Tools and Methodologies for Verifying Answer Set Programs

Zach Hansen
University of Nebraska Omaha
Omaha, Nebraska
zachhansen@unomaha.edu

1 Introduction

Answer Set Programming (ASP) is a powerful declarative programming paradigm commonly used for solving challenging search and optimization problems [14, 15]. The modeling languages of ASP are supported by sophisticated solving algorithms (solvers) that make the solution search efficient while enabling the programmer to model the problem at a high level of abstraction [11]. As an approach to Knowledge Representation and Reasoning, ASP benefits from its simplicity, conciseness and rigorously defined semantics. These characteristics make ASP a straightforward way to develop formally verifiable programs. In the context of artificial intelligence (AI), the clarity of ASP programs lends itself to the construction of explainable, trustworthy AI. In support of these goals, my research is concerned with extending the theory and tools supporting the verification of ASP progams.

The formal verification of ASP programs presents several challenges that differentiate it from the verification of procedural imperative software. First and foremost, ASP programs lack modularity; traditionally, the meaning of a portion of code cannot be determined in isolation from the remainder of the program. Second, since the program semantics are traditionally defined through grounding, it is difficult to reason formally about the correctness of a program without referring to a specific problem instance. Finally, the well-established tools of proof for reasoning about procedural imperative languages (such as loop invariants) are not directly applicable to ASP programs. Instead, we must explore new verification strategies, both manual and automated, to provide a high level of assurance about our programs' behavior.

2 Background

Approaches to defining semantics for logic programs can be roughly divided into two broad categories: fixpoint formalisms, and translational formalisms [5]. The original definition of a stable model from 1988 is an example of the fixpoint approach; it defines the behavior of basic logic programs with negation in terms of reducts [7]. Advances in grounding and solving technology have made the implementation of these semantics efficient, however, reasoning about the correctness of programs in terms of fixpoints can be challenging. Part of the challenge is the inseperability of problem class and problem instance. Another challenge faced by fixpoint approaches is the definition of advanced language constructs, such as arbitrary choice rules, conditional literals, and aggregates. These are useful language features, but their behavior in terms of the grounding and solving process is typically described informally [9]. Thus, most of my research so far has focused on developing alternative translational semantics for advanced ASP language constructs.

Y. Lierler, J.F. Morales, C. Dodaro, V. Dahl, M. Gebser, T. Tekle (Eds.):
International Conference on Logic Programming 2022 (ICLP'22).
EPTCS 364, 2022, pp. 211–216, doi:10.4204/EPTCS.364.36

One example of a translational approach is *program completion* [2, 8]. Programs meeting the syntactic requirement of *tightness*[1] can be converted into a first-order sentence whose models are in one-to-one correspondence with the answer sets of the program. A related approach is the SM operator [5]. Much like completion, applying the SM operator to a logic program first requires a syntactic transformation into a first-order sentence. The SM operator then transforms this sentence into a second-order one and uses predicate quantification to minimize belief in certain *intensional* predicates. When all the predicates occurring in the logic program are treated as intensional, then the models of this second-order sentence correspond to the program's answer sets. These translational approaches have the advantage of avoiding grounding entirely.

ANTHEM is a software system that transforms a non-ground ASP program into an equivalent first-order theory via program completion [4, 13]. Given an ASP program and a formal specification written in first-order logic, ANTHEM translates the program into typed first-order theories and then uses the theorem prover VAMPIRE [12] to verify the adherence of the translation to the specification. This provides a way to automatically verify the correctness of tight logic programs. We wish to extend the capabilities of ANTHEM to include a broader class of logic programs, such as those containing aggregates and conditional literals. The semantics implemented by the answer set solver CLINGO of these language constructs are given by a translation [9] to infinitary propositional logic [17].

3 Current Status

Since joining the University of Nebraska Omaha in Fall 2020, I have been engaged in several research projects under the umbrella topic of *formal verification of ASP programs*. I have worked with my advisor (Dr. Yuliya Lierler) and Dr. Jorge Fandinno to define the semantics of non-basic ASP language constructs such as aggregates and conditional literals in a manner convenient for proving results about their behavior. Additionally, we have extended the modular proof methodology developed by Cabalar, Fandinno, and Lierler (2020) to programs with aggregates occurring in constraints.

3.1 Many-sorted semantics for aggregates ([3])

A project I have recently worked on with Dr. Lierler and Dr. Fandinno provided an alternative characterization of aggregate semantics, which are typically defined through a translation to infinitary propositional logic. Conversely, our approach avoids referring to grounding by applying a many-sorted generalization of the SM operator to a set of many-sorted first-order formulas representing a logic program. Aggregates are defined as functions on sets of tuples, whose members are restricted to those tuples satisfying the list of conditions present in the associated aggregate. To ensure this behavior, we add second-order axioms to the program that fix the behavior of sets and aggregate function symbols. We proved the equivalence of our semantics to the semantics implemented by CLINGO for programs that do not contain positive recursion through aggregates. Furthermore, for tight programs with finite aggregates, we can replace the second-order axiomatization with a first-order one. This results in a fully first-order characterization of the behavior of these programs.

This contribution helps to address one of the fundamental issues identified in the Introduction; namely, it decouples the argument of program correctness from specific problem instances for programs with aggregates. Since aggregates are such useful and common constructs, this substantially broadens the class of ASP programs with formally defined non-ground semantics. We also envision this work as

[1] A program is *tight* if it has an acyclic dependency graph, for details see [1].

part of the foundation required to automatically verify the correctness of programs with aggregates. The long-term goal of this line of research is to extend the capabilities of ANTHEM accordingly (Section 4.1).

3.2 Many-sorted semantics for conditional literals

In a similar vein as the project from Section 3.1, I developed a many-sorted translation and axiomatization for conditional literals. Syntactically and semantically, these constructs resemble set notation in traditional mathematics. I attempted to formalize this intuition by relating conditional literals to sets of tuples satisfying the conditions in the literal. The added axioms then mandate that the (negated) predicate in the head of the conditional literal must (not) hold for every tuple of terms in the associated set. As before, this project is designed to expand the class of ASP programs for which we have a simple, non-ground semantics.

3.3 Conditional literals as nested implications

The many-sorted approach to defining conditional literal semantics (Section 3.2) was appealing because, on the surface, it paralleled the informal presentation of conditional literal behavior found in teaching materials and manuals. For example, conditional literals are traditionally described (in an example-driven way) by a translation to ground forms of basic rules [6]. Our hope was that this new characterization could be useful from a pedagogical perspective in addition to aiding proofs of correctness. However, developing formal justifications for the equivalence of these semantics to those defined via infinitary propositional logic [9] was cumbersome, and the axiomatic characterization itself was confusing. Therefore, we replaced this approach with a simpler translation to unsorted first-order logic following the intuition that conditional literals behave as nested implications. We proved the equivalence of our semantics (defined by the SM operator applied to our translated programs) to those implemented by CLINGO via their equivalence to the infinitary propositional logic semantics. Once again, for tight programs this provides a first-order treatment that could be used to extend ANTHEM.

3.4 Modular proofs of correctness with aggregate constraints

Cabalar, Fandinno, and Lierler (2020) proposed a modular methodology for arguing the correctness of ASP programs. In this approach, the program is divided into various independent modules, whose behavior is captured via the SM operator. They showcase their approach using an encoding that solves the Hamiltonian Cycle problem. We extend this encoding to the Traveling Salesman problem with the addition of a constraint on the cumulative weight of the selected cycle. Importantly, this constraint uses the `sum` aggregate, which necessitated the application of our many-sorted semantics for aggregates (Section 3.1). We "recycle" the proof of correctness developed for the Hamiltonian Cycle encoding, and extend it with our own proof of correctness for the aggregate constraint. This showcases the utility of both the many-sorted aggregate semantics and the modular proof methodology. We show that our approach is also applicable to programs containing choice rules with cardinality bounds by formally proving the correctness of a Graph Coloring encoding.

4 Ongoing Directions

4.1 Extending ANTHEM with sorts

The projects detailed in Sections 3.1, 3.2, and 3.3 are pieces of the foundation for a many-sorted implementation of ANTHEM. Developing the theory and tools for this system will likely be the basis of my dissertation, and will comprise the majority of my research activity going forward. This project presents a number of opportunities and challenges. First and foremost is the issue of automatically verifying tight programs with aggregates. ANTHEM requires an input specification written in first-order logic against which to compare the logic program, and currently there is no clear way to represent aggregates in this specification language. Developing such a representation and demonstrating its validity is an ongoing project for Dr. Lierler, Dr. Fandinno, and I. An alternative approach could be to enhance ANTHEM with the ability to check the equivalence of two programs. Then a simple, human-verified program with aggregates could act as the "specification" and an alternative encoding (perhaps re-written for performance instead of readability) could be checked against it.

Extending ANTHEM to programs with aggregates will also require us to add more sort information to ANTHEM. Specifically, we will need to characterize the behavior of sets and set membership. The semantics introduced in Section 3.1 make assumptions that, for example, set membership behaves "as expected," however such an assumption cannot be enforced in the current versions of ANTHEM and VAMPIRE. The Thousands of Problems for Theorem Provers (TPTP) project has numerous partial axiomatizations of theories written in typed first-order formulas, which makes them compatible with VAMPIRE [16]. We may be able to use these as a starting point for implementing our assumptions. The theory of program completion will have to be defined for programs with multiple sorts, and the capabilities of VAMPIRE will need to be extended as described before we can add aggregates to ANTHEM.

4.2 ASP modules

A project that I would like to undertake is the establishment of a repository of verified ASP sub-programs ("modules") that provide efficient, correct implementations of commonly encountered sub-problems. A simple example of this would be the transitive closure of a binary relation. A user who is not familiar with the nuances of ASP grounding (for instance, a civil engineer trying to apply ASP to a domain-specific problem) might write the rules

$$\begin{aligned} &\mathit{reachable}(\mathit{City}1,\mathit{City}2) \text{ :- } \mathit{road}(\mathit{City}1,\mathit{City}2).\\ &\mathit{reachable}(\mathit{City}1,\mathit{City}3) \text{ :- } \mathit{reachable}(\mathit{City}1,\mathit{City}2),\mathit{reachable}(\mathit{City}2,\mathit{City}3). \end{aligned}$$

This second rule would have an unnecessary negative impact on the grounding size and solving time of their program. A better alternative could be downloaded from an ASP repository:

$$\begin{aligned} &\mathit{transitive}(X,Y) \text{ :- } \mathit{edge}(X,Y).\\ &\mathit{transitive}(X,Z) \text{ :- } \mathit{transitive}(X,Y),\mathit{edge}(Y,Z). \end{aligned}$$

The programmer would have to define the interfaces (e.g. $\mathit{road}/2$ should be taken as the input relation mapped onto $\mathit{edge}/2$, and $\mathit{reachable}/2$ should be the output obtained from $\mathit{transitive}/2$) but otherwise such a module would be easy to integrate. For more complex sub-problems, a formal guarantee about the correctness of the module could be useful. The format of such guarantees may draw inspiration from Hoare triples [10]. For example, in this case the precondition would be the existence of a binary relation

(which will be renamed $edge/2$ within the module) and the postcondition would be the existence of a binary relation representing the transitive closure of the input relation.

Widespread community adoption of such a project would be beneficial to both developers and researchers. Reusing efficient, trustworthy code will mean more efficient, trustworthy applications of ASP in addition to reducing the time and effort required to develop them. Furthermore, creating formal guarantees of program correctness is a laborious process. Taking a modular approach where previous contributions can be shared and reused decreases this workload and might make researchers more apt to contribute such proofs. This is the strategy we undertook in our extension of the Hamiltonian Cycle problem to the Traveling Salesman problem (Section 3.4).

5 Conclusions

I believe that the development of trustworthy software is an important and rewarding direction of research. The formal verification of software is especially crucial in high-consequence or safety-critical systems when we need high levels of assurance. ASP has many strengths to recommend it for use in such systems. In fact, it was the clarity and "verifiability" of ASP programs that attracted me to this paradigm in the first place. As such, I hope that my dissertation research can expand the tools and strategies available to ASP programmers as they develop reliable software.

References

[1] Pedro Cabalar, Jorge Fandinno & Yuliya Lierler (2020): *Modular Answer Set Programming as a Formal Specification Language*. Theory and Practice of Logic Programming, doi:10.1007/978-1-4471-0043-0.

[2] Keith L. Clark (1978): *Negation as Failure*, pp. 293–322. Springer US, Boston, MA, doi:10.1007/978-1-4684-3384-5_11.

[3] Jorge Fandinno, Zach Hansen & Yuliya Lierler (2022): *Axiomatization of Aggregates in Answer Set Programming*. In: Proceedings of the Thirty-six National Conference on Artificial Intelligence (AAAI'22), AAAI Press.

[4] Jorge Fandinno, Vladimir Lifschitz, Patrick Lühne & Torsten Schaub (2020): *Verifying Tight Logic Programs with anthem and vampire*. Theory and Practice of Logic Programming 5(20), pp. 735–750, doi:10.1017/S1471068403001765.

[5] Paolo Ferraris, Joohyung Lee & Vladimir Lifschitz (2011): *Stable models and circumscription*. Artificial Intelligence 175(1), pp. 236–263, doi:10.1016/j.artint.2010.04.011.

[6] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele & Philipp Wanko (2019): *Potassco User Guide*, 2.2.0 edition. University of Potsdam.

[7] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming*. In Robert Kowalski, Bowen & Kenneth, editors: Proceedings of International Logic Programming Conference and Symposium, MIT Press, pp. 1070–1080. Available at http://www.cs.utexas.edu/users/ai-lab?gel88.

[8] Amelia Harrison, Vladimir Lifschitz & Dhananjay Raju (2017): *Program completion in the input language of GRINGO*. Theory and Practice of Logic Programming 17(5-6), pp. 855–871, doi:10.1007/978-1-4684-3384-5_11.

[9] Amelia Harrison, Vladimir Lifschitz & Fangkai Yang (2014): *The Semantics of Gringo and Infinitary Propositional Formulas*. In: Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning, KR'14, AAAI Press, p. 32–41, doi:10.1007/978-1-4684-3384-5_11.

[10] Charles Anthony Richard Hoare (1969): *An Axiomatic Basis for Computer Programming*. Commun. ACM 12(10), p. 576–580, doi:10.1145/363235.363259.

[11] Tomi Janhunen & Ilkka Nimelä (2016): *The Answer Set Programming Paradigm*. AI Magazine 37(3), pp. 13–24, doi:10.1609/aimag.v37i3.2671. Available at https://aaai.org/ojs/index.php/aimagazine/article/view/2671.

[12] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, Lecture Notes in Computer Science 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[13] Vladimir Lifschitz, Patrick Lühne & Torsten Schaub (2020): *Towards Verifying Logic Programs in the Input Language of clingo*. In Andreas Blass, Patrick Cégielski, Nachum Dershowitz, Manfred Droste & Bernd Finkbeiner, editors: *Fields of Logic and Computation III*, Springer International Publishing, Cham, pp. 190–209, doi:10.1007/978-1-4684-3384-5_11.

[14] Victor W. Marek & Miroslaw Truszczyński (1999): *Stable Models and an Alternative Logic Programming Paradigm*, pp. 375–398. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-60085-2_17.

[15] Ilkka Niemelä (1999): *Logic programs with stable model semantics as a constraint programming paradigm*. Annals of Mathematics and Artificial Intelligence 25(3), pp. 241–273, doi:10.1023/A:1018930122475.

[16] Geoff Sutcliffe (2017): *The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0*. Journal of Automated Reasoning 59(4), pp. 483–502, doi:10.1007/s10817-017-9407-7.

[17] Miroslaw Truszczynski: *Connecting First-Order ASP and the Logic FO(ID) through Reducts*. Lecture Notes in Computer Science 7265, Springer Berlin Heidelberg, doi:10.1007/978-3-642-30743-0_37. Available at http://link.springer.com/10.1007/978-3-642-30743-0_37.