A Reductive Semantics for Counting and Choice in Answer Set Programming

Joohyung Lee[1] **Vladimir Lifschitz**[2] **Ravi Palla**[1]

[1] Computer Science and Engineering
School of Computing and Informatics
Arizona State University
Tempe, AZ 85281, USA
{joolee, Ravi.Palla}@asu.edu

[2] Department of Computer Sciences
University of Texas at Austin
1 University Station C0500
Austin, TX 78712, USA
vl@cs.utexas.edu

Abstract

In a recent paper, Ferraris, Lee and Lifschitz conjectured that the concept of a stable model of a first-order formula can be used to treat some answer set programming expressions as abbreviations. We follow up on that suggestion and introduce an answer set programming language that defines the meaning of counting and choice by reducing these constructs to first-order formulas. For the new language, the concept of a safe program is defined, and its semantic role is investigated. We compare the new language with the concept of a disjunctive program with aggregates introduced by Faber, Leone and Pfeifer, and discuss the possibility of implementing a fragment of the language by translating it into the input language of the answer set solver DLV. The language is also compared with cardinality constraint programs defined by Syrjänen.

Introduction

In the stable model semantics (Gelfond and Lifschitz 1988), a logic program with variables is viewed as shorthand for the set of all ground instances of its rules. In the existing proposals on extending this semantics to more general programs, variables are treated, for the most part, in the same way, although the process of grounding often becomes more complicated. For instance, the semantics of disjunctive programs with aggregates from (Faber *et al.* 2004) divides this operation into two parts—a "global substitution" and a "local substitution."

The definition of a stable model for first-order formulas proposed in (Ferraris *et al.* 2007) and reviewed in the next section is an exception: it does not refer to grounding. Instead, it employs a syntactic transformation of formulas with variables that is similar to circumscription (McCarthy 1980). As part of motivation for their work, the authors talk about the possibility of treating choice rules and cardinality constraints with variables[1] as abbreviations for first-order formulas.[2] For instance, the choice rule

$$\{q(x)\} \leftarrow p(x)$$

("for any element of p decide arbitrarily whether or not to include it in q") can be thought of as an abbreviation for the formula

$$\forall x(p(x) \rightarrow (q(x) \vee \neg q(x))). \quad (1)$$

Alternatively, this choice rule can be treated as shorthand for

$$\forall x((p(x) \wedge \neg\neg q(x)) \rightarrow q(x)). \quad (2)$$

(Since formulas (1) and (2) are logically valid, the models of a formula that includes (1) or (2) as a conjunctive term will not change if we drop that term. But the *stable* models of the formula can be affected by such a transformation.[3] In this sense, (1) and (2) are nontrivial.) As another example, consider the cardinality constraint

$$1\,\{\neg q(x) : p(x)\}$$

("there exists at least one element of p that doesn't belong to q"). In the spirit of the approach outlined in (Ferraris *et al.* 2007), this constraint in the body of a rule can be identified with the formula

$$\exists x(\neg q(x) \wedge p(x)).$$

We follow up on that suggestion and introduce here an answer set programming language that defines the meaning of counting and choice by reducing these constructs to first-order formulas. The language is called RASPL-1, for Reductive Answer Set Programming Language, version 1. (In future versions, this language will be extended by aggregates other than counting.) We discuss *safety*, a condition

[1] These two constructs play an important part in the input language of LPARSE (`http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`)—the front-end of SMODELS and several other answer set solvers.

[2] This idea generalizes the approach to propositional choice rules and aggregates investigated in (Ferraris and Lifschitz 2005) and (Ferraris 2005, Section 4).

[3] The class of "strongly equivalent" transformations, which do not change the stable models of a first-order formula, is studied in (Lifschitz *et al.* 2007). It includes all transformations that are sanctioned by intuitionistic logic, and many others. For instance, formulas (1) and (2) are strongly equivalent to each other, although they are not equivalent in intuitionistic logic. Each step involved in the standard process of converting a formula to prenex form is a strongly equivalent transformation (Lee and Palla 2007), although some of these steps are not acceptable intuitionistically.

that answer set solvers usually impose on their input (Leone *et al.* 2006, Section 2.1). Our goal is to extend that concept to RASPL-1 (and to first-order formulas in general, to pave the way for future work on RASPL-2) and to investigate its semantic role. We compare RASPL-1 with the proposal from (Faber *et al.* 2004) mentioned above, and use the result of this analysis to discuss the possibility of implementing a fragment of RASPL-1 by translating it into the input language of the answer set solver DLV. Finally, RASPL-1 is related to the semantics of cardinality constraint programs from (Syrjänen 2004).

Answer Sets of a First-Order Formula

The definition of the "stable model operator" SM in (Ferraris *et al.* 2007) uses notation that was introduced in (Lifschitz 1985) for the purpose of defining parallel circumscription, and we begin with a review of that notation. Let $\mathbf{p}$ be a list of distinct predicate constants $p_1, \dots, p_n$, and let $\mathbf{u}$ be a list of distinct predicate variables $u_1, \dots, u_n$ of the same length as $\mathbf{p}$. By $\mathbf{u} = \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \leftrightarrow p_i(\mathbf{x}))$, where $\mathbf{x}$ is a list of distinct object variables of the same arity as the length of p_i, for all $i = 1, \dots n$. By $\mathbf{u} \leq \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \rightarrow p_i(\mathbf{x}))$ for all $i = 1, \dots n$, and $\mathbf{u} < \mathbf{p}$ stands for $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{u} = \mathbf{p})$.

For any first-order sentence F, SM$[F]$ stands for the second-order sentence

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

where $\mathbf{p}$ is the list $p_1, \dots, p_n$ of all predicate constants occurring in F, $\mathbf{u}$ is a list $u_1, \dots, u_n$ of distinct predicate variables, and $F^*(\mathbf{u})$ is defined recursively:

- $p_i(t_1, \dots, t_m)^* = u_i(t_1, \dots, t_m)$;
- $(t_1 = t_2)^* = (t_1 = t_2)$;
- $\bot^* = \bot$;
- $(F \wedge G)^* = F^* \wedge G^*$;
- $(F \vee G)^* = F^* \vee G^*$;
- $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
- $(\forall x F)^* = \forall x F^*$;
- $(\exists x F)^* = \exists x F^*$.

(There is no clause for negation here, because we treat $\neg F$ as shorthand for $F \rightarrow \bot$.) According to (Ferraris *et al.* 2007), an interpretation of the signature $\sigma(F)$ consisting of the object, function and predicate constants occurring in F is a *stable model* of F if it satisfies SM$[F]$.[4]

Note that the operator $F \mapsto F^*(\mathbf{u})$ replaces each predicate constant with the corresponding predicate variable, and that it commutes with all propositional connectives except implication and with both quantifiers. Consequently, for any formula F that does not contain implication (and negation), $F^*(\mathbf{u})$ is simply the result of substituting $\mathbf{u}$ for $\mathbf{p}$ in F, so that SM$[F]$ is, for such F, the result of circumscribing all predicate constants in F in parallel.

The terms "stable model" and "answer set" are often used in the literature interchangeably. In the context of this discussion of the use of SM in answer set programming, it is convenient to distinguish between them as follows: by an *answer set* of a first-order sentence F that contains at least one object constant we will understand an Herbrand[5] interpretation of $\sigma(F)$ that satisfies SM$[F]$.

Example 1 If F is

$$p(a) \wedge q(b) \wedge \forall x((p(x) \wedge \neg q(x)) \rightarrow r(x)) \tag{3}$$

then SM$[F]$ is equivalent to

$$\forall x(p(x) \leftrightarrow x = a) \wedge \forall x(q(x) \leftrightarrow x = b) \\ \wedge \forall x(r(x) \leftrightarrow (p(x) \wedge \neg q(x)))$$

(see (Ferraris *et al.* 2007), Example 3).[6] Consequently, the only answer set of (3) is

$$\{p(a),\ q(b),\ r(a)\}. \tag{4}$$

Example 2 If F is the conjunction of (1) and

$$p(a) \wedge p(b) \tag{5}$$

then SM$[F]$ is equivalent to

$$\forall x(p(x) \leftrightarrow (x = a \vee x = b)) \wedge \forall x(q(x) \rightarrow (x = a \vee x = b))$$

(see (Ferraris *et al.* 2007), Example 4). Consequently, the answer sets of this conjunction are

$$\{p(a),\ p(b)\},\ \{p(a),\ p(b),\ q(a)\}, \\ \{p(a),\ p(b),\ q(b)\},\ \{p(a),\ p(b),\ q(a),\ q(b)\}. \tag{6}$$

The conjunction of (2) and (5) has the same answer sets.

For any sentences F and G, SM$[F \wedge \neg G]$ is equivalent to SM$[F] \wedge \neg G$. (This is immediate from (Ferraris *et al.* 2007, Proposition 2).) Consequently the answer sets of $F \wedge \neg G$ can be characterized as the answer sets of F that satisfy $\neg G$.

Example 3 As discussed above, the answer sets of the conjunction of (2) and (5) are sets (6). If we append the formula

$$\neg\neg \exists xy(q(x) \wedge q(y) \wedge x \neq y) \tag{7}$$

to that conjunction as an additional term, the resulting formula will have one answer set

$$\{p(a),\ p(b),\ q(a),\ q(b)\} \tag{8}$$

—the only set from list (6) that satisfies (7).

In the next section we will see how Examples 1–3 can be expressed in the syntax of RASPL-1.

[4] The definition of a stable model in that paper is actually more general, because it allows the underlying signature to be a superset of $\sigma(F)$. If this signature contains predicate constants that do not occur in F then it would be reasonable to require in the definition of a stable model that the interpretations of these predicate constants be identically false. The absence of this condition in (Ferraris *et al.* 2007) is an oversight; without it, the assertion of Proposition 1 from that paper is incorrect.

[5] Recall that an *Herbrand interpretation* of a signature σ (containing at least one object constant) is an interpretation of σ such that its universe is the set of all ground terms of σ, and every ground term represents itself. An Herbrand interpretation can be identified with the set of ground atoms (not containing equality) to which it assigns the value *true*.

[6] This fact can be established, for instance, using the results from (Ferraris *et al.* 2007) that relate SM to completion.

Definition of RASPL-1

Syntax

In RASPL-1, a *term* is an object constant or an object variable (so that there are no function constants of positive arity). An *atom* is an expression of the form $P(t_1, \dots, t_n)$ or $t_1 = t_2$, where P is an n-ary predicate constant and each t_i is a term.

An *aggregate expression* is an expression of the form

$$b\,\{\mathbf{x} : F_1, \dots, F_k\} \tag{9}$$

($k \geq 1$), where b is a positive integer ("the bound"), $\mathbf{x}$ is a list of variables (possibly empty), and each F_i is an atom possibly preceded by *not*. This expression reads: there are at least b values of $\mathbf{x}$ such that $F_1, \dots, F_k$.

A *rule* is an expression of the form

$$A_1\ ;\dots;\ A_l \leftarrow\ E_1, \dots, E_m, \mathit{not}\ E_{m+1}, \dots, \mathit{not}\ E_n \tag{10}$$

($l \geq 0$; $n \geq m \geq 0$), where each A_i is an atom, and each E_i is an aggregate expression. A *program* is a list of rules.

Semantics

The semantics of RASPL-1 is defined by a procedure that turns every aggregate, every rule, and every program into a formula of first-order logic, called its *FOL-representation*. Two notational conventions are used in the definition of this procedure. First, we identify the logical connectives $\wedge$, $\vee$ and $\neg$ with their counterparts used in RASPL-1 programs—the comma, the semicolon, and *not*. This convention allows us to treat the list $F_1, \dots, F_k$ in (9) as a conjunction of literals. Second, for any lists of variables $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$ of the same length, $\mathbf{x} = \mathbf{y}$ stands for $x_1 = y_1 \wedge \dots \wedge x_n = y_n$.

The FOL-representation of an aggregate expression $b\,\{\mathbf{x} : F(\mathbf{x})\}$ is the formula

$$\exists \mathbf{x}^1 \cdots \mathbf{x}^b \left[\bigwedge_{1 \leq i \leq b} F(\mathbf{x}^i) \wedge \bigwedge_{1 \leq i < j \leq b} \neg(\mathbf{x}^i = \mathbf{x}^j) \right] \tag{11}$$

where $\mathbf{x}^1, \dots \mathbf{x}^b$ are lists of new variables of the same length as $\mathbf{x}$.

The FOL-representation of a RASPL-1 rule *Head* $\leftarrow$ *Body* is the universal closure of the implication *Body* $\rightarrow$ *Head* with each aggregate expression in *Body* replaced by its FOL-representation.

The FOL-representation of a RASPL-1 program is the conjunction of the FOL-representations of its rules.

For any RASPL-1 program Π containing at least one object constant, an *answer set* of Π is an answer set of the FOL-representation of Π, as defined in the previous section.

Abbreviations and Examples

If an aggregate expression E_i in (10) has the form $1\{:\ A\}$, where A is an atom (so that the list of variables in front of the semicolon is empty) then we will write it as A. This convention is consistent with the semantics of aggregate expressions defined above, in the sense that the FOL-representation of $1\{:\ A\}$ is A. It allows us to view any traditional disjunctive rule, with atoms and negated atoms in the body, as a rule of RASPL-1.

Example 1 (continued) The FOL-representation of the program

$$\begin{array}{l} p(a) \\ q(b) \\ r(x) \leftarrow p(x), \mathit{not}\ q(x) \end{array}$$

is formula (3). Consequently, the only answer set of this program is (4).

If an aggregate expression E_i in (10) with $i > m$ has the form $1\{:\ \mathit{not}\ A\}$ then we will write it as *not* A. This convention is consistent with the semantics of aggregate expressions as well, and it allows us to include "double negations" *not not* A in the body of a RASPL-1 rule.

An expression of the form

$$\{A\} \leftarrow\ E_1, \dots, E_m, \mathit{not}\ E_{m+1}, \dots, \mathit{not}\ E_n$$

where A is an atom, stands for

$$A \leftarrow\ E_1, \dots, E_m, \mathit{not}\ E_{m+1}, \dots, \mathit{not}\ E_n, \mathit{not\ not}\ A.$$

Example 2 (continued) The FOL-representation of the program

$$\begin{array}{l} p(a) \\ p(b) \\ \{q(x)\} \leftarrow p(x) \end{array}$$

is the conjunction of formulas (5) and (2). Consequently, the answer sets of this program are sets (6).

If E_i in (10) with $i > m$ is

$$b\,\{\mathbf{x} : F(\mathbf{x})\}$$

then the term *not* E_i can be written as

$$\{\mathbf{x} : F(\mathbf{x})\}\, b - 1$$

("there are at most $b - 1$ values of $\mathbf{x}$ such that $F(\mathbf{x})$").

Example 3 (continued) The expression $\{x\ :\ q(x)\}\,1$ is shorthand for *not* $2\,\{x\ :\ q(x)\}$, so that its FOL-representation is $\neg\exists xy(q(x) \wedge q(y) \wedge x \neq y)$. Consequently, the FOL-representation of the program

$$\begin{array}{l} p(a) \\ p(b) \\ \{q(x)\} \leftarrow p(x) \\ \leftarrow \{x : q(x)\}\,1 \end{array}$$

is the conjunction of formulas (5), (2) and (7). The only answer set of this program is (8).

Programming in RASPL-1

The idea of answer set programming is to reduce a given search problem to the problem of finding an answer set, and then use an answer set solver to generate a solution. To illustrate the use of RASPL-1 for search, we consider here a classical search problem, finding a large clique in a graph, and show how to encode it in RASPL-1.

Consider the finite graph with a set V of vertices and with edges $\{a_i, b_i\}$ $(i \in I)$. We want to find a clique of cardinality $\geq n$ in this graph or determine that such a clique does not exist. This problem can be represented by the following program:

$$\begin{array}{ll} \mathit{vertex}(a) & (a \in V), \\ \mathit{edge}(a_i, b_i) & (i \in I), \\ \{\mathit{in}(x)\} \leftarrow \mathit{vertex}(x) & \\ \leftarrow \mathit{in}(x),\ \mathit{in}(y),\ \mathit{not}\ \mathit{edge}(x, y),\ \mathit{not}\ x = y & \\ \leftarrow \{x : \mathit{in}(x)\}\, n - 1. & \end{array} \tag{12}$$

To see why the answer sets of this program correspond to cliques of cardinality $\geq n$, let's begin with the rules in the first three lines. This part of the program is similar to Example 2. Its answer sets are the sets consisting of

- the atoms $\mathit{vertex}(a)$ for all $a \in V$,
- the atoms $\mathit{edge}(a_i, b_i)$ for all $i \in I$,
- the atoms $\mathit{in}(a)$ for all a from some subset W of V.

Thus the answer sets of the first three lines of (12) are in a 1–1 correspondence with arbitrary sets W of vertices of the graph. The first of the two constraints in (12) eliminates the sets W that are not cliques, and the second constraint eliminates the sets that contain fewer than n vertices.

Safe Formulas and Rules

Herbrand Models of an Extended Signature

Recall that the answer sets of a sentence F are defined as the Herbrand interpretations of the signature $\sigma(F)$, consisting of the nonlogical constants that occur in F, that satisfy SM$[F]$. What if we extend $\sigma(F)$ by some object constants that do not occur in F, and consider the Herbrand interpretations of the extended signature that satisfy SM$[F]$ — will such interpretations, viewed as sets of ground atoms, be identical to the answer sets of F? Generally, the answer to this question is no. For instance, let F be the formula

$$p(a) \wedge \forall xy(p(x) \rightarrow p(y)), \tag{13}$$

corresponding to the RASPL-1 program

$$\begin{array}{l} p(a) \\ p(y) \leftarrow p(x). \end{array} \tag{14}$$

The only answer set of F is $\{p(a)\}$; on the other hand, the only Herbrand model of the extended signature $\{a, b\}$ that satisfies SM$[F]$ is a different set, $\{p(a), p(b)\}$.

Dependence of the meaning of a program on the presence of "irrelevant" object constants in the signature, such as b in the example above, may be considered unintuitive. Besides, generating answer sets for programs like this presents additional computational difficulties. For these reasons, of interest are syntactic conditions that eliminate "bad" formulas and programs, such as (13) and (14).

Safe Formulas

The definition of a safe sentence below is restricted to sentences in prenex form

$$Q_1 x_1 \cdots Q_n x_n M \tag{15}$$

(each Q_i is $\forall$ or $\exists$; $x_1, \ldots, x_n$ are distinct variables; the matrix M is a quantifier-free formula). This is not an essential limitation, because the usual process of converting formulas to prenex form is a strongly equivalent transformation (see Footnote 3). Furthermore, we assume that (15) does not contain function constants of arity > 0. (This condition is satisfied for FOL-representations of RASPL-1 programs.)

As a preliminary step, we assign to every quantifier-free formula F without function constants of arity > 0 a set $\mathit{RV}(F)$ of its *restricted variables*, as follows: [7]

- For an atomic formula F,
  - if F is an equality between two variables then $\mathit{RV}(F) = \emptyset$;
  - otherwise, $\mathit{RV}(F)$ is the set of all variables occurring in F;
- $\mathit{RV}(\bot) = \emptyset$;
- $\mathit{RV}(F \wedge G) = \mathit{RV}(F) \cup \mathit{RV}(G)$;
- $\mathit{RV}(F \vee G) = \mathit{RV}(F) \cap \mathit{RV}(G)$;
- $\mathit{RV}(F \rightarrow G) = \emptyset$.

For instance, x is restricted in the formula $p(x) \wedge \neg q(x, y)$, and y is not.

A sentence (15) is *safe* if every occurrence of every variable x_i in M is contained in an occurrence of a subformula $F \rightarrow G$ that satisfies two conditions:

- the occurrence of $F \rightarrow G$ is positive[8] if Q_i is $\forall$, and negative if Q_i is $\exists$;
- $x_i \in \mathit{RV}(F)$.

For instance, formulas (1) and (2), as well as the prenex form

$$\forall x(p(a) \wedge q(b) \wedge ((p(x) \wedge \neg q(x)) \rightarrow r(x)))$$

of (3), are safe: in each case, x is restricted in the antecedent of the implication. The prenex form of (7) is

$$\exists xy \neg\neg(q(x) \wedge q(y) \wedge x \neq y);$$

this formula is safe also. Indeed, this expression is shorthand for

$$\exists xy(((q(x) \wedge q(y) \wedge x \neq y) \rightarrow \bot) \rightarrow \bot),$$

and x, y are restricted in the antecedent of

$$(q(x) \wedge q(y) \wedge x \neq y) \rightarrow \bot.$$

(Note that dropping the double negation in (7), which is not a strongly equivalent transformation, would produce an unsafe formula.)

Consider, on the other hand, the prenex form of (13):

$$\forall xy(p(a) \wedge (p(x) \rightarrow p(y))).$$

This sentence is unsafe, because the only implication containing the occurrence of y in the matrix is $p(x) \rightarrow p(y)$, and y is not restricted in its antecedent.

[7] Some parts of this definition are similar to clauses of Definition 16 from (Topor and Sonenberg 1988).

[8] The occurrence of one formula in another is *positive* if the number of implications containing that occurrence in the antecedent is even, and *negative* otherwise.

The safety of a sentence does indeed imply that its meaning does not depend on the presence of irrelevant object constants in the signature:

Proposition 1 *For any safe sentence F containing at least one object constant and any signature σ obtained by adding object constants to $\sigma(F)$, an Herbrand interpretation of σ satisfies* SM$[F]$ *iff it is an answer set of F.*

Safe Rules

Our next goal is to adapt the theorem on safe sentences stated above to the syntax of RASPL-1.

We say that an aggregate expression $b\{\mathbf{x} : F\}$ is *allowed* if every member of $\mathbf{x}$ is restricted in F. For instance, $2\{x : p(x, y)\}$ is allowed; $2\{x : p(y)\}$ and $2\{x : \mathit{not}\ p(x, y)\}$ are not allowed.

We say that a variable v is *restricted* in an aggregate expression $b\{\mathbf{x} : F\}$ if v is restricted in F and does not belong to $\mathbf{x}$. For instance, y is restricted in $2\{x : p(x, y)\}$ and in $2\{x : p(y)\}$, but is not restricted in $2\{x : \mathit{not}\ p(x, y)\}$.

A variable v is *free* in a rule (10) if

- v occurs in the head $A_1 ; \ldots ; A_l$ of the rule, or
- the body $E_1, \ldots, \mathit{not}\ E_n$ of the rule contains an aggregate expression $b\{\mathbf{x} : F\}$ such that v occurs in F and does not belong to $\mathbf{x}$.

A rule (10) is *safe* if

- each aggregate expression in its body is allowed, and
- each of its free variables is restricted in one of the aggregate expressions $E_1, \ldots, E_m$.

A RASPL-1 program is *safe* if each of its rules is safe. For instance, program (14) is not safe, because y is not restricted in the body of the second rule.

Proposition 2 *Let Π be a safe RASPL-1 program containing at least one object constant, and let F be its FOL-representation. For any signature σ obtained by adding object constants to $\sigma(F)$, an Herbrand interpretation of σ satisfies* SM$[F]$ *iff it is an answer set of* Π.

The proof of this proposition is based on the fact that converting the FOL-representation of a safe program to prenex form gives a safe sentence.

Comparison with the Semantics of Counting According to Faber, Leone and Pfeifer

The approach to the semantics of aggregates proposed in (Faber *et al.* 2004) is attractive because it does not produce the same unintuitive results as its predecessors in application to nonmonotonic aggregates, such as sums of families of numbers that can be both positive and negative.[9] This success is achieved using an ingenious modification of the original definition of the reduct from (Gelfond and Lifschitz 1988). The reduct of a program according to (Faber *et al.* 2004) is generated by dropping some of the rules, but the rules that are not dropped remain intact. They are not "reduced" in any way even when they contain negation, as done in the 1988 definition.

[9] See, for instance, (Ferraris and Lifschitz 2005, Footnote 6).

The language RASPL-1 is less expressive than the language introduced by Faber *et al.* in the sense that it incorporates only one aggregate, counting. But in another sense it is more expressive: F_i in an aggregate expression (9) can be a negated atom, which is not allowed by Faber *et al.* The two languages have a significant common part (modulo some syntactic details), [10] and we will talk about it using the following terminology. An aggregate expression (9) is *positive* if each F_i is an atom. If each aggregate expression in a rule is positive then we say that the rule is *semi-positive*, and similarly for programs. Thus a semi-positive program does not have negations inside aggregate expressions, but a negation may occur in front of an aggregate expression in the body of a rule.

The approach of (Faber *et al.* 2004) can be adapted to the semi-positive fragment of RASPL-1 as follows. Let Π be a program without free variables, and let S be a set of ground atoms not containing equality. The *reduct* of Π with respect to S is obtained from Π by dropping all rules r such that S does not satisfy the FOL-representation of the body of r. About a semi-positive program Π without free variables we say that S is *an answer set of* Π *in the sense of Faber, Leone and Pfeifer* if S is minimal among the sets satisfying (as Herbrand models) the FOL-representation of the reduct of Π with respect to S. Finally, to extend this definition to semi-positive programs with free variables, we define the answer sets of such a program Π to be the answer sets of the program obtained from Π by replacing each rule with all its "closed instances"—the rules obtained from it by substituting object constants for free variables in all possible ways.

This semantics is equivalent to the semantics of RASPL-1 when the latter is restricted to semi-positive programs:

Proposition 3 *The answer sets of any semi-positive program are identical to its answer sets in the sense of Faber, Leone and Pfeifer.*

The requirement that a program be semi-positive is important, because "choice rules," such as the rule

$$\{q(x)\} \leftarrow p(x)$$

from Example 2, are not semi-positive. Recall that this is shorthand for a rule containing a nonpositive aggregate expression:

$$q(x) \leftarrow p(x), \mathit{not}\ 1\ \{: \mathit{not}\ q(x)\}. \tag{16}$$

Actually, choice rules cannot be simulated by semi-positive

[10] In the syntax of (Faber *et al.* 2004), $b\,\{\mathbf{x} : F\}$ is written as $\#\mathit{count}\{\mathbf{x} : F\} \geq b$. Incidentally, their language allows us also to write $\#\mathit{count}\{\mathbf{x} : F\} \leq b$. The behavior of this expression is usually similar to the behavior of $\{\mathbf{x} : F\}\,b$ in RASPL-1, but in some contexts properties of $\leq b$ seem unintuitive. For instance, the one-rule program $p(a) \leftarrow \mathit{not}\ \#\mathit{count}\{x : p(x)\} \leq 0$ has one answer set—empty, although "unfolding" this rule

$$p(a) \leftarrow \mathit{not}\ q$$
$$q \leftarrow \#\mathit{count}\{x : p(x)\} \leq 0$$

produces a program with two answer sets, $\{q\}$ and $\{p(a)\}$.

rules without introducing auxiliary predicates.[11] For instance, the third line of the clique program (12) cannot be simulated by semi-positive rules. From the perspective of answer set programming, the availability of choice rules is the main advantage of RASPL-1 in comparison with the language from (Faber *et al.* 2004).[12]

The definition of an answer set in the sense of Faber, Leone and Pfeifer can be extended to arbitrary RASPL-1 programs in an obvious way, but without the assumption that the program is semi-positive the assertion of Proposition 3 would be invalid. Indeed, the set of closed instances

$$q(c) \leftarrow p(c), \mathit{not}\ 1\,\{:\ \mathit{not}\ q(c)\}$$

of rule (16) has essentially the same reduct with respect to any S as the set of trivial rules $q(c) \leftarrow p(c), q(c)$.

A large subset of the language defined in (Faber *et al.* 2004) is implemented in the answer set solver DLV.[13] Proposition 3 above shows that in some cases it is possible to compute the answer sets of a RASPL-1 program simply by running DLV. What are limitations of this method? Here are some preliminary considerations.

The safety condition imposed by DLV on its input is more stringent than safety defined in the previous section. For instance, the RASPL-1 rule

$$p(x) \leftarrow 2\,\{y : q(x,y)\} \tag{17}$$

is safe, but the corresponding rule in the syntax of DLV causes this system to produce the error message `Rule is not safe.`

There is a process, however, that allows us to circumvent this problem, and it has been implemented by the DLV group.[14] Any safe RASPL-1 rule can be turned into a strongly equivalent rule that is "DLV-safe" by appending appropriate atoms to its body. For instance, the rule

$$p(x) \leftarrow 2\,\{y : q(x,y)\}, q(x,z)$$

is strongly equivalent to (17) and does not create any problems for DLV.

What about the RASPL-1 rules that are not semi-positive? It appears that they can be always translated into the language of DLV at the price of introducing auxiliary predicates. For instance, rule (16) can be replaced by

$$q(x) \leftarrow p(x), \mathit{not}\ 1\,\{:\ \mathit{aux}(x)\}$$
$$\mathit{aux}(x) \leftarrow p(x), \mathit{not}\ q(x).$$

On these grounds, we expect that DLV can serve as the basis for the implementation of a large subset of RASPL-1.

[11] The reason is that any semi-positive program has the anti-chain property: one of its answer sets cannot be a proper subset of another. The use of auxiliary predicates is discussed below.

[12] The semantics of programs with aggregates without variables that was proposed in (Ferraris 2005, Section 4) has similar advantages.

[13] `http://www.dbai.tuwien.ac.at/proj/dlv/`

[14] Nicola Leone, personal communication, December 7, 2007.

Comparison with the Semantics of Cardinality Constraints According to Syrjänen

Cardinality constraints in the sense of (Syrjänen 2004) may involve "conditional literals." A conditional literal has the form

$$\mathbf{x}\,.\,L : A \tag{18}$$

where $\mathbf{x}$ is a list of variables, L is a literal, and A is an atom. "Intuitively, $L : A$ can be seen as a conjunction that is evaluated in two phases: first A is checked, and if it is true, then the truth value of L determines the truth value of the whole construct" (Syrjänen 2004, Section 2).

Thus intuitively (18) is somewhat similar to the RASPL-1 expression $\{\mathbf{x} : L, A\}$. We will now show how the idea behind the semantics from (Syrjänen 2004) can be adapted to a fragment of RASPL-1 as follows.

Answer Sets According to Syrjänen

We say that an aggregate expression (9) is *short* if

(i) $b = k = 1$, $\mathbf{x}$ is empty, and F_1 is an atom (recall that we have agreed to identify this aggregate expression with F_1), or

(ii) $k = 2$, and F_2 is an atom.

In other words, short aggregate expressions are atoms and expressions of the form $b\,\{\mathbf{x} : L, A\}$, where L is a literal and A is an atom. A RASPL-1 program is *regular* if it doesn't contain equality and, in each of its rules (10), $l = 1$ and the aggregate expressions $E_1, \dots, E_n$ are short.

For example, the clique program (12) is not regular because of its last rule: the expression $\{x : \mathit{in}(x)\}$ in the body corresponds to (9) with $k = 1$ and non-empty $\mathbf{x}$. We can make the program regular by replacing that expression with $\{x : \mathit{in}(x), \mathit{vertex}(x)\}$.

We will define, for any regular program Π without free variables and any set S of ground atoms not containing equality, the *reduct of* Π *with respect to* S ("reduct in the sense of Syrjänen," if we want to distinguish it from the reduct in the sense of Faber, Leone and Pfeifer introduced above). The reduct of Π with respect to S, as we will see, is a set of formulas, and it will be denoted by Π^S.

The definition uses the following notation: for any ground atom A,

$$A\,?\,S \;=\; \begin{cases} \top, & \text{if } A \in S, \\ \bot, & \text{otherwise;} \end{cases}$$

$$(\mathit{not}\ A)\,?\,S \;=\; \neg(A\,?\,S).$$

The reducts of short aggregate expressions without free variables,[15] and of their negations, are defined as follows. For expressions of type (i), A^S is A; $(\mathit{not}\ A)^S$ is $(\mathit{not}\ A)\,?\,S$. For expressions of type (ii), $(b\,\{\mathbf{x} : L(\mathbf{x}), A(\mathbf{x})\})^S$ stands for

$$\bigvee_{\substack{C \subseteq \{\mathbf{c}:\, A(\mathbf{c}) \in S\} \\ |C| = b}} \bigwedge_{\mathbf{c} \in C} L(\mathbf{c})^S,$$

[15] The *free variables* of an aggregate expression (9) are the variables that occur in this expression but do not belong to $\mathbf{x}$.

where **c** ranges over all tuples of object constants of the same length as **x**; $(\mathit{not}\ b\,\{\mathbf{x} : L(\mathbf{x}), A(\mathbf{x})\})^S$ stands for

$$\neg \bigvee_{\substack{C \subseteq \{\mathbf{c}:\, A(\mathbf{c}) \,\in\, S\} \\ |C| \,=\, b}} \bigwedge_{\mathbf{c} \in C} (L(\mathbf{c})\,?\,S).$$

Finally, the reduct Π^S of a regular program Π without free variables is the set of formulas

$$E_1^S \wedge \cdots \wedge E_m^S \wedge (\mathit{not}\ E_{m+1})^S \wedge \cdots \wedge (\mathit{not}\ E_n)^S \rightarrow A$$

corresponding to the rules

$$A \leftarrow\ E_1, \ldots, E_m, \mathit{not}\ E_{m+1}, \ldots, \mathit{not}\ E_n$$

of Π.

It is clear that Π^S consists of implications such that the consequent of every implication is a ground atom, and the antecedent of every implication is a propositional combination of ground atoms; furthermore, all occurrences of ground atoms in the antecedents are positive. (There may be negations in the antecedents, but they are only applied to propositional combinations of the 0-place connectives $\top$, $\bot$.) Consequently, Π^S is equivalent to a set of definite clauses, and has a unique minimal Herbrand model. If this model equals S then we say that S is an *answer set of* Π *in the sense of Syrjänen.*

Let us check, for instance, that the set

$$\{p(a), q(a), r(b)\} \tag{19}$$

is an answer set of the program

$$\begin{array}{l} p(a) \\ q(a) \\ r(b) \leftarrow 1\,\{x : p(x), q(x)\} \end{array} \tag{20}$$

in the sense of Syrjänen. The reduct of (20) with respect to (19) is

$$p(a) \wedge q(a) \wedge (p(a) \rightarrow r(b)). \tag{21}$$

The minimal Herbrand model of this formula is (19).

To extend the definition of an answer set in the sense of Syrjänen to regular programs with free variables, we replace rules with their closed instances.

The definition above follows the method of (Syrjänen 2004) very closely, although we use different terminology and notation. The most essential difference is that our condition $|C| = b$ refers to the number of *tuples* **c**, rather than the number of *literals* $L(\mathbf{c})$. These numbers can differ if some of the variables **x** do not actually occur in $L(\mathbf{x})$.

Conditional Literals vs. Conjunctions

The quote from (Syrjänen 2004, Section 2) at the beginning of this section suggests that L and A in a cardinality constraint (18) are treated somewhat asymmetrically in Syrjänen's semantics even when L is an atom. For instance, if we replace $p(x), q(x)$ in the last rule of (20) with $q(x), p(x)$ then the reduct of the program will change: the last conjunctive term of (21) will turn into $q(a) \rightarrow r(b)$.

On the other hand, the meaning of $F_1, \ldots, F_k$ in our aggregate expression (9) is invariant with respect to changing the order of the conjunctive terms, just as in classical logic. Because of this difference, it is not surprising that the answer sets of a regular program in the sense of Syrjänen are not necessarily identical to its answer sets in the sense of RASPL-1. The program

$$\begin{array}{l} p(a) \\ q(a) \leftarrow 1\,\{x : p(x), q(x)\} \end{array} \tag{22}$$

is an example. According to the semantics of RASPL-1, its only answer set is $\{p(a)\}$; according to Syrjänen, its answer sets are $\{p(a)\}$ and $\{p(a), q(a)\}$. Program (22) illustrates the difference between conditional literals and conjunctions.

The treatment of the atom A in a conditional literal (18) can be reflected in our reductive approach by inserting two negations in front of A, as follows. In the FOL-representation of a regular program Π, each aggregate expression of the form $b\,\{\mathbf{x} : L(\mathbf{x}), A(\mathbf{x})\}$ is represented by the subformula

$$\exists \mathbf{x}^1 \cdots \mathbf{x}^b \left[\bigwedge_{1 \leq i \leq b} (L(\mathbf{x}^i) \wedge A(\mathbf{x}^i)) \wedge \bigwedge_{1 \leq i < j \leq b} \neg(\mathbf{x}^i = \mathbf{x}^j) \right].$$

The result of inserting $\neg\neg$ in front of every $A(\mathbf{x}^i)$ in every such subformula will be called the *modified FOL-representation* of Π. For instance, the FOL-representation of program (22) is

$$p(a) \wedge (\exists x(p(x) \wedge q(x)) \rightarrow q(a)),$$

and the modified FOL-representation of this program is

$$p(a) \wedge (\exists x(p(x) \wedge \neg\neg q(x)) \rightarrow q(a)).$$

Proposition 4 *For any regular program* Π*, the answer sets of* Π *in the sense of Syrjänen are identical to the answer sets of the modified FOL-representation of* Π*.*

Strongly Regular Programs

Example (22) shows that the assertion of Proposition 4 will become incorrect if we drop the word "modified." Under some conditions, however, inserting double negations in the FOL-representation of a regular program as described above has no effect on the answer sets.

The *predicate dependency graph* of a RASPL-1 program Π is the directed graph such that

- its vertices are the predicate constants occurring in Π, and
- it has an edge from a vertex p to a vertex q if there is a rule (10) in Π such that p occurs in its head $A_1; \ldots; A_l$, and q occurs positively (i.e., not preceded by *not*) in one of the aggregate expressions $E_1, \ldots, E_m$.[16]

A regular program Π is *strongly regular* if, for every aggregate expression of the form $b\,\{\mathbf{x} : L, A\}$ occurring as one of $E_1, \ldots, E_m$ in a rule (10) of Π, there is no path in the predicate dependency graph of Π from the predicate constant in A to the predicate constant in the head of the rule. For

[16] This is essentially a special case of the definition of the predicate dependency graph of a first-order formula from (Ferraris *et al.* 2007, Section 5.3).

instance, the predicate dependency graph of program (20) has two edges, from r to p and from r to q; this program is strongly regular, because r is not reachable from q in this graph. The predicate dependency graph of (22) has the edges from q to p and from q to q; this program is not strongly regular, because q is reachable from q.

Proposition 5 *The answer sets of any strongly regular program are identical to its answer sets in the sense of Syrjänen.*

From the practical point of view, the condition defining strongly regular programs, like the safety condition, is very general. A program for which it is violated, such as (22), would cause LPARSE to produce an error message.

Conclusion

The language RASPL-1, proposed in this note, combines useful constructs available in the best known implemented answer programming languages: choice rules (LPARSE) and counting (DLV). Its definition in terms of the syntactic operator SM exemplifies the reductive approach to the semantics of answer set programming languages with variables, which does not rely on grounding. We hope that the simplicity of such definitions will facilitate proving the correctness of programs written in RASPL-1 and similar languages.

We plan to extend this work in several directions: extend RASPL-1 by aggregates other than counting and by other useful features; implement an extension of RASPL-1 on top of DLV; conduct experiments with the use of RASPL-1 and its extensions for solving knowledge representation and search problems.

Acknowledgements

Many thanks to Selim Erdoğan, Paolo Ferraris, Michael Gelfond, Yuliya Lierler and David Pearce for useful discussions related to the topic of this paper. The first and the third authors were partially supported by the DTO AQUAINT program. The second author was partially supported by the National Science Foundation under Grant IIS-0712113.

References

Wolfgang Faber, Nicola Leone, and Gerard Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, 2004. Revised version: http://www.wfaber.com/research/papers/jelia2004.pdf.

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

Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.

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

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.

Joohyung Lee and Ravi Palla. Yet another proof of the strong equivalence between propositional theories and logic programs. In *Working Notes of the Workshop on Correspondence and Equivalence for Nonmonotonic Theories*, 2007.

Nicola Leone, Wolfgang Faber, Gerald Pfeifer, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

Vladimir Lifschitz, David Pearce, and Agustin Valverde. A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2007.

Vladimir Lifschitz. Computing circumscription. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 121–127, 1985.

John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980.

Tommi Syrjänen. Cardinality constraint programs. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, pages 187–199, 2004.

R. W. Topor and E. A. Sonenberg. On domain independent databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 217–240. Morgan Kaufmann, San Mateo, CA, 1988.