

Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016



Summary of Last Lecture

Example

```
is_map([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]))).
```

```
coloured_map([Region|Regions], Colours) :-
    coloured_region(Region,Colours),
    coloured_map(Regions,Colours).
coloured_map([],Colours).
```

```
coloured_region(region(Name,Colour,Neighbours), Colours) :-
    select(Colour,Colours,Colours1),
    sublist_of(Neighbours,Colours1).
```

```
test_colour(Map) :-
    is_map(Map),
    is_colours(Colours),
    coloured_map(Map,Colours).
```

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

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

Example (parse tree)

```

is_expr (node (1, [])).                % tree elem
is_expr (node (+, [ExprL, ExprR])) :- % operation
    is_expr (ExprL),
    is_expr (ExprR).

expr (node (1, [])) -->
    "1".

expr (node (+, [Expr1, Expr2])) -->
    "(",
    expr (Expr1),
    "+",
    expr (Expr2),
    ")" .

:- phrase (Expr, "(1+1)").
:-/ - Xs = [_ , _ , _ , _], phrase (expr, Xs).
:-/ - & phrase (expr, Xs), false .

```

Example

```
seq ([]) —>
    [].
```

```
seq ([C|Cs]) —>
    [C],
    seq(Cs).
```

```
:- phrase(seq([1,2,3,4]), Xs), Xs = [1,2,3,4].
```

```
:- phrase(seq('abcd'), 'abcd').
```

```
invseq ([]) —>                                % aka reverse
    [].
```

```
invseq ([C|Cs]) —>
    invseq(Cs),
    [C].
```

```
:- phrase(invseq('abcd'), Xs), Xs = 'dcba'.
```

Example

```
palindrom -->
    seq(Xs),
    invseq(Xs).
```

```
palindrom -->
    seq(Xs),
    [-],
    invseq(Xs).
```

```
:- phrase(palindrom, 'abba').
```

```
palindrom2 --> [].
palindrom2 --> [-].
palindrom2 -->
    [X],
    palindrom2,
    [X].
```

Example (genome sequencing)

```
base —> "A".           % Adenin
base —> "C".           % Cytosin
base —> "G".           % Guanin
base —> "T".           % Tymin
```

```
basen —> "" .
basen —> base , basen .
```

```
tandemrepeat(Alpha) —>
    seq(Alpha) , seq(Alpha) ,
    { dif(Alpha , []) } .
```

```
subseq_tandemrepeat(Alpha) —>
    seq(_Prefix) ,
    tandemrepeat(Alpha) ,
    seq(_Suffix) .
```

```
:- genom(Gen) , phrase(subseq_tandemrepeat(Alpha) , Gen) ,
   Alpha='TGA' .
```


Example

```
numberpair(pair(X,Y)) :-  
    is_number(X),  
    is_number(Y).
```

```
:- numberpair(pair(s(s(0)),s(0))).
```

```
%:- numberpair(pair(A,B)), A = s(s(0)), B = s(s(s(0))).
```

Example

```

numberpair(pair(X,Y)) :-
    is_number(X),
    is_number(Y).

:- numberpair(pair(s(s(0)),s(0))).
%:- numberpair(pair(A,B)), A = s(s(0)), B = s(s(s(0))).

```

Example

```

numberpairD(pair(X,Y)) :-
    is_number(Z),
    plus(X,Y,Z).

:- numberpairD(pair(s(s(0)),s(0))).
:- numberpairD(pair(A,B)), A = s(s(0)), B = s(s(s(0))).

```

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \text{S E N D} \\ \text{M O R E} \\ \hline \text{M O N E Y} \end{array} +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \text{S E N D} \\ \text{M O R E} \\ \hline \text{1 O N E Y} \end{array} +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \text{S E N D} \\ \text{1 O R E} \\ \hline \text{1 O N E Y} \end{array} +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \text{S E N D} \\ \text{1 0 R E} \\ \hline \text{1 0 N E Y} \end{array} +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r}
 9 \text{ E N D} \\
 1 \text{ 0 R E} \\
 \hline
 1 \text{ 0 N E Y}
 \end{array}
 +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r}
 9\ E\ N\ D \\
 1\ 0\ 8\ E \\
 \hline
 1\ 0\ N\ E\ Y
 \end{array}
 +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r}
 95ND \\
 108\color{red}{5} \\
 \hline
 10N5Y
 \end{array}
 +$$

Cryptarithmic

Definition

- a cryptarithmic problem is a puzzle in which each letter represents a unique digit ≤ 9
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9567 \\ 1085 \\ \hline 10652 \end{array} +$$

First Attempt

generate and test

```

solve ([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
    Digits = [D, E, M, N, O, R, S, Y],
    Carries = [C1,C2,C3,C4],
    selects(Digits, [0,1,2,3,4,5,6,7,8,9]),
    members(Carries, [0,1]),
    M      == C4,
    O + 10 * C4 == S + M + C3,
    N + 10 * C3 == E + O + C2,
    E + 10 * C2 == N + R + C1,
    Y + 10 * C1 == D + E,
    M > 0, S > 0.

:- solve(X),
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]].

```

Discussion

very inefficient

```
?- time(solve(X)).
```

```
% 133,247,057 inferences ,
```

```
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
```

```
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

Discussion

very inefficient

```
?- time(solve(X)).
```

```
% 133,247,057 inferences ,
```

```
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
```

```
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

Discussion

very inefficient

```
?- time(solve(X)).
```

```
% 133,247,057 inferences ,
```

```
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
```

```
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

improvement

- move testing into generating
- destroys clean structure of program

Discussion

very inefficient

```
?- time(solve(X)).
```

```
% 133,247,057 inferences ,
```

```
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
```

```
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

improvement

- move testing into generating
- destroys clean structure of program
- any other ideas?

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in `Xs` must be in the given range
- `all_different(Xs)` specifies that all values in `Xs` are different
- `label(Xs)` all variables in `Xs` are evaluated to become values
- `#=`, `#\=`, `#>`, ... like the arithmetic comparison operators, but may contain (constraint) variables

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in `Xs` must be in the given range
- `all_different(Xs)` specifies that all values in `Xs` are different
- `label(Xs)` all variables in `Xs` are evaluated to become values
- `#=`, `#\=`, `#>`, ... like the arithmetic comparison operators, but may contain (constraint) variables

standard approach

- load the library
- specify all constraints
- call `label` to start efficient computation of solutions

Second Attempt

constraint logic program

```

solve ([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
    Digits = [D, E, M, N, O, R, S, Y],
    Carries = [C1,C2,C3,C4],
    Digits ins 0 .. 9, all_different(Digits),
    Carries ins 0 .. 1,
    M           $\neq$           C4,
    O + 10 * C4  $\neq$  S + M + C3,
    N + 10 * C3  $\neq$  E + O + C2,
    E + 10 * C2  $\neq$  N + R + C1,
    Y + 10 * C1  $\neq$  D + E,
    M  $\#>$  0, S  $\#>$  0,
    label(Digits).

```

8 queens

```
queens(Xs) :- template(Xs), solution(Xs).
```

```
template([1/_Y1,2/_Y2,3/_Y3,4/_Y4,  
          5/_Y5,6/_Y6,7/_Y7,8/_Y8]).
```

```
solution([]).
```

```
solution([X/Y|Others]) :-  
    solution(Others),  
    member(Y, [1,2,3,4,5,6,7,8]),  
    noattack(X/Y, Others).
```

```
noattack(_, []).
```

```
noattack(X/Y, [X1/Y1|Others]) :-  
    Y \= Y1,  
    Y1 - Y \= X1 - X,  
    Y1 - Y \= X - X1,  
    noattack(X/Y, Others).
```

n-queens (using clp)

```
nqueens(N,Qs) :-  
    length(Qs,N),  
    Qs ins 1 .. N, all_different(Qs),  
    constraint_queens(Qs),  
    label(Qs).
```

```
constraint_queens([]).  
constraint_queens([Q|Qs]) :-  
    noattack(Q,Qs,1),  
    constraint_queens(Qs).
```

```
noattack(_,[],_).  
noattack(X,[Q|Qs],N) :-  
    X #\= Q+N,  
    X #\= Q-N,  
    M is N+1,  
    noattack(X,Qs,M).
```

Definition

- **Sudoku** is a well-known logic puzzle; usually played on a 9×9 grid
- \forall *cells*: $cells \in \{1, \dots, 9\}$
- \forall *rows*: all entries are different
- \forall *columns*: all entries are different
- \forall *blocks*: all entries are different

Definition

- **Sudoku** is a well-known logic puzzle; usually played on a 9×9 grid
- \forall *cells*: $cells \in \{1, \dots, 9\}$
- \forall *rows*: all entries are different
- \forall *columns*: all entries are different
- \forall *blocks*: all entries are different

Main Loop (using clp)

```
sudoku(Puzzle) :-  
    show(Puzzle),  
    flatten(Puzzle, Cells),  
    Cells ins 1 .. 9,  
    rows(Puzzle),  
    cols(Puzzle),  
    blocks(Puzzle),  
    label(Cells),  
    show(Puzzle).
```

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

row/1

```
rows ([ ]).  
rows ([R|Rs]) :-  
    all_different(R), rows(Rs).
```

auxiliary predicates

- *flatten/2* flattens a list
- *show/1* prints the current puzzle

row/1

```
rows ([ ] ).  
rows ([R|Rs]) :-  
    all_different(R), rows(Rs).
```

row/1 (alternative)

```
rows(Rs) :- maplist(all_distinct, Rs).
```

cols/1

```
cols ([[ ] | - ]).
```

```
cols ([
```

```
    [X1|R1],
```

```
    [X2|R2],
```

```
    [X3|R3],
```

```
    [X4|R4],
```

```
    [X5|R5],
```

```
    [X6|R6],
```

```
    [X7|R7],
```

```
    [X8|R8],
```

```
    [X9|R9] ] ) :-
```

```
    all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
```

```
    cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]).
```

cols/1

```
cols ([[ ] | - ] ).  
cols ([  
    [X1|R1] ,  
    [X2|R2] ,  
    [X3|R3] ,  
    [X4|R4] ,  
    [X5|R5] ,  
    [X6|R6] ,  
    [X7|R7] ,  
    [X8|R8] ,  
    [X9|R9] ] ) :-  
    all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]) ,  
    cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]) .
```

cols/1 (alternative)

use *maplist/2*

blocks/1

```
blocks([]).  
blocks([[],[],[]|Rs]) :- blocks(Rs).  
blocks([[X1,X2,X3|R1],  
        [X4,X5,X6|R2],  
        [X7,X8,X9|R3] |Rs]) :-  
    all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),  
    blocks([R1,R2,R3|Rs]).
```

blocks/1

```

blocks ( []).
blocks ([], [], [] | Rs) :- blocks(Rs).
blocks ([ [X1,X2,X3 | R1],
          [X4,X5,X6 | R2],
          [X7,X8,X9 | R3] | Rs) :-
    all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
    blocks([R1,R2,R3 | Rs]).

```

Example

```

:- sudoku([ [1, -, -, -, -, -, -, -, -, -],
            [ -, -, 2, 7, 4, -, -, -, -, -],
            [ -, -, -, 5, -, -, -, -, 4],
            [ -, 3, -, -, -, -, -, -, -],
            [ 7, 5, -, -, -, -, -, -, -],
            [ -, -, -, -, -, 9, 6, -, -],
            [ -, 4, -, -, -, 6, -, -, -],
            [ -, -, -, -, -, -, -, 7, 1],
            [ -, -, -, -, -, -, 1, -, 3, -] ]).

```

Wrong Solutions

Example

```
/* A farmer has chicken and cows with in total 24 legs and 9 heads.  
How many chicken and cows does the farmer own? */
```

Wrong Solutions

Example

/* A farmer has chicken and cows with in total 24 legs and 9 heads.
How many chicken and cows does the farmer own? */

Example

```
solve_riddle(Hens,Cows) :-  
    4 * Cows + 2 * Hens #= 24,  
    Cows + Hens #= 9,  
    Cows #>= 0,  
    Hens #>= 0.  
:- solve_riddle(Hens,Cows), Hens=6, Cows=3.
```

```
solve_riddle2(Hens,Cows) :-  
    4 * Cows + 2 * Hens #= 24,  
    Cows + Hens #= 9.  
:- solve_riddle2(Hens,Cows), Hens=6, Cows=3.
```