The s(ASP) Predicate Answer Set Programming System

Kyle Marple, Elmer Salazar, Zhuo Chen, Gopal Gupta

Department of Computer Science
The University of Texas at Dallas

Abstract: We present the s(ASP) system that computes stable models of normal logic programs, i.e., logic programs extended with negation, in the presence of predicates with arbitrary terms. Such programs need not have a finite grounding, so traditional SAT solver-based methods do not apply. Our method relies on the use of an extended Herbrand universe, as well as coinduction, constructive negation and a number of other novel techniques. Using our method, a normal logic program with predicates can be executed directly under the stable model semantics without requiring it to be grounded either before or during execution and without requiring that its variables range over a finite domain. Our method is quite general and supports the use of terms as arguments, including lists and complex data structures. A prototype implementation has been realized and non-trivial applications have been developed to demonstrate the feasibility of our method.

1 Introduction

Answer Set Programming (ASP) [1] has emerged as a successful paradigm for developing intelligent reasoning applications. ASP is based on adding *negation as failure* to logic programming under the stable model semantics regime [2]. ASP allows for sophisticated reasoning mechanisms that are employed by humans (common sense reasoning, default reasoning, counterfactual reasoning, abductive reasoning, etc.) to be modeled elegantly. Numerous systems have been built to execute answer set programs that are extremely sophisticated and efficient. CLASP is the best representative of these systems [8]. These systems restrict programs to predicates that only have variables and constants as arguments (general structures are not allowed). Answer sets (or stable models) of such programs are computed by grounding the program rules with the (finite) Herbrand universe, suitably transforming it, and then using a SAT solver to compute models of the transformed program. These models of the transformed program are the stable models of the original Answer Set Program. There are many problems with these model-finding, SAT solver-based approaches:

1. Since SAT solvers can only handle propositional programs, these approaches only work for finitely-groundable programs. That is, programs with structures and lists occurring in arguments of predicates cannot be executed, as grounding of such programs will result in an infinite-sized program (due to the Herbrand universe being infinite). In many instances, lists and structures are essential for representing information.
2. Grounding of the program can lead to an exponential blowup in program size. For programs to be executable in such a system, a programmer has to be aware of how the grounding process works and how the ASP solver works and then they have to write their code in such a way that this blowup is minimized. This places undue burden on the programmer, as the programmer has to have knowledge of the grounding procedure as well as the model-finding process.
3. If the number of constants in the program is large, then a SAT-based approach may be infeasible due to the size of the grounded program that will be created. It is next to impossible to build a large general-purpose knowledge-based system—for example, a medical diagnosis system—using such an approach, as such a knowledge-based system will potentially have tens of thousands of constants. The challenge is exacerbated by the fact that such systems are generally developed incrementally over time.
4. SAT-based ASP solvers do not allow reasoning with real numbers.
5. SAT-based model-finding approaches compute the entire model. That is obviously an overkill. Most of the time users are interested in a specific piece of information. Thus, if we were to develop a general purpose knowledge-based system, then the current ASP systems will compute the entire model, i.e., everything that can be inferred from the knowledge-base will be computed.
6. Often, it is hard to isolate the solution that is embedded in the model that is produced by the SAT solver. For example, if one solves the Tower of Hanoi problem using a SAT-based ASP solver, then the answer set will contain a large set of moves that are in the model. One cannot easily isolate the sequence of moves that represent the solution to the problem.
7. Since ASP systems compute the entire model, even a minor inconsistency in a narrow part of the knowledgebase will result in the system concluding that no answer set exists. A practical, large, real-world knowledgebase is very likely going to contain inconsistencies.
8. Justification for why an atom is present in an answer set is difficult to produce.

While SAT solver-based ASP systems such as CLASP are very powerful, their use is limited to solving specific well-defined problems, e.g., resource optimization problems or planning problems. Their use is suboptimal for implementing large-scale, general-purpose knowledge-based system due to the problems mentioned above.

2 Query-driven Execution of Answer Set Programs

We have been working on designing query-driven answer set programming systems [3] for last several years. A query-driven system computes the partial answer set that contains the query. Thus, it does not compute the entire answer set; it computes only those parts of the answer set that are relevant to answering the query. Having a query-driven system addresses problems 5, 6, 7, and 8 mentioned above [4], however, issues mentioned in points 1, 2, 3 and 4 above still remain as problems. To alleviate problems 1, 2, 3 and 4 above, we have extended our system to allow general-purpose predicates. Thus, our extended system, called s(ASP), admits answer set programs containing predicates that are allowed to have variables, constants and structures as arguments [5, 6].

Our s(ASP) system does not ground the program. It can be thought of as full Prolog extended with negation-as-failure under the stable model semantics regime [6]. Problem 1, 2, 3 and 4 above are eliminated by s(ASP), since programs do not have to be grounded prior to execution. Note also that incorporating the stable model semantics within Prolog means that not everything has to be modeled using ASP, as is the case if one uses traditional ASP systems. For example, generator predicates can be written using standard Prolog predicates such as member and select; they don't have to be always written using choice rules alone.

As we extend ASP to allow direct execution of predicates, a number of subtleties arise. As an example, consider the following simple program:

```
d(1).
p(X) :- not d(X).
```

If we ground this program, we obtain:

```
d(1).
p(1) :- not d(1).
```

Given the query `?- p(X).`, its execution will always fail for the grounded program (the answer set of the grounded program is {`d(1)`}). However,

execution of the same query with the original program with predicates under s(ASP) should succeed and return the solution {`p(X), not d(X) (X ≠ 1)`} (note that since s(ASP) computes the partial answer set containing the query, both positive and negative atoms have to be specified). Note that the answer set computed by s(ASP) is still compatible with the grounded program, if we restrict `X` to range over the domain {1}.

As an example of an actual s(ASP) program, consider the answer set program for finding Hamiltonian cycles in a graph taken from [1]:

```
reachable(V) :- chosen(U, V), reachable(U).
reachable(0) :- chosen(V, 0).

% Every vertex must be reachable.
:- vertex(U), not reachable(U).

% Choose exactly one edge from each vertex.
other(U, V) :-
    vertex(U), vertex(V), vertex(W),
    V \= W, chosen(U, W).
chosen(U, V) :-
    vertex(U), vertex(V),
    edge(U, V), not other(U, V).

% Two edges cannot be incident on the same vertex.
:- chosen(U, W), chosen(V, W), U \= V.

% Sample graph: vertices and connecting edges.
vertex(0).              vertex(3).
vertex(1).              vertex(4).
vertex(2).

edge(0, 1).             edge(1, 2).
edge(2, 3).             edge(3, 4).
edge(4, 0).             edge(4, 1).
edge(4, 2).             edge(4, 3).
```

Notice that the above program has an odd loop over negation. The partial answer set produced for the query `?- reachable(0).` is shown below.

```
{ chosen(0,1), chosen(1,2), chosen(2,3), chosen(3,4), chosen(4,0),
edge(0,1), edge(1,2), edge(2,3), edge(3,4), edge(4,0), edge(4,1),
```

```
edge(4,2), edge(4,3), other(0,0), other(0,2), other(0,3),
other(0,4), other(1,0), other(1,1), other(1,3), other(1,4),
other(2,0), other(2,1), other(2,2), other(2,4), other(3,0),
other(3,1), other(3,2), other(3,3), other(4,1), other(4,2),
other(4,3), other(4,4), reachable(0), reachable(1), reachable(2),
reachable(3), reachable(4), vertex(0), vertex(1), vertex(2),
vertex(3), vertex(4), not chosen(0,0), not chosen(0,2), not
chosen(0,3), not chosen(0,4), not chosen(0,Var644) ( Var644 \= 0,
Var644 \= 1, Var644 \= 2, Var644 \= 3, Var644 \= 4 ), not
chosen(1,0), not chosen(1,1), not chosen(1,3), not chosen(1,4), not
chosen(1,Var710) ( Var710 \= 0, Var710 \= 1, Var710 \= 2, Var710 \=
3, Var710 \= 4 ), not chosen(2,0), not chosen(2,1), not chosen(2,2),
not chosen(2,4), not chosen(2,Var776) ( Var776 \= 0, Var776 \= 1,
Var776\= 2, Var776 \= 3, Var776 \= 4 ), not chosen(3,0), not
chosen(3,1), not chosen(3,2), not chosen(3,3), not chosen(3,Var842)
( Var842 \= 0, Var842 \= 1, Var842 \= 2, Var842 \= 3, Var842 \= 4 ),
not chosen(4,1), not chosen(4,2), not chosen(4,3), not chosen(4,4),
not chosen(4,Var908) ( Var908 \= 0, Var908 \= 1, Var908 \= 2, Var908
\= 3, Var908 \= 4 ), not chosen(Var627,_) ( Var627 \= 0, Var627 \=
1, Var627 \= 2, Var627 \= 3, Var627 \= 4 ), not chosen(Var663,1) (
Var663 \= 0, Var663 \= 1, Var663 \= 2, Var663 \= 3, Var663 \= 4 ),
not chosen(Var734,2) ( Var734 \= 0, Var734 \= 1, Var734 \= 2, Var734
\= 3, Var734 \= 4 ), not chosen(Var805,3) ( Var805 \= 0, Var805 \=
1, Var805 \= 2, Var805 \= 3, Var805 \= 4 ), not chosen(Var876,4) (
Var876 \= 0, Var876 \= 1, Var876 \= 2, Var876 \= 3, Var876 \= 4 ),
not chosen(Var922,0) ( Var922 \= 0, Var922 \= 1, Var922 \= 2, Var922
\= 3, Var922 \= 4 ), not edge(0,0), not edge(0,2), not edge(0,3),
not edge(0,4), not edge(1,0), not edge(1,1), not edge(1,3), not
edge(1,4), not edge(2,0), not edge(2,1), not edge(2,2), not
edge(2,4), not edge(3,0), not edge(3,1), not edge(3,2), not
edge(3,3), not edge(4,4), not other(0,1), not other(1,2), not
other(2,3), not other(3,4), not other(4,0), not vertex(Var31) (
Var31 \= 0, Var31 \= 1, Var31 \= 2, Var31 \= 3, Var31 \= 4 ) } .
```

The s(ASP) system makes several novel contributions:

- It implements a top-down, query-driven method that can execute normal logic programs with arbitrary predicates, thus solving a problem that was hitherto considered unsolvable.
- Our method can be thought of as providing an operational semantics to normal logic programs with predicates (or, Prolog extended with

negation as failure) under the stable model semantics. This can be combined with other advanced features of logic programming such as constraints over reals to develop extremely powerful applications in an elegant manner, such as automated planning under real-time constraints [11].
- The stable model semantics and answer set programming have been shown to support powerful reasoning techniques such as default reasoning, counter-factual reasoning, abductive reasoning, etc. These reasoning capabilities now become available within Prolog.
- Abductive reasoning under the stable model semantics can be elegantly realized, as it is not possible to find the minimum set of abducibles using SAT solver-based approaches.
- Justification (or a proof trace) for a given query can be produced much more readily and naturally.

3 Implementation and Applications

A prototype implementation of the s(ASP) system has been developed [5]. The system supports abductive reasoning. It also provides justification (proof trace) for a given query. The implementation relies on several novel techniques [6] that include:

- Coinduction: The execution algorithm relies on positive and negative coinduction to detect even and odd cycles through negation.
- Dual Rules: Dual rules have been used to implement ASP systems, however, their use has to be cleverly extended for the predicate case to make sure that appropriate variables are bound to appropriate values.
- Constructive Negation: Unification algorithm has to be extended to allow for negatively constrained variables; essentially, for each unbound variable the system has to also explicitly keep track of values that the variable cannot be bound to.
- Extended Herbrand Universe: The Herbrand universe has to be extended to be a superset of the standard countably infinite Herbrand universe.

Unfortunately, more details cannot be given due to lack of space. They will appear in a forthcoming paper [6]. The correctness of the s(ASP) method has been established. We show that the s(ASP) method is sound for all legal programs and argue that while completeness is in fact impossible to achieve, the method is still useful for a vast majority of practical programs [6]. For finite, ground, legal programs, the method is indeed

complete. Note that legal programs are those that obey the following restrictions: (i) operands of arithmetic operations are ground at the time they are executed; (ii) left recursion cannot lead to success; and, (iii) a negatively constrained variable cannot be *disunified with or constrained against* another negatively constrained variable.

The s(ASP) system is publicly available [5], and has been used to develop a number of non-trivial applications based on ASP; it has also been used to organize an AI hackathon [9]. Some of these applications cannot be executed on traditional ASP systems such as CLASP, as these applications make use of lists and structures to represent information. They have been developed by people who are not experts in ASP. These applications include:

- **Degree Audit System:** A system for automatically performing a degree audit of a student's undergraduate transcript at a US University, i.e., automatically determining whether a student can graduate with a degree or not, has been developed using the s(ASP) system [10]. The system represents the graduation requirements laid out in the course catalog as ASP clauses. Use of negation is important for representing these requirements. The system has to make use of lists, and has hundreds of courses that appear as constants in the program (hence its grounding will produce an inordinately large program).
- **Physician Advisory System:** A system for disease management, particularly, for chronic heart failure has been developed using the s(ASP) system [7]. This system automates the 80-page guidelines (that the American College of Cardiology has developed) by representing them in ASP. While the current system can be run under systems such as CLASP due to the number of constants not being too large, the final system that models a doctor's full knowledge will have quite a few constants, and advanced data-structures may be needed.
- **Automating Textbook Knowledge:** A system that represents high-school level knowledge about cells (in the discipline of biology) as answer set programs has been developed using s(ASP). It can answer high-school level questions posed as s(ASP) queries. The goal is to represent the knowledge in the entire introductory biology textbook as an answer set program, and then be able to automatically answer questions that would be asked of a student (the questions have to be translated into ASP queries that are then executed to find the answer).
- **Birthday Gift Advisor:** A recommendation system for birthday gifts has also been developed using the s(ASP) system. This system

codes a human's knowledge about friends, level of friendship, a person's wealth level, generosity level, and hobbies as answer set programs. When queried, the system can recommend a birthday present for a particular friend (e.g., on one's Facebook page). Note that other similar (web-based) recommendation systems can be built too using s(ASP).

4 Conclusion

We believe that the ASP paradigm is a very powerful paradigm that allows for complex human thought processes to be elegantly emulated. Complex reasoning patterns that humans use can be elegantly modeled using ASP [7]. However, as argued above, the current model-finding, SAT solver-based approaches are not able to realize the full power of ASP. We argue that query-driven implementations of predicate ASP are crucial to the paradigm's success. An additional advantage of a query-driven approach over model-finding approaches is that in the latter case, everything has to be modeled in the ASP paradigm, while in the former case both the standard logic programming paradigm and the ASP paradigm can be made to work together. Abductive reasoning under the stable model semantics is also better realized in a query-driven approach such as s(ASP).

Significant progress has been made with the realization of the s(ASP) system, however, a considerable amount of research remains to be done. The s(ASP) system suffers from some of the same disadvantages that Prolog systems suffer (e.g., certain left-recursive programs may not terminate or the search may have to be guided by the programmer). For this reason it is important to extend the s(ASP) system with (i) constraint logic programming over finite domains, (ii) constraint logic programming over reals, and (iii) tabled logic programming. Note that due to top-down, query-driven nature of execution, both constraints and tabling can be naturally integrated into s(ASP). Finally, the problem of determining the query-relevant part of a given knowledgebase has been solved for top-down propositional ASP in the Galliwasp system, it remains a subject of research in the predicate case.

Acknowledgment: Support from NSF Grant IIS 1423419 is gratefully acknowledged.

References

1. C. Baral. Knowledge Representation - Reasoning & Declarative Problem Solving. Cambridge Univ. Press. 2003.

2. M. Gelfond, V. Lifschitz. The stable model semantics for logic programming. Proc. Joint International Conference and Symposium on Logic Programming, 1070–1080. 1988.
3. K. Marple, A. Bansal, R. Min, G. Gupta. Goal-directed execution of answer set programs. Proc. PPDP 2012: 35-44
4. K. Marple, G. Gupta. Dynamic Consistency Checking in Goal-Directed Answer Set Programming. TPLP 14(4-5): 415-427 (2014)
5. K. Marple, E. Salazar, G. Gupta. The s(ASP) system. `https://sourceforge.net/projects/sasp-system/`
6. K. Marple, E. Salazar, G. Gupta. Computing Stable Models of Normal Logic Programs without Grounding. Forthcoming paper. March 2017.
7. Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil. A Physician Advisory System for Chronic Heart Failure management based on knowledge patterns. TPLP 16(5-6):604-618, 2016.
8. M. Gebser, B. Kaufmann, A. Neumann, T. Schaub. clasp: A Conflict-Driven Answer Set Solver, Proc. LPNMR07, 2007. `https://potassco.org/`.
9. The UT Dallas AI Society. s(ASP) Hackathon. `https://hackai16.devpost.com/submissions`
10. A. Sobhi, S. Srirangapalli, K. Marple, E. Salazar, G. Gupta. The UT Dallas Degree Audit System. 2016. `http://gradaudit.xyz/`
11. Ajay Bansal, Neda Saeedloei, Gopal Gupta. Timed Planning. Proc. 23rd FLAIRS Conference. AAAI Press. 2010.