Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Definition

the time complexity of a (Prolog) program expresses the runtime of a program as a function of the size of its input

Definition

the space complexity of a (Prolog) program expresses the memory requirement of a program as a function of the size of its input

Observations

- space usage depends on the depth of recursion
- if full unification is not employed, the number of reductions asymptotically bounds the runtime
- in general size of search space and size of input terms needs to be taken into account, even for measuring time

Howto Improve Performance

Suggestion ①

use better algorithms

Suggestion ②

tuning, via:

1. good goal order
2. elimination of (unwanted) nondeterminism by using explicit conditions and cuts
3. exploit clause indexing (order arguments suitably)
   indexing performs static analysis to detect clauses which are applicable for reduction

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

Generate and Test

Example

```
map(test,[region(a,A,[B,C,D]), region(b,B,[A,C,E]),
    region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
    region(e,E,[B,C,F]), region(f,F,[C,D,E])]).
```

Generate and Test

Example

```
map(test,[region(a,A,[B,C,D]), region(b,B,[A,C,E]),
    region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
    region(e,E,[B,C,F]), region(f,F,[C,D,E])]).

colour_map([Region|Regions], Colours) :-
    colour_region(Region,Colours),
    colour_map(Regions,Colours).
colour_map([],Colours).
```

Generate and Test

Example

```
map(test,[region(a,A,[B,C,D]), region(b,B,[A,C,E]),
    region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
    region(e,E,[B,C,F]), region(f,F,[C,D,E])]).

colour_map([Region|Regions], Colours) :-
    colour_region(Region,Colours),
    colour_map(Regions,Colours).
colour_map([],Colours).

colour_region(region(Name,Colour,Neighbours), Colours) :-
    select(Colour,Colours,Colours1),
    members(Neighbours,Colours1).
```

Generate and Test

Example

```
map(test,[region(a,A,[B,C,D]), region(b,B,[A,C,E]),
    region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
    region(e,E,[B,C,F]), region(f,F,[C,D,E])]).

colour_map([Region|Regions], Colours) :-
    colour_region(Region,Colours),
    colour_map(Regions,Colours).
colour_map([],Colours).

colour_region(region(Name,Colour,Neighbours), Colours) :-
    select(Colour,Colours,Colours1),
    members(Neighbours,Colours1).

test_colour(Name,Map) :-
    map(Name,Map),
    colours(Name,Colours),
    colour_map(Map,Colours).
```

Howto Test for Variants

Example

```
numbervars('$VAR'(N),N,N1) :- N1 is N+1.
numbervars(Term,N1,N2) :-
    nonvar(Term), functor(Term,Name,N),
    numbervars(0,N,Term,N1,N2).

numbervars(N,N,Term,N1,N1).
numbervars(I,N,Term,N1,N3) :-
    I < N, I1 is I+1, arg(I1,Term,Arg),
    numbervars(Arg,N1,N2), numbervars(I1,N,Term,N2,N3).
```

Howto Test for Variants

Example

```
numbervars('$VAR'(N),N,N1) :- N1 is N+1.
numbervars(Term,N1,N2) :-
    nonvar(Term), functor(Term,Name,N),
    numbervars(0,N,Term,N1,N2).
numbervars(N,N,Term,N1,N1).
numbervars(I,N,Term,N1,N3) :-
    I < N, I1 is I+1, arg(I1,Term,Arg),
    numbervars(Arg,N1,N2), numbervars(I1,N,Term,N2,N3).
```

Example

```
verify(Goal) :- \+ \+ Goal.
variant(Term1,Term2) :-
    verify((numbervars(Term1,0,N),
            numbervars(Term2,0,N),Term1=Term2)).
```

Nondeterministic Programming

Example

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$*q_2$	$\varnothing$	$\varnothing$

Nondeterministic Programming

Example

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$*q_2$	$\varnothing$	$\varnothing$

Definition

A NFA is quintuple $(Q, \Sigma, \Delta, I, F)$ such that

1. Q is a set of states
2. Σ is an alphabet
3. Δ is relation on $(Q \times \Sigma) \times Q$
4. I are the initial states
5. F are the final states

Example

```
accept(S) :-
    initial(Q),
    accept(Q,S).
accept(Q,[X|Xs]) :-
    delta(Q,X,Q1),
    accept(Q1,Xs).
accept(Q,[]) :-
    final(Q).
```

Example

```
accept(S) :-
    initial(Q),
    accept(Q,S).
accept(Q,[X|Xs]) :-
    delta(Q,X,Q1),
    accept(Q1,Xs).
accept(Q,[]) :-
    final(Q).
initial(q0).
final(q2).
```

Example

```
accept(S) :-
    initial(Q),
    accept(Q,S).
accept(Q,[X|Xs]) :-
    delta(Q,X,Q1),
    accept(Q1,Xs).
accept(Q,[]) :-
    final(Q).
initial(q0).
final(q2).
delta(q0,0,q0).
delta(q0,0,q1).
delta(q0,1,q0).
delta(q1,1,q2).
```

Example

```
accept(S) :-
    initial(Q),
    accept(Q,S).
accept(Q,[X|Xs]) :-
    delta(Q,X,Q1),
    accept(Q1,Xs).
accept(Q,[]) :-
    final(Q).
initial(q0).
final(q2).
delta(q0,0,q0).
delta(q0,0,q1).
delta(q0,1,q0).
delta(q1,1,q2).
:- accept([0,0,0,1,0,1]).
```

Incomplete Data Structures

Observation

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

1. `[1,2,3] = [1,2,3] \ []`

Incomplete Data Structures

Observation

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

1. `[1,2,3] = [1,2,3] \ []`
2. `[1,2,3] = [1,2,3,4,5] \ [4,5]`

Incomplete Data Structures

Observation

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

1. `[1,2,3] = [1,2,3]` $\setminus$ `[]`
2. `[1,2,3] = [1,2,3,4,5]` $\setminus$ `[4,5]`
3. `[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. [1,2,3] = [1,2,3] \ []
2. [1,2,3] = [1,2,3,4,5] \ [4,5]
3. [1,2,3] = [1,2,3,8] \ [8]
4. [1,2,3] = [1,2,3|Xs] \ Xs

Incomplete Data Structures

Observation

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

1. `[1,2,3] = [1,2,3] \ []`
2. `[1,2,3] = [1,2,3,4,5] \ [4,5]`
3. `[1,2,3] = [1,2,3,8] \ [8]`
4. `[1,2,3] = [1,2,3|Xs] \ Xs`

Definition

the difference of two lists is denotes 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. `[1,2,3] = [1,2,3] \ []`
2. `[1,2,3] = [1,2,3,4,5] \ [4,5]`
3. `[1,2,3] = [1,2,3,8] \ [8]`
4. `[1,2,3] = [1,2,3|Xs] \ Xs`

Definition

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

Example

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

Application of Difference Lists

Recall

```
flatten([X|Xs],Ys) :-
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X ≠ [].
flatten([],[]).
```

Application of Difference Lists

Recall

```
flatten([X|Xs],Ys) :-
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X ≠ [].
flatten([],[]).
```

Example

```
flatten(Xs,Ys) :- flatten_dl(Xs,Ys \ []).
flatten_dl([X|Xs],Ys \ Zs) :-
    flatten_dl(X,Ys \ Ys1), flatten_dl(Xs,Ys1 \ Zs).
flatten_dl(X,[X|Xs] \ Xs) :- constant(X), X ≠ [].
flatten_dl([],Xs \ Xs).
```

Difference Lists Implement Accumulators Top-Down

Example (Flatten with Difference Lists)

```
flatten(Xs,Ys) :- flatten_dl(Xs,Ys \ []).

flatten_dl([X|Xs],Ys \ Zs) :-
    flatten_dl(X,Ys \ Ys1), flatten_dl(Xs,Ys1 \ Zs).
flatten_dl(X,[X|Xs] \ Xs) :- constant(X), X ≠ [].
flatten_dl([],Xs \ Xs).
```

Difference Lists Implement Accumulators Top-Down

Example (Flatten with Difference Lists)

```
flatten(Xs,Ys) :- flatten_dl(Xs,Ys \ []).
flatten_dl([X|Xs],Ys \ Zs) :-
    flatten_dl(X,Ys \ Ys1), flatten_dl(Xs,Ys1 \ Zs).
flatten_dl(X,[X|Xs] \ Xs) :- constant(X), X ≠ [].
flatten_dl([],Xs \ Xs).
```

Example (Flatten Using Accumulator)

```
flatten(Xs,Ys) :- flatten(Xs,[],Ys).
flatten([X|Xs],Zs,Ys) :-
    flatten(Xs,Zs,Ys1), flatten(X,Ys1,Ys).
flatten(X,Xs,[X|Xs]) :-
    constant(X), X ≠ [].
flatten([],Xs,Xs).
```

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

```
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 $\setminus$ simplifies reading, but can be eliminated: "As $\setminus$ 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 $\backslash$ 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

Observations

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

Difference-structures

Example

consider the following task: convert the sum $(a + b) + (c + d)$ into $(a + (b + (c + (d + 0))))$

Difference-structures

Example

consider the following task: convert the sum $(a+b)+(c+d)$ into $(a+(b+(c+(d+0))))$

Definition

we make use of difference-sums: $E1$++$E2$, where $E1$, $E2$ are incomplete; the empty sum is denoted by 0

Difference-structures

Example

consider the following task: convert the sum $(a + b) + (c + d)$ into $(a + (b + (c + (d + 0))))$

Definition

we make use of difference-sums: $E1{+}{+}E2$, where $E1$, $E2$ are incomplete; the empty sum is denoted by 0

Example

```
normalise(Exp,Norm) :- normalise_ds(Exp,Norm ++ 0).
normalise_ds(A+B, Norm ++ Space) :-
    normalise_ds(A, Norm ++ NormB),
    normalise_ds(B, NormB ++ Space).
normalise_ds(A,(A + Space) ++ Space) :-
    constant(A).
```

Example

consider the following tasks

- create
- use
- maintain

a set of values indexed by keys

Example

consider the following tasks

- create
- use
- maintain

a set of values indexed by keys

Example

```
lookup(Key,[(Key,Value) | Dictionary],Value).
lookup(Key,[(Key1,Value1) | Dictionary],Value) :-
    Key ≠ Key1,
    lookup(Key,Dictionary,Value).
```

```
:- Dict = [(arnold,8881), (barry,4513), (cathy,5950) | Xs].
```

Example

consider the following tasks

- create
- use
- maintain

a set of values indexed by keys

Example

```
lookup(Key,[(Key,Value) | Dictionary],Value).
lookup(Key,[(Key1,Value1) | Dictionary],Value) :-
    Key ≠ Key1,
    lookup(Key,Dictionary,Value).
```

```
:- Dict = [(arnold,8881), (barry,4513), (cathy,5950) | Xs].
:- lookup(david,Dict,1199).
Dict ↦ [(arnold,8881), (barry,4513),
    (cathy,5950), (david,1199) | Xs]
```

Example (Freeze and Melt)

```
copy(A,B) :- assert ('$foo'(A)), retract('$foo'(B)).
```

Example (Freeze and Melt)

```
copy(A,B) :- assert ('$foo'(A)), retract('$foo'(B)).
freeze(A,B) :- copy(A,B), numbervars(B,0,N).
```

Example (Freeze and Melt)

```
copy(A,B) :- assert ('$foo'(A)), retract('$foo'(B)).
freeze(A,B) :- copy(A,B), numbervars(B,0,N).
melt(A,B) :- melt(A,B,Dictionary), !.
```

Example (Freeze and Melt)

```
copy(A,B) :- assert ('$foo'(A)), retract('$foo'(B)).

freeze(A,B) :- copy(A,B), numbervars(B,0,N).

melt(A,B) :- melt(A,B,Dictionary), !.

melt('$VAR'(N),X,Dictionary) :- lookup(N,Dictionary,X).
melt(X,X,Dictionary) :- constant(X).
melt(X,Y,Dictionary) :-
    compound(X),
    functor(X,F,N),
    functor(Y,F,N),
    melt(N,X,Y,Dictionary).
```

Example (Freeze and Melt)

```
copy(A,B) :- assert ('$foo'(A)), retract('$foo'(B)).

freeze(A,B) :- copy(A,B), numbervars(B,0,N).

melt(A,B) :- melt(A,B,Dictionary), !.

melt('$VAR'(N),X,Dictionary) :- lookup(N,Dictionary,X).
melt(X,X,Dictionary) :- constant(X).
melt(X,Y,Dictionary) :-
    compound(X),
    functor(X,F,N),
    functor(Y,F,N),
    melt(N,X,Y,Dictionary).

melt(N,X,Y,Dictionary) :-
    N > 0, arg(N,X,ArgX),
    melt(ArgX,ArgY,Dictionary),
    arg(N,Y,ArgY), N1 is N-1,
    melt(N1,X,Y,Dictionary).
melt(0,X,Y,Dictionary).
```

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, $V \cap \Sigma = \varnothing$
3. R finite set of rules
4. $S \in \mathcal{V}$ 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, $V \cap \Sigma = \varnothing$
3. R finite set of rules
4. $S \in \mathcal{V}$ the start symbol of G

a rule is a pair $P \to Q$ 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, $V \cap \Sigma = \varnothing$
3. R finite set of rules
4. $S \in \mathcal{V}$ the start symbol of G

a rule is a pair $P \to Q$ 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 $P \to Q$:

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

Example

```
sentence → noun_phrase, verb_phrase.

noun_phrase → determiner, noun_phrase2.
noun_phrase → noun_phrase2.

noun_phrase2 → adjective, noun_phrase2.
noun_phrase2 → noun.

verb_phrase → verb, noun_phrase.
verb_phrase → verb.

determiner → [the].
determiner → [a].

noun → [pie-plate].
noun → [surprise].

adjective → [decorated].

verb → [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).
```

Example

$\texttt{sentence(PT)} \overset{*}{\Rightarrow}$ `‘‘the decorated pie-plate contains a surprise’’`

$\texttt{sentence(PT)} \overset{*}{\Rightarrow}$ `‘‘the decorated pie-plates contain a surprise’’`

Example

sentence(PT) $\overset{*}{\Rightarrow}$ ''the decorated pie-plate contains a surprise''
sentence(PT) $\overset{*}{\Rightarrow}$ ''the decorated pie-plates contain a surprise''

Example

determiner(det(the)) → [the].
determiner(det(a)) → [a].

noun(noun(pie-plate)) → [pie-plate].
noun(noun(pie-plates)) → [pie-plates].
noun(noun(surprise)) → [surprise].
noun(noun(surprises)) → [surprises].

adjective(adj(decorated)) → [decorated].

verb(verb(contains)) → [contains].
verb(verb(contain)) → [contain].

sentence(PT) $\overset{*}{\Rightarrow}$ ''the decorated pie-plates contains a surprise''

Extension: Number Agreement

Example

```
sentence(sentence(NP,VP),Num) →
    noun_phrase(N,Num), verb_phrase(V,Num).

⋮

determiner(det(the),Num) → [the].
determiner(det(a),singular) → [a].
noun(noun(pie-plate),singular) → [pie-plate].
noun(noun(pie-plates),plural) → [pie-plates].
noun(noun(surprise),singular) → [surprise].
noun(noun(surprises),plural) → [surprises].
adjective(adj(decorated)) → [decorated].
verb(verb(contains),singular) → [contains].
verb(verb(contain),plural) → [contain].
```

sentence(PT) $\overset{*}{\Rightarrow}$ ''the decorated pie-plates contain a surprise''