Logic Programming

Cezary Kaliszyk **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015

Summary of Last Lecture

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

GUPU

Example (constraint logic programming)

- 59, 60 (solution space)
- 61 (termination)
- 66 (core relation)

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

Efficient Constraint Logic Programmming

Strategies for Solutions

- take termination seriously
  non-termination is a sign of inefficiency
- choose suitable labeling strategies
- use system predicates

```
:- Zs = [A,B,C], Zs ins 1..2,
   A #\= B, B #\= C, A #\= C.
:/- Zs = [A,B,C], Zs ins 1..2,
    all_different(Zs).
```

- make use of redundant constraints
  recall the magic square example, where the sums equal $n \cdot (n^2 + 1)/2$; using this insight redundant constrains are prevented, and the search is quicker; however, in general a predefined search strategy doesn't need to be more efficient

Labeling Strategies

Strategies for Solutions (cont'd)

- minimise the solution space
  consider the exclusion of rotations and symmetries for magic square
- improve representation of solutions
  inefficient/redundant representations increase the solution space unnecessarily

Labeling Strategies

Strategies for Solutions (cont'd)

- minimise the solution space
  consider the exclusion of rotations and symmetries for magic square
- improve representation of solutions
  inefficient/redundant representations increase the solution space unnecessarily

Definition

labeling (+Options,+Vars) assign a value to each variable in *Vars*; three categories of options exist

- variable selection strategy
- value order strategy
- branching strategy

Definition (variable selection strategy)

- leftmost, select the variables in the order they occur in *Vars* (default)
- min, select the leftmost variable with lowest lower bound next

```
:- X in 1..2, Y in 3..4, labeling([min],[X,Y]).
X = 1, Y = 3;
X = 1, Y = 4;
X = 2, Y = 3;
X = 2, Y = 4
```

- max, select the leftmost variable with highest upper bound next

```
:- X in 1..2, Y in 3..4, labeling([min],[X,Y]).
X = 1, Y = 3;
X = 2, Y = 3;
X = 1, Y = 4;
X = 2, Y = 4
```

- ff, first fail, select the leftmost variable with smallest domain next, in order to detect infeasibility early

Definition (variable selection strategy (cont'd))

- ffc, from the variables with smallest domain, select the one occurring most often in constraints

Definition (variable selection strategy (cont'd))

- ffc, from the variables with smallest domain, select the one occurring most often in constraints

Definition (value order strategy)

- up, try the elements of the domain in ascending order
- down, in descending order

Definition (variable selection strategy (cont'd))

- ffc, from the variables with smallest domain, select the one occurring most often in constraints

Definition (value order strategy)

- up, try the elements of the domain in ascending order
- down, in descending order

Definition (branching strategy)

- step, for each variable X, the choice is between $X = V$ and X #\= V (V determined by value order)
- enum, enumerate the domain of X according to the value order
- bisect, choice is between X #=< M and X #> M (M the midpoint of the domain)

The New Kid on the Block

Answer Set Programming

- novel approach to modelling and solving search and optimisation problems
- $\neg$ programming, but a specification language
- $\neg$ Turing complete
- purely declarative
- restricted to finite models

The New Kid on the Block

Answer Set Programming

- novel approach to modelling and solving search and optimisation problems
- $\neg$ programming, but a specification language
- $\neg$ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport

The New Kid on the Block

Answer Set Programming

- novel approach to modelling and solving search and optimisation problems
- $\neg$ programming, but a specification language
- $\neg$ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport
- expert system in space shuttle

The New Kid on the Block

Answer Set Programming

- novel approach to modelling and solving search and optimisation problems
- $\neg$ programming, but a specification language
- $\neg$ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport
- expert system in space shuttle
- natural language processing
- ...

Propositional Setting

Definitions

- atoms, facts, rules are defined as before

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head
- an answer set is a set of atoms corresponding to the minimal model of the program

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head
- an answer set is a set of atoms corresponding to the minimal model of the program

Example (Negation as Failure)

```
light_on :- power_on, not broken.
power_on.
```

answer set: $\{power\_on, light\_on\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example

```
a | b.
a | c.
```

answer sets: $\{a\}$ and $\{b, c\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example

```
a | b.
a | c.
```

answer sets: $\{a\}$ and $\{b, c\}$

```
a | b.
a :- b.
```

answer set: $\{a\}$, but not $\{b\}$ nor $\{a, b\}$

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Example

```
a :- not a, b.
```

any answer set must not contain *b* and simplifies to

```
:- b.
```

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Example

```
a :- not a, b.
```

any answer set must not contain b and simplifies to

```
:- b.
```

Additional Features

- finite choice functions: $\{fact_1, fact_2, fact_3\}$.
- choice and counting: $1\{fact_1, fact_2, fact_3\}2$.
  "1" or "2" may be missing

First-Order Setting

Definition

- extension of first-order language
- no function symbols

First-Order Setting

Definition

- extension of first-order language
- no function symbols

Example (3-colouring)

```
red(X) | green(X) | blue(X).
:- red(X), red(Y), edge(X,Y).
:- green(X), green(Y), edge(X,Y).
:- blue(X), blue(Y), edge(X,Y).
```

First-Order Setting

Definition

- extension of first-order language
- no function symbols

Example (3-colouring)

```
red(X) | green(X) | blue(X).
:- red(X), red(Y), edge(X,Y).
:- green(X), green(Y), edge(X,Y).
:- blue(X), blue(Y), edge(X,Y).
```

Example ((part of) 8-queens problem)

```
:- not (1 = count(Y : queen(X,Y))), row(X)
```

expresses that exactly one queen appears in every row and column

Grounders and Solvers

Grounders and Solvers

Grounders

- DLV (DLV Systems, Calabria)
- Gringo (University of Potsdam)
- lparse (University of Helsinki)

Grounders and Solvers

Grounders

- DLV (DLV Systems, Calabria)
- Gringo (University of Potsdam)
- lparse (University of Helsinki)

Solvers

- clasp (University of Potsdam)
- cmodels (University of Austin)
- smodels (University of Helsinki)

Prolog and Answer Set Programming

- proof search
- Turing complete
- control
- efficiency

- model search
- finite domain
- specification language
- generality

Prolog and Answer Set Programming

- proof search
- Turing complete
- control
- efficiency

- model search
- finite domain
- specification language
- generality

Example

```
hanoi(0,_,_,_,[]).
hanoi(N,X,Y,Z,Ls) :-
     N > 0, M is N - 1,
     hanoi(M,X,Z,Y,Ls0),
     append(Ls0,[move(N,X,Z)],Ls1),
     hanoi(M,Y,X,Z,Ls2),
     append(Ls1,Ls2,Ls).
```

Example

```
disk(1..n).                      peg(a;b;c).
transition(0..pathlength-1).     situation(0..pathlength).
location(Peg) :- peg(Peg).       location(Disk) :- disk(Disk).
#domain disk(X;Y).        #domain peg(P;P1;P2).
#domain transition(T). #domain situation(I).
#domain location(L;L1).

on(X,L,T+1) :- on(X,L,T), not otherloc(X,L,T+1).
otherloc(X,L,I) :- on(X,L1,I), L1!=L.
:- on(X,L,I), on(X,L1,I), L!=L1.
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I).    inpeg(P,P,I).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I).
covered(L,I) :- on(X,L,I).
:- on(X,Y,I), X>Y.
on(X,L,T+1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T).
:- move(P1,P2,T), top(P1,P1,T). movement(P1,P2) :- P1 != P2.
1 {move(A,B,T) : movement(A,B) } 1.
on(n,a,0).                           on(X,X+1,0) :- X<n.
onewrong :- not inpeg(X,c,pathlength).
:- onewrong.
```