Answer Set Programming and Extensions
Unit 2 – Program Evaluation

Thomas Eiter

Institut für Informationsysteme, TU Wien

VTSA Summer School 2016, Liège, August 29-September 2, 2016

Austrian Science Fund (FWF) grants P24090, P26471, P27730
Unit Outline

1. Introduction
2. Solvers
3. Intelligent Grounding
4. Solving
5. Grounding Bottleneck
ASP Program Evaluation

Efficient evaluation of answer set programs is challenging

- NP-/$\Sigma^P_2$-completeness of consistency already in the ground (propositional) case
- different reasoning tasks
  - single solution
  - solution enumeration
  - query answering (reasoning from multiple models)
    projection, intersection, union
  - (multi-objective) optimization and preference
  - additional constructs (weak constraints, aggregates, function symbols, etc.)
- complex data in applications (large data volumes)
Answer Set Solvers

(see also http://en.wikipedia.org/wiki/Answer_set_programming)

- ASPERIX  www.info.univ-angers.fr/pub/claire/asperix/
- ASSAT  assat.cs.ust.hk/
- CLASP¹  potassco.sourceforge.net/#clasp/
- CMODELS  www.cs.utexas.edu/users/tag/cmodels/
- DLV²  www.dbai.tuwien.ac.at/proj/dlv/
- ASPTOOLS  research.ics.aalto.fi/software/asp/
- 0 ME-ASP  www.mat.unical.it/ricca/me-asp/
- OMIGA  www.kr.tuwien.ac.at/research/systems/omiga
- SMODELS  www.tcs.hut.fi/Software/smodels/
- WASP  www.mat.unical.it/ricca/wasp/
- XASP  xsb.sourceforge.net/,
  distributed with XSB

¹ + CLASP\textsuperscript{D}, CLINGO, CLINGCON etc. (http://potassco.sourceforge.net/)
² + DLV\textsuperscript{HEX}, DLV\textsuperscript{DB}, DLT, DLV-COMPLEX, ONTO-DLV etc.

- Many ASP solvers are available (mostly function-free programs)
- Efforts to realize tractable fragments (downscaling)
- clasp was first ASP solver competitive to top SAT solvers
- ASP Solver competition (at LPNMR, since 2007)
Evaluation Approaches

Different methods and evaluation approaches:

- resolution-based [Bonatti et al., 2008]
- forward chaining [Lefèvre et al., 2015]
- lazy grounding: [Palù et al., 2009], [Dao-Tran et al., 2012], [de Cat et al., 2012]
- translation-based (see below)
- meta-interpretation [E_ et al., 2003]

Predominant:

intelligent grounding + model search (solving)
Architecture of ASP Solvers

Common: two level architecture

1 Intelligent Grounding

Given a program \( P \), generate a (subset) of \( \text{grnd}(P) \) that has the same models

2 Solving: Model Search

More complicated than in SAT/CSP Solving:

- candidate generation (classical model)
- model checking (stability, foundedness!)

- for SAT, model checking is in logarithmic space (in fact in \( \text{ ALOGTIME} \))
- for normal propositional programs, model checking is PTime-complete
- for disjunctive propositional programs, model checking is \( \text{co-NP-complete} \)
Intelligent Grounding

- Grounding is a hard problem
  
  \[ bit(0). \quad bit(1). \]  
  \[ p(X_1, \ldots, X_n) \leftarrow bit(X_1), \ldots, bit(X_n). \]  

naive: \(2^n\) rules

- In the worst case, grounding time is exponential in the input size.
- Getting the “right” rules is difficult, already for positive programs
  - rule matching is NP-hard
  - deciding rule relevance is \textsc{Exptime}-hard

Efficient grounding is at the heart of current systems

- dlv’s grounder (built-in);
- lparse (smodels), gringo (clasp)

Special techniques used:

- “safe rules” (dlv): each variable in a rule occurs in the body in an unnegated atom with non-built-in predicate (exception: \(X = c\))
- domain-restriction (smodels)
- Deductive DB methods: semi-naive evaluation, magic sets, …
Grounding: Basic Ideas

- Avoid useless rule instances not applicable in any answer set

Example

\[
\begin{align*}
  &\text{c}(1, 2). \\
  &\text{a}(X)|\text{b}(Y) \leftarrow \text{c}(X, Y).
\end{align*}
\]

The full instantiation of the only rule yields

\[
\begin{align*}
  &\text{a}(1) \mid \text{b}(1) \leftarrow \text{c}(1, 1). \\
  &\text{a}(2) \mid \text{b}(1) \leftarrow \text{c}(2, 1). \\
  &\text{a}(2) \mid \text{b}(2) \leftarrow \text{c}(2, 2). \\
  &\text{a}(1) \mid \text{b}(2) \leftarrow \text{c}(1, 2).
\end{align*}
\]

- (5)-(7) are useless: \text{c}(1,1), \text{c}(2,1), and \text{c}(2,2) do not occur in heads

Basic instantiation:

- body matching of variables (assignment)
- backtracking on assignment
- use evaluation order: \[r_1 : p \leftarrow B_1. \quad r_2 : q \leftarrow p, B_2.\]
  - must evaluate \(r_1\) before \(r_2\)
  - \(\Rightarrow\) use syntactic dependency graph
Optimizations

Intelligent grounders use a big deal of optimization techniques

- **literal ordering** in the body ( = join optimization in DBs)
- **backjumping algorithm** [Perri et al., 2008]
- **magic sets**
  - classic **magic sets technique** (standard Datalog): emulate goal-directed query answering (top-down) in bottom up computation
    - restrict rule instances using “magic predicates”
  - **dynamic magic sets** [Alviano et al., 2012] for disjunctive programs with negation, exploit magic set information also during search [Alviano and Faber, 2011]
  - useful for hard problems in consistent query answering [Manna et al., 2015]

- **parallel instantiation:**
  - exploit modern multi-core/-proc architectures with load balancing and granularity control [Perri et al., 2013]
  - lparse [Pontelli et al., 2003]
Grounding: Function Symbols

- Function symbols: infinite grounding, if even answer sets are finite

**Example**

\[
\begin{align*}
p(0). & & (9) \\
p(X) & \leftarrow p(f(X)), \text{not } q(X). & (10) \\
q(X) & \leftarrow p(f(X)), \text{not } p(X). & (11)
\end{align*}
\]

- dlv-grounder, gringo admit recursive function symbols

- Termination for *finitely-ground (FG) programs* [Calimeri et al., 2008]
  - problem: only semi-decidable
  - Turing complete (each computable function expressible)

- **Practice**: decidable syntactic / semantic restrictions (preprocessing)
  - \(\omega\)-restrictedness [Syrjänen, 2001] (lparse)
  - \(\lambda\)-restrictedness [Gebser et al., 2007] (gringo x, x<3)
  - argument-restrictedness [Lierler and Lifschitz, 2009]
  - finite domain [Calimeri et al., 2008] (dlv)
  - \(\Gamma\)-acyclic programs [Greco et al., 2012]
Finitely-Ground (FG) Programs

Idea:

- **Modularization**: split a program $P$ into modules $P_1 \ldots P_n$, define a proper module ordering $\prec$ for bottom up evaluation
- **Simplification**:
  - instantiate a given module $P_i$ of $P$ by exploiting instantiations of previous modules $\rightarrow$ ”intelligent” instantiation
  - add only ground rules whose heads have a chance to be true in some answer set

- All $P_i$ are finitely instantiated along $\prec \Rightarrow P'$ as above is found

Pros:

- FG programs are very expressive: they correspond to terminating computations of Turing Machines
- FG programs have a finite set of finite answer sets

Cons:

- Undecidable (semi-decidable) in general, not recognizable
Finite Domain (FD) Programs

- Simple, decidable subclass of FG programs [Calimeri et al., 2008]

- All arguments of a program $P$ must be finite domain

- Use an argument dependency graph $AG(P)$ of $P$:
  - nodes = argument positions of predicates in $P$; eg. edge[1], path[2],
  - edge $p[i] \rightarrow q[j]$, if in some rule,
    - (i) $p(t_1, \ldots, t_n)$ is in the head,
    - (ii) $q(s_1, \ldots, s_m)$ occurs positive in the body, and
    - (iii) $t_i$ and $s_j$ share some variable.

- Program $P$ is finite domain if, for each atom $p(t_1, \ldots, t_n)$ in the head of a rule $r \in P$, and for each argument $p[i]$ of $p$, either
  - (i) $t_i$ is variable-free; or
  - (ii) $t_i$ occurs as a (sub)term of an atom not under “not” in the body; or
  - (iii) each variable occurring in $t_i$ occurs in a positive body atom $q(s_1, \ldots, s_m)$ in some $s_j$ s.t. $p[i]$ and $q[j]$ are not on a cycle in $AG(P)$. 
Examples

More than two shared interests

\[ \text{sharedInterests}(U_1, U_2, \#\text{intersection}(S_1, S_2)) \leftarrow \text{user}(U_1, S_1), \text{user}(U_2, S_2), U_1 \neq U_2. \]

\[ \text{proposeConnection(pair}(U_1, U_2)) \leftarrow \text{sharedInterests}(U_1, U_2, S), \#\text{card}(S) > 2. \]

is finite domain.
Examples (ctd.)

Paths in a graph

\[
\begin{align*}
\text{path}([X, Y]) & \leftarrow \text{edge}(X, Y). \\
\text{path}([X|[Y|W]]) & \leftarrow \text{edge}(X, Y), \text{path}([Y|W]), \text{not} \#\text{member}(X, [Y|W]).
\end{align*}
\]

is not finite domain:

\[
\begin{tikzpicture}
    \node (path1) at (0,0) {path[1]};
    \node (edge1) at (-1,-1) {edge[1]};
    \node (edge2) at (1,-1) {edge[2]};
    \node (member1) at (-1,-2) {\#member[1]};
    \node (member2) at (1,-2) {\#member[2]};
    \draw[->] (path1) to (edge1);
    \draw[->] (path1) to (edge2);
    \draw[->] (edge1) to (member1);
    \draw[->] (edge2) to (member2);
\end{tikzpicture}
\]

- **dlv**: “Termination is not guaranteed ... Use option -nofinitecheck in order to evaluate this program anyway.”
- still finitely-ground for finite edge ⇒ dlv terminates
- easily modified to compute e.g. all Hamiltonian paths from s to t:
Solving: Model Search

- Applied to ground programs.
- Early solvers (e.g. smodels, dlv): native methods
  - inspired by Davis-Putnam-Logemann Loveland (DPLL) for SAT
    - 3 basic operations: decision, propagate, backtrack
  - special propagation for ASP, e.g.,
    - smodels: 5 propagators
    - dlv: *well-founded, must-be-true* propagation (supportedness)

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

- important: heuristics (which atom/rule is next?) [Faber et al., 2001]
- chronological backtrack-search was improved introducing backjumping and look-back heuristics [Maratea et al., 2008]
- Recent: Abstract solving [Brochenin et al., 2014]

- Stability check: unfounded sets, reductions to UNSAT [Leone et al., 1997b], [Koch et al., 2003]
ASP Solving Approaches

- Predominant to date: modern SAT techniques
- Export of techniques for optimal answer sets to SAT
- **Genuine conflict-driven ASP solvers**
  - clasp [Gebser et al., 2012], wasp [Alviano et al., 2013]

- **Translation based solving**
  - SAT: assat [Lin and Zhao, 2004], cmodels [Lierler, 2005], lpsat [?], [Giunchiglia et al., 2006] (multiple SAT solver calls)
  - Mixed IP: [Bell et al., 1994], [Liu et al., 2012]; CPLEX backend

- **Cross translation**: intermediate format to ease cross translation
  - SAT modulo acyclicity [Gebser et al., 2014]
    - interconnect graph based constraints with clausal constraints
    - can postpone choice of the target format to last step

- **Portfolio solvers**
  - claspfolio [Gebser et al., 2011]: combines variants of clasp
  - me-asp [Maratea et al., 2014]: multi-engine portfolio ASP solver
Program Decomposition

- A set $S$ of (ground) atoms is a **splitting set** of a (ground) program $P$, if for each rule $r : H(r) \leftarrow B(r)$ in $P$ either (i) $H(r) \cup B(r) \subseteq S$ or $H(r) \cap S = \emptyset$.
- $b_S(P)$ and $t_S(P)$ denotes all rules (i) and (ii), resp.

**Example**

E.g. $P = \left\{ \begin{array}{l} f_1 \quad \text{man}(d). \\ r_1 \quad \text{husband}(d) \lor \text{single}(d) \leftarrow \text{man}(d). \\ r_2 \quad \text{bachelor}(d) \leftarrow \text{man}(d) \text{ not husband}(d). \end{array} \right\}$

$S = \{ \text{man}(d), \text{husband}(d), \text{single}(d) \}$, $b_S(P) = \{ f_1, r_1 \}$, $t_S(P) = \{ r_2 \}$

**Splitting Set Theorem, Lifschitz and Turner [1994]**

For any splitting set $S$ of program $P$, $\text{AS}(P) = \bigcup_{M \in \text{AS}(b_S(P))} \text{AS}(t_S(P) \cup M)$.

- Splitting sets allow for (de)composition and bottom up evaluation
  - $\text{AS}(b_S(P)) = \{ M_1 = \{ \text{man}(d), \text{single}(d) \}, M_2 = \{ \text{man}(d), \text{husband}(d) \} \}$,
  - $\text{AS}(t_S(P) \cup M_1) = M_1 \cup \{ \text{bachelor}(d) \}$, $\text{AS}(b_S(P) \cup M_2) = M_2$

- has also nonground versions
- solvers identify splitting sets (e.g. strongly connected components)
Characterizing Answers Sets in Classical Logic

- Answer sets of a program $P$ are special classical models of $P$
  - view rule $r : a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$ ($= a \leftarrow B(r)$) as implication $b_1 \land \cdots \land b_m \land \neg c_1 \land \cdots \land \neg c_n \rightarrow a$ ($=: BF(r) \rightarrow a$)

- Clark [1978]: for Horn $P$, turn $\rightarrow$ into $\leftrightarrow$:

$$CF(P) = \{ \bigvee_{a \leftarrow B(r) \in P} BF(r) \rightarrow a \mid a \text{ is an atom} \}$$

- “Completion” $CF(P)$ captures $AS(P)$ for special normal programs

**Theorem**

*For normal logic programs $P$ without cyclic positive recursion (called tight programs), $AS(P)$ is captured by $CF(P)$ (i.e., its classical models).*

**Example**

- $P = \{ p \leftarrow q \}$: $CF(P) = \{ p \leftrightarrow q, q \leftrightarrow \bot \}$ $P$ is tight ✓

- $P = \{ p \leftarrow q; q \leftarrow p, \text{not } r \}$: $CF(P) = \{ p \leftrightarrow q, q \leftrightarrow p \land \neg r, r \leftrightarrow \bot \}$ $P$ is not tight ×
Loop Formulas

\( AS(P) \) is captured by adding loop formulas \( LF(P) \) [Lin and Zhao, 2002]

**Theorem**

For any normal program \( P \), \( AS(P) = CF(P) \cup LF(P) \).

- Informally, \( LF(P) \) contains clauses enforcing that a positive cycle \( C \) can only be true if some \( r : a \leftarrow B(r) \) exists s.t. \( a \in C \), \( B(r) \) is satisfied and no atom from \( C \) occurs positively in \( B(r) \).

**Example**

- \( P = \{p \leftarrow q\} \): \( LF(P) = \emptyset \)

\[
CF(P) \cup LF(P) = \{p \leftrightarrow q, q \leftrightarrow \bot\} \equiv \neg p \land \neg q
\]

- \( P = \{p \leftarrow q, q \leftarrow p, \text{not } r\} \):

\[
LF(P) = \{p \land q \rightarrow \bot\}
\]

\[
CF(P) \cup LF(P) = \{p \leftrightarrow q, q \leftrightarrow p \land \neg r, r \leftrightarrow \bot, p \land q \rightarrow \bot\} \equiv \neg p \land \neg q \land \neg r
\]
Loop Formulas /2

- Also feasible for disjunctive programs, non-propositional program [Lee and Lifschitz, 2003], [Chen et al., 2006], and aggregates.

- Based on this, some ASP solvers employ SAT solvers.

- Semantically, loop formulas correspond to *unfounded-freeness* [Leone et al., 1997a], [Lee, 2005]

- **Downside:** exponentially many loop formulas exist in general [Lifschitz and Razborov, 2006]
  - add loop formulas *on the fly*

- Alternative: avoid explicit loop formulas, do cautious description of reachability (*ordered completion*, [Asuncion et al., 2012])
Conflict-Driven ASP Solving

■ **Breakthrough in SAT:** conflict driven clause learning (CDCL)  
[Silva *et al.*, 2009]

- represent clauses as nogoods (inadmissible assignments)
- propagate with conflict-driven backtracking
- clause (nogood) learning from conflicts

**Example**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>¬(a \lor b \lor c), ((d \lor a)), ((d \lor \neg b))</td>
<td>(\leadsto) nogoods {(Ta, Fb, Fc}), {(Fd, Fa}), {(Fd, Tb}}</td>
<td></td>
</tr>
<tr>
<td>assignment {(Fc)}; set (Fd)</td>
<td>(\Rightarrow) derive (Ta, Fb): conflict!</td>
<td></td>
</tr>
<tr>
<td>learn noogod {(Fc, Fd)}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

■ **ASP: key idea** [Gebser *et al.*, 2012]

- exploit \(AS(P) = CF(P) + LF(P)\)
- use completion nogoods for \(CF(P)\), loop nogoods for \(LF(P)\)
- map inferences in ASP onto unit propagation on nogoods
- enable learning
A Glimpse on clasp

Basic CDCL algorithm

1. initializes
2. loop
   - propagate completion, loop, and recorded nogoods
3. if no conflict then
   - if all variables assigned then
     - return variable assignment
   - else
     - if no nogood recorded then decide
     - else
       - if top-level conflict then return unsatisfiable
       - else
         - analyze
   - otherwise
4. backjump

Use unit propagation for completion nogoods (to a fixpoint)

Special loop nogood propagator

Workflow:

1. $P \Rightarrow P'$ simplify
2. $P' \Rightarrow CF(P')$ (which is static)
3. construct nogoods for $LF(P')$ on demand, using a dependency graph
multi-threading: parallel execution of clasp (search space splitting, competitive strategies)
  - learned conflict nogoods can be exchanged

further post propagation is usable for theory-specific propagation
  - exploited e.g. by solvers clingcon, dlvhex
Preference and Optimization

Two classes [Delgrande *et al.*, 2004]

1. **Prescriptive**: condition on how rules are/must be applied
   - Implementation by compilation [Delgrande *et al.*, 2003] or meta-interpretation [E_* et al.*, 2003]

2. **Descriptive**: how answer sets relate (comparison)
   - Preferences among the answer sets of a program [Brewka *et al.*, 2003], [Sakama and Inoue, 2000], [Son and Pontelli, 2006]

Class 2) has usually higher computational complexity than 1)

Tools:

- ASPRIN system [Brewka *et al.*, 2015]: general and flexible framework to compute optimal answer sets relative to preferences among them

- plp tool [Delgrande *et al.*, 2000]: compiler for preferences
  http://www.cs.uni-potsdam.de/~torsten/plp/
Grounding Bottleneck

- Recall: Grounding is expensive in general
- **Practical Grounding Bottleneck** [E_ et al., 2007b]
  - Unless \( \text{EXPTIME} = \text{PSPACE} \), grounding + solving needs unavoidably exponential space in general
  - Certain programs are evaluable in polynomial space, while the top grounders still produce an exponentially large ground programs
- Example: programs \( P \) with predicate arities bounded by a constant

**Theorem** [E et al., 2007a]

In the datalog case, deciding whether a bounded-arity program \( P \) has some answer set is (i) \( \Sigma_2^P \)-complete if \( P \) is normal, and (ii) \( \Sigma_2^P \)-complete if \( P \) is disjunctive.

- For some families \( P_n = P_1, P_2, \ldots \) of bounded arity programs, the grounders produce exponentially growing output
Examples

- reachability over paths of length $k$ in a directed graph $G$, given by $e(v_1, v_2)$:

  \[
p_k(X_1, X_k) \leftarrow e(X_1, X_2), \ldots, e(X_{k-1}, X_k).
  \]

  \[
  \text{reachable}(X, Y) \leftarrow p_k(X, Y).
  \]

  \[
  \text{reachable}(X, Y) \leftarrow \text{reachable}(X, Z), p_k(Z, Y).
  \]

- for $k = 2n + 1$ there are $2^n$ paths from $v_1$ to $v_{n+1} \Rightarrow p_k(v_1, v_{n+1})$ has $2^n$ derivations
- if $G$ is nondeterministically chosen by other rules (possibly depending on $p_k$) grounders don’t optimize
Monadic choice:

\[ b(0). \ b(1). \quad (1) \]
\[ p(X) \leftarrow b(X), \text{not} \ q(X). \quad (2) \]
\[ q(X) \leftarrow b(X), \text{not} \ p(X). \quad (3) \]
\[ r(X_1) \leftarrow p(X_1), \ldots, p(X_k), b(X_1), \ldots b(X_k). \quad (4) \]

- as above, exponential grounding result
- modular evaluation (1),(2) before (3) can be spoiled e.g. with adding
  \[ q(X) \leftarrow r(X) \]
Nonground ASP: Special Techniques

Techniques to address grounding explosion

- **lazy grounding**
- for **bounded predicate arities**
  - few generating rules property [E_ et al., 2010]
  - answer set size and number of supporting rules is polynomial
  - can run in polynomial space

- **rule decomposition** [Morak and Woltran, 2012], [Bichler et al., 2016]
  - split (long) rule \( r : H \leftarrow B \) using decomposition of its hypergraph \( H_r = (V_r, E_r) \)
  - \( V_r = \) variables in \( B \), \( E_r = \) literals (labeled with predicate)
  - the decomposition effect relies on the treewidth of \( H_r \), \( \text{tw}(H_r) \) ("treelikeness")
  - avoids explosion for \( \text{tw}(H_r) \leq k \), constant \( k \) (thus bounded arities)
  - tool (for full ASP-syntax)
    http://dbai.tuwien.ac.at/research/project/lpopt/
Example

- Single rule \( r : h(X, W) \leftarrow e(X, Y), e(Y, Z), \text{not } e(Z, W), e(W, X) \)
  - \( V_r = \{X, Y, X, W\} \),
  - \( E_r = \{e_1 = \{X, W\}, e_2 = \{X, Y\}, e_3 = \{Y, Z\}, e_4 = \{Z, W\}, e_5 = \{W, X\}\} \)

- Tree Decomposition: \( \mathcal{T} = (T, \chi), T = (\{n_1, n_2\}, \{n_1 \rightarrow n_2\}) \)

  \[
  n_1 \quad \chi(n_1) = \{X, Y, W\}, \text{ covers } h(X, W), e(X, Y), e(W, X)
  \]

  \[
  \downarrow
  \]

  \[
  n_2 \quad \chi(n_2) = \{Y, Z, W\}, \text{ covers } e(Y, Z), \text{not } e(Z, W)
  \]

- Decomposition algorithm yields (fresh predicate \( t_1, \text{dom}_W \)):

  \[
  h(X, W) \leftarrow e(X, Y), e(X, W), t_1(Y, W).
  \]

  \[
  \text{dom}_W(W) \leftarrow e(W, X).
  \]

  \[
  t_1(Y, W) \leftarrow e(Y, Z), \text{not } e(Z, W), \text{dom}_W(W).
  \]

- Runs for bounded treewidth, i.e. \( \max_{n \in T} |\chi(n_i)| - 1 = O(1) \) in polynomial space.
Application: Propositional ASP and QSAT

Bichler et al. [2016] (ICLP 2016)

- Transform prop. disjunctive ASP $P$ into normal ASP w/ bounded arities
  - represent $P$ as facts
  - few fixed rules (independent of $P$): guess $M$, check $M \models P$
  - few long rules: check foundedness of $M$

- Customized Encoding: 2-QSAT ($\forall \exists$-QBF satisfiability)
- Similarly, encode $\Sigma_3^P$ problems into disjunctive ASP w/ bounded arities
  - 3-QSAT ($\exists \forall \exists$-QBF satisfiability)
  - stable cautious abductive reasoning [E_ et al., 1997]

- For 2-QSAT, results are quite competitive to state-of-the-art solver DepQBF Solver [Lonsing and Egly, 2015]; for 3-QSAT problems, complementary
- Also good results for abductive reasoning
QSAT Encodings

QBF  \( \Phi = \forall x_1, \ldots, x_m \exists y_1, \ldots, y_n (c_1 \land \cdots \land c_k) \),  
where  \( c_i = \ell_{i,1} \lor \ell_{i,2} \lor \ell_{i,3} \)

- “classical” encoding:

  \[
  \begin{align*}
  &\text{var}(x_i). \text{ for all } x_i \\
  &\text{var}(y_j). \text{ exists}(y_j). \text{ for all } y_j \\
  &\text{occ}_h(c_i, z, 0). \text{ s.t. } \ell_{i,h} = \neg z, \quad \text{occ}_h(c_i, z, 1). \text{ s.t. } \ell_{i,h} = z \\
  (1) &\text{ass}(X, 1) \lor \text{ass}(X, 0) \leftarrow \text{var}(X). \\
  (2) &\text{ass}(Y, 0) \leftarrow \text{sat}, \text{exists}(Y). \\
  (3) &\text{ass}(Y, 1) \leftarrow \text{sat}, \text{exists}(Y). \\
  (4) &\text{sat} \leftarrow (\text{occ}_i(C, X_i, A_i), \text{ass}(X_i, 1 - A_i))_{i=1}^3. \\
  (5) &\leftarrow \text{not sat}.
  \end{align*}
\]

- Informally, (1) guesses an assignment for all vars
- by (5), we must derive \( \text{sat} \)
- (2), (3), (4): if some assignment to the \( y_j \) violates a clause, do saturate

Note: some answer set exists iff \( \Phi \) evaluates to false

- Maratea et al. [2008] used a propositional version of the classic encoding; 
  \( \text{dlv} \) was competitive with state-of-the-art QBF solvers then
QSAT Encodings /2

QBF \( \Phi = \forall x_1, \ldots, x_m \exists y_1, \ldots, y_n (c_1 \land \cdots \land c_k) \), where \( c_i = \ell_{i,1} \lor \ell_{i,2} \lor \ell_{i,3} \)

- **Bounded arity encoding**

  
  \begin{align*}
  (1) \quad & t(x_i) \lor f(x_i) \leftarrow . \quad \text{for all } x_i \\
  (2) \quad & c_i(\bar{t}) \leftarrow t(x_j). \quad \text{for all } \bar{t} \in \bar{Y}(c_i), \ x_j \in \{\ell_{i,1}, \ell_{i,2}, \ell_{i,3}\} \\
  (3) \quad & c_i(\bar{t}) \leftarrow f(x_j). \quad \text{for all } \bar{t} \in \bar{Y}(c_i), \ -x_j \in \{\ell_{i,1}, \ell_{i,2}, \ell_{i,3}\} \\
  (4) \quad & c_i(\bar{t}). \quad \bar{t} \in \bar{Y}(c_i) \setminus \{\bar{c}_i\} \\
  (5) \quad & \leftarrow c_1(\eta(c_1)), \ldots, c_k(\eta(c_k)).
  \end{align*}

- \( \bar{Y}(c_i) = \{0, 1\}^{h_i} \) are all assignments of 0 and 1 to the literals \( \ell_{i,j_1}, \ldots, \ell_{i,j_{h_i}} \) over \( y_1, \ldots, y_n \) in \( c_i (h_i \leq 3) \)

- \( \bar{c}_i \) is the single such assignment that makes them false

- \( \eta(c_i) \) results from \( \ell_{i,j_1}, \ldots, \ell_{i,j_{h_i}} \) by replacing each \( y_j \) and \( \neg y_j \) with \( Y_j \)

  Example: for \( c = \neg y_1 \lor x_2 \lor y_3 \), we have \( \ell_{i,j_1}, \ldots, \ell_{i,j_{h_i}} = \neg y_1, y_3 \) and thus \( \bar{Y}(c) = \{0, 1\}^2, \bar{c} = (1, 0), \) and \( \eta(c) = Y_1, Y_3 \)

- (1) can be replaced with “shifting” \( t(x_i) \leftarrow \text{not } f(x_i). \) and \( f(x_i) \leftarrow \text{not } t(x_i). \)

- **Lifted to QBFs** \( \Phi = \exists x_1, \ldots, x_l \forall x_{l+1}, \ldots, x_m \exists y_1, \ldots, y_n (c_1 \land \cdots \land c_k) \)
Evaluation Results

Publicly available 2-QBF (∀∃) competition instances
For (1), (2), preprocessing by DepQBFSolver, (for (2), decomposition), then clingo
For (3), DepQBFSolver
200 instances, 2 secs: (2) solved 111, (1) 88, and (3) 107
Specific benchmark “stmt”: (3) > (2) > (1)
3-QBF: 151 inst. from the Eval-2012 data set of latest QBF competition
- (3) solved 47; (2) solved only 18, but 10 not solved by (3)
Mario Alviano and Wolfgang Faber.
Dynamic magic sets and super-coherent answer set programs.

Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone.
Magic sets for disjunctive datalog programs.

Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca.
WASP: A native ASP solver based on constraint learning.

Vernon Asuncion, Fangzhen Lin, Yan Zhang, and Yi Zhou.
Ordered completion for first-order logic programs on finite structures.

Michael Bartholomew and Joohyung Lee.
System aspmt2smt: Computing ASPMT theories by SMT solvers.
C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian.  

Manuel Bichler, Michael Morak, and Stefan Woltran.  
The power of non-ground rules in answer set programming.  

Piero A. Bonatti, Enrico Pontelli, and Tran Cao Son.  
Credulous resolution for answer set programming.  

Gerhard Brewka, Ilkka Niemelä, and Miroslaw Truszczynski.  
Answer set optimization.  
References III

Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub.  
asprin: Customizing answer set preferences without a headache.  

Rémi Brochenin, Yuliya Lierler, and Marco Maratea.  
Abstract disjunctive answer set solvers.  

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone.  
Computable functions in asp: Theory and implementation.  

Yin Chen, Fangzhen Lin, Yisong Wang, and Mingyi Zhang.  
First-order loop formulas for normal logic programs.  
References IV

Keith L. Clark.
Negation as failure.

Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl.
OMiGA: An open minded grounding on-the-fly answer-set solver.

Broes de Cat, Marc Denecker, and Peter J. Stuckey.
Lazy model expansion by incremental grounding.
In Dovier and Costa [2012], pages 201–211.

James P. Delgrande, Torsten Schaub, and Hans Tompits.
A compiler for ordered logic programs.

James P. Delgrande, Torsten Schaub, and Hans Tompits.
A framework for compiling preferences in logic programs.
References V

A classification and survey of preference handling approaches in nonmonotonic reasoning.

Agostino Dovier and Vítor Santos Costa, editors.

Thomas Eiter, Georg Gottlob, and Nicola Leone.
Abduction From Logic Programs: Semantics and Complexity.

Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer.
Computing preferred answer sets by meta-interpretation in answer set programming.

T. Eiter, W. Faber, M. Fink, and S. Woltran.
Complexity Results for Answer Set Programming with Bounded Predicate Arities.

Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran.
Complexity results for answer set programming with bounded predicate arities and implications.
References VI


Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller.
A portfolio solver for answer set programming: Preliminary report.

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub.
Conflict-driven answer set solving: From theory to practice.

Martin Gebser, Tomi Janhunen, and Jussi Rintanen.
Answer set programming as SAT modulo acyclicity.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea.
Answer set programming based on propositional satisfiability.


Nicola Leone, Pasquale Rullo, and Francesco Scarcello.

Yuliya Lierler and Vladimir Lifschitz.
One more decidable class of finitely ground programs.

Yuliya Lierler.
cmodels - SAT-Based Disjunctive Answer Set Solver.

Vladimir Lifschitz and Alexander A. Razborov.
Why are there so many loop formulas?

V. Lifschitz and H. Turner.
Splitting a Logic Program.
Fangzhen Lin and Yuting Zhao.
ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers.

Fangzhen Lin and Yuting Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.

Guohua Liu, Tomi Janhunen, and Ilkka Niemelä.
Answer set programming via mixed integer programming.

Florian Lonsing and Uwe Egly.
Depqbf: An incremental QBF solver based on clause groups.

Marco Manna, Francesco Ricca, and Giorgio Terracina.
Taming primary key violations to query large inconsistent data via ASP.
Marco Maratea, Francesco Ricca, Wolfgang Faber, and Nicola Leone.  
Look-back techniques and heuristics in dlv: Implementation, evaluation, and comparison to qbf solvers.  

Marco Maratea, Luca Pulina, and Francesco Ricca.  
A multi-engine approach to answer-set programming.  

Michael Morak and Stefan Woltran.  
Preprocessing of complex non-ground rules in answer set programming.  
In Dovier and Costa [2012], pages 247–258.

Ilkka Niemelä.  
Integrating answer set programming and satisfiability modulo theories.  

Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi.  
Gasp: Answer set programming with lazy grounding.  


Tran Cao Son and Enrico Pontelli.
Planning with preferences using logic programming.

Tommi Syrjänen.
Omega-restricted logic programs.