Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016

Summary of Last Lecture

Definitions

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

Example (generate and test)

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).
ordered([X]).
ordered([X,Y|Ys]) :- X ⩽ Y, ordered([Y|Ys]).
```

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

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), correctness proofs, meta-logical predicates, cuts non-deterministic programming, efficient programs, complexity

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arity *Arity*

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arity *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arity *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Example

```
:- functor(father(haran,lot),F,A)
```

F ↦ father

A ↦ 2

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arity *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Example

```
:- functor(father(haran,lot),F,A)
F ↦ father
A ↦ 2
```

Example

```
:- arg(2,father(haran,lot),X)
X ↦ lot
```

Example

```
subterm(Term,Term).
subterm(Sub,Term) :-
    compound(Term),
    functor(Term,F,N),
    subterm(N,Sub,Term).

subterm(N,Sub,Term) :-
    N > 1,
    N1 is N - 1,
    subterm(N1,Sub,Term).
subterm(N,Sub,Term) :-
    arg(N,Term,Arg),
    subterm(Sub,Arg).

:- subterm(X,f(a,f(a,b))), X = a
```

Example

```
subterm(Term,Term).
subterm(Sub,Term) :-
    compound(Term),
    functor(Term,F,N),
    subterm(N,Sub,Term).

subterm(N,Sub,Term) :-
    N > 1,
    N1 is N - 1,
    subterm(N1,Sub,Term).
subterm(N,Sub,Term) :-
    arg(N,Term,Arg),
    subterm(Sub,Arg).

:- subterm(X,f(a,f(a,b))), X = a
:- subterm(X,f(U,f(V,W))), X = f(V,W).
```

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Definition

- *Term* =.. *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator =.. is also called univ

Example

```
:- father(haran,lot) =.. Xs
```

$X \mapsto$ `[father,haran,lot]`

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

```
:- father(haran,lot) =.. Xs
```

X $\mapsto$ `[father,haran,lot]`

Remark

- programs written with `functor` and `arg` can also be written with `univ`

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

```
:- father(haran,lot) =.. Xs
```

$X \mapsto$ `[father,haran,lot]`

Remark

- programs written with `functor` and `arg` can also be written with `univ`
- programs using `univ` are typically simpler
- programs using `functor` and `arg` are more efficient
- `univ` can be built from `functor` and `arg`

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,?)        :- X =  Z                    .
delete([X|Xs],Z,?)        :- dif(X,Z)                    .
```

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,Ys)     :- X =  Z , delete(Xs,Z,Ys).
```

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,Ys)      :- X =  Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,?)       :- dif(X,Z)                   .
```

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,Ys)     :- X =  Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) :- dif(X,Z) , delete(Xs,Z,Ys).
```

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,Ys)     :- X = Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) :- dif(X,Z) , delete(Xs,Z,Ys).
delete([],X,[]).
```

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

```
delete([X|Xs],Z,Ys)      :- X =  Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) :- dif(X,Z) , delete(Xs,Z,Ys).
delete([],X,[]).

delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- dif(X,Z), delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- dif(X,Z), delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :-  delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete2([X|Xs],X,Ys) :- delete2(Xs,X,Ys).
delete2([X|Xs],Z,[X|Ys]) :-  delete2(Xs,Z,Ys).
delete2([],X,[]).
```

Example

```
delete₂([X|Xs],X,Ys) :- delete₂(Xs,X,Ys).
delete₂([X|Xs],Z,[X|Ys]) :-  delete₂(Xs,Z,Ys).
delete₂([],X,[]).
```

```
:- delete₂([a,b,c,b],b,[a,c])
true
:- delete₂([a,b,c,b],b,[a,b,c,b])
true
```

Example

```
delete₂([X|Xs],X,Ys) :- delete₂(Xs,X,Ys).
delete₂([X|Xs],Z,[X|Ys]) :-  delete₂(Xs,Z,Ys).
delete₂([],X,[]).
```

```
:- delete₂([a,b,c,b],b,[a,c])
true
:- delete₂([a,b,c,b],b,[a,b,c,b])
true
```

Example (Select $\approx$ Delete$_2$)

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

```
:- delete₂([a],b,[a])
true
:- select(b,[a],X)
false
```

Example (non termination)

```
% infinite <- defines an uniformly nonterminating relation

infinite :- infinite.
```

Example (non termination)

```
% infinite <- defines an uniformly nonterminating relation

infinite :- infinite.
```

Example (again, but different)

```
% winfinite <- uniformly nonterminating relation

winfinite :- winfinite.
winfinite.
```

Example (non termination)

```
% infinite <- defines an uniformly nonterminating relation

infinite :- infinite.
```

Example (again, but different)

```
% winfinite <- uniformly nonterminating relation

winfinite :- winfinite.
winfinite.
```

Example (non termination, yet again)

```
% hinfinte <- not strongly terminating, weakly terminating

hinfinite.
hinfinite :- hinfinite.

:- hinfinite.
```

Termination Analysis

Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

Termination Analysis

Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

Example (specialised)

```
ancestor_of_2(Ancestor, Descendant) :- false,
    child_of(Descendant, Ancestor),
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

Termination Analysis

Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

Example (specialised)

```
ancestor_of_2(Ancestor, Descendant) :- false,
    child_of(Descendant, Ancestor),
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

equivalently

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

- Ancestor doesn't occur in first goal (= recursive call)
- no influence on termination behaviour

Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

- Ancestor doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- Descendant remains unchanged

Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

- Ancestor doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- Descendant remains unchanged
- last goal has no effect → let's remove (generalisation)

Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

- Ancestor doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- Descendant remains unchanged
- last goal has no effect → let's remove (generalisation)

Example (specialised and generalised)

```
% ancestor_of_2 <- uniform nontermination

ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant), false,
    child_of(Person, Ancestor).
```

Fact

suppose the solution set for `Goal` *is infinite, then the query*

```
:- Goal, false.
```

cannot terminate

Fact

suppose the solution set for `Goal` *is infinite, then the query*

```
:- Goal, false.
```

cannot terminate

Example

```
:- hinfinite, false % false, but does not terminate
```

Fact

suppose the solution set for Goal *is infinite, then the query*

```
:- Goal, false.
```

cannot terminate

Example

```
:- hinfinite, false % false, but does not terminate
```

Example (ancestor_of specialised)

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

:- \+ child_of(X,X).
:- ancestor_of(Ancestor, Descendant), false. % terminates
:- false, ancestor_of(Ancestor, Descendant). % remark order
```

Termination Domains

Example (recall)

```
% infinite <- defines an uniformly nonterminating relation

infinite :- infinite.

% winfinite <- uniformly nonterminating relation

winfinite :- winfinite.
winfinite.
```

Termination Domains

Example (recall)

```
% infinite <- defines an uniformly nonterminating relation

infinite :- infinite.

% winfinite <- uniformly nonterminating relation

winfinite :- winfinite.
winfinite.
```

Observation

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

Example

```
married(X,Y) :- married(Y,X).

parent_of(X,Y) :- child_of(Y,X).
child_of(X,Y) :- parent_of(Y,X).
```

Example

```
married(X,Y) :- married(Y,X).

parent_of(X,Y) :- child_of(Y,X).
child_of(X,Y) :- parent_of(Y,X).
```

Definitions

- a domain is a set of goals closed under the instance relation
- a termination domain of a program P is a domain D such that P terminates on all goals in D

Example

```
married(X,Y) :- married(Y,X).

parent_of(X,Y) :- child_of(Y,X).
child_of(X,Y) :- parent_of(Y,X).
```

Definitions

- a domain is a set of goals closed under the instance relation
- a termination domain of a program P is a domain D such that P terminates on all goals in D

Example (domain)

```
is_list([]).          is_list([X|Xs]) :- is_list(Xs).
:- is_list([a,X,b]).
```

Definition

recursive (grammar) rules which have the recursive goal as the first goal in the body are called left recursive

Definition

recursive (grammar) rules which have the recursive goal as the first goal in the body are called left recursive

Example (cont'd)

```
are_married(X,Y) :- married(X,Y).
are_married(X,Y) :- married(Y,X).
```

Definition

recursive (grammar) rules which have the recursive goal as the first goal in the body are called left recursive

Example (cont'd)

```
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

Definition

recursive (grammar) rules which have the recursive goal as the first goal in the body are called left recursive

Example (cont'd)

```
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

Definition

recursive (grammar) rules which have the recursive goal as the first goal in the body are called left recursive

Example (cont'd)

```
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

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions space/time

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time
- number of unification attempts — time

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time
- number of unification attempts — time
- size of proof tree — time

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time
- number of unification attempts — time
- size of proof tree — time
- logical inferences per second (LIPS) — time

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time
- number of unification attempts — time
- size of proof tree — time
- logical inferences per second (LIPS) — time
- size of terms — space

Complexity of Programs

- as soon as we know the termination domain of a program, we can ask about the complexity (= efficiency) of the program
- in general resource analysis is even more difficult than termination analysis; in particular this holds for automation

Definition

suitable complexity measures are

- cardinality of the set/multiset of solutions — space/time
- size of SLD tree — time
- number of unification attempts — time
- size of proof tree — time
- logical inferences per second (LIPS) — time
- size of terms — space
- full cost of SLD resolution — space/time

Example (ancestor_of, specialised)

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

:- ancestor_of(joseph_II,Descendant).
:- ancestor_of(Ancestor,joseph_II).
```

Example (ancestor_of, specialised)

```
ancestor_of(Ancestor, Descendant) :- false,
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

:- ancestor_of(joseph_II,Descendant).
:- ancestor_of(Ancestor,joseph_II).
```

Example (ancestor_of, specialised)

```
ancestor_of(Ancestor, Descendant) :- false,
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

:- ancestor_of(joseph_II,Descendant).
:- ancestor_of(Ancestor,joseph_II).
```

Example (cont'd)

we can ignore Descendant as it has no effect on the number of steps:

```
ancestor_of'(Ancestor) :-
    child_of(Person, Ancestor),
    ancestor_of'(Person).
```

Analysis

- in goal ancestor_of (joseph_II) we know the first argument: number of inferences bounded by number of descendants of Joseph II

Analysis

- in goal ancestor_of (joseph_II) we know the first argument: number of inferences bounded by number of descendants of Joseph II
- consider goal ancestor_of (Ancestor, joseph_II); here the 2nd argument is irrelevant for the complexity of the program

Analysis

- in goal ancestor_of (joseph_II) we know the first argument: number of inferences bounded by number of descendants of Joseph II
- consider goal ancestor_of (Ancestor, joseph_II); here the 2nd argument is irrelevant for the complexity of the program
- child_of /2 is called with free variables, hence the solution space is given by the whole database
- hence, all ancestors of all persons are computed

Analysis

- in goal ancestor_of (joseph_II) we know the first argument: number of inferences bounded by number of descendants of Joseph II
- consider goal ancestor_of (Ancestor, joseph_II); here the 2nd argument is irrelevant for the complexity of the program
- child_of /2 is called with free variables, hence the solution space is given by the whole database
- hence, all ancestors of all persons are computed

Example (reversed search)

```
ancestor_of_3(Ancestor, Descendant) :-
    child_of(Descendant, Ancestor).
ancestor_of_3(Ancestor,Descendant) :-
    child_of(Descendant, Person),
    ancestor_of_3(Ancestor,Person).

:- ancestor_of(Ancestor,joseph_II).
```