First-Order Modular Logic Programs and their Conservative Extensions (Extended Abstract)

Amelia Harrison
University of Texas at Austin
ameliaj@cs.utexas.edu

Yuliya Lierler
University of Nebraska Omaha
ylierler@unomaha.edu

Abstract

This paper introduces first-order modular logic programs, which provide a way of viewing answer set programs as consisting of many independent, meaningful modules. We also present conservative extensions of such programs. This concept helps to identify strong relationships between modular programs as well as between traditional programs. For example, we illustrate how the notion of a conservative extension can be used to justify the common projection rewriting. This is a short version of a paper was presented at the 32nd International Conference on Logic Programming [Harrison and Lierler, 2016].

1 Introduction

Answer set programming is a prominent knowledge representation paradigm with roots in logic programming [Leake, 2016]. It is especially useful for addressing combinatorial search problems. In answer set programming, a given computational problem is represented by a declarative program that describes the properties of a solution to the problem. Then, an answer set solver is used to generate answer sets, also called stable models, for the program. These models correspond to solutions to the original problem.

In this paper we show how some logic programs under the answer set semantics can be viewed as consisting of various "modules", and how stable models of these programs can be computed by composing the stable models of the modules. We call collections of such modules first-order modular programs. To illustrate this approach consider the following two rules

$$r(X,Y) \leftarrow in(X,Y). \quad (1)$$
$$r(X,Y) \leftarrow r(X,Z), r(Z,Y). \quad (2)$$

Intuitively, these rules encode that the relation r is the transitive closure of the relation in. The empty set is the only answer set of the program composed of these rules alone. Thus, in some sense the meaning of these two rules in isolation is the same as the meaning of any program that has a single answer set that is empty. (The empty program is an example of another program with a single empty answer set.) We show how to view rules (1) and (2) as a module and use the operator SM introduced by Ferraris et al. (2011) to define a semantics that corresponds more accurately to the intuitive meaning of these rules. The operator SM provides a definition of the stable model semantics for first-order logic programs that does not refer to grounding or fixpoints as does the original definition [Gelfond and Lifschitz, 1988]. The operator SM has proved an effective tool for studying properties of logic programs with variables, which are the focus of this paper.

Modularity is essential for modeling large-scale practical applications. Here we propose *first-order modular programs* and argue their utility for reasoning about answer set programs. We use the *Hamiltonian Cycle* problem as a running example to illustrate that a "modular" view of a program gives us

- a more intuitive reading of the parts of the program;
- the ability to incrementally develop modules or parts of a program that have stand-alone meaning and that interface with other modules via a common signature;
- a theory for reasoning about modular rewritings of individual components with a clear picture of the overall impact of such changes.

First-order modular programs can be viewed as a generalization of propositional modular logic programs [Lierler and Truszczyński, 2013]. In turn, propositional modular logic programs generalize the concept of modules introduced by Oikarinen and Janhunen 2008. ASP-FO logic [Denecker *et al.*, 2012] is another related formalism. It is a modular formalization of *generate-define-test* answer set programming [Lifschitz, 2002] that allows for unrestricted interpretations as models, non-Herbrand functions, and first-order formulas in the bodies of rules. An ASP-FO theory is a set consisting of modules of three types: G-modules (G for generate), D-modules (D for define), and T-modules (T for test). In contrast, there is no notion of type among modules in the modular programs introduced here.

We also define conservative extensions for first-order modular programs. This concept is related to strong equivalence for logic programs [Lifschitz *et al.*, 2001]. If two rules are strongly equivalent, we can replace one with the other within the context of any program and the answer sets of the resulting program coincide with those of the original one. Conservative extensions allow us to reason about rewritings even

when the rules in question have different signatures. The theorem stated at the end of this paper, for instance, shows that conservative extensions can be used to justify the projection rewriting [Faber *et al.*, 1999], which is commonly employed to improve program performance. Consider the rule

$$\leftarrow not\ r(X,Y), edge(X,Z), edge(Z',Y). \tag{3}$$

which says that every vertex must be reachable from every other vertex. This rule can be replaced with the following three rules without affecting the stable models in an "essential way"

$$\leftarrow not\ r(X,Y) \wedge v1(X) \wedge v2(Y).$$
$$v1(X) \leftarrow edge(X,Y).$$
$$v2(Y) \leftarrow edge(X,Y).$$

Furthermore, this replacement is valid in the context of any program, as long as that program does not already contain either of the predicates *v1* and *v2*. Currently, these performance-enhancing rewritings are done manually. We expect the theory about conservative extensions developed here to provide a platform for automating such rewritings in the future. We note that conservative extensions are related to the notion of knowledge forgetting in [Wang *et al.*, 2014]. However, that work applies only to propositional programs.

2 Review: Traditional Programs

A *(traditional logic) program* is a finite set of *rules* of the form

$$a_1;\ldots;a_k \leftarrow \quad a_{k+1},\ldots,a_l, not\ a_{l+1},\ldots,not\ a_m, \quad not\ not\ a_{m+1},\ldots,not\ not\ a_n, \tag{4}$$

$(0 \leq k \leq l \leq m \leq n)$, where each a_i is an atomic formula, possibly containing function symbols, variables, or the equality symbol with the restriction that atomic formulas $a_1,\ldots,a_k$ and $a_{m+1},\ldots,a_n$ may not contain the equality symbol. The expression containing atomic formulas a_{k+1} through a_n is called the *body* of the rule. A rule with an empty body is called a *fact*. An *instance* of a rule R in a program Π is a rule that can be formed by replacing all variables in R with ground terms formed from function symbols and object constants occurring in Π. The process of *grounding* a traditional logic program consists of the following steps:

1. each rule is replaced with all of its instances by substituting ground terms for variables;
2. in each instance, every atomic formula of the form $t_1 = t_2$ is replaced by $\top$ if t_1 is the same as t_2 and by $\bot$ otherwise.

It is easy to see that the resulting ground program does not have equality symbols and can be viewed as a propositional program. The *answer sets* of a traditional program Π are stable models of the result of grounding Π, where stable models are defined using the fixpoint operation introduced in [Ferraris, 2005].

Traditional programs do not include some constructs available in ASP input languages. (For example, aggregate expressions and arithmetic are not covered.) Even so, they do cover a substantial practical fragment. In particular, according to [Ferraris and Lifschitz, 2005] and [Ferraris, 2005], rules of the form (4) are sufficient to capture the meaning of the commonly used choice rule construct. For instance, the choice rule $\{\mathtt{p(X)}\} \leftarrow \mathtt{q(X)}$ is understood as the rule

$$p(X) \leftarrow q(X),\ not\ not\ p(X).$$

Consider the *Hamiltonian Cycle* problem on an undirected graph. A *Hamiltonian Cycle* is a subset of the set of edges in a graph that forms a cycle going though each vertex exactly once. A sample program for finding such a cycle can be constructed by adding rules (1), (2), and (3) to the following:

$$edge(a,a').\ \ldots\ edge(c,c'). \tag{5}$$
$$edge(X,Y) \leftarrow edge(Y,X). \tag{6}$$
$$\{in(X,Y)\} \leftarrow edge(X,Y). \tag{7}$$
$$\leftarrow in(X,Y), in(X,Z), Y \neq Z. \tag{8}$$
$$\leftarrow in(X,Z), in(Y,Z), X \neq Y. \tag{9}$$
$$\leftarrow in(X,Y), in(Y,X). \tag{10}$$

Each answer set of the *Hamiltonian Cycle* program above corresponds to a Hamiltonian cycle of the given graph, specified by facts (5), so that the predicate *in* encodes these cycles. If an atom $in(a,b)$ appears in an answer set it says that the edge between a and b is part of the subset forming the Hamiltonian cycle. Intuitively,

- the facts in (5) define a graph instance by listing its edges, and rule (6) ensures that this *edge* relation is symmetric (since we are dealing with an undirected graph); the vertices of the graph are implicit—they are objects that occur in the edge relation;[1]
- rule (7) says that any edge may belong to a Hamiltonian cycle;
- rules (8) and (9) impose the restriction that no two edges in a Hamiltonian cycle may start or end at the same vertex, and rule (10) requires that each edge appears at most once in a Hamiltonian cycle (recall that $in(a,b)$ and $in(b,a)$ both encode the information that the edge between a and b is included in a Hamiltonian cycle);
- rules (1) and (2) define a relation r (reachable) that is the transitive closure of relation *in*;
- rule (3) ensures that every vertex in a Hamiltonian cycle must be reachable from every other vertex.

Clearly, rules in the program can be grouped according to intuitive meaning. Yet, considering these groups separately will not produce "meaningful" logic programs under the answer set semantics as discussed in the introduction. In this paper, we show how we can view each of these groups of rules as a separate module, and then use the SM operator [Ferraris *et al.*, 2011; 2009], along with a judicious choice of "intensional" and "extensional" predicates to achieve a more accurate correspondence between the intuitive reading of the groups of rules and their model-theoretic semantics.

3 Operator SM

The SM operator was introduced by Ferraris et al. (2011; 2009). There are a few key differences between the original stable model semantics and the semantics provided by the

[1] This precludes graphs that include isolated vertices.

SM operator that make the latter a convenient formalism for facilitating a view of groups of rules in a program as separate units or "modules". First, the SM operator does not rely on grounding, but instead operates on first-order sentence representations of logic programs. Secondly, the semantics as defined by SM allows for a distinction between "extensional" and "intensional" predicates. Intuitively, "extensional" predicates correspond to the input of the program, or module, and "intensional" predicates correspond to output or auxiliary concepts. Finally, unlike the original stable model semantics, the SM operator does not involve a fixpoint calculation, but instead defines stable models as models of a second-order formula. The result of applying the SM operator to a first-order sentence F with intensional predicates $\mathbf{p}$ is a second-order formula, denoted $\mathrm{SM}_{\mathbf{p}}[F]$.

The SM operator applies to first-order sentences, rather than to logic programs. Yet, traditional logic programs can be identified with the first-order sentences of a particular form. For example, we understand the *Hamiltonian Cycle* program presented in Section 2 as an abbreviation for the conjunction of the following formulas

$$\begin{array}{l} edge(a,a') \wedge \ \ldots \ \wedge edge(c,c') \\ \forall xy(edge(y,x) \rightarrow edge(x,y)) \\ \forall xy((\neg\neg in(x,y) \wedge edge(x,y)) \rightarrow in(x,y)) \\ \forall xyz((in(x,y) \wedge in(x,z) \wedge \neg(y=z)) \rightarrow \bot) \\ \forall xyz((in(x,z) \wedge in(y,z) \wedge \neg(x=y)) \rightarrow \bot) \\ \forall xy((in(x,y) \wedge in(y,x)) \rightarrow \bot) \\ \forall xy(in(x,y) \rightarrow r(x,y)) \\ \forall xyz((r(x,z) \wedge r(z,y)) \rightarrow r(x,y)) \\ \forall xyzz'((\neg r(x,y) \wedge edge(x,z) \wedge edge(z',y)) \rightarrow \bot) \end{array} \tag{11}$$

where $a, a', \ldots c, c'$ are object constants and x, y, z, z' are variables.

The answer sets of any traditional program Π that contains at least one object constant coincide with Herbrand models of $\mathrm{SM}_{\mathbf{p}}[\Pi]$, where $\mathbf{p}$ is the list of all predicates occurring in Π.

4 Modular Logic Programs

A first-order modular logic program is a collection of logic programs, where the SM operator is used to compute models of each individual logic program in the collection.

We call a formula of the form $\mathrm{SM}_{\mathbf{p}}[F]$, where $\mathbf{p}$ is a tuple of predicate symbols and F is a traditional program, a *defining module (of* $\mathbf{p}$ *in* F*)* or a *def-module*. We can view any traditional program Π as a def-module $\mathrm{SM}_{\mathbf{p}}[\Pi]$, where $\mathbf{p}$ is the list of all predicates occurring in Π. A *first-order modular logic program* (or, *modular program*) P is a finite set of def-modules

$$\{\mathrm{SM}_{\mathbf{p_1}}[F_1], \ldots, \mathrm{SM}_{\mathbf{p_n}}[F_n]\}.$$

By $\sigma(P)$ we denote the set of all function and predicate symbols occurring in a modular program P, also called the *signature of* P. A *stable model* of a modular program P is any interpretation over $\sigma(P)$ that is a model of the conjunction of all def-modules in P.

We now illustrate how modular programs capture the encoding (11) of the *Hamiltonian Cycle* so that each of its modules carries its intuitive meaning. The modular program P_{hc} consists of five def-modules:

$$\mathrm{SM}_{edge}[edge(a,a') \wedge \ \ldots \ \wedge edge(c,c') \wedge \forall xy(edge(y,x) \rightarrow edge(x,y))] \tag{12}$$

$$\mathrm{SM}_{in}[\forall xy((\neg\neg in(x,y) \wedge edge(x,y)) \rightarrow in(x,y))] \tag{13}$$

$$\mathrm{SM}[\forall xyz((in(x,y) \wedge in(x,z) \wedge \neg(y=z)) \rightarrow \bot) \wedge \forall xyz((in(x,z) \wedge in(y,z) \wedge \neg(x=y)) \rightarrow \bot) \wedge \forall xy((in(x,y) \wedge in(y,x)) \rightarrow \bot)] \tag{14}$$

$$\mathrm{SM}_{r}[\forall xy(in(x,y) \rightarrow r(x,y)) \wedge \forall xyz((r(x,z) \wedge r(z,y)) \rightarrow r(x,y))] \tag{15}$$

$$\mathrm{SM}[\forall xyzz'((\neg r(x,y) \wedge edge(x,z) \wedge edge(z',y)) \rightarrow \bot)] \tag{16}$$

The def-modules shown above correspond to the intuitive groupings of rules of the *Hamiltonian Cycle* encoding discussed in Section 2.

- A model of def-module (12) is any interpretation I over $\sigma(P_{hc})$ such that the extension[2] of the *edge* predicate in I corresponds to the symmetric closure of the facts in (5).
- A model of def-module (13) is any interpretation I over $\sigma(P_{hc})$ such that the extension of the predicate *in* in I is a subset of the extension of the predicate *edge* in I.
- A model of def-module (14) is any interpretation I over $\sigma(P_{hc})$ that satisfies the conjunction in (14).
- A model of def-module (15) is any interpretation I over $\sigma(P_{hc})$, where relation r is the transitive closure of relation *in*.
- A model of def-module (16) is any interpretation I over $\sigma(P_{hc})$ that satisfies the implication in (16).

Any interpretation over $\sigma(P_{hc})$ that satisfies the conditions imposed by every module is a stable model of P_{hc}.

5 Relating Modular Programs and Traditional Programs

We view a traditional logic program as an abbreviation for a first-order sentence formed as a conjunction of formulas of the form

$$\widetilde{\forall}(a_{k+1} \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n \rightarrow a_1 \vee \cdots \vee a_k), \tag{17}$$

which corresponds to rule (4). The symbol $\widetilde{\forall}$ denotes universal closure. We call the disjunction in the consequent of a rule (17) its *head*, and the conjunction in the antecedent its *body*. The conjunction $a_{k+1} \wedge \cdots \wedge a_l$ constitutes the *positive part of the body*. A modular program is called *simple* if for every def-module $\mathrm{SM}_{\mathbf{p}}[F]$, every predicate symbol p occurring in the head of a rule in F occurs also in the tuple $\mathbf{p}$. For instance, P_{hc} is a simple modular program.

The *dependency graph* [Ferraris *et al.*, 2009] of a simple modular program P, denoted $DG[P]$, is a directed graph that

- has all intensional predicates in P as vertices, and

[2]The *extension* of a predicate in an interpretation is the set of tuples that satisfy the predicate in that interpretation.

- has an edge from p to q if there is a def-module $SM_{\mathbf{p}}[F] \in P$ containing a rule with p occurring in the head and q occurring in the positive part of the body.

We call a simple modular program P *coherent* if

(i) no two def-modules in P have overlapping intensional predicates, and

(ii) every strongly connected component in the dependency graph of P is contained within $\mathbf{p}$ for some def-module $SM_{\mathbf{p}}[F]$ in P.

From the symmetric splitting result from [Ferraris *et al.*, 2009] it follows that the Herbrand stable models of a coherent modular program P that (i) contains at least one object constant and (ii) has each predicate symbol in P occurring in $\mathbf{p}$ for some def-module $\mathrm{SM}_{\mathbf{p}}[F]$, coincide with the answer sets of the traditional program constructed as the conjunction of all first order sentences occurring in this modular program.

The strongly connected components of the dependency graph of P_{hc} each consist of a single vertex. It is easy to check that the *Hamiltonian Cycle* program P_{hc} is coherent and that all of its predicate symbols are intensional in some def-module. Therefore, the Herbrand models of P_{hc} coincide with the answer sets of (11) so that answer set solvers can be used to find these models.

Arguably, when answer set practitioners develop their applications they intuitively associate meaning with components of their programs. We believe that modular programs as introduced here provide us with a suitable model for understanding the meaning of these components.

6 Conservative Extensions

In this section, we study the question of how to formalize common rewriting techniques used in answer set programming, such as projection, and argue their correctness.

For an interpretation I over signature σ and a function symbol (or, predicate symbol) t from σ by t^I we denote a function (or, relation) assigned to t by I. Let σ and Σ be signatures so that $\sigma \subset \Sigma$. For interpretation I over Σ, by $I_{|\sigma}$ we denote the interpretation over σ such that for every function or predicate symbol t in σ, $t^I = t^{I_{|\sigma}}$. Let P and P' be modular logic programs such that the set of all predicates occurring in P is a subset of the set of all predicates in P' and both programs share the same function symbols. We say that P' *is a conservative extension of* P if $M \mapsto M_{|\sigma(P)}$ is a 1-1 correspondence between the models of P' and the models of P. It turns out that we can replace def-modules in a modular program with their conservative extensions and are guaranteed to obtain a conservative extension of the original modular program. Thus, conservative extensions of def-modules allow us to establish something similar to strong equivalence accounting for the possibility of different signatures.

For example, consider the choice rule $\{\mathtt{p}\}$, a shorthand for the rule $p \leftarrow \mathit{not}\ \mathit{not}\ p$. In some answer set programming dialects double negation is not allowed in the body of a rule. It is then common to simulate a choice rule as above by introducing an auxiliary atom $\hat{p}$ and using the rules $p \leftarrow \neg\hat{p}$ and $\hat{p} \leftarrow \neg p$. It is easy to check that $\mathrm{SM}_{p,\hat{p}}[(\neg\hat{p} \to p) \wedge (\neg p \to \hat{p})]$ is a conservative extension of $\mathrm{SM}_p[\neg\neg p \to p]$. It follows that we can replace the latter with the former within the context of any modular program not containing the predicate symbol $\hat{p}$, and get a conservative extension of the original program.

Similarly, replacing def-module (16) in the P_{hc} by

$$\begin{array}{l}\mathrm{SM}_{v1,v2}[\ \forall xy((\neg r(x,y) \wedge v1(x) \wedge v2(y)) \to \bot) \wedge \\ \qquad \forall xz(edge(x,z) \to v1(x)) \wedge \\ \qquad \forall z'y(edge(z',y) \to v2(y))]\end{array} \tag{18}$$

results in a conservative extension of the original program. This follows from a more general fact about the projection rewriting stated below.

Let R be a rule (17) occurring in a traditional logic program F, and let $\mathbf{x}$ be a non-empty tuple of variables occurring only in the body of R. By $\alpha(\mathbf{x},\mathbf{y})$ we denote the conjunction of all conjunctive terms in the body of R that contain at least one variable from $\mathbf{x}$, where $\mathbf{y}$ denotes all the variables occurring in these conjunctive terms but not occurring in $\mathbf{x}$. By β we denote the set of all conjunctive terms in the body of R that do not contain any variables in $\mathbf{x}$. By γ we denote the head of R. Let t be a predicate symbol that does not occur in F. Then the *result of projecting variables* $\mathbf{x}$ *out of* R *using predicate symbol* t is the conjunction

$$\widetilde{\forall}((t(\mathbf{y}) \wedge \beta) \to \gamma) \wedge \forall\mathbf{xy}\,(\alpha(\mathbf{x},\mathbf{y}) \to t(\mathbf{y}))\,.$$

We can project variables out of a traditional logic program by successively projecting variables out of rules. For example, first projecting z out of the traditional logic program in (16) and then projecting z' out of the first rule of the resulting program yields program (18).

Theorem *Let $SM_{p_1,\ldots,p_k}[F]$ be a def-module and R be a rule in F. Let $\mathbf{x}$ denote a non-empty tuple of variables occurring in the body of R, but not in the head. If G is constructed from F by replacing R in F with the result of projecting variables $\mathbf{x}$ out of R using a predicate symbol p_{k+1} that is not in the signature of F, then $SM_{p_1,\ldots,p_{k+1}}[G]$ is a conservative extension of $SM_{p_1,\ldots,p_k}[F]$.*

This theorem can be restated in terms of traditional logic programs using the fact that any traditional program can be viewed as a def-module.

7 Conclusion

In this paper, we introduced first-order modular logic programs as a way of viewing logic programs as consisting of many independent, meaningful modules. We also defined conservative extensions, which like strong equivalence for traditional programs, can be useful for reasoning about traditional programs and modular programs. We showed how these concepts justify the common projection rewriting.

Acknowledgments

Many thanks to Joshua Irvin, Vladimir Lifschitz, and Miroslaw Truszczynski for useful discussions regarding ideas in this paper. Thanks as well to the anonymous referees for helpful comments. Amelia Harrison was partially supported by the National Science Foundation under Grant IIS-1422455. Yuliya Lierler was partially supported by University Committee on Research and Creative Activity from the University of Nebraska at Omaha in Summer 2016.

References

[Denecker *et al.*, 2012] Marc Denecker, Yuliya Lierler, Miroslaw Truszczynski, and Joost Vennekens. A tarskian informal semantics for answer set programming. 2012.

[Faber *et al.*, 1999] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. pages 135–139, 1999.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1–2):45–74, 2005.

[Ferraris *et al.*, 2009] Paolo Ferraris, Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 797–803, 2009.

[Ferraris *et al.*, 2011] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

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

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[Harrison and Lierler, 2016] Amelia Harrison and Yuliya Lierler. First-order modular logic programs and their conservative extensions. *Theory and Practice of Logic Programming*, 16(5-6):755–770, 2016.

[Leake, 2016] David Leake, editor. *AI Magazine*, volume 37(3). 2016. Special Issue on Answer Set Programming.

[Lierler and Truszczyński, 2013] Yuliya Lierler and Miroslaw Truszczyński. Modular answer set solving. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, 2013.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 2002] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Oikarinen and Janhunen, 2008] Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for Smodels programs. *Theory and Practice of Logic Programming*, 5–6:717–761, 2008.

[Wang *et al.*, 2014] Yisong Wang, Yan Zhang, Yi Zhou, and Mingyi Zhang. Knowledge forgetting in answer set programming. *Journal of Artificial Intelligence Research*, 50(1):31–70, 2014.