Next Generation Logic Programming Systems: Reaching for the Holy Grail of Computer Science

Gopal Gupta

Department of Computer Science
The University of Texas at Dallas
July 2017

1 Motivation

In the last 40 years logic programming (LP) has emerged as a powerful paradigm for intelligent reasoning and deductive problem solving. In the last few decades several powerful and successful extensions and enhancements of logic programming have been proposed and researched. These include *constraint logic programming (CLP)* [15], *tabled logic programming* [18], *concurrent/parallel logic programming* [7], *deterministic coroutining (the Andorra principle)* [17], *answer set programming* [1], and *coinductive logic programming* [13]. Logic programming, through *inductive logic programming*, has also proven to be foundational in the area of machine learning [16]. These extensions have led to very powerful applications, for example, CLP has been used in industry for intelligently and efficiently solving large scale scheduling and resource allocation problems, tabled logic programming has been used for verification problems, and answer set programming for practical planning problems, etc.

However, all these powerful extensions of logic programming have been developed by extending standard logic programming (Prolog) systems, and each one has been developed in isolation from others. While each extension has resulted in very powerful applications in reasoning, considerably more powerful applications will become possible if all these extensions were combined into a *single system*. This power will come about due to a user having the ability to specify the problem at a very high level, and the underlying LP system (equipped with all these extensions) finding the solution(s) quickly in near-optimal time. Incorporating all these extensions into a single system not only leads to easing a user's burden in specifying the problem as a logic program, it also results in the search space being dramatically pruned, reducing the time taken to find a solution (in other words, reaching the holy grail).

At the outset it should be mentioned that automated reasoning and logic programming remain very important areas in computer science:

– Logic programming relates very closely to various important and foundational areas of computer science. It relates to databases (LP languages are very powerful query languages), compiler-writing (parsers and code generators can be naturally crafted in LP), software engineering (executable specifications are easily developed in LP), OS (process view of predicates), verification & model checking (model checkers and verifiers can be elegantly

developed), AI (intelligent applications can be developed in LP), optimization (through incorporation of constraints in LP). One can argue that logic programming constitutes the *grand unifying theory* of computer science [5]. This should not be surprising at all, as logic is the underpinning of computer science, and logic programming brings in "execution efficiency" by introducing an operational semantics into (a subset of) logic, and execution efficiency is an important aspect of computing.

– While the whole world is currently mesmerized by machine learning and its applications, once the underlying rules are learned from examples, one still needs an automated reasoning mechanism to draw inferences from these rules.

When Prolog was devised it had a very rigid control regime: goals in rules are executed by the Prolog system left to right, while the rules that match a given goal themselves are tried in the textual order in which they appear in the program file (in LP parlance, this is termed as the *computation rule* [3]). Robert A. Kowalski, the inventor of logic programming, proclaimed that

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

and that users should only specify the declarative logic of the problem in their programs, the execution environment should automatically figure out the optimal control strategy for executing the program. This control strategy should be such that it will discover the solution by exploring the smallest possible search space. However, in practice this has not been achieved, and one could argue that much of the research in LP since has been driven by the desire to improve on Prolog's computation's rule and achieve Kowalski's dream. Thus, research has been directed towards making goal selection dynamic (constraints [15], deterministic coroutining [17]) and making clause selection dynamic (tabling [18], or-parallelism [7]). If the logic programming system supports tabling and or-parallelism *programmers don't have to worry about the order of rules*. If the logic programming system supports CLP or deterministic coroutining *programmers don't have to worry about order of goals*. It would therefore be desirable to support CLP, or-parallelism, deterministic coroutining, tabling, and or-parallelism, so that programmers *don't have to worry about either the order of rules or the order of subgoals*, yet find solutions efficiently. Programmers will find it considerably easier to program complex problems because they will be completely freed from worrying about issues of control [8].

The holy grail of computer science, as stated in the call for papers, is that *the user simply states the problem and the computer solves it*. In this position paper we argue that indeed a logic programming system incorporating all the extensions mentioned above is the holy grail of computer science. Specifically, we argue that CLP alone is not the holy grail, incorporating negation-as-failure through answer set programming (ASP) is equally important. This is because standard logic programming only works with least fixed point semantics of the program. Including greatest fixed point (GFP) semantics [13] and negation [1] adds to the expressive power of logic programming. Thus, to reach the "holy

grail," we must at least incorporate techniques such as ASP and coinductive LP that facilitate working with GFP semantics within LP.

As an aside, it should be mentioned that realizing multiple or all the above-mentioned extensions elegantly, in a single system has been rendered difficult by the enormous complexity of implementing reasoning systems in general and logic programming systems in particular. Implementing standard Prolog systems is itself quite complex due to the built-in unification and backtracking that have to be supported. Extending a Prolog system to incorporate constraints, or tabling, or parallelism, further presents formidable challenges. Developing really simple implementation techniques for various extensions should be part of the research community's agenda.

2 Predicate ASP

Constraint logic programming, especially, CLP(FD) has been a phenomenal success (the author himself has developed tools for automatically assigning TAs to courses at his institution that are in use for the last 15 years). However, its success is limited to solving constraint satisfaction and optimization problems. The program for solving Sudoku is just perhaps 30 lines long, as one merely states the constraints that describe the Sudoku puzzle. However, there are many other applications that cannot be simply modeled as CSP problems stated by identifying a set of variables each of which range over a finite set of values, followed by specifying a set of constraints over these variables. Often, the requirement of finiteness becomes a restriction, as general data structures have to be designed to represent and solve the problem efficiently.

Also, it should be noted that constraints by themselves are not enough, ability to manipulate constraints through computation has equally contributed to the success of the CP paradigm (constraints *and* programming). CP/CLP is a framework in which we write programs that dynamically generates constraints that are periodically solved by calling a solver and plugging the results of this constraint-solving back into the computation and continuing.

The computation part can be made simpler by bringing in negation and answer set programming (ASP). ASP allows human-style common sense reasoning to be simulated elegantly—default reasoning, counterfactual reasoning, abductive reasoning, etc. Simulating human-style common sense reasoning in a natural manner is important, as a sizable number of intelligent applications attempt to simulate common sense reasoning. A self-driving car should be simulating the "perfect" human driver, for example. ASP is particularly good at modeling reasoning based on default rules (and exceptions to these default rules) [1]. Humans simplify their burden of reasoning by relying on default rules. For example, we will automatically assume that if Tweety is a bird, it flies. In general, we discount the possibility that Tweety might be a penguin, an ostrich, or it may be wounded, etc. We may not consider these exceptional situations right in the beginning. However, once we learn that that one of the exceptional situation holds, we will withdraw our earlier conclusion that Tweety can fly. Likewise, if I call a

very close friend of mine on his mobile phone in the morning on a weekday, and he doesn't pick it up, then I would immediately conclude that he is busy getting ready for work (because normally he picks up the phone immediately). We will discount less likely reasons such as he may be sick and may be in the hospital or may have had to leave for work early, unless we learn that that is indeed the case. In fact, I may take further actions based on the default conclusion I reached (e.g., I need to talk to my friend urgently, so next I will call his home phone, because I have concluded that he is home, he is just not picking up his mobile phone).

There are many other such reasoning patterns that humans use that can be elegantly modeled by ASP as well, e.g., *preferences, anti-recommendation, concomitant choice, indispensable choice, incompatible choice, etc.* [14]. One could argue that for many applications, such as self-driving cars, human-styled common sense reasoning is a must. Arguably, ASP should be sufficient for modeling human-style reasoning. That is because well-founded reasoning (inductive reasoning) as well as cyclical reasoning (establishing mutual consistency or *coinductive* reasoning) can be both represented in ASP, permitting a wide-range of reasoning patterns to be modeled [13]. Our experience modeling a physician advisory system for chronic heart failure indeed bears this out [14].

ASP provides a very natural way of writing programs, where each rule is easily understood (though understanding the meaning of the whole answer set program is another matter). ASP is an ideal vehicle for knowledge representation and building intelligent applications, however, ASP systems have been implemented using model-finding approaches that rely on grounding the program and using SAT-solvers. This model-finding approach has a number of limitations [6]:

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 is infeasible due to the size of the grounded program that will be created. It is next to impossible to build a general-purpose knowledge-based system using such an approach, as such a knowledge-based system will potentially have tens of thousands of constants.
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 over kill. Most of the time users are interested in a specific piece of information. Thus, if we have 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.

For these reasons, ASP, we believe, has been more focused on solving optimization problems (where SAT solvers shine) and, in fact, competing with CLP(FD). In some instances, it even outperforms CLP, as really sophisticated solvers have been developed for ASP [4].

We believe that restricting ASP to predicates that can only contain constants and variables (i.e., no general terms allowed) and using a model-finding approach based on a SAT solver severely restricts the applicability of ASP to automated reasoning tasks and for building practical knowledge-based systems. Designing query-driven, goal-directed ASP system that support general predicates that can contains general terms as arguments will result in a much more powerful, practical reasoning system. Practical applications that involve common sense reasoning can then be quickly built.

We have been working on designing query-driven answer set programming systems [9]. A query-driven system computes the partial answer set that contains the query (thus, it does not compute the entire answer set). Having a query-driven system addresses problems 5, 6 and 7 mentioned above [10], 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 [11, 12].

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 [12]. Problem 1, 2, 3, and 4 above are eliminated by s(ASP), since programs do not have to be grounded prior to execution. The s(ASP) system is publicly available [11], and has been used to develop a number of non-trivial applications based on ASP. Some of these applications cannot be executed on traditional ASP systems such as CLASP, as these applications make use of lists and structure to represent information. They have been developed by people who are not experts in ASP. These applications include:

- A system for automatically performing degree audit of a student's undergraduate transcript at a US University, i.e., automatically determining if a student can graduate with a degree or not. 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).
- A system for disease management, particularly, for chronic heart failure. This system automates the 80-page guidelines (that the American College of Cardiology has developed) by representing them in ASP. Our tests indicate that the system is able to make recommendations that doctors have missed [14].
- A system that represents high-school level knowledge about cells (in the discipline of biology) as answer set programs. 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).
- A recommendation system for birthday gifts: 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.

Many other applications by non-expert users have been developed using the s(ASP) system.

We believe that the ASP paradigm is a very powerful paradigm that allows for complex human thought processes to be elegantly emulated [14]. Common sense reasoning is an important application of automated reasoning and we believe that being able to model common sense reasoning is critical to building practical automated reasoning systems. However, as argued above, the current model-finding, SAT-solver based approaches are not able to realize the full-power of ASP. We believe that query-driven implementations of predicate ASP are crucial to the ASP paradigm's success and that such implementations are needed to make the ASP technology practical for building large, scalable knowledge-based applications. 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. Integration with other LP techniques such as CLP(FD), CLP(R), Tabled LP, etc., is more readily possible in a query-driven approach.

ASP can be thought of as a better way of realizing "expert systems". The expert system enterprise failed in the past, we believe, due to the fact that standard logic-based formalisms that are monotonic in nature are inadequate for modelling human reasoning and common sense reasoning. Earlier implementations of expert system relied on these monotonic logics that are brittle and

cannot tolerate inconsistencies (in the presence of an inconsistency every proposition as well as its negation becomes true). ASP, based on negation as failure, stable model semantics, and a non-monotonic logic, in contrast, is quite capable of faithfully modeling human reasoning. However, generalization to predicates and a query-driven execution model are critical to the success of using ASP for building expert systems, we believe.

3 Reaching for The Holy Grail

Significant progress has been made in developing query-driven predicate ASP systems with the implementation of our s(ASP) system. However, a query-driven system brings all the problems that Prolog has suffered from: program efficiency is affected by the order of rules and goals. However, these problems can be eliminated by incorporating extensions outlined in section 1, namely, CLP(FD), CLP(R), Tabled LP, deterministic coroutining and or-parallelism, to free the user from determining the order of search.

Thus, we believe that the holy grail can be reached, if we build a single system that extends Prolog with stable model semantics based negation, CLP(FD), CLP(R), Tabling, deterministic coroutining, coinduction, and or-parallelism. It will be quite challenging to build such a system, but significant progress is being made.

Considerable amount of research remains to be done. My own research program over the last 20 years has been guided by this desire to reach the holy grail. We urge the research community to invest more effort in this direction.

Acknowledgment: Support from NSF Grant IIS 1423419 is gratefully acknowledged. I am indebted to many people for discussions, especially, my current and past research group members.

References

1. Baral, C. 2003. *Knowledge Representation - Reasoning & Declarative Problem Solving*. Cambridge Univ. Press.
2. Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *International Conference and Symposium*, 1070–1080.
3. J. Lloyd. *Foundations of Logic Programming* (2nd. Ed.). Springer Verlag, 1987.
4. Martin Gebser, Benjamin Kaufmann, André Neumann, Torsten Schaub. clasp: A Conflict-Driven Answer Set Solver, LPNMR07, 2007. `https://potassco.org/`.
5. G. Gupta. Logic Programming: The Grand Unifying Theory of Computer Science. Talk given at UT Dallas, 2004.
6. G. Gupta, et al. A Case for Predicate ASP (Position Paper). Proc. of ARCADE 2017 Workshop (CADE 2017).
7. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, M. Hermenegildo, Parallel Execution of Prolog Programs: A Survey. In *ACM TOPLAS* 23(4): 472-602 (2001)
8. G. Gupta. Next Generation Logic Programming Systems. Technical Report. UT Dallas CS Department. 2003.

9. K. Marple, G. Gupta Goal-directed execution of answer set programs. Proc. PPDP 2012: 35-44
10. K. Marple, G. Gupta. Dynamic Consistency Checking in Goal-Directed Answer Set Programming. TPLP 14(4-5): 415-427 (2014)
11. K. Marple, E. Salazar, G. Gupta. The s(ASP) system. `https://sourceforge.net/projects/sasp-system/`
12. K. Marple, E. Salazar, G. Gupta. Computing Stable Models of Normal Logic Programs without Grounding. Forthcoming paper. March 2017.
13. G. Gupta, L. Simon, A. Mallya, R. Min, A. Bansal. Coinductive Logic Programming and Its Applications. Invited Tutorial. Proc. ICLP 2007. pp. 27-44.
14. Zhuo Chen, Kyle Marple, Elmer Salazar, Gopal Gupta, Lakshman Tamil. A Physician Advisory System for Chronic Heart Failure management based on knowledge patterns. TPLP 16(5-6):604-618, 2016.
15. K. Marriott, P. Stuckey. Programming with Constraints. MIT Press. 1998.
16. S. Muggleton, L. De Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, Vol. 19/20, 1994. pp. 629–679.
17. V. Santos Costa, R. Yang. Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism. In *Proceedings of ACM PPoPP*, Apr. 1991. pp. 83–93.
18. Terrance Swift, David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. TPLP 12(1-2): 157-187 (2012)