Answer Set Programming for the Semantic Web

Tutorial

Technische Universität Wien
Vienna University of Technology
Università della Calabria
Campus di Arcavacata
Universidad Rey Juan Carlos

Thomas Eiter, Roman Schindlauer (TU Wien)
Giovambattista Ianni (TU Wien, Univ. della Calabria)
Axel Polleres (Univ. Rey Juan Carlos, Madrid)

Supported by IST REWERSE, FWF Project P17212-N04, CICyT project TIC-2003-9001-C02.

Unit 1 – ASP Basics

T. Eiter

KBS Group, Institute of Information Systems, TU Vienna

European Semantic Web Conference 2006

presented by A.Polleres, G. Ianni

Unit Outline

Sudoku

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Task

Fill in the grid so that every row, every column, and every 3x3 box contains the digits 1 through 9

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Social Dinner Example

- Imagine the ESWC organizers are planning a fancy dinner for the ASP tutorial attendees.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees' preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.

Wanted!

A general-purpose approach for modeling and solving these and many other problems

Issues:

- Diverse domains
- Spatial and temporal reasoning
- Constraints
- Incomplete information
- Preferences and priority

Proposal:

Answer Set Programming (ASP) paradigm!

Wanted!

A general-purpose approach for modeling and solving these and many other problems

Issues:

- Diverse domains
- Spatial and temporal reasoning
- Constraints
- Incomplete information
- Preferences and priority

Proposal:

Answer Set Programming (ASP) paradigm!

Roots of ASP – Knowledge Representation (KR)

How to model

- An agent's belief sets
- Commonsense reasoning
- Defeasible inferences
- Preferences and priority

Approach

- use a logic-based formalism
- Inherent feature: nonmonotonicity

Many logical formalisms for knowledge representation have been developed.

Logic Programming – Prolog revisited

Logic as a Programming Language (?)

Kowalski (1979):

ALGORITHM = LOGIC + CONTROL

- Knowledge for problem solving (LOGIC)
- "Processing" of the knowledge (CONTROL)

Prolog

Prolog = "Programming in Logic"

- Basic data structures: terms
- Programs: rules and facts
- Computing: Queries (goals)
  - Proofs provide answers
  - SLD-resolution
  - unification - basic mechanism to manipulate data structures
- Extensive use of recursion

Simple Social Dinner Example

From simple.dlv:

- Wine bottles (brands) "a", . . . , "e"
- plain ontology natively represented within the logic program.
- preference by facts

```
% A suite of wine bottles and their kinds
wineBottle("a").   isA("a","whiteWine").   isA("a","sweetWine").
wineBottle("b").   isA("b","whiteWine").   isA("b","dryWine").
wineBottle("c").   isA("c","whiteWine").   isA("c","dryWine").
wineBottle("d").   isA("d","redWine").   isA("d","dryWine").
wineBottle("e").   isA("e","redWine").   isA("e","sweetWine").

% Persons and their preferences
person("axel").   preferredWine("axel","whiteWine").
person("gibbi").   preferredWine("gibbi","redWine").
person("roman").   preferredWine("roman","dryWine").

% Available bottles a person likes
compliantBottle(X,Z) :- preferredWine(X,Y), isA(Z,Y).
```

Example: Recursion

```
append([],X,X) .
append([X|Y],Z,[X|T]) :- append(Y,Z,T) .

reverse([],[]).
reverse([X|Y],Z) :- append(U,[X],Z), reverse(Y,U) .
```

- both relations defined recursively
- terms represent complex objects: lists, sets, ...

Problem:

Reverse the list `[a,b,c]`

Ask query: `?- reverse([a,b,c],X).`

- A proof of the query yields a substitution: `X=[c,b,a]`
- The substitution constitutes an answer

Prolog /2

The key: Techniques to search for proofs

- Understanding of the resolution mechanism is important
- It may make a difference which logically equivalent form is used (e.g., termination).

```
reverse([X|Y],Z) :- append(U,[X],Z), reverse(Y,U) .
```

vs

```
reverse([X|Y],Z) :- reverse(Y,U), append(U,[X],Z) .
```

Query: `?- reverse([a|X],[b,c,d,b])`

Is this truly declarative programming?

Negation in Logic Programs

Why negation?

- Natural linguistic concept
- Facilitates declarative descriptions (definitions)
- Needed for programmers convenience

Clauses of the form:

$$p(\vec{X}) \text{:-} q_1(\vec{X_1}), \ldots, q_k(\vec{X_k}), not\ \ r_1(\vec{Y_1}), \ldots, not\ \ r_l(\vec{Y_l})$$

Things get more complex!

Negation in Prolog

- "*not* (·)" means "Negation as Failure (to prove)"
- **Different from negation in classical logic!**

Example

```
compliantBottle("axel","a"),

bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
bottleSkipped(X) :- fail.  % dummy declaration
```

Query:

```
?- bottleChosen(X).
   X = "a"
```

Programs with Negation /2

Modified rule:

```
compliantBottle("axel","a").

bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
bottleSkipped(X) :- not bottleChosen(X),  compliantBottle(Y,X).
```

Result ????

Problem: not a single minimal model!

Two alternatives:

- M_1 = { `compliantBottle("axel","a")`, `bottleChosen("a")` },
- M_2 = { `compliantBottle("axel","a")`, `bottleSkipped("a")` }.

Which one to choose?

Semantics of Logic Programs with Negation

Great Logic Programming Schism

Single Intended Model Approach:

- Select a single model of all classical models
- Agreement for so-called "stratified programs": " Perfect model"

Multiple Preferred Model Approach:

- Select a subset of all classical models
- Different selection principles for non-stratified programs

Stratified Negation

Intuition: For evaluating the body of a rule containing *not* $r(\vec{t})$, the value of the "negative" predicates $r(\vec{t})$ should be known.

1. Evaluate first $r(\vec{t})$
2. if $r(\vec{t})$ is false, then *not* $r(\vec{t})$ is true,
3. if $r(\vec{t})$ is true, then *not* $r(\vec{t})$ is false and rule is not applicable.

Example:

```
compliantBottle("axel","a").
bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
```

Computed model

$M = \{$ `compliantBottle("axel","a")`, `bottleChosen("a")` $\}$.

Note: this introduces *procedurality* (violates declarativity)!

Program Layers

- Evaluate predicates bottom up in layers
- Methods works if there is no cyclic negation (layered negation)

Example:

```
L0: compliantBottle("axel","a"). wineBottle("a"). expensive("a").

L1: bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
L0: bottleSkipped(X) :- expensive(X), wineBottle(X).
```

Unique model resulting by layered evaluation (“perfect model”):

$M = \{$ compliantBottle("axel","a"), wineBottle("a"), expensive("a"), bottleSkipped("a")$\}$

Multiple preferred models

Unstratified Negation makes layering ambiguous:

```
L0: compliantBottle("axel","a").
L?: bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
L?: bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
```

- Assign to a program (theory) not one but **several** intended models! For instance: Answer sets!
- How to interpret these semantics? Answer set programming caters for the following views:
  1. *skeptical* reasoning: Only take entailed answers, i.e. true in all models
  2. *brave* reasoning: each model represents a different solution to the problem
  3. additionally: one can define to consider only a subset of *preferred models*
- (Alternative: well-founded inference takes a more “agnostic” view: One model, leaving ambiguous literals unknown.)

Multiple preferred models

Unstratified Negation makes layering ambiguous:

```
L0: compliantBottle("axel","a").
L?: bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
L?: bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
```

- Assign to a program (theory) not one but **several** intended models! For instance: Answer sets!
- How to interpret these semantics? Answer set programming caters for the following views:
  1. *skeptical* reasoning: Only take entailed answers, i.e. true in all models
  2. *brave* reasoning: each model represents a different solution to the problem
  3. additionally: one can define to consider only a subset of *preferred models*
- (Alternative: well-founded inference takes a more "agnostic" view: One model, leaving ambiguous literals unknown.)

Multiple preferred models

Unstratified Negation makes layering ambiguous:

```
L0: compliantBottle("axel","a").
L?: bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
L?: bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
```

- Assign to a program (theory) not one but **several** intended models! For instance: Answer sets!
- How to interpret these semantics? Answer set programming caters for the following views:
  1. *skeptical* reasoning: Only take entailed answers, i.e. true in all models
  2. *brave* reasoning: each model represents a different solution to the problem
  3. additionally: one can define to consider only a subset of *preferred models*
- (Alternative: well-founded inference takes a more "agnostic" view: One model, leaving ambiguous literals unknown.)

Multiple preferred models

Unstratified Negation makes layering ambiguous:

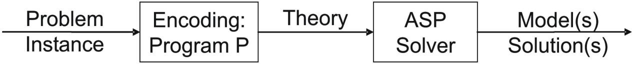
```
L0: compliantBottle("axel","a").
L?: bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
L?: bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
```

- Assign to a program (theory) not one but **several** intended models! For instance: Answer sets!
- How to interpret these semantics? Answer set programming caters for the following views:
  1. *skeptical* reasoning: Only take entailed answers, i.e. true in all models
  2. *brave* reasoning: each model represents a different solution to the problem
  3. additionally: one can define to consider only a subset of *preferred models*
- (Alternative: well-founded inference takes a more "agnostic" view: One model, leaving ambiguous literals unknown.)

Answer Set Programming Paradigm

General idea: Models are Solutions!

Reduce solving a problem instance I to computing models



1. *Encode* I as a (non-monotonic) logic program P, such that solutions of I are represented by models of P
2. *Compute* some model M of P, using an ASP solver
3. *Extract* a solution for I from M.

Variant: Compute multiple models (for multiple / all solutions)

Applications of ASP

ASP facilitates *declarative problem solving*

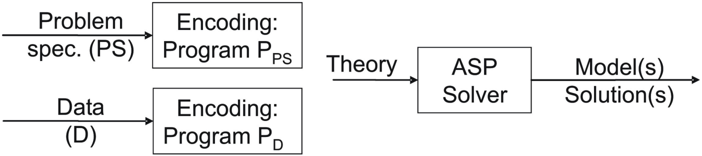
Problems in different domains (some with substantial amount of data), see

`http://www.kr.tuwien.ac.at/projects/WASP/report.html`

- information integration
- constraint satisfaction
- planning, routing
- semantic web
- diagnosis
- security analysis
- configuration
- computer-aided verification
- ...

ASP Showcase: `http://www.kr.tuwien.ac.at/projects/WASP/showcase.html`

ASP in Practice



Uniform encoding:

Separate problem specification, *PS* and input data *D* (usually, facts)

- Compact, easily maintainable representation: Disjunctive Logic programs with constraints: This is more than we saw so far!
- Integration of KR, DB, and search techniques
- Handling dynamic, knowledge intensive applications: data, defaults, exceptions, closures, …

Example: Sudoku

Problem specification *PS*

$tab(i,j,n)$: cell (i,j), $i,j \in \{0,...,8\}$ has digit n

From sudoku.dlv:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
   #int(X),  0 <= X,  X <= 8, #int(Y),  0 <= Y,  Y <= 8.

% Check rows and columns
 :-  tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
 :-  tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
 div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
  div(X,Y,Z) :- XminusDelta = Y*Z,  X = XminusDelta + Delta, Delta < Y.
```

Example: Sudoku

Problem specification *PS*

$tab(i,j,n)$: cell (i,j), $i,j \in \{0,...,8\}$ has digit n

From sudoku.dlv:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
   #int(X),  0 <= X,  X <= 8, #int(Y),  0 <= Y,  Y <= 8.

% Check rows and columns
 :-  tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
 :-  tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
 div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
  div(X,Y,Z) :- XminusDelta = Y*Z,  X = XminusDelta + Delta, Delta < Y.
```

Example: Sudoku

Problem specification *PS*

$tab(i,j,n)$: cell (i,j), $i,j \in \{0,...,8\}$ has digit n

From sudoku.dlv:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
   #int(X),  0 <= X,  X <= 8, #int(Y),  0 <= Y,  Y <= 8.

% Check rows and columns
 :-  tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
 :-  tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
 div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
  div(X,Y,Z) :- XminusDelta = Y*Z,  X = XminusDelta + Delta, Delta < Y.
```

Example: Sudoku

Problem specification *PS*

$tab(i,j,n)$: cell (i,j), $i,j \in \{0,...,8\}$ has digit n

From sudoku.dlv:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
   #int(X),  0 <= X,  X <= 8, #int(Y),  0 <= Y,  Y <= 8.

% Check rows and columns
 :-  tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
 :-  tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
 div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
  div(X,Y,Z) :- XminusDelta = Y*Z,  X = XminusDelta + Delta, Delta < Y.
```

Example: Sudoku

Problem specification *PS*

$tab(i,j,n)$: cell (i,j), $i,j \in \{0,...,8\}$ has digit n

From sudoku.dlv:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
   #int(X),  0 <= X,  X <= 8, #int(Y),  0 <= Y,  Y <= 8.

% Check rows and columns
 :-  tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
 :-  tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
 div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
 :-  tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
  div(X,Y,Z) :- XminusDelta = Y*Z,  X = XminusDelta + Delta, Delta < Y.
```

Sudoku (cont'd)

Data *D*:

```
% Table positions X=0..8, Y=0..8
tab(0,1,6). tab(0,3,1). tab(0,5,4). tab(0,7,5).
tab(1,2,8). tab(1,3,3). tab(1,5,5). tab(1,6,6).
...
```

Solution:

Task

Run suduko.dlv using our Web interface!

Sudoku (cont'd)

Data *D*:

```
% Table positions X=0..8, Y=0..8
tab(0,1,6). tab(0,3,1). tab(0,5,4). tab(0,7,5).
tab(1,2,8). tab(1,3,3). tab(1,5,5). tab(1,6,6).
...
```

Solution:

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Task

Run suduko.dlv using our Web interface!

ASP - Desiderata

Expressive Power

Capable of representing a range of problems, hard problems
Disjunctive ASP: $\mathrm{NEXP}^{\mathrm{NP}}$-complete problems !

Ease of Modeling

- Intuitive semantics
- Concise encodings: Availability of predicates and variables
  Note: SAT solvers do *not* support predicates and variables
- Modular programming: global models can be composed from local models of components

Performance

Fast solvers available

Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

```
(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`.
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more on that later!

Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

```
% These rules generate multiple answer sets:
(1) bottleSkipped(X)  :-  not bottleChosen(X),
                           compliantBottle(Y,X).
(2) bottleChosen(X) :-  not bottleSkipped(X),
                         compliantBottle(Y,X).

(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`.
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more on that later!

Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

```
% These rules generate multiple answer sets:
(1) bottleSkipped(X)  :-  not bottleChosen(X),
                           compliantBottle(Y,X).
(2) bottleChosen(X) :-  not bottleSkipped(X),
                         compliantBottle(Y,X).

(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`.
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more on that later!

Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

```
% Alternatively we could use disjunction:

(4) bottleSkipped(X) v bottleChosen(X) :- compliantBottle(Y,X).


(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`.
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more on that later!

Answer Set Semantics

- Variable-free, non-disjunctive programs first!
- Rules

$$a \text{:- } b_1, \ldots, b_m, not\ \ c_1, \ldots, not\ \ c_n$$

where all a, b_i, c_j are atoms

- a *normal logic program* P is a (finite) set of such rules
- $HB(P)$ is the set of all atoms with predicates and constants from P.

Example

```
compliantBottle("axel","a").  wineBottle("a").

bottleSkipped("a") :- not bottleChosen("a"),
                      compliantBottle("axel","a").

bottleChosen("a") :- not bottleSkipped("a"),
                     compliantBottle("axel","a").

hasBottleChosen("axel") :- bottleChosen("a"),
                           compliantBottle("axel","a").
```

- $HB(P) =$ { `wineBottle("a")`, `wineBottle("axel")`, `bottleSkipped("a")`, `bottleSkipped("axel")`, `bottleChosen("a")` `bottleChosen("axel")`, `compliantBottle("axel","a")`, `compliantBottle("axel","axel")`, ... `compliantBottle("a","axel")` }

Answer Sets /2

Let

- P be a normal logic program
- $M \subseteq HB(P)$ be a set of atoms

Gelfond-Lifschitz (GL) Reduct P^M

The reduct P^M is obtained as follows:

1. remove from P each rule

$$a \text{:-}\ b_1, \ldots, b_m, not\ \ c_1, \ldots, not\ \ c_n$$

   where some c_i is in M

2. remove all literals of form $not\ \ p$ from all remaining rules

Answer Sets /3

- The reduct P^M is a Horn program
- It has the least model $lm(P^M)$

Definition

$M \subseteq HB(P)$ is an answer set of P if and only if $M = lm(P^M)$

Intuition:

- M makes an **assumption** about what is true and what is false
- P^M derives positive facts under the assumption of $not\,(\cdot)$ as by M
- If the result is M, then the assumption of M is "stable"

Computation of $lm(P)$

The least model of a *not*-free program can be computed by fixpoint iteration.

Algorithm Compute_LM(P)

Input: Horn program P;
Output: $lm(P)$

$new\_M := \emptyset$;
repeat
$M := new\_M$;
$new\_M := \{a \mid a \text{:-} b_1, \ldots, b_m \in P,\ \{b_1, \ldots, b_m\} \subseteq M\}$
until $new\_M == M$
return M

Examples

```
compliantBottle("axel","a"). wineBottle("a").
hasBottleChosen("axel") :- bottleChosen("a"),
                           compliantBottle("axel","a").
```

- P has no *not* (i.e., is Horn)
- thus, $P^M = P$ for every M
- the single answer set of P is
  $M = lm(P) =$
  { `wineBottle("a"), compliantBottle("axel","a")` }.

Examples II

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                          compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                          compliantBottle("axel","a").
```

Take M = { `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleSkipped("a")` }

- Rule (2) "survives" the reduction (cancel `not bottleChosen("a")`)
- Rule (3) is dropped

$lm(P^M) = M$, and thus M is an answer set

Examples II

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                          compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                          compliantBottle("axel","a").
```

Take M = { wineBottle("a"), compliantBottle("axel","a"), bottleSkipped("a") }

- Rule (2) "survives" the reduction (cancel not bottleChosen("a"))
- Rule (3) is dropped

$lm(P^M) = M$, and thus M is an answer set

Examples II

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                          compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                          compliantBottle("axel","a").
```

Take M = { wineBottle("a"), compliantBottle("axel","a"), bottleSkipped("a") }

- Rule (2) "survives" the reduction (cancel `not bottleChosen("a")`)
- Rule (3) is dropped

$lm(P^M) = M$, and thus M is an answer set

Examples III

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                         compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                                compliantBottle("axel","a").
```

Take M = { `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleChosen("a")`, `hasBottleChosen("axel")` }

- Rule (2) is dropped
- Rule (3) "survives" the reduction (cancel `not bottleSkipped("a")`)

$Im(P^M) = M$, and therefore M is another answer set

Examples III

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                         compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                                compliantBottle("axel","a").
```

Take M = { wineBottle("a"), compliantBottle("axel","a"), bottleChosen("a"), hasBottleChosen("axel") }

- Rule (2) is dropped
- Rule (3) "survives" the reduction (cancel not bottleSkipped("a"))

$Im(P^M) = M$, and therefore M is another answer set

Examples III

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                         compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                                compliantBottle("axel","a").
```

Take M = { wineBottle("a"), compliantBottle("axel","a"), bottleChosen("a"), hasBottleChosen("axel") }

- Rule (2) is dropped
- Rule (3) "survives" the reduction (cancel `not bottleSkipped("a")`)

$lm(P^M) = M$, and therefore M is another answer set

Examples IV

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                         compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                                compliantBottle("axel","a").
```

Take M = { `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleChosen("a")`, `bottleSkipped("a")`, `hasBottleChosen("axel")`, }

- Rules (2) and (3) are dropped

$Im(P^M)$ = { `wineBottle("a")`, `compliantBottle("axel","a")`} $\neq M$

Thus, M is not an answer set

Examples IV

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
                          compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
                         compliantBottle("axel","a").
(4) hasBottleChosen("axel") :-  bottleChosen("a"),
                                compliantBottle("axel","a").
```

Take M = { wineBottle("a"), compliantBottle("axel","a"), bottleChosen("a"), bottleSkipped("a"), hasBottleChosen("axel"), }

- Rules (2) and (3) are dropped

$Im(P^M)$ = { wineBottle("a"), compliantBottle("axel","a")} $\neq M$
Thus, M is not an answer set

Programs with Variables

- Like in Prolog, consider Herbrand models only!
- Adopt in ASP: no function symbols ("Datalog")
- Each clause is a shorthand for all its ground substitutions, i.e., replacements of variables with constants

E.g., `b(X) :- not s(X), c(Y,X).`
is with constants "axel","a" short for:

```
b("a") :- not s("a"), c("a","a").
b("a") :- not s("a"), c("axel","a").
b("axel") :- not s("axel"), c("axel","axel").

b("axel") :- not s("axel"), c("axel","a").
```

Programs with Variables /2

- The *Herbrand base of P*, $HB(P)$, consists of all ground (variable-free) atoms with predicates and constant symbols from P
- The grounding of a rule r, $Ground(r)$, consists of all rules obtained from r if each variable in r is replaced by some ground term (over P, unless specified otherwise)
- The grounding of program P, is $Ground(P) = \bigcup_{r \in P} Ground(r)$

Definition

$M \subseteq HB(P)$ is an answer set of P if and only if M is an answer set of $Ground(P)$

Inconsistent Programs

Program

```
p :- not p.
```

- This program has NO answer sets
- Let P be a program and p be a new atom
- Adding

  ```
  p :- not p.
  ```

  to P "kills" all answer sets of P

Constraints

- Adding

$$\texttt{p :- q}_1\texttt{,...,}\ \texttt{q}_m\ \texttt{, not r}_1\texttt{, ..., not r}_n\texttt{, not p.}$$

to P "kills" all answer sets of P that:

  - contain $\texttt{q}_1\texttt{,...,}\ \texttt{q}_m$, and
  - do not contain $\texttt{r}_1\texttt{,...,}\ \texttt{r}_n$

- Abbreviation:

$$\texttt{:- q}_1\texttt{,...,}\ \texttt{q}_m\ \texttt{, not r}_1\texttt{, ..., not r}_n\texttt{.}$$

This is called a "**constraint**" (cf. integrity constraints in databases)

Social Dinner Example II

Task

Add a constraint to simpleGuess.dlv in order to filter answer sets in which for some person no bottle is chosen

```
% This rule generates multiple answer sets:
(1) bottleSkipped(X)  :-  not bottleChosen(X),
    compliantBottle(Y,X).
(2) bottleChosen(X) :- not bottleSkipped(X),
    compliantBottle(Y,X).
% Ensure that each person gets a bottle.
(3) hasBottleChosen(X) :- bottleChosen(Z),
                                compliantBottle(X,Z).
(4) :- person(X), ?
```

Solution at simpleConstraint.dlv

Social Dinner Example II

Task

Add a constraint to simpleGuess.dlv in order to filter answer sets in which for some person no bottle is chosen

```
% This rule generates multiple answer sets:
(1) bottleSkipped(X)  :-  not bottleChosen(X),
    compliantBottle(Y,X).
(2) bottleChosen(X) :- not bottleSkipped(X),
    compliantBottle(Y,X).
% Ensure that each person gets a bottle.
(3) hasBottleChosen(X) :- bottleChosen(Z),
                               compliantBottle(X,Z).
(4) :- person(X), not hasBottleChosen(X).
```

Solution at simpleConstraint.dlv

Main Reasoning Tasks

Consistency

Decide whether a given program P has an answer set.

Cautious (resp. Brave) Reasoning

Given a program P and ground literals $l_1, \ldots, l_n$, decide whether $l_1, \ldots l_n$ simultaneously hold in every (resp., some) answer set of P

Query Answering

Given a program P and non-ground literals $l_1, \ldots, l_n$ on variables $X_1, \ldots, X_k$, list all assignments of values ν to $X_1, \ldots, X_k$ such that $l_1\nu, \ldots, l_n\nu$ is cautiously resp. bravely true.

- seamless integration of query language and rule language
- expressivity beyond traditional query languages, e.g. SQL)

Answer Set Computation

Compute some / all answer sets of a given program P.

Main Reasoning Tasks

Consistency

Decide whether a given program P has an answer set.

Cautious (resp. Brave) Reasoning

Given a program P and ground literals $l_1, \ldots, l_n$, decide whether $l_1, \ldots l_n$ simultaneously hold in every (resp., some) answer set of P

Query Answering

Given a program P and non-ground literals $l_1, \ldots, l_n$ on variables $X_1, \ldots, X_k$, list all assignments of values ν to $X_1, \ldots, X_k$ such that $l_1\nu, \ldots, l_n\nu$ is cautiously resp. bravely true.

- seamless integration of query language and rule language
- expressivity beyond traditional query languages, e.g. SQL)

Answer Set Computation

Compute some / all answer sets of a given program P.

Main Reasoning Tasks

Consistency

Decide whether a given program P has an answer set.

Cautious (resp. Brave) Reasoning

Given a program P and ground literals $l_1, \ldots, l_n$, decide whether $l_1, \ldots l_n$ simultaneously hold in every (resp., some) answer set of P

Query Answering

Given a program P and non-ground literals $l_1, \ldots, l_n$ on variables $X_1, \ldots, X_k$, list all assignments of values ν to $X_1, \ldots, X_k$ such that $l_1\nu, \ldots, l_n\nu$ is cautiously resp. bravely true.

- seamless integration of query language and rule language
- expressivity beyond traditional query languages, e.g. SQL)

Answer Set Computation

Compute some / all answer sets of a given program P.

Main Reasoning Tasks

Consistency

Decide whether a given program P has an answer set.

Cautious (resp. Brave) Reasoning

Given a program P and ground literals $l_1, \ldots, l_n$, decide whether $l_1, \ldots l_n$ simultaneously hold in every (resp., some) answer set of P

Query Answering

Given a program P and non-ground literals $l_1, \ldots, l_n$ on variables $X_1, \ldots, X_k$, list all assignments of values ν to $X_1, \ldots, X_k$ such that $l_1\nu, \ldots, l_n\nu$ is cautiously resp. bravely true.

- seamless integration of query language and rule language
- expressivity beyond traditional query languages, e.g. SQL)

Answer Set Computation

Compute some / all answer sets of a given program P.

Simple Social Dinner Example – Reasoning

- For our simple Social Dinner Example (simple.dlv), we have a single answer set
- Therefore, cautious and brave reasoning coincides.
- *compliantBottle("axel","a")* is both a cautious and a brave consequence of the program.
- For the query *person(X)*, we obtain the answers *"axel"*, *"gibbi"*, *"roman"*.

Social Dinner Example II – Reasoning

For simpleConstraint.dlv:

- The program has 20 answer sets.
- They correspond to the possibilities for all bottles being chosen or skipped.
- The cautious query *bottleChosen("a")* fails.
- The brave query *bottleChosen("a")* succeeds.
- For the nonground query *bottleChosen(X)*, we obtain under cautious reasoning an empty answer.

ASP vs Prolog

Under answer set semantics,

- the order of program rules does not matter;
- the order of subgoals in a rule does not matter;

"Pure" declarative programming, different from Prolog

- no (unrestricted) function symbols in ASP solvers available (finitary programs; other work in progress)

Disjunctive ASP

- The use of disjunction in rule heads is natural

```
man(X) v woman(X) :- person(X)
```

- ASP has thus been extended with disjunction

$$a_1 \vee a_2 \vee \cdots \vee a_k \text{:- } b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n$$

- The interpretation of disjunction is "minimal" (in LP spirit)
- Disjunctive rules thus permit to encode choices

Social Dinner Example II – Disjunctive Version

Task

Replace the choice rules in simpleConstraint.dlv

```
bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
```

with an equivalent disjunctive rule

```
 ?    ∨  ?    :-compliantBottle(Y,X).
```

Solution at simpleDisj.dlv. This form is more natural and intuitive!

- Very often, disjunction corresponds to such cyclic negation
- However, disjunction is more expressive in general, and can not be efficiently eliminated

Social Dinner Example II – Disjunctive Version

Task

Replace the choice rules in simpleConstraint.dlv

```
bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
```

with an equivalent disjunctive rule

```
bottleSkipped(X) ∨ bottleChosen(X) :-compliantBottle(Y,X).
```

Solution at simpleDisj.dlv. This form is more natural and intuitive!

- Very often, disjunction corresponds to such cyclic negation
- However, disjunction is more expressive in general, and can not be efficiently eliminated

Social Dinner Example II – Disjunctive Version

Task

Replace the choice rules in simpleConstraint.dlv

```
bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
bottleChosen(X)  :- not bottleSkipped(X), compliantBottle(Y,X).
```

with an equivalent disjunctive rule

```
bottleSkipped(X) ∨ bottleChosen(X) :-compliantBottle(Y,X).
```

Solution at simpleDisj.dlv. This form is more natural and intuitive!

- Very often, disjunction corresponds to such cyclic negation
- However, disjunction is more expressive in general, and can not be efficiently eliminated

Answer Sets of Disjunctive Programs

Define answer sets similar as for normal logic programs

Gelfond-Lifschitz Reduct P^M

Extend P^M to disjunctive programs:

1. remove each rule in $Ground(P)$ with some literal *not* a in the body such that $a \in M$
2. remove all literals *not* a from all remaining rules in $Ground(P)$

However, $lm(P^M)$ does not necessarily exist (multiple minimal models!)

Definition

$M \subseteq HB(P)$ is an answer set of P if and only if M is a minimal (wrt. $\subseteq$) model of P^M

Example

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") v bottleChosen("a") :-
    compliantBottle("axel","a").
(3) hasBottleChosen("axel") :- bottleChosen("a"),
                               compliantBottle("axel","a").
```

This program contains no *not* , so $P^M = P$ for every M
Its answer sets are its minimal models:

- M_1 = { `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleSkipped("a")` }
- M_2 = { `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleChosen("a")`, `hasBottleChosen("axel")` }

This is the same as in the non-disjunctive version!

Example

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") v bottleChosen("a") :-
    compliantBottle("axel","a").
(3) hasBottleChosen("axel") :- bottleChosen("a"),
                               compliantBottle("axel","a").
```

This program contains no *not* , so $P^M = P$ for every M
Its answer sets are its minimal models:

- M_1 = { wineBottle("a"), compliantBottle("axel","a"), bottleSkipped("a") }
- M_2 = { wineBottle("a"), compliantBottle("axel","a"), bottleChosen("a"), hasBottleChosen("axel") }

This is the same as in the non-disjunctive version!

Example

```
(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") v bottleChosen("a") :-
    compliantBottle("axel","a").
(3) hasBottleChosen("axel") :- bottleChosen("a"),
                               compliantBottle("axel","a").
```

This program contains no *not* , so $P^M = P$ for every M
Its answer sets are its minimal models:

- $M_1 = \{$ `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleSkipped("a")` $\}$
- $M_2 = \{$ `wineBottle("a")`, `compliantBottle("axel","a")`, `bottleChosen("a")`, `hasBottleChosen("axel")` $\}$

This is the same as in the non-disjunctive version!

Properties of Answer Sets

Minimality:

Each answer set M of P is a minimal Herbrand model (wrt $\subseteq$).

Generalization of Stratified Semantics:

If negation in P is layered ("P is stratified"), then P has a unique answer set, which coincides with the perfect model.

NP-Completeness:

Deciding whether a normal propositional program P has an answer set is NP-complete in general.
$\Rightarrow$ Answer Set Semantics is an expressive formalism;
Higher expressiveness through further language constructs (disjunction, weak/weight constraints)

Answer Set Solvers

NP-completeness:

Efficient computation of answer sets is not easy!
Need to handle

1. complex data
2. search

Approach:

- Logic programming and deductive database techniques (for 1.)
- SAT/Constraint Programming techniques for 2.

Different sophisticated algorithms have been developed (like for SAT solving)
There exist many ASP solvers (function-free programs only)

Answer Set Solvers on the Web

DLV	`http://www.dbai.tuwien.ac.at/proj/dlv/`
SModels	`http://www.tcs.hut.fi/Software/smodels/`
GnT	`http://www.tcs.hut.fi/Software/gnt/`
Cmodels	`http://www.cs.utexas.edu/users/tag/cmodels/`
ASSAT	`http://assat.cs.ust.hk/`
NoMore	`http://www.cs.uni-potsdam.de/~linke/nomore/`
XASP	distributed with XSB v2.6 `http://xsb.sourceforge.net`
aspps	`http://www.cs.engr.uky.edu/ai/aspps/`
ccalc	`http://www.cs.utexas.edu/users/tag/cc/`

- Some provide a number of extensions to the language described here.
- Rudimentary extension to include function symbols exist ($\Rightarrow$ finitary programs, Bonatti)
- Answer Set Solver Implementation: see Niemelä's ICLP tutorial [61]

Architecture of ASP Solvers

Typically, a two level architecture

1. Grounding Step

Given a program P with variables, generate a (subset) of its grounding which has the same models
DLV's grounder; lparse (Smodels), XASP, aspps
Special techniques used:

- "Safe rules" (DLV)
- domain-restriction (Smodels)

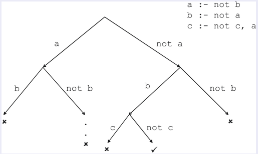
Architecture of ASP Solvers /2

2. Model search

This is applied for ground programs.

Techniques:

- Translations to SAT (e.g. Cmodels, ASSAT)
- Special-purpose search procedures (Smodels, dlv, NoMore, aspps)



- Backtracking procedures for assigning truth value to atoms
- Similar to DPPL algorithm for SAT Solving
- Important: Heuristics (which atom/rule to consider next)

Questiontime. . .

Coffee Break!

Questiontime. . .

Coffee Break!