

Answer Set Programming ^{*a*}

Victor W. Marek
University of Kentucky

Winter 2008

^{*a*}Research supported by the National Science Foundation under Grant IIS-0325063 and by the Kentucky Science and Engineering Foundation under Grant 1036-RDE-008.

What is Answer Set Programming?

- ▶ A different way of using logic to solve problems (in this case NP-search problems)
- ▶ Based on searching the space of models and **not** the least model
- ▶ Like in satisfiability, solutions are models (and not variable assignments)
- ▶ Unlike PROLOG programming language, it possesses a meaningful semantics
- ▶ But the price paid for this is a significantly weaker expressibility

Outline

- ▶ **History**
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, Complexity results
- ▶ Computation of stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

History

- ▶ Logic Programming (LP) has 35+ year history, grounded in the work of Herbrand, Robinson, Kowalski, Smullyan, Colmerauer and others
- ▶ It started as a way to search of a least Herbrand model of a Horn theory
- ▶ Here a Horn theory is a *program*, and a solution to a query to that program is an assignment to variables in the query so that a corresponding atomic sentence is true in the least Herbrand model
- ▶ The programming language PROLOG implements (open) Horn fragment of predicate logic
- ▶ But from the very beginning LP had serious problems

Supported models

- ▶ Extralogical constructs (such as *cut*) created problems
- ▶ The implementations of Logic Programming did not follow unification algorithm
- ▶ The implementation of Logic Programming allowed for *negation*, and semantics of negation in Logic Programming was not clear at all
- ▶ The first serious attempt to handle negation in Logic Programming was due to Clark, who introduced *completion* of a program
- ▶ Completion of a program is a logical theory

History, cont'd

- ▶ Models of completion: presence of atom , say p is supported by satisfaction of a body of a clause with the head p
- ▶ But circular arguments are possible here
- ▶ Example: when in the program P we have only two rules with p in the head

$$p \leftarrow q$$

$$p \leftarrow s, \neg t$$

- ▶ Then we put in the completion

$$p \equiv q \vee (s \wedge \neg t)$$

- ▶ but s may depend on p

History, cont'd

- ▶ Theory formed out of completion formulas is completion of the program and models of completion are called *supported models*
- ▶ The \subseteq -least model of the Horn program is supported
- ▶ Presence of an atom p in a supported model is determined by satisfaction the body of some clause with head p
- ▶ But there is more to the least model of the Horn program; it is, of course, minimal
- ▶ Supported models of programs do not need to be minimal
- ▶ This was considered a significant limitation of Clark's semantics

Outline, again

- ▶ History
- ▶ **Terminology**
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, Complexity results
- ▶ Computation of stable models, Constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

A bit of terminology

- ▶ A clause is a string of the form

$$p \leftarrow q_1, \dots, q_m, \neg r_1, \dots, \neg r_n$$

- ▶ Here, $p, q_1, \dots, q_m, r_1, \dots, r_n$ are atoms
- ▶ p is the *head* of our clause
- ▶ $q_1 \wedge \dots \wedge q_m \wedge \neg r_1 \wedge \dots \wedge \neg r_n$ is the body of our clause
- ▶ Body splits into *PosBody* and *NegBody*
- ▶ We will limit to Herbrand semantics (i.e. models over Herbrand algebra of the language). Thus we can assume that our clauses have no variables
- ▶ A *program* is a collection of clauses
- ▶ A clause is *Horn* if $n = 0$, and Horn program consists of Horn clauses

Semantics

- ▶ Models: collections of atomic sentences
- ▶ Model M satisfies p is $p \in M$
- ▶ $\text{body}(C)$, in the body interpreted as a conjunction
- ▶ M satisfies a clause $C = p \leftarrow q_1, \dots, q_m, \neg r_1, \dots, \neg r_n$ if either M does not satisfy the body of C or M satisfies p
- ▶ M is a *model* of a program P is M satisfies all clauses of P
- ▶ Thus \leftarrow is interpreted as *reverse implication*

Operator T_P

- ▶ The operator T_P (Kowalski-van Emden, Apt-van Emden) – usual meaning

$$T_P(M) = \{head(C) : C \in P \text{ and } M \models body(C)\}$$

- ▶ Fixpoints of T_P are precisely supported models of P
- ▶ In particular the least model of a Horn program is supported
- ▶ When P is Horn, T_P is monotone and (upper-half) continuous and so the least fixpoint exists and is reached in at most ω steps
- ▶ In general T_P does not have to be monotone, and the existence of fixpoints is not guaranteed

More on T_P

- ▶ But even for Horn programs the operator T_P is not lower-half continuous, i.e. in general

$$T_P\left(\bigcap_{i \in N} M_i\right) \neq \bigcap_{i \in N} T_P(M_i)$$

whenever $\langle M_i \rangle_{i \in N}$ is monotone decreasing (i.e. for all i, j
 $i < j \Rightarrow M_j \subseteq M_i$)

- ▶ An example shows this
- ▶ Variables: $At := \{p\} \cup \{p_i : i \in N\}$
- ▶ Clauses $C_i := p \leftarrow p_i$
- ▶ $P = \{C_i : i \in N\}$
- ▶ $M_i = At \setminus \{p_1, \dots, p_i\}$

Stable models

- ▶ Introduced by Gelfond and Lifschitz in 1988
- ▶ Gelfond-Lifschitz reduct: Given M , delete clauses C in P such that $M \not\models \text{negBody}(C)$
- ▶ In the remaining clauses eliminate negBody
- ▶ Resulting program P_M is Horn, and has least model N_M
- ▶ If $M = N_M$, M is called *stable* model of P
- ▶ (Gelfond-Lifschitz) Such models, when exist, are indeed models, and they are fixpoints of T_P (thus they are supported). Moreover, stable models are minimal
- ▶ Stable models of logic programs turned out closely related to so-called *Default Logic* of R. Reiter, while supported models are closely related to *Autoepistemic Logic* of R. Moore

The Gelfond-Lifschitz operator

- ▶ Let us fix program P
- ▶ Then we have an assignment $M \mapsto P_M$, that is we have Gelfond-Lifschitz reduct of P by M
- ▶ Let N_M be *least model* of P_M
- ▶ Gelfond-Lifschitz operator, GL_P associated with P assigns to M the set N_M
- ▶ M is a stable model of P if and only if M is a fixpoint of GL_P
- ▶ The operator GL_P is antimonotone

Useful properties of antimonotone operators

- ▶ In general antimonotone operators in (complete) lattices do not have to possess fixpoints
- ▶ But if O is an antimonotone operator, then the operator O^2 is monotone, thus it possesses a least and largest fixpoints
- ▶ When O is natimonotone then the least fixpoint of O^2 , f , and the largest fixpoint of O^2 , F , provide bounds for fixpoints of O . Specifically:

$$f \leq M \leq F$$

for every fixpoint M of O

Looking at properties of stable models

- ▶ Stable models form an antichain
- ▶ (Truszczyński and VM) The existence of stable model for (finite, propositional) logic programs is an NP-complete problem
- ▶ (Reduction from kernel)
- ▶ But are there “easier cases”?
- ▶ Of course Horn programs have a unique stable model (the least model)
- ▶ It turned out that so-called *stratified programs* of Apt, Blair and Walker also have unique stable model
- ▶ (It is easy to test program for existence of stratification)

What about the predicate case?

- ▶ The classic result due to Kleene and Smullyan: all recursively enumerable sets are definable by means of Horn programs
- ▶ (Apt and Blair) All arithmetic sets are definable by means of stratified programs
- ▶ Nerode, Remmel and VM subjected programs to systematic studies from the point of recursion theory

Outline, again

- ▶ History
- ▶ Terminology
- ▶ **Proof schemes, characterizing stable models**
- ▶ FSP programs, complexity results
- ▶ Computing stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

Proof schemes

- ▶ It is quite easy to define a proof-system that derives from a Horn program all elements of the least model (and only them)
- ▶ Then one can define M -proofs out of programs, where M controls the applicability of rules
- ▶ The idea now is to abstract from a set M , and carry the necessary information to make it work within the proof itself
- ▶ Instead of M we will have the concept of *support*. This is important, because regardless of the size of At , the support of the proof scheme is *finite*

Proof schemes, cont'd

- ▶ We formalize this notion in a Hilbert-style proof (that we call *proof scheme* (but we could use a form of resolution instead))
- ▶ A proof scheme is a sequence of the following form

$$S := \langle \langle C_1, p_1 \rangle, \dots, \langle C_k, p_k \rangle, U \rangle$$

- ▶ Where C_j , $1 \leq j \leq k$ are clauses of P , p_j , $1 \leq j \leq k$, is the head of C_j and U is a finite set of variables

Proof schemes, cont'd

- ▶ p_k will be called the *conclusion* of S
- ▶ U is *support* of S , denoted $\text{supp}(S)$
- ▶ Proof schemes defined inductively
- ▶ $\langle \langle C, \text{head}(C) \rangle, \{p : \neg p \in \text{negBody}(C)\} \rangle$ is a proof scheme for p , providing $\text{posBody}(C) = \emptyset$
- ▶ if $\langle \langle C_1, p_1 \rangle, \dots, \langle C_k, p_k \rangle, U \rangle$ is a proof scheme, and C is a clause of P , and

$$\text{posBody}(C) \subseteq \{p_1, \dots, p_k\},$$

then

$$\langle \langle C_1, p_1 \rangle, \dots, \langle C_k, p_k \rangle, \langle C, \text{head}(C) \rangle, U \cup \{p : \neg p \in \text{negBody}(C)\} \rangle$$

is also a proof scheme

An example

- ▶ Let P be

$$C_1 := p \leftarrow$$

$$C_2 := q \leftarrow p, \neg r$$

$$C_3 := r \leftarrow \neg q$$

$$C_4 := s \leftarrow \neg t$$

- ▶ Then $\langle\langle C_1, p \rangle, \langle C_2, q \rangle, \{r\}\rangle$ is a proof scheme, with conclusion q
- ▶ But also $\langle\langle C_1, p \rangle, \langle C_2, q \rangle, \langle C_3, r \rangle, \{q, r\}\rangle$ is a proof scheme. This one has conclusion r and support $\{q, r\}$
- ▶ Yet another example is $\langle\langle C_1, p \rangle, \langle C_4, s \rangle, \{t\}\rangle$
- ▶ There are funny things about the second and third example. The second one involves a contradiction; the third one involves proving extraneous variables

Proof Schemes, cont'd

- ▶ A set of variables M admits a proof scheme S if $M \cap \text{supp}(S) = \emptyset$
- ▶ Here is the basic characterization of stable models via proof schemes. A set of variables M is a stable model of P if
 - For every variable $p \in M$ there is a proof scheme with conclusion p that is admitted by M
 - For every variable $p \notin M$ there is no proof scheme with conclusion p admitted by M

Operator GL and proof schemes

- ▶ The operator GL_P maps subsets of At to subsets of At
- ▶ P_M is Gelfond-Lifschitz reduct of P w.r.t. M
- ▶ $N_M = GL_P(M)$ is the least model of the reduced program P_M
- ▶ N_M consists precisely of conclusions of proof schemes (w.r.t. P) admitted by M
- ▶ Among supports of proof schemes only those that are inclusion-minimal count
- ▶ The reason is that once M admits a proof scheme S with a support U then any proof scheme S' with *smaller* support U' is also admitted
- ▶ Indeed, if $M \cap U = \emptyset$ and $U' \subseteq U$ then $M \cap U' = \emptyset$

Defining equations

- ▶ When U is finite set of variables, $\neg U$ is the conjunction of negations of variables in U
- ▶ A *defining equation* for a variable p (w.r.t. P) is a formula

$$p \leftrightarrow \neg U_1 \vee \neg U_2 \dots$$

where $\langle U_1, U_2, \dots \rangle$ is the list of minimal supports of proof schemes for p

- ▶ In principle the defining equation for a variable may be infinitary
- ▶ Defining equations are similar to completion
- ▶ When there is no proof scheme with conclusion p then the defining equation is $p \leftrightarrow \perp$
- ▶ When p has a proof scheme with empty support then its defining equation is equivalent to p

If you use proof schemes...

- ▶ Then few things like:
 - Fages theorem on programs w/o recursion via positive part of the body (stable and supported models coincide)
 - Erdem and Lifschitz generalization (but only for SLP, not DLP)
 - Dung theorem on equivalence of programs and purely negative programs
- ▶ become very easy

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ **FSP programs, complexity results**
- ▶ Computing stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

FSP programs

- ▶ FSP programs (finite support programs) are programs for which the defining equations are *finite* for every variable p in At
- ▶ Of course finite programs are FSP
- ▶ Let us look at an example
 - Let us consider four predicate clauses (generating infinitely many clauses)

$$p(0) \leftarrow q(X)$$

$$q(X) \leftarrow \neg r(0)$$

$$nat(0)$$

$$nat(s(X)) \leftarrow nat(X)$$

- Then $p(0)$ is a conclusion of infinitely many proof schemes. But they have same support, $\{r(0)\}$ so the defining equation is finite

More on FSP programs

- ▶ Thus in FSP programs for every variable p there is a finite family \mathcal{X}_p of finite sets consisting of inclusion-minimal supports of proof schemes with the conclusion p
- ▶ (inclusion-minimality means that for U_1, U_2 in \mathcal{X}_p if $U_1 \neq U_2$ then $U_1 \not\subseteq U_2$ and $U_2 \not\subseteq U_1$)
- ▶ Thus the family \mathcal{X}_p is an antichain

Continuity properties of GL operator

- ▶ The operator GL is antimonotone, that is:

$$M_1 \subseteq M_2 \Rightarrow GL_P(M_2) \subseteq GL_P(M_1)$$

- ▶ So when investigating continuity of GL_P we need to say clearly what is meant by continuity
- ▶ The first property we show does not involve FSP
- ▶ The operator GL_P is lower-half continuous that is for a *decreasing* family of sets of variables $\langle M_i \rangle_{i \in N}$ we have:

$$GL_P\left(\bigcap_{i \in N} M_i\right) = \bigcup_{i \in N} GL_P(M_i)$$

Monotone increasing sequences

- ▶ The following are equivalent:
 - (a) P is an FSP program
 - (b) The operator GL_P is upper-half continuous, that is:

$$GL_P\left(\bigcup_{i \in N} M_i\right) = \bigcap_{i \in N} GL_P(M_i)$$

for every monotonic increasing sequence of sets of variables
 $\langle M_i \rangle_{i \in N}$

Trees and stable models

- ▶ Since the proof schemes are finite sequences we can effectively build trees out of them (one needs to be a bit careful, but not much)
- ▶ Conversely, we can code trees (say over positive integers) via programs so that there is a one-to-one, recursive correspondence between stable models of the coding programs and infinite paths through the trees
- ▶ Recursion-theorists (Jockusch and Soare, Clote, Jockusch, Lewis and Remmel) studied paths through the trees and their degrees
- ▶ Not surprisingly these classifications are related to various classes of programs, allowing for classification of programs
- ▶ With some effort finite programs (but with function symbols) correspond to various classes of trees (Nerode, Remmel, VM)

Trees and stable models, cont'd

- ▶ Thus the stable semantics for logic programs (finite programs with function symbols, or infinite propositional programs) is *overexpressive*
- ▶ We can, for instance, have a unique stable model of a finite program, but it will be very high in Davis-Mostowski hierarchy
- ▶ So, if we want to use the stable semantics as a computational tool, something needs to be abandoned, for instance function symbols
- ▶ if so, then what, and what can be added instead?

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, complexity results
- ▶ **Computing stable models, constraints in ASP**
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

What about computation of stable models?

- ▶ Given an input program P
- ▶ Doing it naively:
 - guess M ,
 - compute reduct
 - compute N_M (using, for instance Dowling-Gallier linear time algorithm for computing the least model of a Horn program)
 - Check if $M = N_M$
- ▶ If the last test fails, check next M

Computing stable models, cont'd

- ▶ It turns out that the three-valued interpretation

$$\langle lfp(GL_P^2), gfp(GL_P^2) \rangle$$

the property that reducing P by that interpretation leads to a residual program which has models that are in 1-1 correspondence with the models of P

- ▶ This three-valued interpretation called *well-founded model* (van Gelder, Ross and Schlipf) has other interesting properties, for instance, if it is 2-valued then it is a unique stable model of P

Computing stable models, cont'd

- ▶ Clearly, computation of stable models is NP-complete search problem
- ▶ In fact, it is uniform (Remmel, VM): for every search problem \mathcal{S} in the class NP there is a single logic program $P_{\mathcal{S}}$ and an encoding f of instances $f : \mathcal{I}_{\mathcal{S}} \rightarrow At$ so that there is a one-to-one correspondence between the solution to \mathcal{S} for instance I and the stable models of $f(I) \cup P_{\mathcal{S}}$.
- ▶ Thus ASP solvers can be used for solving NP-search problems

Practical computation with stable models

- ▶ In 1991, Nerode, Remmel and VM observed encodings of specific search problems as logic programs
- ▶ In 1995 Niemelä, and separately Truszczyński and VM observed that stable semantics of logic programs can be used as a *constraint solving paradigm*
- ▶ The term *Answer Set Programming* has been coined by Lifschitz, who showed how a variety of planning problems can be solved using stable semantics of logic programs
- ▶ Officially, in 2001 the first AAAI workshop on Answer Set Programming was organized in Palo Alto, CA

Constraints in ASP

- ▶ When Niemelä and his collaborators implemented ASP system *smodels* they implemented an extension of stable semantics, allowing for constructs such as *cardinality constraints* and more generally, *weight constraint*
- ▶ A cardinality atom kXl where $k \leq l$, $k, l \in N$, $X \subseteq At$ is a constraint on the satisfying valuation: *no less than k but no more than l of atoms in X are satisfied*
- ▶ Clearly, $p \equiv 1\{p\}1$, $\neg p \equiv 0\{p\}0$
- ▶ (Cardinality constraints, and more generally weight constraints come from 0-1 integer programming)

Constraints cont'd

- ▶ Niemelä et.al. proposed stable semantics for programs where such construct appear both in *heads and/or bodies* of program clauses
- ▶ But there are many other constraints, for instance SQL constraints (on SUM, AVG, MAX etc), parity constraints, “same size” constraints, etc.
- ▶ We can talk about *set constraints*, which are families of subsets of *At* with satisfaction generalized directly from cardinality constraints
- ▶ Remmel and VM showed a semantics for programs admitting set constraints (in heads and/or bodies)
- ▶ But it is not the only proposal, and this is still an area of research in ASP

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, complexity results
- ▶ Computing stable models, constraints in ASP
- ▶ **ASP and SAT**
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ Conclusions

An example: Hamiltonian cycle problem

- ▶ We want to compute Hamiltonian cycle in a graph, we use ASP
- ▶ Data about the graph - “extensional database”, edb_G
 - $\{vtx(a) : a \in V\}$
 - $\{edge(a, b) : (a, b) \in E\}$
 - $initvtx(a_0)$
- ▶ Intentional predicates:
 - $in(X, Y)$
 - $out(X, Y)$
 - $reached(X)$

Example, cont'd

► Program P :

- $1\{in(X, Y), out(X, Y)\}1 \leftarrow: edge(X, Y)$
- $\leftarrow in(X, Y), in(X, Z), Y \neq Z$
- $\leftarrow in(X, Y), in(T, Y), X \neq T$
- $reached(Y) \leftarrow in(X, Y), reached(Y)$
- $reached(Y) \leftarrow in(X, Y), initvtx(Y)$
- $\leftarrow \neg reached(X), vtx(X)$

► Hamiltonian cycles in G can be read off stable models of $edb_G \cup P$

ASP and SAT

- ▶ We have noticed that ASP (in its stable logic programming version) and SAT capture the same class of search problems, namely NP-search problems
- ▶ In fact it is very easy to (uniformly) translate SAT into ASP
- ▶ But, of course, we would like to reduce ASP to SAT in the sense that stable models could be computed on SAT solvers
- ▶ There is a reason: since the breakthroughs of mid-1990ies (with the partial closure under resolution a.k.a. *clause learning*, better data structures *watch literals*) there were immense speed-ups in SAT solving
- ▶ The issue is, then: can we compute stable models using SAT solver as a *back engine*

ASP and SAT, cont'd

- ▶ In 2002, Lin and Zhao found the technique that allows us to do so
- ▶ The idea was to analyze why the supported models are not necessarily stable models (supported models can be computed on SAT solvers directly)
- ▶ The reason for that is that there may be cycles in the *call graph* of the program and that those cycles must be “broken” to get the stable models
- ▶ This can be achieved in a variety of ways, the only problem is that (Lifschitz and Razborov) there can be exponential number of cycles to be broken
- ▶ A number of other techniques has been proposed for reduction, some with low space complexity

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, complexity results
- ▶ Computing stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ **Programming in ASP**
- ▶ Software for ASP
- ▶ Conclusions

Getting serious about *programming*

- ▶ Once we decide to use ASP as a *computational paradigm* a number of steps are necessary
 - Object variables must get in (because nobody with a sound mind would allow for typing same condition over and over again)
 - Function symbols (but the functions must be tabulated) are nice
 - Some forms of *software engineering* are needed
- ▶ One needs to handle connections to other programming languages and to databases
- ▶ One needs to be able to process strings
- ▶ In complex problems one needs *modules*, subprograms that solve specific tasks

Getting serious about *programming*, cont'd

- ▶ The solving engine processes *propositional programs*. Therefore one needs programs that convert programs with variables to programs without variables via process known as *grounding*
- ▶ Starting with Niemelä's *lparse* a number of grounders are available including *psground* of East and Truszczyński
- ▶ Niemelä's *smodels* allow for computation of values used by variables in other programming languages
- ▶ Both Vienna/Calabria *d1v* and Kentucky *aspps* allow for database connections

Getting serious about *programming*, cont'd

- ▶ Modules are handled via *strong equivalence*
- ▶ There is a natural equivalence relation between programs:

$P \simeq P'$ if P, P' have the same families of stable models

- ▶ But because of the nonmonotonic character of programs (a bigger program does not have less models) a stronger property is needed:

$$P \cong P' \text{ if for all } P'', P \cup P'' \simeq P' \cup P''$$

- ▶ (this is called *strong equivalence*)
- ▶ Amazingly, this property \cong is related to so-called *Smetanich logic* (also known as *here-and-there* logic, HT)

Getting serious about *programming*, cont'd

- ▶ H-T logic is an extension of intuitionistic logic by the scheme

$$F \vee (F \Rightarrow G) \vee \neg G$$

- ▶ It is semantically characterized by frames with two worlds
- ▶ Let us think about \leftarrow as (reverse) implication – *Body* implies *head*
- ▶ Then (Lifschitz-Pearce-Valverde): Two programs are strongly equivalent if and only if they are equivalent in HT-logic

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, complexity results
- ▶ Computing stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ **Software for ASP**
- ▶ Conclusions

Existing ASP systems

- ▶ As we have seen, ASP is a logic-based system used to solve search problems
- ▶ In addition to the conceptually simplest framework (stable semantics for logic programs), more complex formalisms (default logic, disjunctive logic programming) have been investigated and tried as computational mechanisms
- ▶ The oldest is the *DeReS* system based on Default Logic
- ▶ A very successful system (based on *disjunctive logic programming*) is *d1v* (Eiter, Leone) out of Vienna UT and Calabria

Existing ASP systems, cont'd

- ▶ The best known ASP system is *smodels* (Niemelä, Simons, Syrjanen) out of Helsinki UT. Likely best supported and instrumented
- ▶ There are other systems, for instance *ASSAT* out of Hong Kong, *NoMoRe* out of Potsdam, *cmodes* out of Austin, TX
- ▶ Periodic competitions for ASP solvers

Existing ASP systems, and their use

- ▶ Homegrown (i.e. from Lexington) is *aspps* a hybrid system (some aspects of SAT, some aspects of ASP)
- ▶ Generally, ASP solvers are *slower* than SAT on similar problems, but (of course) not always
- ▶ Programming with ASP is generally *harder* than SAT, because the programmer needs to take into account the nonmonotonic character of programs
- ▶ Unlike SAT, only rarely can the semantics of ASP programs be explained to *juniors*
- ▶ But if your problem involves *change*, ASP is absolutely easier to use

Outline, again

- ▶ History
- ▶ Terminology
- ▶ Proof schemes, characterizing stable models
- ▶ FSP programs, complexity results
- ▶ Computing stable models, constraints in ASP
- ▶ ASP and SAT
- ▶ Programming in ASP
- ▶ Software for ASP
- ▶ **Conclusions**

Conclusions

- ▶ Caveat: ASP is pretty well researched, with a conference series devoted exclusively to the subject (LPNMR, every other year, next time in 2009) and several conference series (ICLP, IJCAI, AAI, ECAI) accepting quality papers in the area
- ▶ Thus we covered only a fragment of the work in the area
- ▶ Like SAT, ASP is logic-based, with very strong connections to logical formalisms
- ▶ Like SAT, ASP was developed by logicians, but coming, mainly, from AI-logic background

A message to take home

- ▶ Logic changed in the past half century
- ▶ Some aspects of logic became very practical, in particular SAT
- ▶ (Of course there are other successful applications of logic, for instance Automated Theorem Proving)
- ▶ Today (and in fact for a number of years) papers that are clearly logicians' work are published in very practical venues (say CAV and like)
- ▶ ASP is one of areas where logicians work with the idea that their work will become practically useful
- ▶ These slides available at <http://www.cs.uky.edu/~marek>