What Is Answer Set Programming?

Vladimir Lifschitz
Department of Computer Sciences
University of Texas at Austin
1 University Station C0500
Austin, TX 78712
vl@cs.utexas.edu

Abstract

Answer set programming (ASP) is a form of declarative programming oriented towards difficult search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications. ASP programs consist of rules that look like Prolog rules, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the creation of fast satisfiability solvers for propositional logic.

Introduction

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications. ASP is based on the stable model (answer set) semantics of logic programming (Gelfond & Lifschitz 1988), which applies ideas of autoepistemic logic (Moore 1985) and default logic (Reiter 1980) to the analysis of negation as failure.[1]

In ASP, search problems are reduced to computing stable models, and answer set solvers—programs for generating stable models—are used to perform search. The search algorithms used in the design of many answer set solvers are enhancements of the Davis-Putnam-Logemann-Loveland procedure, and they are somewhat similar to the algorithms used in efficient SAT solvers (Gomes *et al*. 2008).[2] Unlike SLDNF resolution employed in Prolog, such algorithms, in principle, always terminate.

The ASP methodology is about ten years old. The planning method proposed by Dimopoulos, Nebel, & Koehler (1997) is an early example of answer set programming. Their approach is based on the relationship between plans and stable models described in (Subrahmanian & Zaniolo 1995). Soininen & Niemelä (1998) applied what is now known as answer set programming to the problem of product configuration. The use of answer set solvers for search was identified as a new programming paradigm in (Marek & Truszczyński 1999) (the term "answer set programming" was used for the first time as the title of a part of the collection where that paper appeared) and in (Niemelä 1999).

[1] In a more general sense, ASP includes all applications of answer sets to knowledge representation (Baral 2003; Gelfond 2008).

[2] The reduction of ASP to SAT invented by Fangzhen Lin and Yuting Zhao (2004) has led to the creation of several "SAT-based" answer set solvers. These systems performed well in a recent competition (Gebser *et al*. 2007). On the other hand, as noted in (Lifschitz & Razborov 2006), ASP appears to be stronger than SAT in the sense of the "comparative linguistics" approach to knowledge representation formalisms described in (Gogic *et al*. 1995).

An answer set programming language

System LPARSE was originally created as a front-end for the answer set solver SMODELS,[3] and it is now used in the same way in most other answer set solvers.[4]

Some rules found in LPARSE programs are traditional "Prolog-style" rules, such as

```
p :- q.
```

or

```
q :- not r.
```

A collection of Prolog-style rules often has a unique stable model, and that model often consists of all queries to which Prolog would answer `yes`. For instance, the program consisting of the two rules above has one stable model, which consists of `p` and `q`.

The input of LPARSE can include also "choice rules," such as

```
{s,t} :- p.
```

Intuitively, this rule means: if `p` is included in the stable model then choose arbitrarily which of the atoms `s`, `t` to include. If we instruct SMODELS to find all stable models of the program P consisting of all three rules shown above, it will produce the following output:

```
Answer: 1
Stable Model: p q
Answer: 2
Stable Model: t p q
Answer: 3
```

[3] `http://www.tcs.hut.fi/Software/smodels/` .

[4] System DLV (`http://www.dbai.tuwien.ac.at/proj/dlv/`) is an exception; the syntax of ASP programs written for DLV is somewhat different.

```
Stable Model: s p q
Answer: 4
Stable Model: s t p q
```

The head of a choice rule can include numerical bounds. For instance, the rule

```
1 {s,t} :- p.
```

says: if `p` is generated then generate at least one of the atoms `s`, `t`. If we substitute this rule for the last rule of P then Answer 1 will disappear from the output of SMODELS.

The definition of a stable model from (Gelfond & Lifschitz 1988) was extended to programs with choice rules and other rules involving numerical bounds by Niemelä, Simons, & Soininen (1999). At the end of this paper we reproduce a simple definition of a stable model due to Paolo Ferraris (2005) that is sufficiently general to cover these and other useful types of rules.

A *constraint*[5] is a rule with the empty head, such as

```
:- s, not t.
```

The effect of adding a constraint to a program is to eliminate some of its stable models. For instance, the constraint above prohibits generating `s` if `t` is not generated. Adding this constraint to P eliminates Answer 3.

A program with variables is grounded (replaced by an equivalent program without variables) by LPARSE before it is passed on to SMODELS.[6] For instance, grounding turns

```
p(a).  p(b).  p(c).
q(X) :- p(X).
2 {r(X) : p(X)}.
```

into

```
p(a).  p(b).  p(c).
q(a) :- p(a).
q(b) :- p(b).
q(c) :- p(c).
2 {r(a),r(b),r(c)}.
```

Programming methodology: Generate, define, test

Recall that a *clique* in a graph is a set of pairwise adjacent vertices. If our goal is to find a clique of cardinality ≥ 10 in a given graph then the following program C can be used:

```
10 {in(X) : vertex(X)}.
:- in(X), in(Y), vertex(X), vertex(Y),
   X!=Y, not edge(X,Y), not edge(Y,X).
```

To use this program, we combine it with a description of the graph, such as

```
vertex(1..99).  % 1,...,99 are vertices
edge(3,7).      % 3 is adjacent to 7
. . .
```

The atoms of the form `in(...)` in a stable model will represent the vertices in a clique of size ≥ 10. If SMODELS reports that the program has no stable models then the graph has no cliques of required size.

The structure of program C illustrates the "generate-and-test" organization that is often found in simple ASP programs. The first line of C is a choice rule that describes a set of "potential solutions"—an easy to describe superset of the set of solutions to the given search problem; in this case, a potential solution is any set consisting of at least 10 vertices. The second rule is a constraint that eliminates all "bad" potential solutions; in this case, all sets that are not cliques. (The search process employed by SMODELS is *not* based of course, on examining potential solutions one by one, just as SAT solvers do not operate by testing all assignments.)

More complex LPARSE programs include, in addition to choice rules ("generate") and constraints ("test"), a third part, which defines auxiliary predicates that are used in the constraints. This "define" part usually consists of traditional Prolog-style rules.

Such a definition is needed for instance, if we want to use ASP to find a Hamiltonian cycle in a given directed graph (a closed path that passes through each vertex of the graph exactly once). The ASP program below should be combined with definitions of the predicates `vertex` and `edge`, as in the previous example. It uses the predicate `in` to express that an edge belongs to the path; we assume that `0` is one of the vertices.

```
{in(X,Y)} :- edge(X,Y).

:- 2 {in(X,Y) : edge(X,Y)}, vertex(X).
:- 2 {in(X,Y) : edge(X,Y)}, vertex(Y).

r(X) :- in(0,X), vertex(X).
r(Y) :- r(X), in(X,Y), edge(X,Y).

:- not r(X), vertex(X).
```

The choice rule in line 1 of this program is its "generate" part. It says that an arbitrary set of edges is counted as a potential solution. (Without it, LPARSE would have decided that the predicate `in` is identically false.) The "test" part consists of three constraints. Line 2 prohibits sets of edges containing "forks"—pairs of different edges that start at the same vertex `X`. Line 3 prohibits "forks" with two different edges ending at the same vertex `Y`. The constraint in the last line of the program requires that every vertex `X` be reachable from `0` by a non-empty path consisting entirely of `in`-edges. This reachability condition is expressed by the auxiliary predicate `r`, recursively defined in lines 4 and 5. These two lines form the "define" component of the program.

Representing incomplete information

From the perspective of knowledge representation, a set of atoms can be thought of as a description of a complete state of knowledge: the atoms that belong to the set are known

[5] Expressions with numerical bounds, such as `1{s,t}` in the rule above, are sometimes called "cardinality constraints" or "weight constraints." This is not related to the use of the term "constraint" in this paper.

[6] Thus the search process employed by SMODELS does not involve unification. In this sense, in answer set solvers we see a more radical departure from traditional Prolog search than in constraint logic programming, which generalizes unification to constraint solving.

to be true, and the atoms that do not belong to the set are known to be false. A possibly incomplete state of knowledge can be described using a consistent but possibly incomplete set of literals; if an atom `p` does not belong to the set and its negation does not belong to the set either then it is not known whether `p` is true.

In the context of logic programming, this idea leads to the need to distinguish between two kinds of negation: negation as failure, used above, and strong (or "classical") negation, which is denoted in the language of LPARSE by `-` (Gelfond & Lifschitz 1991). The following example, illustrating the difference between the two kinds of negation, belongs to John McCarthy. A bus may cross railway tracks under the condition that there is no approaching train. The rule

```
cross :- not train.
```

is not an adequate representation of this idea: it says that it is okay to cross in the absence of information about an approaching train. The weaker rule, that uses strong negation instead of negation as failure, is preferable:

```
cross :- -train.
```

It is okay to cross if we know that no train is approaching.

Combining both forms of negation in the same rule allows us to express the closed world assumption—the assumption that a predicate does not hold whenever there is no evidence that it does (Reiter 1978). For instance, the rule

```
-q(X,Y) :- not q(X,Y), p(X), p(Y).
```

says that the binary relation `q` does not hold for a pair of elements of `p` if there is no evidence that it does. The program obtained by adding this rule to the rules

```
p(a).  p(b).  p(c).  p(d).
q(a,b).  q(c,d).
```

has a unique stable model, which includes the "positive facts" shown in these two lines and 14 "negative facts" about `q`: `-q(a,a), -q(a,c),`....

An ASP program with strong negation can include the closed world assumption rules for some of its predicates and leave the other predicates in the realm of the open world assumption.

ASP solution to the frame problem

The frame problem is the problem of formalizing the commonsense law of inertia: *Everything is presumed to remain in the state in which it is* (Leibniz, "An Introduction to a Secret Encyclopædia", *c*. 1679). Ray Reiter expressed this default in default logic (Reiter 1980, Section 1.1.4).[7] Here is an ASP counterpart of his "frame default":

```
p(T+1) :- p(T), not -p(T+1), time(T).
```

Like the formalization of the closed world assumption above, this rule uses both negation as failure and strong negation. It says that if the truth-valued fluent `p` is true at time `T`, and there is no evidence that it becomes false at time `T+1`, then it remains true.

This solution to the frame problem is widely used in research on the automation of reasoning about actions, including the work on decision support for the Space Shuttle mentioned in the following section.

Applications of ASP

Answer set programming has been applied to several areas of science and technology. Here are three examples.

Automated Product Configuration. The early work in this area mentioned in the introduction has led to the creation of a web-based product configurator (Tiihonen *et al*. 2003). This technology has been commercialized.[8]

Decision Support for the Space Shuttle. An ASP system capable of solving some planning and diagnostic tasks related to the operation of the Space Shuttle (Nogueira *et al*. 2001) has been designed by a group led by Michael Gelfond, one of the creators of ASP, in collaboration with Matthew Barry of United Space Alliance.

Inferring Phylogenetic Trees. An ASP-based method for reconstructing a phylogeny for a set of taxa has been applied to historical analysis of languages and to historical analysis of parasite-host systems (Brooks *et al*. 2007). The group of authors includes a zoologist, two linguists, and two specialists on answer set programming.

By the way, what's the definition of "stable"?

The definition of a stable model below tells us when a model of a propositional formula F (that is, a truth assignment satisfying F) is considered "stable." It provides a semantics for grounded ASP programs in view of two conventions. First, we agree to treat rules and programs without variables as propositional formulas "written in logic programming notation." For instance, we identify the LPARSE program P shown above with the formula

$$(q \to p) \wedge (\neg r \to q) \wedge (p \to ((s \vee \neg s) \wedge (t \vee \neg t))).^{9} \quad (1)$$

The second convention is to identify any set X of atoms with the truth assignment that makes all elements of X true and makes all other atoms false.

According to (Ferraris 2005), the *reduct* F^X of a propositional formula F relative to a set X of atoms is the formula obtained from F by replacing each maximal subformula that is not satisfied by X with $\bot$ (falsity). We say that X is a *stable model* of F if X is minimal among the sets satisfying F^X. The minimality of X is understood here in the sense of set inclusion: no proper subset of X satisfies F^X.

[7] Steve Hanks and Drew McDermott (1987) argued, on the basis of their Yale shooting example, that Reiter's solution to the frame problem is unsatisfactory. Hudson Turner (1997) showed, however, that it works correctly in the presence of appropriate additional postulates.

[8] `http://www.variantum.com/en/ .`

[9] The third conjunctive term of (1) is, of course, a tautology, so that dropping it would not change the set of models of this formula. But such a simplification would change the set of its *stable* models. Many equivalent transformations preserve the stable models of a formula, but there are exceptions. Examples of "forbidden" equivalent transformations include replacing an implication $\neg p \to q$ with its contrapositive $\neg q \to p$, and replacing the formula $\neg\neg p$ (which represents the constraint `:- not p`) with p. See (Lifschitz, Pearce, & Valverde 2001) for detailed analysis.

Clearly, every set that is a stable model of F according to this definition is a model of F. Indeed, if X does not satisfy F then F^X is $\bot$.

For instance, to check that $\{p, q, s\}$ is a stable model of (1) we take the reduct of (1) relative to that set

$$(q \to p) \wedge (\neg\bot \to q) \wedge (p \to ((s \vee \bot) \wedge (\bot \vee \neg\bot)))$$

or, equivalently,

$$(q \to p) \wedge q \wedge (p \to s),$$

and verify that $\{p, q, s\}$ is minimal among its models.

Acknowledgements

Many thanks to Esra Erdem, Selim Erdoğan, Michael Gelfond, Joohyung Lee, Yuliya Lierler, Victor Marek, Yana Todorova and Mirosław Truszczyński for useful comments. This research was partially supported by the National Science Foundation under Grant IIS-0712113.

References

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

Brooks, D. R.; Erdem, E.; Erdoğan, S. T.; Minett, J. W.; and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning* 39:471–511.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in non-monotonic logic programs. In Steel, S., and Alami, R., eds., *Proceedings of European Conference on Planning*, 169–181. Springer-Verlag.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 119–131.

Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczynski, M. 2007. The First Answer Set Programming System Competition. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 3–17. Springer-Verlag.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Proceedings of International Logic Programming Conference and Symposium*, 1070–1080. MIT Press.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9:365–385.

Gelfond, M. 2008. Answer sets. In van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of Knowledge Representation*. Elsevier.

Gogic, G.; Kautz, H.; Papadimitriou, C.; and Selman, B. 1995. The comparative linguistics of knowledge representation. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 862–869.

Gomes, C. P.; Kautz, H.; Sabharwal, A.; and Selman, B. 2008. Satisfiability solvers. In van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of Knowledge Representation*. Elsevier.

Hanks, S., and McDermott, D. 1987. Nonmonotonic logic and temporal projection. *Artificial Intelligence* 33(3):379–412.

Lifschitz, V., and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7:261–268.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2:526–541.

Lin, F., and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence* 157:115–137.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 375–398.

Moore, R. 1985. Semantical considerations on nonmonotonic logic. *Artificial Intelligence* 25(1):75–94.

Niemelä, I.; Simons, P.; and Soininen, T. 1999. Stable model semantics for weight constraint rules. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 317–331.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25:241–273.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, 169–183.

Reiter, R. 1978. On closed world data bases. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. New York: Plenum Press. 119–140.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* 13:81–132.

Soininen, T., and Niemelä, I. 1998. Developing a declarative rule language for applications in product configuration. In Gupta, G., ed., *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, 305–319. Springer-Verlag.

Subrahmanian, V., and Zaniolo, C. 1995. Relating stable models and AI planning domains. In *Proceedings of International Conference on Logic Programming (ICLP)*, 233–247.

Tiihonen, J.; Soininen, T.; Niemelä, I.; and Sulonen, R. 2003. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design*, 1290–1299.

Turner, H. 1997. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming* 31:245–298.