2.0 VO 184.205, WS 2005/06

Verarbeitung deklarativen Wissens
(Declarative Knowledge Processing)

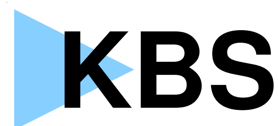
Answer Set Programming /1

Thomas Eiter

Institute of Information Systems

Knowledge Based Systems Group

Vienna University of Technology



`http://www.kr.tuwien.ac.at/staff/eiter`

Roadmap

1. Introduction & Background
2. Concepts
3. Disjunctive Logic Programs
4. Answer Set Solvers
5. Guess and Check Paradigm
6. The DLV System – Core
7. ASP Extensions
8. KR Applications
9. DLV Frontends

1. Introduction & Background

- Motivation to ASP
- Roots
- Logic Programming – Prolog revisited

French Phrases, Italian Soda (*Dell Logic Puzzles*)

- Six people sit at a round table
- Each drinks a different kind of soda
- Each plans to visit a different French-speaking country
- The person who is planning a trip to Quebec, who drank either blueberry or lemon soda, didn't sit in seat number one.
- Jeanne didn't sit next to the person who enjoyed the kiwi soda.
- The person who has a plane ticket to Belgium, who sat in seat four or seat five, didn't order the tangelo soda.
- ...

Question: *What is each of them drinking, and where is each of them going ?*

Example: 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	

- Fill in numbers from 1,...,9
- Each row / column must contain each number exactly once
- Each subtable must contain each number exactly once

Wanted!

- A general-purpose approach for modeling and solving these and many other problems
- Issues:
  - Diverse domains
  - Spatial and temporal reasoning
  - Constraints
  - Incomplete information
  - Frame problem
- Proposal: *Answer Set Programming (ASP) paradigm!*

Roots of ASP – Knowledge Representation (KR)

- How to model:
  - An agent's belief sets
  - Commonsense reasoning
  - Defeasible inferences
  - The Frame Problem
- 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 = “Programming in Logic”

Prolog

- Kowalski (1979):

ALGORITHM = LOGIC + CONTROL

- 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

Example

```
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, ...

Example, cont'd

- Problem: reverse 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
- Query `?- reverse([a|X],[b,c,d,a])` returns `X=[d,c,b]`
- Query `?- reverse([a|X],[b,c,d,b])` returns no substitutions (there is no 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:
  - `reverse([X|Y],Z) :- append(U,[X],Z), reverse(Y,U) .`
  - `reverse([X|Y],Z) :- reverse(Y,U), append(U,[X],Z) .`
  - termination vs. non-termination for query:

    `?- reverse([a|X],[b,c,d,b])`
- **Is this truly declarative programming?**

Logic Programming Desiderata

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

"Pure" declarative programming

- Prolog does not satisfy these desiderata
- Satisfied e.g. by the stable semantics of logic programs (introduced later)

2. Concepts

- Negation in Logic Programs
- Stratified Negation
- Answer Set Programming
- Stable Semantics

Negation in Logic Programs

- Why negation?
  - Natural linguistic concept
  - Facilitates declarative descriptions (definitions)
  - Needed for programmers convenience
- Clauses of the form:

$$p(\vec{X}) :- q_1(\vec{X}_1), \ldots, q_k(\vec{X}_k), \mathrm{not}(r_1(\vec{Y}_1)), \ldots, \mathrm{not}(r_l(\vec{Y}_l))$$

- Things get more complex!

Programs with negation

Prolog: “not (·)” means “Negation as Failure (to prove)”

Different from negation in classical logic!

Example:

```
man(dilbert).
single(X) :- man(X), not(husband(X)).
husband(X) :- fail.  % dummy declaration
```

Query:

```
?- single(X).
   X = dilbert
```

Programs with Negation /2

Modified rule:

```
man(dilbert).
single(X) :- man(X), not husband(X).
husband(X) :- man(X), not single(X).
```

Result ????

Problem: not a single minimal model!

Two alternatives:

- M_1= { `man(dilbert), single(dilbert)` },
- M_2 = { `man(dilbert), husband(dilbert)` }.

Which one to choose?

Semantics of logic programs with negation

Great Logic Programming Schism

1. **Single Intended Model Approach:**

   Select a *single* model of all classical models

   Agreement for so-called “stratified programs” (⇒ lecture on NMR):

   “ Perfect model”

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

Example: $\mathtt{boring}(\mathtt{chess}) :- \mathrm{not}(\mathtt{interesting}(\mathtt{chess}))$

Computed model $M = \{\mathtt{boring}(\mathtt{chess})\}$.

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

Program Layers

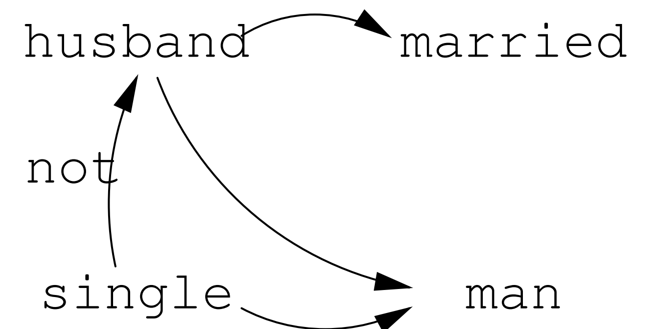
- Evaluate predicates bottom up in layers
- Methods works if there is no cyclic negation (layered negation)

Example:

P: $\texttt{man}(\texttt{dilbert}).$

$\texttt{husband}(\texttt{X}) :- \texttt{man}(\texttt{X}), \texttt{married}(\texttt{X}).$

$\texttt{single}(\texttt{X}) :- \texttt{man}(\texttt{X}), \text{not } \texttt{husband}(\texttt{X}).$



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

$M = \{\texttt{man}(\texttt{dilbert}), \texttt{single}(\texttt{dilbert})\}$

Multiple preferred models

```
man(dilbert).
single(X) :- man(X), not husband(X) .
husband(X) :- man(X), not single(X) .
```

- Assign to a program (theory) not one but **several** intended models!
  - For instance, all stable models (⇒ course NMR)
- How to interpret these semantics?
  - skeptical reasoning — resolution-based approaches complex (⇒ course on NMR)
  - *preferred models represent different solutions to a problem*

Answer Set Programming Paradigm

- Fundamental concept:

  Models, not proofs, **represent solutions**!

- Therefore, need techniques to **compute models** (not to compute proofs)

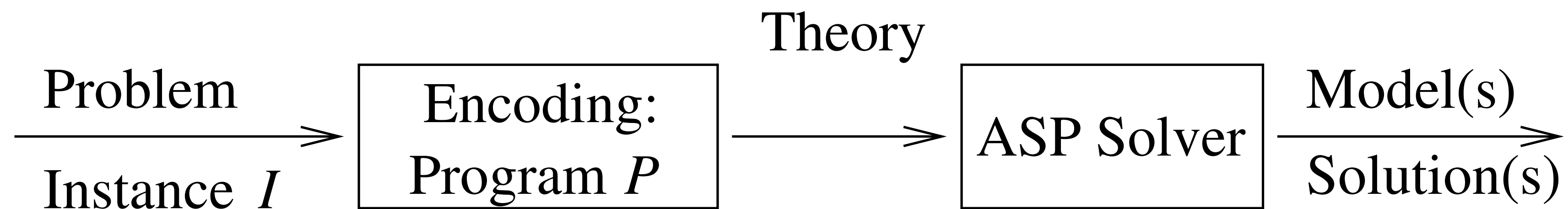
- What is model generation good for?

  Solve *search problems*

  - E.g., one/all solution(s) to the N-queens problem;
  - error diagnoses for a faulty system;
  - routes to reach the airport, ...

Solving search problems with ASP

General idea:



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)

Example: Graph 3-Coloring Problem

- **Given:** Graph $G = (V, E)$
- **Task:** Find an assignment of one of 3 colors (red, green, blue) to each vertex, such that adjacent vertices have different color.

Encoding to a logic program P (using disjunction here):

for each node $v \in V$	for each edge $(u, v) \in E$:
$\mathtt{v_r} \vee \mathtt{v_g} \vee \mathtt{v_b}.$	
$\mathtt{:-}\ \mathtt{v_r}, \mathtt{v_g}.$	$\mathtt{:-}\mathtt{u_r}, \mathtt{v_r}.$
$\mathtt{:-}\ \mathtt{v_r}, \mathtt{v_b}.$	$\mathtt{:-}\mathtt{u_g}, \mathtt{v_g}.$
$\mathtt{:-}\ \mathtt{v_b}, \mathtt{v_g}.$	$\mathtt{:-}\mathtt{u_b}, \mathtt{v_b}.$

The 3-colorings of G and models of P are in correspondence

Applications of ASP

ASP facilitates *declarative problem solving*

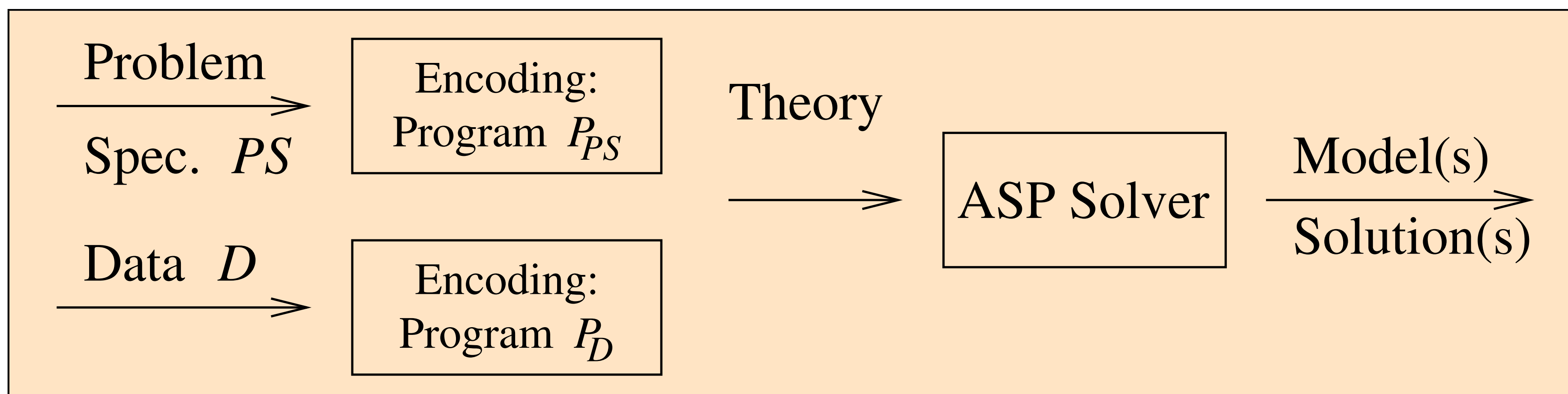
Problems in different domains (some with substantial amount of data), see

http://www.kr.tuwien.ac.at/research/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
- Integration of KR, DB, and search techniques
- Handling dynamic, knowledge intensive applications: data, defaults, exceptions, closures, ...

Example: 3-Coloring

- Problem specification PS: 3-Colorability condition

$$P_{PS} = \{\ \texttt{col}(\texttt{X},\texttt{r}) \vee \texttt{col}(\texttt{X},\texttt{g}) \vee \texttt{col}(\texttt{X},\texttt{b})\ \texttt{:-}\ \texttt{node}(\texttt{X}).$$
$$\texttt{:-}\ \texttt{node}(\texttt{X}), \texttt{col}(\texttt{X},\texttt{C}), \texttt{col}(\texttt{X},\texttt{D}), \texttt{C} <> \texttt{D}.$$
$$\texttt{:-}\ \texttt{edge}(\texttt{X},\texttt{Y}), \texttt{col}(\texttt{X},\texttt{C}), \texttt{col}(\texttt{Y},\texttt{C}).\}$$

- Data D: Graph $G = (V, E)$

$$P_D = \{\ \texttt{node}(v) \mid v \in V\} \cup \{\ \texttt{edge}(u,v) \mid (u,v) \in E\}.$$

- Correspondence 3-Colorings $\rightleftharpoons$ models:

v is colored with $c \in \{r, g, b\}$ iff $\texttt{col}(v, c)$ is in the model.

Example: Sudoku

- **Problem specification PS:**

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

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

Sudoku (cont'd)

- **Data D:**

```
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

ASP - Desiderata

- **Expressive power:**

  Capable of representing a range of problems

- **Ease of modeling:**
  - Intuitive semantics
  - Concise encodings: Availability of predicates and variables

    Not: SAT solvers do *not* support predicates and variables
  - Modular programming: global models can be composed from local models of components

- **Performance:**

  Fast solvers available

Stable Model Semantics

- Overview
  - Normal Logic Programs (NLP; propositional case)
  - Stable models (propositional case)
  - Programs with variables
  - Properties
- For background, see [1, 3, 4]
- Note: “Stable Models” amount to “Answer Sets” of extended NLPs

Normal Logic Programs

- Propositional case (no variables) first!
- Propositional language over a set of atoms *At*
- Clause

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

  - a, b_i, c_j are atoms
  - a is the *head* of the clause
  - literals $b_i, \text{not } c_j$ form the *body* of the rule

Stable models

Let

- P be a normal logic program
- M be a set of atoms

The *Gelfond-Lifschitz (GL) Reduct* P^M is obtained as follows:

1. remove each rule with some negative literal $\text{not}\ a$ in the body such that $a \in M$
2. remove all negative literals from all remaining rules

Stable models /2

- Reduct P^M is a Horn program
- It has the least model $lm(P^M)$
- **Definition:**

$M \subseteq HB(P)$ is a stable model 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 $\mathrm{not}(\cdot)$ as by M
- If the result is M, then the assumption of M is “stable”

Examples

The program P:

```
man.
 single :- man, not husband.
husband :- man, not single.
```

has two stable models: $M_1 = \{\texttt{man}, \texttt{single}\}$ and $M_2 = \{\texttt{man}, \texttt{husband}\}$

Case of M_1:

- Reduct P^{M_1} consists of

```
man.
single :- man.
```

- The least model of P^{M_1} is $\{\texttt{man}, \texttt{single}\}$ (= M_1)

Examples /2

M_2: Symmetry

Note: $M_3 = \{\mathtt{man}, \mathtt{single}, \mathtt{husband}\}$ is a classical model, but not a stable model:

- P^{M_3} consists only of

  ```
  man.
  ```

- Thus, $lm(P^{M_3}) = \{\mathtt{man}\}$ ($\neq M_3$)

Examples /3

- Program $P = Q \cup R$:

Q
```
p :- q, not s.
r :- p, not q , not s.
- - - - - - - - - - - - -
```
R
```
s :- not q.
q :- not s.
```

- P has two stable models: M_1 ={p,q} and M_2 ={s}

  Case of M_1:

  – P^{M_1}: `p :- q.  q.`

  – $lm(P^{M_1})$ ={p,q}= M_1

- Notice: R “feeds” its stable models {s} , {q} into higher module Q

Computation of $lm(P)$

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

Algorithm COMPUTE_LM(P)

Input: not-free 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

Inconsistent Programs

Program

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

- This program has NO stable models
- Let P be a program and p be a new atom
- Adding

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

  to P “kills” all stable models of P

“Killer” clauses — constraints

- Adding

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

  to P “kills” all stable models of P that:

  – contain $\texttt{q}_1\texttt{, ..., } \texttt{q}_m$, and

  – do not contain $\texttt{r}_1\texttt{, ..., } \texttt{r}_n$

- Abbreviation:

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

  Terminology: *“Constraint”*

Programs with Variables

- Like in Prolog, consider Herbrand models only!
- Adopt: No function symbols (“Datalog”)
- Program clause — a shorthand for all its ground substitutions

```
edge(1,2).
edge(1,3).
edge(2,4).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Programs with Variables /2

- Clause `path(X,Y) :- edge(X,Y).` represents:

```
path(1,1) :- edge(1,1).
path(1,2) :- edge(1,2).
path(2,1) :- edge(2,1).
path(2,2) :- edge(2,2).
path(1,3) :- edge(1,3).
...
```

- "Grounding:" Substitute the variables with values (ground terms, i.e., constants), in all possible ways.

Programs with Variables /3

- 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)

Note: *Ground*(P) is a propositional logic program

Definition:

$M \subseteq HB(P)$ is a stable model of P if and only if M is a stable model of *Ground*(P)

Example

```
r1:  man(dilbert).
r2:  woman(alice).
r3:   single(X) :- man(X), not husband(X).
r4:  husband(X) :- man(X), not single(X).
```

Ground(**r3**) = { `single(dilbert) :- man(dilbert)., not husband(dilbert).,`
`single(alice) :- man(alice), not husband(alice).` }

Ground(P):

```
man(dilbert).
woman(alice).
  single(dilbert) :- man(dilbert), not husband(dilbert).
  single(alice) :- man(alice), not husband(alice).
husband(dilbert) :- man(dilbert), not single(dilbert).
husband(alice) :- man(alice), not single(alice).
```

Example /2

Stable models:

- $M_1 = \{\texttt{man}(\texttt{dilbert}), \texttt{woman}(\texttt{alice}), \texttt{single}(\texttt{dilbert})\}$
- $M_2 = \{\texttt{man}(\texttt{dilbert}), \texttt{woman}(\texttt{alice}), \texttt{husband}(\texttt{dilbert})\}$

Properties of Stable Models

- **Minimality:**

  Each stable model 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 stable model, which coincides with the perfect model. ($\Rightarrow$ course on NMR)

- **NP-Completeness:**

  Deciding whether a propositional program P has a stable model is NP-complete in general. (NP-hardness e.g. by reduction from 3-Coloring).

$\Rightarrow$ Stable Semantics is an expressive formalism;

Higher expressiveness through further language constructs (e.g. disjunction, weak/weight constraints)

ASP vs Prolog

Under stable model 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)

ASP vs Prolog: Greatest Common Divisor

Problem: Compute the greatest common divisor of two positive integers n and m

- Classical Method: Euclid's algorithm

  recursion scheme (DLV code):

```
gcd(X,X,X) :- #int(X), X>1.
gcd(T,X,Y) :- X<Y, gcd(T,X,Y1), Y = Y1+X.
gcd(T,X,Y) :- X>Y, gcd(T,X1,Y), X = X1+Y.
```

  Similar code in Prolog

- But: A genius like Euclid is *not* the average user

- Question: Natural encoding ?

Natural Greatest Common Divisor Encoding

- ASP (DLV code):

```
% Declare when T divides a number N.
divisor(T,N) :- #int(T), #int(N), #int(M), N=T*M.
% Declare common divisors

cd(T,N1,N2) :- divisor(T,N1), divisor(T,N2).
% Single out non-maximal common divisors T

larger_cd(T,N1,N2) :- cd(T,N1,N2), cd(T1,N1,N2), T < T1.
% Apply double negation: take non non-maximal divisor

gcd(T,N1,N2) :- cd(T,N1,N2), not larger_cd(T,N1,N2).
```

- note: stratified program, use of “double negation”.

- A similar “natural” encoding in Prolog is tricky!

Disjunctive Logic Programs

- Normal logic programs have been extended with disjunction in rule heads:

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

- **Example**: Program P, describing the behavior of a reviewer while reviewing a paper:

$$\begin{aligned} \mathit{good} \vee \mathit{bad} &:- \mathit{paper} \\ \mathit{happy} &:- \mathit{good} \\ \mathit{angry} &:- \mathit{bad} \\ \mathit{smoke} &:- \mathit{happy} \\ \mathit{smoke} &:- \mathit{angry} \\ \mathit{paper} &:- \end{aligned}$$

Semantics

- Positive programs P with disjunction do not always have a single minimal model (i.e., $lm(P)$ may not exist)
- In the above example, the following models are minimal:

$$M_1 = \{\textit{paper}, \textit{good}, \textit{happy}, \textit{smoke}\} \text{ and}$$
$$M_2 = \{\textit{paper}, \textit{bad}, \textit{angry}, \textit{smoke}\}.$$

- In a multiple model view, the minimal models have been proposed as preferred models (cf. also Closed World Reasoning, Circumscription)
- The stable models of a positive disjunctive logic program P are its minimal (Herbrand) models, denoted $mmod(P)$

Disjunction – Examples

- Disjunction is interpreted *minimally*

```
a v b v c.
```

mmod(P) = { {a}, {b}, {c} }

- actually *subset minimal*

```
a v b.
a v c.
```

mmod(P) = { {a}, {b,c} }

- **Definition:**

Model $M \subseteq HB(P)$ of P is minimal, if no model $N \subset M$ of P exists.
By $mmod(P)$ we denote the set of minimal models of P,

Disjunction – Examples /2

- ```
  a v b.
  a :- b.
  ```

  Only {a} is minimal; {b,a} is not minimal!

- Minimal interpretation may not be *exclusive*

  ```
  a v b.
  b v c.
  a v c.
  ```

  $mmod$(P) = { {a,b}, {a,c}, {b,c} }
  ```

Stable models of Disjunctive Programs with Negation

- Define stable models similar as for normal logic programs
- Extend the Gelfond-Lifschitz Reduct P^M to disjunctive programs:

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

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

$M \subseteq HB(P)$ is a stable model of P if and only if $M \in mmod(P^M)$

Example

$$
\begin{aligned}
\textit{good} \vee \textit{bad} &:- \textit{paper} \\
\textit{happy} &:- \textit{good} \\
\textit{angry} &:- \textit{bad} \\
\textit{smoke} &:- \textit{happy} \\
\textit{smoke} &:- \textit{angry} \\
\textit{paper} &:- \\
\textit{listen_radio} &:- \mathrm{not}\ \textit{angry}
\end{aligned}
$$

Stable models:

$$M_1 = \{\textit{paper}, \textit{good}, \textit{happy}, \textit{smoke}, \textit{listen_radio}\} \text{ and}$$

$$M_2 = \{\textit{paper}, \textit{bad}, \textit{angry}, \textit{smoke}\}.$$

Disjunction vs Unstratified Negation

- Disjunction may be rewritten to normal rules (sometimes)

P_1: *man(dilbert)*

husband$(X) \lor$ *single*(X) :− *man*(X)

and

P_2: *man(dilbert)*

husband(X) :− *man*(X), not *single*(X)

single(X) :− *man*(X), not *husband*(X)

have the same stable models:

$$M_1 = \{\textit{dilbert}, \textit{husband}\} \text{ and } M_2 = \{\textit{dilbert}, \textit{single}\}.$$

- Here, "v" can be eliminated by "shifting" literals from the head to the body.

- However, the usage of disjunction is more natural

4. Answer Set Solvers

- **NP-completeness:** Efficient computation of answer sets is not easy!

  Need to handle

  1. complex data
  2. search

- Approach used:
  - 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 exists 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 (1, 2)	`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
  (⇒ *finitary programs*, Bonatti)
- Answer Set Solver Implementation: see Niemelä's ICLP tutorial [5]

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, discussed later)
- domain-restriction (Smodels)

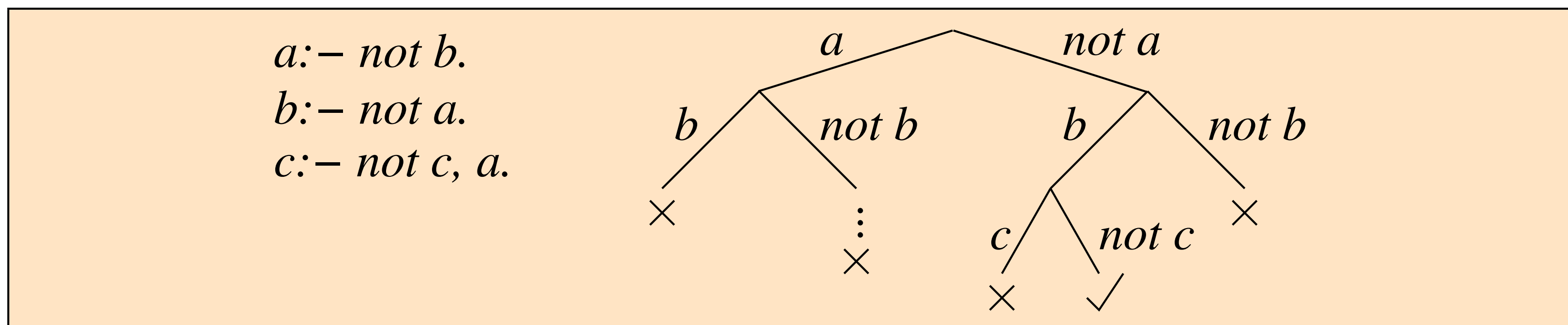
Architecture of ASP Solvers /2

2. Model search:

This is applied for ground programs.

Techniques:

- Translations to SAT (e.g. Cmodels, Cmodels2, ASSAT)
- Special-purpose search procedures (Smodels, dlv, NoMore, aspps)
  - Backtracking procedures for assigning truth value to atoms
  - Similar to Davis-Putnam-Logemann-Loveland algorithm for SAT Solving



  - Important: Heuristics (which atom/rule to consider next)

Bibliography

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, 2003.

[2] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes and J. A. Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Proceedings*, volume 3229 of *LNCS*, pages 200–212. Springer Verlag, 2004.

[3] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[4] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

Remark: A very elegant characterization of stable models/answer sets has been recently given in [2]: M is a stable model of P iff M is a minimal model of the program $\{H \mathbin{:\!-} B \in \textit{Ground}(P) \mid M \models B\}$.

[5] I. Niemelä. The implementation of answer set solvers, 2004. Tutorial at ICLP 2004. Available at `http://www.tcs.hut.fi/~init/papers/niemela-iclp04-tutorial.ps.gz/`.

[6] A. Provetti and S. T. Cao, editors. *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, Stanford, CA, March 2001. AAAI Press.