

Logic Programming

Cezary Kaliszyk **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015



Summary of Last Lecture

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 $Term$

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**

Composing Recursive Programs

Example

```

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

```

Example

```

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

```

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, unification, database and recursive programming, termination

Incomplete Data Structures and Constraints

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

Full Prolog

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

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, unification, database and recursive programming, **termination**

Incomplete Data Structures and Constraints

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

Full Prolog

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

Termination Revisited

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**

Termination Revisited

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

Termination Revisited

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

Observation

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

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

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).
```

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

```
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

```
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

```
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

```
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

Efficiency of Programs

Observations

- 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 automatable techniques

Efficiency of Programs

Observations

- 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 automatable techniques

Definition

suitable complexity measures are

- | | |
|---------------------------------------|------------|
| • cardinality of the set of solutions | space/time |
| • number of inferences | time |
| • number of resolution steps | time |
| • size of terms | space |

Efficiency of Programs

Observations

- 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 automatable techniques

Definition

suitable complexity measures are

- | | |
|---------------------------------------|------------|
| • cardinality of the set of solutions | space/time |
| • number of inferences | time |
| • number of resolution steps | time |
| • size of terms | space |

Example (specialised ancestor_of/2)

```
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 (specialised ancestor_of/2)

```

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)

```

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
- `child_of/2` is called with free variables, hence the solution space is given by the whole database
- **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
- all ancestors of all persons are computed

Example

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

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


Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$\mathbf{1} \quad [1,2,3] = [1,2,3] \setminus []$$

Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$1 \quad [1,2,3] = [1,2,3] \setminus []$$

$$2 \quad [1,2,3] = [1,2,3,4,5] \setminus [4,5]$$

Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$1 \quad [1,2,3] = [1,2,3] \setminus []$$

$$2 \quad [1,2,3] = [1,2,3,4,5] \setminus [4,5]$$

$$3 \quad [1,2,3] = [1,2,3,8] \setminus [8]$$

Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$1 \quad [1,2,3] = [1,2,3] \setminus []$$

$$2 \quad [1,2,3] = [1,2,3,4,5] \setminus [4,5]$$

$$3 \quad [1,2,3] = [1,2,3,8] \setminus [8]$$

$$4 \quad [1,2,3] = [1,2,3|Xs] \setminus Xs$$

Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$1 \quad [1,2,3] = [1,2,3] \setminus []$$

$$2 \quad [1,2,3] = [1,2,3,4,5] \setminus [4,5]$$

$$3 \quad [1,2,3] = [1,2,3,8] \setminus [8]$$

$$4 \quad [1,2,3] = [1,2,3|Xs] \setminus Xs$$

Definition

the difference of two lists is denoted as $As \setminus Bs$ and called **difference list**

Incomplete Data Structures

Observation

given a list $[1,2,3]$ it can be **represented** as the **difference** of two lists

$$1 \quad [1,2,3] = [1,2,3] \setminus []$$

$$2 \quad [1,2,3] = [1,2,3,4,5] \setminus [4,5]$$

$$3 \quad [1,2,3] = [1,2,3,8] \setminus [8]$$

$$4 \quad [1,2,3] = [1,2,3|Xs] \setminus Xs$$

Definition

the difference of two lists is denoted as $As \setminus Bs$ and called **difference list**

Example

```
append_dl(Xs \ Ys, Ys \ Zs, Xs \ Zs).
```

Application of Difference Lists

Example

```
reverse(Xs,Ys) :- reverse_dl(Xs, Ys \ []).  
reverse_dl([X|Xs], Ys \ Zs) :-  
    reverse_dl(Xs, Ys \ [X | Zs]).  
reverse_dl([], Xs \ Xs).
```

Application of Difference Lists

Example

```
reverse(Xs,Ys) :- reverse_dl(Xs, Ys \ []).
reverse_dl([X|Xs], Ys \ Zs) :-
    reverse_dl(Xs, Ys \ [X | Zs]).
reverse_dl([], Xs \ Xs).
```

Example

```
quicksort(Xs,Ys) :- quicksort_dl(Xs, Ys \ []).
quicksort_dl([X|Xs], Ys \ Zs) :-
    partition(Xs,X,Littles, Bigs),
    quicksort_dl(Littles,Ys \ [X|Ys1]),
    quicksort_dl(Bigs,Ys1 \ Zs).
quicksort_dl([], Xs \ Xs).
```


Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator `\` simplifies reading, but can be eliminated:
“As `\` Bs” \rightarrow “As , Bs”

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator `\` simplifies reading, but can be eliminated:
“As `\` Bs” \rightarrow “As , Bs”
- the explicit constructor should be removed, if time or space efficiency is an issue

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator \backslash simplifies reading, but can be eliminated: “ $As \backslash Bs$ ” \rightarrow “ As , Bs ”
- the explicit constructor should be removed, if time or space efficiency is an issue

More Observations

- the tail Bs of a difference list acts like a pointer to the end of the first list As

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator \backslash simplifies reading, but can be eliminated: “ $As \backslash Bs$ ” \rightarrow “ As , Bs ”
- the explicit constructor should be removed, if time or space efficiency is an issue

More Observations

- the tail Bs of a difference list acts like a pointer to the end of the first list As
- this works as As is an **incomplete** list

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator \backslash simplifies reading, but can be eliminated: “ $As \backslash Bs$ ” \rightarrow “ As, Bs ”
- the explicit constructor should be removed, if time or space efficiency is an issue

More Observations

- the tail Bs of a difference list acts like a pointer to the end of the first list As
- this works as As is an **incomplete** list
- thus we represent a concrete list as the difference of two incomplete data structures

Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator \backslash simplifies reading, but can be eliminated: “ $As \backslash Bs$ ” \rightarrow “ As, Bs ”
- the explicit constructor should be removed, if time or space efficiency is an issue

More Observations

- the tail Bs of a difference list acts like a pointer to the end of the first list As
- this works as As is an **incomplete** list
- thus we represent a concrete list as the difference of two incomplete data structures
- generalises to other recursive data types

Context-Free Grammars

Definition

a **grammar** G is a tuple $G = (V, \Sigma, R, S)$, where

- 1 V finite set of **variables** (or **nonterminals**)
- 2 Σ alphabet, the **terminal symbols**, $V \cap \Sigma = \emptyset$
- 3 R finite set of **rules**
- 4 $S \in V$ the **start symbol** of G

Context-Free Grammars

Definition

a **grammar** G is a tuple $G = (V, \Sigma, R, S)$, where

- 1 V finite set of **variables** (or **nonterminals**)
- 2 Σ alphabet, the **terminal symbols**, $V \cap \Sigma = \emptyset$
- 3 R finite set of **rules**
- 4 $S \in V$ the **start symbol** of G

a **rule** is a pair $P \rightarrow Q$ of words, such that $P, Q \in (V \cup \Sigma)^*$ and there is at least one variable in P

Context-Free Grammars

Definition

a **grammar** G is a tuple $G = (V, \Sigma, R, S)$, where

- 1 V finite set of **variables** (or **nonterminals**)
- 2 Σ alphabet, the **terminal symbols**, $V \cap \Sigma = \emptyset$
- 3 R finite set of **rules**
- 4 $S \in V$ the **start symbol** of G

a **rule** is a pair $P \rightarrow Q$ of words, such that $P, Q \in (V \cup \Sigma)^*$ and there is at least one variable in P

Definition

grammar $G = (V, \Sigma, R, S)$ is **context-free**, if \forall rules $P \rightarrow Q$:

- 1 $P \in V$
- 2 $Q \in (V \cup \Sigma)^*$

Example

sentence \rightarrow noun_phrase, verb_phrase.

noun_phrase \rightarrow determiner, noun_phrase2.

noun_phrase \rightarrow noun_phrase2.

noun_phrase2 \rightarrow adjective, noun_phrase2.

noun_phrase2 \rightarrow noun.

verb_phrase \rightarrow verb, noun_phrase.

verb_phrase \rightarrow verb.

determiner \rightarrow [the].

determiner \rightarrow [a].

noun \rightarrow [pie-plate].

noun \rightarrow [surprise].

adjective \rightarrow [decorated].

verb \rightarrow [contains].

sentence $\stackrel{*}{\Rightarrow}$ ‘‘the decorated pie-plate contains a surprise’’

Example

```

sentence(S \ S0) :- noun_phrase(S \ S1), verb_phrase(S1 \ S0).
noun_phrase(S \ S0) :-
    determiner(S \ S1), noun_phrase2(S1 \ S0).
noun_phrase(S) :- noun_phrase2(S).
noun_phrase2(S \ S0) :-
    adjective(S \ S1), noun_phrase2(S1 \ S0).
noun_phrase2(S) :- noun(S).
verb_phrase(S \ S0) :- verb(S \ S1), noun_phrase(S1 \ S0)
verb_phrase(S) :- verb(S).
determiner([the|S] \ S).
determiner([a|S] \ S).
noun([pie-plate|S] \ S).
noun([surprise|S] \ S).
adjective([decorated|S] \ S).
verb([contains|S] \ S).

```

Extension: Add Parsetree

Example

```
sentence(sentence(N,V), S \ S0) :-  
    noun_phrase(N, S \ S1),  
    verb_phrase(V, S1 \ S0).
```

Extension: Add Parsetree

Example

```
sentence(sentence(N,V), S \ S0) :-
    noun_phrase(N, S \ S1),
    verb_phrase(V, S1 \ S0).
```

Example (Definite Clause Grammars)

```
sentence(sentence(N,V)) → noun_phrase(N), verb_phrase(V).
noun_phrase(np(D,N)) → determiner(D), noun_phrase2(N).
noun_phrase(np(N)) → noun_phrase2(N).
noun_phrase2(np2(A,N)) → adjective(A), noun_phrase2(N).
noun_phrase2(np2(N)) → noun(N).
verb_phrase(vp(V,N)) → verb(V), noun_phrase(N).
verb_phrase(vp(V)) → verb(V).
```

GUPU

Example (termination and efficiency)

- Example 35
- Example 36

GUPU

Example (termination and efficiency)

- Example 35
- Example 36

Example (definite clause grammars)

- Example 40