TLP **16** (5–6): 755–770, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000430

First-order modular logic programs and their conservative extensions

AMELIA HARRISON
University of Texas at Austin
(*e-mail:* ameliaj@cs.utexas.edu)

YULIYA LIERLER
University of Nebraska Omaha
(*e-mail:* ylierler@unomaha.edu)

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

Modular logic programs provide a way of viewing logic programs as consisting of many independent, meaningful modules. This paper introduces first-order modular logic programs, which can capture the meaning of many answer set programs. We also introduce conservative extensions of such programs. This concept helps to identify strong relationships between modular programs as well as between traditional programs. We show how the notion of a conservative extension can be used to justify the common projection rewriting.

1 Introduction

Answer set programming (ASP) is a prominent knowledge representation paradigm rooted in logic programming. In ASP, a software developer represents a given computational problem by a program whose answer sets (also called stable models) correspond to solutions. Then, the developer uses an answer set solver to generate stable models for the program. In this paper we show how some logic programs 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

$$r(X,Y) \leftarrow in(X,Y). \tag{1}$$
$$r(X,Y) \leftarrow r(X,Z), r(Z,Y). \tag{2}$$

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.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

We show how we can view these rules as forming a module and use the operator SM introduced by Ferraris *et al.* (2011) to define a semantics that corresponds more accurately to the intuition associated with the rules above. 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. The operator SM has proved to be an effective tool for studying the properties of logic programs with variables. Since such programs are the focus of this paper, we chose the operator SM as a technical tool here.

Modularity is essential for modeling large-scale practical applications. Yet research on modular answer set programming is at an early stage. 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 introduced here 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 lp-modules 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 any program and the answer sets of the resulting program will coincide with those of the original one. Conservative extensions allow us to reason about rewritings even when the rules in question have different signatures. We can justify the common projection rewriting described in Faber *et al.* (1999) using this concept. For example, the rule

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

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"

$$\leftarrow not\ r(X,Y) \wedge vertex1(X) \wedge vertex2(Y).$$
$$vertex1(X) \leftarrow edge(X,Y).$$
$$vertex2(Y) \leftarrow edge(X,Y).$$

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

Furthermore, this replacement is valid in the context of any program, as long as that program does not already contain either of the predicates *vertex1* and *vertex2*. Such rewritings can be justified using conservative extensions. Conservative extensions provide a theoretical justification for rewriting techniques already commonly in use. Projection is one such technique, which often improves the performance of answer set programs. Currently, these performance-enhancing rewritings are done manually. We expect the theory about conservative extensions developed here will 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.

This paper is structured as follows. In Sections 2 and 3 we review traditional programs and the operator SM. In Section 4, we define first-order modular logic programs, and in Section 5 we show how they are related to traditional logic programs. Finally, in Section 6, we introduce conservative extensions and show how they can be used to justify program rewritings.

2 Review: traditional programs

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

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

($0 \leqslant k \leqslant l \leqslant m \leqslant n$), 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 occurring in a program Π is a rule that can be formed by replacing all variables occurring 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 understood as in (Ferraris 2005).

According to (Ferraris and Lifschitz 2005) and (Ferraris 2005), rules of the form (4) are sufficient to capture the meaning of the choice rule construct commonly used in answer set programming. For instance, the choice rule $\{\mathtt{p(X)}\} \leftarrow \mathtt{q(X)}$ is understood as the rule

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

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

In this paper we adopt choice rule notation. Traditional logic programs cover a substantial practical fragment of the input languages used in developing answer set programming applications.

Consider the traditional program consisting of the rule

$$s(X,Z) \leftarrow p(Z), q(X,Y), r(X,Y) \tag{5}$$

and the facts

$$p(2).\ q(1,1).\ q(1,2).\ q(2,2).\ r(1,1).\ r(1,2).\ r(2,1). \tag{6}$$

Grounding this program results in eight instances of (5) and the facts in (6). The only answer set of this program is

$$\{p(2), q(1,1), q(1,2), q(2,2), r(1,1), r(1,2), r(2,1), s(1,2)\}. \tag{7}$$

Consider the *Hamiltonian Cycle* problem on an undirected graph. This problem is often used to introduce answer set programming. 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 that encodes this can be constructed by adding rules (1), (2), and (3) to the following:

$$edge(a,a').\ \ \ldots\ \ edge(c,c'). \tag{8}$$

$$edge(X,Y) \leftarrow edge(Y,X). \tag{9}$$

$$\{in(X,Y)\} \leftarrow edge(X,Y). \tag{10}$$

$$\leftarrow in(X,Y), in(X,Z), Y \neq Z. \tag{11}$$

$$\leftarrow in(X,Z), in(Y,Z), X \neq Y. \tag{12}$$

$$\leftarrow in(X,Y), in(Y,X). \tag{13}$$

Each answer set of the *Hamiltonian Cycle* program above corresponds to a Hamiltonian cycle of the given graph, specified by facts (8), 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 (8) define a graph instance by listing its edges, and rule (9) 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 (10) says that any edge may belong to a Hamiltonian cycle;
- rules (11) and (12) impose the restriction that no two edges in a Hamiltonian cycle may start or end at the same vertex, and rule (13) 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*;

[1] This precludes graphs that include isolated vertices, but such vertices can be safely ignored when computing Hamiltonian cycles.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

- rule (3) imposes the restriction that every vertex in a Hamiltonian cycle must be reachable from every other vertex.

Groups of rules of the *Hamiltonian Cycle* program have clear intuitive meanings as shown above. 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 introduced by Ferraris *et al.* (2011), 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 Review: operator SM

The SM operator introduced by Ferraris *et al.* (2011) gives a definition for the semantics of logic programs with variables different than that described in the previous section. The SM operator bypasses grounding and provides a mechanism for viewing groups of rules in a program as separate units or "modules". Consider rule (5). Intuitively, we attach a meaning to this rule: it expresses that relation s holds for a pair of objects when property p holds of the second object and some object is in relation q and relation r with the first object. A program consisting only of this rule has a single answer set that is empty, which is inadequate to capture these intuitions. Ferraris *et al.* (2011) partition predicate symbols of a program into two groups: "intensional" and "extensional". If the predicate s is the only intensional predicate in rule (5), then the SM operator captures the intuitive meaning of this rule seen as a program.

We now review the operator SM following (Ferraris *et al.* 2011). The symbols $\bot, \wedge, \vee, \rightarrow, \forall$, and $\exists$ are viewed as primitives. The formulas $\neg F$ and $\top$ are abbreviations for $F \rightarrow \bot$ and $\bot \rightarrow \bot$, respectively. If p and q are predicate symbols of arity n then $p \leqslant q$ is an abbreviation for the formula $\forall \mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x}))$, where $\mathbf{x}$ is a tuple of variables of length n. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ of predicate symbols then $\mathbf{p} \leqslant \mathbf{q}$ is an abbreviation for the conjunction

$$(p_1 \leqslant q_1) \wedge \cdots \wedge (p_n \leqslant q_n),$$

and $\mathbf{p} < \mathbf{q}$ is an abbreviation for $(\mathbf{p} \leqslant \mathbf{q}) \wedge \neg(\mathbf{q} \leqslant \mathbf{p})$. We apply the same notation to tuples of predicate variables in second-order logic formulas. If $\mathbf{p}$ is a tuple of predicate symbols $p_1, \ldots, p_n$ (not including equality), and F is a first-order sentence then $\mathrm{SM}_{\mathbf{p}}[F]$ (called the *stable model operator with intensional predicates* $\mathbf{p}$) denotes the second-order sentence

$$F \wedge \neg \exists \mathbf{u}(\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u}),$$

where $\mathbf{u}$ is a tuple of distinct predicate variables $u_1, \ldots, u_n$, and $F^*(\mathbf{u})$ is defined recursively:

- $p_i(\mathbf{t})^*$ is $u_i(\mathbf{t})$ for any tuple $\mathbf{t}$ of terms;
- F^* is F for any atomic formula F that does not contain members of $\mathbf{p}$;

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

- $(F \wedge G)^*$ is $F^* \wedge G^*$;
- $(F \vee G)^*$ is $F^* \vee G^*$;
- $(F \rightarrow G)^*$ is $(F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
- $(\forall x F)^*$ is $\forall x F^*$;
- $(\exists x F)^*$ is $\exists x F^*$.

Note that if $\mathbf{p}$ is the empty tuple then $\mathrm{SM}_{\mathbf{p}}[F]$ is equivalent to F. For intuitions regarding the definition of the SM operator we direct the reader to (Ferraris *et al.* 2011, Sections 2.3, 2.4).

A *signature* is a set of function and predicate symbols. A function symbol of arity 0 is an *object constant*. 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 constructed from I so that for every function or predicate symbol t in σ, $t^I = t^{I_{|\sigma}}$.

By $\sigma(F)$ we denote the set of all function and predicate symbols occurring in formula F (not including equality). We will call this the *signature of* F. An interpretation I over $\sigma(F)$ is a $\mathbf{p}$-stable model of F if it satisfies $\mathrm{SM}_{\mathbf{p}}[F]$, where $\mathbf{p}$ is a tuple of predicates from $\sigma(F)$. We will sometimes refer to $\mathbf{p}$-stable models where $\mathbf{p}$ denotes a *s*et rather than a tuple of predicates. Since the cardinality of $\mathbf{p}$ will always be finite, the meaning should be clear. It is easy to see that any $\mathbf{p}$-stable model of F is also a model of F. Similarly, it is clear that for any interpretation I, if $I_{|\sigma(F)}$ is a $\mathbf{p}$-stable model of F then I satisfies $\mathrm{SM}_{\mathbf{p}}[F]$. We may refer to such an interpretation as a $\mathbf{p}$-stable model as well.

From this point on, we view logic program rules as alternative notation for particular types of first-order sentences. For example, rule (5) is seen as an abbreviation for the first-order sentence

$$\forall xyz((p(z) \wedge q(x,y) \wedge r(x,y)) \rightarrow s(x,z)). \tag{14}$$

Similarly, we understand the *Hamiltonian Cycle* program presented in Section 2 as an abbreviation for the conjunction of the following formulas

$$\begin{array}{l} edge(a,a') \wedge \ \ldots \ \wedge edge(c,c') \\ \forall xy(edge(y,x) \rightarrow edge(x,y)) \\ \forall xy((\neg\neg in(x,y) \wedge edge(x,y)) \rightarrow in(x,y)) \\ \forall xyz((in(x,y) \wedge in(x,z) \wedge \neg(y=z)) \rightarrow \bot) \\ \forall xyz((in(x,z) \wedge in(y,z) \wedge \neg(x=y)) \rightarrow \bot) \\ \forall xy((in(x,y) \wedge in(y,x)) \rightarrow \bot) \\ \forall xy(in(x,y) \rightarrow r(x,y)) \\ \forall xyz((r(x,z) \wedge r(z,y)) \rightarrow r(x,y)) \\ \forall xyzz'((\neg r(x,y) \wedge edge(x,z) \wedge edge(z',y)) \rightarrow \bot) \end{array} \tag{15}$$

where $a, a', \ldots c, c'$ are object constants and x, y, z, z' are variables.[2]

[2] In logic programming it is customary to use uppercase letters to denote variables. In the literature on logic it is the specific letter used that indicates whether a symbol is an object constant or a variable

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

Let S denote sentence (14). We now illustrate the definition of **p**-stable models. If s is the only intensional predicate occurring in S then $S^*(s)$ is

$$\forall xyz(((p(z) \wedge q(x,y) \wedge r(x,y)) \rightarrow u(x,z)) \wedge ((p(z) \wedge q(x,y) \wedge r(x,y)) \rightarrow s(x,z)))$$

and $\mathrm{SM}_s[S]$ is

$$S \wedge \neg\exists u((\forall xz(u(x,z) \rightarrow s(x,z)) \wedge \neg\forall xz(s(x,z) \rightarrow u(x,z))) \wedge S^*(s)$$

This second-order sentence is equivalent to the first-order sentence

$$\forall xz(s(x,z) \leftrightarrow (p(z) \wedge \exists y(q(x,y) \wedge r(x,y)))),$$

which reflects the intuitive meaning of the rule (5) seen as a program.

By $\pi(F)$ we denote the set of all predicate symbols (excluding equality) occurring in F. The following theorem is slight generalization of Theorem 1 from (Ferraris *et al.* 2011) as we consider quantifier-free formulas that may contain equality.

Theorem 1
Let Π be a traditional logic program. If $\sigma(\Pi)$ contains at least one object constant then for any Herbrand interpretation X of $\sigma(\Pi)$ the following conditions are equivalent

- X is an answer set of Π;
- X is a $\pi(\Pi)$-stable model of Π.

This theorem illustrates that the set of Herbrand $edge, r, in$-stable models of program (15) coincide with the set of its answer sets.

4 Modular logic programs

In this section, we introduce first-order modular logic programs, which are similar to the propositional modular logic programs introduced in (Lierler and Truszczyński 2013). In a nutshell, 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. The semantics of a modular program is computed by finding the "intersection" of the interpretations that are models of its components. We call any formula of the form $\mathrm{SM}_{\mathbf{p}}[F]$, where **p** is a tuple of predicate symbols and F is traditional logic program viewed as a first-order formula, a *defining module (of* **p** *in* F*)* or a *def-module*. A *first-order modular logic program* (or, *modular program*) P is a finite set of def-modules

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

Let P be a modular program. By $\sigma(P)$ we denote the set

$$\bigcup_{\mathrm{SM}_{\mathbf{p}}[F] \in P} \sigma(F),$$

(with letters drawn from the beginning of the alphabet typically used for the former and letters from the end of the alphabet for the latter). We utilize both of these traditions depending on the context.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

called the *signature of* P. We say that an interpretation I over the signature $\sigma(P)$ is a *stable model* of modular program P if for every def-module $\mathrm{SM}_{\mathbf{p}}[F]$ in P, $I_{|\sigma(F)}$ is a $\mathbf{p}$-stable model of F.

Let P, Q, and R stand for formulas

$$p(2), \tag{16}$$

$$q(1,1) \wedge q(1,2) \wedge q(2,2), \text{ and} \tag{17}$$

$$r(1,1) \wedge r(1,2) \wedge r(2,1), \tag{18}$$

respectively. Consider a modular program consisting of four def-modules

$$\{\mathrm{SM}_p[P], \mathrm{SM}_q[Q], \mathrm{SM}_r[R], \mathrm{SM}_s[S]\}, \tag{19}$$

where S is defined as in the previous section. The Herbrand interpretation (7) is a stable model of this modular program.

The stable models of modular program (19) coincide with the p,q,r,s-stable models of

$$\mathrm{SM}_{p,q,r,s}[P \wedge Q \wedge R \wedge S]. \tag{20}$$

Recall that $P \wedge Q \wedge R \wedge S$ can be viewed as the logic program consisting of the facts (6) and the rule (5). By Theorem 1, the Herbrand p,q,r,s-stable models of (20) coincide with the answer sets of the logic program composed of rules in (5) and (6). These facts hint at the close relationship between modular logic programs and traditional logic programs as written by answer set programming practitioners. In the following, we formalize the relationship between modular logic programs and traditional logic programs. This formalization is rooted in prior work on splitting logic programs from Ferraris *et al.* (2009).

5 Relating modular programs and traditional programs

As mentioned earlier, we view a traditional logic program as an abbreviation for a first-order sentence formed as a conjunction of formulas of the form

$$\widetilde{\forall}(a_{k+1} \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n \rightarrow a_1 \vee \cdots \vee a_k), \tag{21}$$

which corresponds to rule (4). The symbol $\widetilde{\forall}$ denotes universal closure. We call the disjunction in the consequent of a rule (21) 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*. It is sometimes convenient to abbreviate the body of a rule with the letter B and represent rule (21) as

$$\widetilde{\forall}(B \rightarrow a_1 \vee \cdots \vee a_k). \tag{22}$$

Let P denote a modular program. By $\pi(P)$ we denote the set

$$\bigcup_{\mathrm{SM}_{\mathbf{p}}[F] \in P} \pi(F),$$

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

called the *predicate signature* of P. Similarly, by $\iota(P)$ we denote the set

$$\bigcup_{\mathrm{SM}_{\mathbf{p}}[F]\in P} \mathbf{p}$$

called the *intensional signature* of P. By $\mathscr{F}(P)$ we denote the formula

$$\bigwedge_{\mathrm{SM}_{\mathbf{p}}[F]\in P} F.$$

A modular program is called *simple* when 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, modular program (19) is simple. We note that this restriction is, in a sense, inessential. Indeed, consider a def-module $\mathrm{SM}_{\mathbf{p}}[F]$ that is not simple. There is a straightforward syntactic transformation that can be performed on each rule in F, resulting in a formula F' such that $\mathrm{SM}_{\mathbf{p}}[F]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F']$. Let R be a rule of the form (22) and $\mathbf{p}$ be a tuple of predicate symbols. By $shift_{\mathbf{p}}(R)$ we denote the universal closure of the following formula

$$B \ \wedge \bigwedge_{\substack{\pi(a_i)\notin\mathbf{p},\\ 1\leqslant i\leqslant k}} \neg a_i \ \rightarrow \bigvee_{\substack{\pi(a_i)\in\mathbf{p},\\ 1\leqslant i\leqslant k}} a_i.$$

In other words, any atomic formula in the head of a rule whose predicate symbol is not in $\mathbf{p}$ is moved to the body of the rule and preceded by negation. For a traditional logic program F, $shift_{\mathbf{p}}(F)$ is the conjunction of formulas obtained by applying $shift_{\mathbf{p}}$ to each rule in F. Theorem 5 from Ferraris *et al.* (2011) shows that if the equivalence between any two first-order formulas can be derived intuitionistically from the law of excluded middle formulas for all extensional predicates occurring in those formulas, then they have the same stable models. The following observation is a consequence of that theorem.

Observation 1
For a traditional logic program F, def-modules $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}}[shift_{\mathbf{p}}(F)]$ are equivalent.

For any simple modular program P, the *dependency graph* of P, denoted $DG[P]$, is a directed graph that

- has all members of the intensional signature $\iota(P)$ as its vertices, and
- has an edge from p to q if there is a def-module $\mathrm{SM}_{\mathbf{p}}[F] \in P$ containing a rule with p occurring in the head and q occurring in the positive part of the body.

For instance, the dependency graph of simple modular program (19) consists of four vertices p, q, r, s and edges from s to p, from s to q, and from s to r. It is easy to see that this graph has four strongly connected components, each consisting of a single vertex.

We call a simple modular program P *coherent* if

(i) for every pair of distinct def-modules $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}'}[F']$ in P, tuples $\mathbf{p} \cap \mathbf{p}' = \emptyset$, and

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

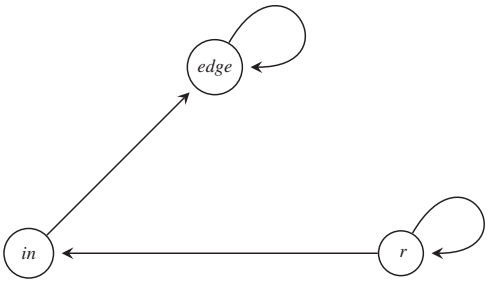


Fig. 1. Dependency graph for P_{hc}.

(ii) for every strongly connected component **c** in the dependency graph of P there is a def-module $\mathrm{SM}_{\mathbf{p}}[F] \in P$ such that **p** contains all vertices in **c**.

It is easy to see, for example, that modular program (19) is coherent.

The following theorem is similar to the Splitting Theorem from Ferraris *et al.* (2009). That theorem says that under certain conditions the stable models of a conjunction of two formulas coincide with those interpretations that are stable models of both individual formulas with respect to different sets of intensional predicates. The theorem below presents a similar result for coherent programs and is more general in the sense that it applies to any finite number of def-modules, rather than just two.

Theorem 2 (*Splitting Theorem*)
If P is a coherent modular program, then an interpretation I is an $\iota(P)$-stable model of $\mathscr{F}(P)$ iff it is a stable model of P.

Since modular program (19) is coherent, it is not by chance that its stable models coincide with the Herbrand p, q, r, s-stable models of (20). Rather, this is an instance of a general fact. The following theorem, which follows from Theorems 1 and 2, describes the relationship between modular programs and traditional logic programs.

Theorem 3
For a coherent modular program P such that $\sigma(P)$ contains at least one object constant and $\pi(P) = \iota(P)$ and any Herbrand interpretation X of $\sigma(P)$ the following conditions are equivalent

- X is an answer set of $\mathscr{F}(P)$;
- X is a stable model of P.

A modular program $\{\mathrm{SM}_p[q(1) \to p(1)], \mathrm{SM}_q[p(1) \to q(1)]\}$ is an example of a non-coherent program. Consider the Herbrand interpretation $\{p(1), q(1)\}$. This interpretation is a stable model of this program. Yet, it is not an answer set of the traditional program consisting of the two rules $q(1) \to p(1).$ and $p(1) \to q(1)$. The only answer set of this traditional program is the empty set.

We now illustrate how modular programs capture the encoding (15) of the *Hamiltonian Cycle* so that each of its modules carries its intuitive meaning. The

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

Hamiltonian Cycle modular program presented below consists of five def-modules:

$$\mathrm{SM}_{edge}[edge(a,a') \wedge \ \ldots \ \wedge edge(c,c') \wedge \forall xy(edge(y,x) \rightarrow edge(x,y)] \quad (23)$$

$$\mathrm{SM}_{in}[\forall xy((\neg\neg in(x,y) \wedge edge(x,y)) \rightarrow in(x,y))] \quad (24)$$

$$\begin{aligned}\mathrm{SM}[&\forall xyz((in(x,y) \wedge in(x,z) \wedge \neg(y=z)) \rightarrow \bot) \wedge \\ &\forall xyz((in(x,z) \wedge in(y,z) \wedge \neg(x=y)) \rightarrow \bot) \wedge \\ &\forall xy((in(x,y) \wedge in(y,x)) \rightarrow \bot)]\end{aligned} \quad (25)$$

$$\begin{aligned}\mathrm{SM}_r[&\forall xy(in(x,y) \rightarrow r(x,y)) \wedge \\ &\forall xyz((r(x,z) \wedge r(z,y)) \rightarrow r(x,y))]\end{aligned} \quad (26)$$

$$\mathrm{SM}[\forall xyzz'((\neg r(x,y) \wedge edge(x,z) \wedge edge(z',y)) \rightarrow \bot)] \quad (27)$$

We call this modular program P_{hc}. The def-modules shown above correspond to the intuitive groupings of rules of the *Hamiltonian Cycle* encoding discussed in Section 2.

- An *edge*-stable model of def-module (23) is any interpretation I over $\sigma(P_{hc})$ such that the extension[3] of the *edge* predicate in I corresponds to the symmetric closure of the facts in (8).
- An *in*-stable model of def-module (24) 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.
- An $\emptyset$-stable model of def-module (25) is any interpretation I over $\sigma(P_{hc})$ that satisfies the conjunction in (25).
- An r-stable model of def-module (26) is any interpretation I over $\sigma(P_{hc})$, where relation r is the transitive closure of relation *in*.
- An $\emptyset$-stable model of def-module (27) is any interpretation I over $\sigma(P_{hc})$ that satisfies the conjunction in (27).

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

The dependency graph of P_{hc} is shown in Figure 1. The strongly connected components of this graph each consist of a single vertex. It is easy to verify that the *Hamiltonian Cycle* program P_{hc} is coherent. By Theorem 3, it follows that the Herbrand models of *Hamiltonian Cycle* coincide with the answer sets of (15) 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 the program. We believe that modular programs as introduced here provide us with a suitable model for understanding the meaning of components of the program.

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.

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

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

Let F and G be second-order formulas such that $\pi(F) \subseteq \pi(G)$ and both formulas share the same function symbols. We say that G *is a conservative extension of* F if

- $\{M|M \text{ is a model of } F\} = \{M_{|\sigma(F)}|M \text{ is a model of } G\}$, and
- there are no distinct models M and M' of G such that $M_{|\sigma(F)} = M'_{|\sigma(F)}$.

The definition of a conservative extension for second-order formulas gives us a definition of a conservative extension for def-modules, as they are second-order formulas. It is interesting to note that the first condition of the definition holds if and only if F has the same models as the second-order formula $\exists p_1 \dots p_n\ G$, where $\{p_1, \dots p_n\} = \pi(G) \setminus \pi(F)$. The second condition adds another intuitive restriction. For example, consider the broadly used Tseitin transformation. In this transformation, an arbitrary propositional formula is converted into conjunctive normal form by (i) augmenting the original formula with "explicit definitions" and (ii) applying equivalent transformations. The resulting formula is of a new signature, but both of the conditions of the definition hold between the original formula and the result of Tseitin transformation. We can state the definition of a conservative extension more concisely by saying that G is a conservative extension of F if $M \mapsto M_{|\sigma(F)}$ is a 1-1 correspondence between the models of G and the models of F.

In view of Theorem 1, the definition of a conservative extension can be applied to traditional logic programs: If Π_1 and Π_2 are traditional programs such that $\pi(\Pi_1) \subseteq \pi(\Pi_2)$ and both programs share the same function symbols, then Π_2 is a conservative extension of Π_1 if $M \mapsto M_{|\sigma(\Pi_1)}$ is a 1-1 correspondence between the answer sets of Π_2 and the answer sets of Π_1.

As an illustration of a conservative extension, consider the following formulas:

$$\forall xz(s(x,z) \leftrightarrow (p(z) \wedge \exists y(q(x,y) \wedge r(x,y)))) \tag{28}$$

$$\forall xz(s(x,z) \leftrightarrow (p(z) \wedge t(x))) \wedge \forall v(t(v) \leftrightarrow \exists w(q(v,w) \wedge r(v,w))). \tag{29}$$

It is easy to verify that the models of formulas (28) and (29) are in 1-1 correspondence so that

$$\{M \mid M \text{ is a model of formula (28)}\} = \{M_{|\{s,p,q,r\}} \mid M \text{ is a model of formula (29)}\}.$$

In fact, formula (29) is obtained from formula (28) by introducing an explicit definition using predicate symbol t. Recall the notion of an explicit definition: to extend a formula F by *an explicit definition using predicate symbol* t means to add to the signature of F a new predicate symbol t of arity n, and to add a conjunctive term to F of the form

$$\forall x_1 \dots x_n(t(x_1, \dots, x_n) \leftrightarrow G), \tag{30}$$

where $x_1 \dots x_n$ are distinct variables and G is a formula over the signature of F. The result of adding such a definition is a formula that is a conservative extension of F. Furthermore, constructing a formula from F by

- substituting every occurrence of subformula G in F with $t(x_1, \dots, x_n)$ (modulo proper substitution of terms) and
- extending this formula with a conjunctive term (30)

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

results in a conservative extension as well. This is the procedure that is used to obtain formula (29) from (28).

Recall that S denotes sentence (14). By S' we denote the sentence

$$\forall xz((t(x) \wedge p(z)) \rightarrow s(x,z)) \wedge \forall xy((q(x,y) \wedge r(x,y)) \rightarrow t(x)). \tag{31}$$

It can be verified that (28) is equivalent to $\mathrm{SM}_s[S]$, and that (29) is equivalent to $\mathrm{SM}_{s,t}[S']$.

The next proposition provides a general method for showing that one def-module is a conservative extension of another.

Proposition 1
For any def-modules $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]$ such that $\pi(F) \subseteq \pi(G)$, both formulas share the same function symbols, and $\mathbf{p}'$ is a subset of predicate symbols $\pi(G) \setminus \pi(F)$, if $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]$ are equivalent to first-order formulas F' and G' respectively, and G' is a conservative extension of F' then $\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]$ is a conservative extension of $\mathrm{SM}_{\mathbf{p}}[F]$.

An analogous property holds for traditional programs:

Proposition 2
For any traditional programs Π_1 and Π_2 such that $\pi(\Pi_1) \subseteq \pi(\Pi_2)$ and both programs share the same function symbols and contain at least one object constant, if $\mathrm{SM}_{\pi(\Pi_1)}[\Pi_1]$ and $\mathrm{SM}_{\pi(\Pi_2)}[\Pi_2]$ are equivalent to first-order formulas Π_1' and Π_2' respectively, and Π_2' is a conservative extension of Π_1', then traditional program Π_2 is a conservative extension of Π_1.

We now lift the definition of a conservative extension to the case of modular programs. We say that modular program 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.

Let us recall the notion of strong equivalence (Lifschitz *et al.* 2001). Traditional programs Π_1 and Π_2 are *strongly equivalent* if for every traditional program Π, programs $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets. Strong equivalence can be used to argue the correctness of some program rewritings used by answer set programming practitioners. However, the projection rewriting technique, exemplified by replacing rule (14) with rules (31), cannot be justified using the notion of strong equivalence. This rewriting technique is commonly used to improve the performance of answer set programs (Buddenhagen and Lierler 2015). Strong equivalence is inappropriate for justifying this rewriting for a simple reason: the signature of the original program is changed. In what follows we attempt to "adjust" the notion of strong equivalence to the context of modular programs so that we may formally reason about the correctness of projection and other similar rewriting techniques. We then translate these notions to the realm of traditional programs. We start by restating the definition of strong equivalence given in (Ferraris *et al.* 2011) and recalling some of its properties.

First-order formulas F and G are *strongly equivalent* if for any formula H, any occurrence of F in H, and any list $\mathbf{p}$ of distinct predicate constants, $\mathrm{SM}_{\mathbf{p}}[H]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[H']$, where H' is obtained from H by replacing F by G.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

In (Lifschitz *et al.* 2007) the authors show that first-order formulas F and G are strongly equivalent if they are intuitionistically equivalent.

The following theorem, which is easy to verify, illustrates that classical equivalence between second-order formulas is sufficient to capture the condition of "strong equivalence" for modular programs. In other words, replacing a def-module by an equivalent def-module with the same intensional predicates does not change the semantics of a modular program.

Theorem 4
Let $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}}[G]$ be def-modules. Then the following two conditions are equivalent:

(i) for any modular program P, the programs $P \cup \{\mathrm{SM}_{\mathbf{p}}[F]\}$ and $P \cup \{\mathrm{SM}_{\mathbf{p}}[G]\}$ have the same stable models;
(ii) $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}}[G]$ are equivalent.

In (Ferraris *et al.* 2011, Section 5.2), the authors observe that if first-order formulas F and G are strongly equivalent then def-modules of the form $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathrm{SM}_{\mathbf{p}}[G]$ are equivalent. Consequently, to show that replacing $\mathrm{SM}_{\mathbf{p}}[F]$ by $\mathrm{SM}_{\mathbf{p}}[G]$ in any modular program results in a program with the same models it is sufficient to prove that F and G are intuitionistically equivalent.

The following theorem lifts Theorem 4 to conservative extensions.

Theorem 5
Let $\mathrm{SM}_{\mathbf{p}}[F]$, $\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]$ be def-modules such that $\pi(F) \subseteq \pi(G)$, both formulas share the same function symbols, and $\mathbf{p}'$ is $\pi(G) \setminus \pi(F)$. Then the following two conditions are equivalent:

(i) for any modular program P such that $\pi(P)$ contains no elements from $\mathbf{p}'$, modular program $P \cup \{\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]\}$ is a conservative extension of $P \cup \{\mathrm{SM}_{\mathbf{p}}[F]\}$;
(ii) $\mathrm{SM}_{\mathbf{p}\cup\mathbf{p}'}[G]$ is a conservative extension of $\mathrm{SM}_{\mathbf{p}}[F]$.

Theorem 5 tells us 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 for modular programs with possibly different signatures.

For example, consider the choice rule $\{\mathtt{p}\}$, a shorthand for the rule $p \leftarrow \mathit{not\ not}\ p$. 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 $\neg\hat{p} \rightarrow p$ and $\neg p \rightarrow \hat{p}$. It is easy to check that $\mathrm{SM}_{p,\hat{p}}[(\neg\hat{p} \rightarrow p) \wedge (\neg p \rightarrow \hat{p})]$ is a conservative extension of $\mathrm{SM}_{p}[p \vee \neg p]$. By Theorem 5, 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.

Proposition 1 and Theorem 5 equip us with a method for establishing the correctness of program rewritings. For instance, the fact that formulas (28) and (29) are equivalent to def-modules $\mathrm{SM}_{s}[S]$ and $\mathrm{SM}_{s,t}[S']$ respectively, translates into the

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

assertion that the latter is a conservative extension of the former. Thus, replacing def-module $\mathrm{SM}_s[S]$ in modular program (19) with $\mathrm{SM}_{s,t}[S']$ results in a modular program that is a conservative extension of (19). Similarly, replacing def-module (27) in the *Hamiltonian Cycle* modular program presented in Section 5 by the def-module

$$\begin{array}{l}\mathrm{SM}_{vertex1,vertex2}[\forall xy((\neg r(x,y) \wedge vertex1(x) \wedge vertex2(y) \rightarrow \bot) \ \wedge \\ \forall xz(edge(x,z) \rightarrow vertex1(x)) \ \wedge \\ \forall z'y(edge(z',y) \rightarrow vertex2(y))]\end{array} \tag{32}$$

results in a conservative extension of the original program. This is an instance of projection rewriting. We now introduce some notation used to state a result about the general case of projection that will support our claim that (32) is a conservative extension of (27).

Let R be a rule (21) 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 occurring 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 of the following two rules

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

For example, the result of projecting y out of (14) using predicate symbol t is (31). 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 (27) and then projecting z' out of the first rule of the resulting program yields the traditional logic program in (32).

Theorem 6

Let $\mathrm{SM}_{p_1,\ldots,p_k}[F]$ 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 $\mathrm{SM}_{p_1,\ldots,p_{k+1}}[G]$ is a conservative extension of $\mathrm{SM}_{p_1,\ldots,p_k}[F]$.

We now restate Theorem 6 in terms of traditional logic programs using the link between def-modules and traditional programs established in Theorem 1.

Corollary 1

Let Π be a traditional logic program containing at least one object constant and R be a rule in Π. Let $\mathbf{x}$ denote a non-empty tuple of variables occurring in the body of R, but not in the head. If Π' is constructed from Π by replacing R in Π with the result of projecting variables $\mathbf{x}$ out of R using a predicate symbol p that does not occur in Π, then Π' is a conservative extension of Π.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press

7 Conclusion

In this paper, we introduced first-order modular logic programs that provide 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 may be used to 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.

References

Buddenhagen, M. and Lierler, Y. (2015). Performance tuning in answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27–30, 2015. Proceedings*, 186–198.

Denecker, M., Lierler, Y., Truszczynski, M. and Vennekens, J. (2012). A Tarskian informal semantics for answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, LIPIcs, 17, 277–289.

Faber, W., Leone, N., Mateis, C. and Pfeifer, G. (1999). Using database optimization techniques for nonmonotonic reasoning, 135–139.

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

Ferraris, P., Lee, J. and Lifschitz, V. (2011). Stable models and circumscription. *Artificial Intelligence*, *175*, 236–263.

Ferraris, P., Lee, J., Lifschitz, V. and Palla, R. (2009). Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 797–803.

Ferraris, P. and Lifschitz, V. (2005). Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 1–2, 45–74.

Lierler, Y. and Truszczyński, M. (2013). Modular answer set solving. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*.

Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence 138*, 39–54.

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

Lifschitz, V., Pearce, D. and Valverde, A. (2007). A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 188–200.

Oikarinen, E. and Janhunen, T. (2008). Achieving compositionality of the stable model semantics for Smodels programs. *Theory and Practice of Logic Programming 5–6*, 717–761.

Wang, Y., Zhang, Y., Zhou, Y. and Zhang, M. (2014). Knowledge forgetting in answer set programming. *Journal of Artificial Intelligence Research 50* 1, 31–70.

https://doi.org/10.1017/S1471068416000430 Published online by Cambridge University Press