal formula in T_i is eventually expanded, and

Definition

a strategy S details:

1. which expansion rule is supposed to be applied
2. or that no expansion rule can be applied

a strategy may use extra information which is updated

Definition

a strategy S is fair if for sequence of tableaux $T_1, T_2, \ldots$ following S :

1. any non-literal formula in T_i is eventually expanded, and
2. any γ-formula occurrence in T_i has the γ-rule applied to it arbitrarily often

Definition

a strategy S details:

1. which expansion rule is supposed to be applied
2. or that no expansion rule can be applied

a strategy may use extra information which is updated

Definition

a strategy S is fair if for sequence of tableaux $T_1, T_2, \dots$ following S :

1. any non-literal formula in T_i is eventually expanded, and
2. any γ-formula occurrence in T_i has the γ-rule applied to it arbitrarily often

Exercise +

make sure your implementation of free-variable tableaux is fair