TLP **17** (5–6): 855–871, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000394 First published online 23 August 2017

Program completion in the input language of GRINGO[*]

AMELIA HARRISON, VLADIMIR LIFSCHITZ and DHANANJAY RAJU
Department of Computer Science, Austin, Texas, USA
(*e-mails:* `ameliaj@cs.utexas.edu,vl@cs.utexas.edu,draju@cs.utexas.edu`)

submitted 18 July 2017; revised 18 July 2017; accepted 27 July 2017

Abstract

We argue that turning a logic program into a set of completed definitions can be sometimes thought of as the "reverse engineering" process of generating a set of conditions that could serve as a specification for it. Accordingly, it may be useful to define completion for a large class of Answer Set Programming (ASP) programs and to automate the process of generating and simplifying completion formulas. Examining the output produced by this kind of software may help programmers to see more clearly what their program does, and to what degree its behavior conforms with their expectations. As a step toward this goal, we propose here a definition of program completion for a large class of programs in the input language of the ASP grounder GRINGO, and study its properties.

KEYWORDS: formal methods, program completion, Answer Set Programming.

1 Introduction

Our interest in defining completion (Clark 1978) for programs in the input language of the ASP grounder GRINGO (`https://potassco.org`) is motivated by the goal of extending formal methods for software verification to answer set programming. Turning a logic program into a set of completed definitions can be sometimes thought of as the "reverse engineering" process of generating a set of conditions that could serve as a specification for it. Consider, for instance, the condition "set r is the union of sets p and q." In the language of logic programming, this definition of r is represented by the pair of rules

$$\begin{array}{l} r(X) \leftarrow p(X), \\ r(X) \leftarrow q(X). \end{array} \tag{1.1}$$

The corresponding completed definition

$$\forall X(r(X) \leftrightarrow p(X) \vee q(X))$$

is the usual definition of union in set theory. Turning program (1.1) into a completed definition gives us a plausible specification that could have led to this program in

[*] This work was partially supported by the National Science Foundation under Grant IIS-1422455.

the first place. The stable model semantics of program (1.1) matches the completed definition, because the program is tight (Fages 1994; Erdem and Lifschitz 2003).

It may be useful to define completion for a large class of ASP programs and to automate the process of generating and simplifying completion formulas. (Simplifying is essential because "raw" completion rarely provides such a clean specification as in the example above.) Examining the output produced by this kind of software may help programmers to see more clearly what their program does, and to what degree its behavior conforms with their expectations. If the programming project started with a formal specification, then they may be able to verify the correctness of the program relative to that specification by comparing the given specification with the "engineered specification" extracted from the program.

As a step toward this goal, we propose here a definition of program completion for a large class of GRINGO programs. Three issues need to be addressed. First, GRINGO programs often include constraints and choice rules, which are not covered by Clark's theory. Extending completion to these constructs has been discussed in the literature; see, for instance, Ferraris *et al.* (2011, Section 6.1).

Second, we need to take into account the fact that in the language of GRINGO a ground term may denote a set of values, rather than a single value. For instance, the term 1..8 denotes the set $\{1,\dots,8\}$, and the condition $X = 1..8$ in the body of a rule expresses that X is an element of that set. In standard mathematical notation, this condition would be expressed using the set membership symbol rather than equality. The syntax of GRINGO allows us to write also

$$X..X+1 = Y..Y+1,$$

which is understood as

$$X \text{ and } Y \text{ are integers, and } \{X, X+1\} \cap \{Y, Y+1\} \neq \emptyset.$$

Third, the semantics of aggregate expressions in the language of GRINGO depends on the distinction between local and global variables. This is similar to the distinction between bound and free variables familiar from first-order logic, except that the definition of a local variable does not refer to quantifiers. The expression $sum\{X{\times}Y : p(X,Y)\}$ in the body of a rule[1] may correspond to any of the expressions

$$\sum_{X,Y\,:\,p(X,Y)} X\times Y, \quad \sum_{X\,:\,p(X,Y)} X\times Y, \quad \sum_{Y\,:\,p(X,Y)} X\times Y$$

depending on where X and Y occur in other parts of the rule. Our way of translating aggregate expressions takes into account this feature. Otherwise, it is similar to the approach proposed by Ferraris and Lifschitz (2010), which is closely related to the use of generalized quantifiers by Lee and Meng (2009, 2012). One of their results (Lee and Meng 2012, Theorem 4) relates stable models of formulas with generalized quantifiers to program completion.

[1] We use here an "abstract" syntax, which disregards some details related to writing rules as strings of ASCII characters (Gebser *et al.* 2015, Section 1). In an actual GRINGO program, this expression would be written as `#sum{X*Y:p(X,Y)}`.

We start by discussing a class of programs that do not contain aggregate expressions. Sections 2 and 3 define a language of programs and a language of formulas—the source and the target of the completion operator. Section 4 describes the process of representing rules by formulas, which is used in the definition of completion in Section 5. We discuss tight programs in Section 6 and give an example of calculating an engineered specification in Section 7. Incorporating aggregate expressions is described in Section 8. In Section 9, the class of formulas is further extended by adding variables for integers, which can be often used to simplify formulas that involve arithmetic operations. The definition of a stable model for the class of programs defined in Section 8 is given in Appendix A. Proofs of theorems are given in Appendix B, posted online.

2 Programs

We assume that four disjoint sets of symbols are selected: *numerals*; *symbolic constants*; *variables*; and *operation names* of various arities. We assume that these sets do not contain the *interval symbol*

$$..$$

the *relation symbols*

$$= \quad \neq \quad < \quad > \quad \leqslant \quad \geqslant$$

and the symbols

$$\mathit{inf} \quad \mathit{sup} \quad \mathit{not} \quad \wedge \quad \vee \quad \leftarrow$$

$$, \quad ; \quad : \quad (\quad) \quad \{ \quad \}$$

$$\in \quad \neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow \quad \forall \quad \exists$$

We assume that a 1–1 correspondence between the set of numerals and the set $\mathbf{Z}$ of integers is chosen. For every integer n, the corresponding numeral will be denoted by $\overline{n}$. We will identify a numeral with the corresponding integer when this does not lead to confusion.

We assume that for every operation name op, a function $\widehat{op}$ from a subset of $\mathbf{Z}^n$ to $\mathbf{Z}$ is chosen, where n is the arity of op. For instance, we can choose $plus$ as a binary operation name, define $\widehat{plus}$ as the addition of integers, and use t_1+t_2 as shorthand for $plus(t_1, t_2)$.

Terms are defined recursively, as follows:

- numerals, symbolic constants, variables, and the symbols inf and sup are terms,
- if f is a symbolic constant and $\mathbf{t}$ is a non-empty tuple of terms (separated by commas), then $f(\mathbf{t})$ is a term,
- if op is an n-ary operation name and $\mathbf{t}$ is an n-tuple of terms, then $op(\mathbf{t})$ is a term,
- if t_1 and t_2 are terms, then $(t_1..t_2)$ is a term.

A term, or another syntactic expression, is *ground* if it does not contain variables. A ground term is *precomputed* if it contains neither operation names nor the interval symbol. According to the semantics of terms defined in Appendix A.1, every ground term t denotes a finite set $[t]$ of precomputed terms, which are called the *values* of t. For instance,

$$[\overline{8}] = \{\overline{8}\},\ [\overline{1}..\overline{8}] = \{\overline{1}, \dots, \overline{8}\},\ [abc + \overline{1}] = \emptyset$$

if abc is a symbolic constant.

We assume a total order on precomputed terms such that inf is its least element, sup is its greatest element, and, for any integers m and n, $\overline{m} \leqslant \overline{n}$ iff $m \leqslant n$.

Atoms are expressions of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of terms, possibly empty. An atom of the form $p()$ will be written as p. *Literals* are atoms (*positive literals*) and atoms preceded by *not* (*negative literals*). A *comparison* is an expression of the form $(t_1 \prec t_2)$ where t_1, t_2 are terms and $\prec$ is a relation symbol.

A *choice expression* is an expression of the form $\{A\}$ where A is an atom.

A *rule* is an expression of the form

$$Head \leftarrow Body \tag{2.1}$$

where

- $Body$ is a conjunction (possibly empty) of literals and comparisons, and
- $Head$ is either an atom [then we say that (2.1) is a *basic rule*], or a choice expression [then (2.1) is a *choice rule*], or empty [then (2.1) is a *constraint*].

If the body of a basic rule or choice rule is empty, then the arrow will be dropped.

A *program* is a set of rules.

An *interpretation* is a set of atoms of the form $p(\mathbf{t})$ where $\mathbf{t}$ is a tuple of precomputed terms. Every program denotes a set of interpretations, which are called its *stable models* (Appendix A).

3 Formulas

The language defined in this section is essentially a first-order language with variables for precomputed terms.

An *argument* is a term that contains neither operation names nor the interval symbol[2]. *Formulas* are defined recursively:

(a) if p is a symbolic constant and $\mathbf{arg}$ is a tuple of arguments, then $p(\mathbf{arg})$ is a formula,
(b) if arg_1 and arg_2 are arguments and $\prec$ is a relation symbol, then $(arg_1 \prec arg_2)$ is a formula,
(c) if arg is an argument and t is a term, then $arg \in t$ is a formula,

[2] Thus, precomputed terms (Section 2) can be alternatively described as ground arguments. This will not be the case, however, when we extend the definition of an argument in Section 8.2 to incorporate aggregates.

(d) $\bot$ ("false") is a formula,
(e) if F and G are formulas, then $(F \rightarrow G)$ is a formula,
(f) if F is a formula and X is a variable, then $\forall X F$ is a formula.

We will drop parentheses in formulas when it does not lead to confusion. Propositional connectives other than implication, and the existential quantifier, are defined as abbreviations in the usual way. Free and bound occurrences of variables, closed formulas, and the universal closure of a formula are defined as usual in first-order logic.

Note that a term that is not an argument can occur in a formula in only one position—to the right of the $\in$ symbol. For example, $X \in \overline{1}..\overline{8}$ and $X \in Y+\overline{1}$ are formulas, but $X = \overline{1}..\overline{8}$ and $X = Y+\overline{1}$ are not. The reason why we do not allow $Y+\overline{1}$ in equalities is that substituting a precomputed term for Y in this expression (for instance, *abc*) may give a term that has no values.

If F is a formula, X is a variable, and r is a precomputed term, then F^X_r stands for the formula obtained from F by substituting r for all free occurrences of X.

The truth value $F^{\mathscr{I}}$, assigned by an interpretation $\mathscr{I}$ to a closed formula F, is defined as t or f, in accordance with the following rules:

(a) $p(\mathbf{arg})^{\mathscr{I}}$ is t if $p(\mathbf{arg}) \in \mathscr{I}$ (and f otherwise),
(b) $(arg_1 \prec arg_2)^{\mathscr{I}}$ is t if $arg_1 \prec arg_2$,
(c) $(arg \in t)^{\mathscr{I}}$ is t if $arg \in [t]$,
(d) $\bot^{\mathscr{I}}$ is f,
(e) $(F \rightarrow G)^{\mathscr{I}}$ is f if $F^{\mathscr{I}}$ is t and $G^{\mathscr{I}}$ is f,
(f) $(\forall X F)^{\mathscr{I}}$ is t if, for every precomputed term r, $(F^X_r)^{\mathscr{I}}$ is t.

We say that an interpretation $\mathscr{I}$ *satisfies* a closed formula F if $F^{\mathscr{I}} = $ t.

For example, the interpretation $\{p(\overline{2}), p(\overline{3}), p(\overline{4})\}$ satisfies the formula $\exists X(p(X) \wedge X \in \overline{1}..\overline{8})$. Indeed, it satisfies $p(\overline{3})$, because it includes $p(\overline{3})$; it also satisfies $\overline{3} \in \overline{1}..\overline{8}$, because $[\overline{1}..\overline{8}]$ is $\{\overline{1}, \dots, \overline{8}\}$, and $\overline{3}$ is an element of this set. Consequently, it satisfies the conjunction $p(\overline{3}) \wedge \overline{3} \in \overline{1}..\overline{8}$.

A formula is *universally valid* if its universal closure is satisfied by all interpretations. A formula F is *equivalent* to a formula G if $F \leftrightarrow G$ is universally valid. Since our definition of satisfaction treats propositional connectives, quantifiers, and equality in the same way as the standard definition of satisfaction applied to the domain of precomputed terms, all equivalent transformations sanctioned by classical first-order logic can be used in this setting as well. The following additional observations about equivalence will be useful.

Observation 1. *For any argument arg and any ground term t, $arg \in t$ is equivalent to $\bigvee_{r \in [t]}(arg = r)$.*

This is immediate from the definition of satisfaction.

For example, for any integers m and n, $arg \in \overline{m}..\overline{n}$ is equivalent to $\bigvee_{i=m}^{n}(arg = \overline{i})$.

Observation 2. *For any arguments arg_1 and arg_2, $arg_1 \in arg_2$ is equivalent to $arg_1 = arg_2$.*

It is sufficient to check this claim for the case when arg_1, arg_2 are ground. In this case, it follows from the fact that $[arg_2]$ is the singleton $\{arg_2\}$.

For example, $X \in Y$ is equivalent to $X = Y$.

4 Representing rules by formulas

In this section, we define a syntactic transformation ϕ that turns rules and their subexpressions into formulas—their *formula representations*.

Formula representations of literals and comparisons are defined as follows:

- $\phi\, p(\mathbf{t})$ is $\exists \mathbf{X}(\mathbf{X} \in \mathbf{t} \wedge p(\mathbf{X}))$[3],
- $\phi(not\;\; p(\mathbf{t}))$ is $\exists \mathbf{X}(\mathbf{X} \in \mathbf{t} \wedge \neg p(\mathbf{X}))$,
- $\phi(t_1 \prec t_2)$ is $\exists X_1 X_2(X_1 \in t_1 \wedge X_2 \in t_2 \wedge X_1 \prec X_2)$;

here, $\mathbf{X}$ is a tuple of new variables of the same length as $\mathbf{t}$, and X_1, X_2 are new variables.

For example, the transformation ϕ turns $p(X)$ into $\exists Y(Y \in X \wedge p(Y))$; this formula is equivalent to $\exists Y(Y = X \wedge p(Y))$, and consequently to $p(X)$. The formula representation of $p(\overline{1}..X)$ is $\exists Y(Y \in \overline{1}..X \wedge p(Y))$. The representation of $X = \overline{1}..\overline{8}$ is

$$\exists X_1 X_2(X_1 \in X \wedge X_2 \in \overline{1}..\overline{8} \wedge X_1 = X_2);$$

this formula is equivalent to $X \in \overline{1}..\overline{8}$.

If each of the expressions $C_1, \dots, C_k$ is a literal or a comparison, then $\phi(C_1 \wedge \dots \wedge C_k)$ stands for $\phi C_1 \wedge \dots \wedge \phi C_k$.

The formula representation of a basic rule

$$p(\mathbf{t}) \leftarrow Body \tag{4.1}$$

is defined as the implication

$$\mathbf{V} \in \mathbf{t} \wedge \phi(Body) \rightarrow p(\mathbf{V}), \tag{4.2}$$

where $\mathbf{V}$ is a tuple of new variables of the same length as $\mathbf{t}$. For example, the formula representation of the rule

$$q(X + \overline{1}) \leftarrow p(X) \wedge X = \overline{1}..\overline{8} \tag{4.3}$$

is

$$V \in X + \overline{1} \wedge \phi\, p(X) \wedge \phi\,(X = \overline{1}..\overline{8}) \rightarrow q(V);$$

after applying equivalent transformations to the antecedent, this formula becomes

$$V \in X + \overline{1} \wedge p(X) \wedge X \in \overline{1}..\overline{8} \rightarrow q(V). \tag{4.4}$$

The formula representation of a choice rule

$$\{p(\mathbf{t})\} \leftarrow Body \tag{4.5}$$

[3] If $\mathbf{X}$ is $X_1, \dots, X_n$, and $\mathbf{t}$ is $t_1, \dots, t_n$, then $\mathbf{X} \in \mathbf{t}$ stands for the conjunction $\bigwedge_{i=1}^{n} X_i \in t_i$.

is defined as the (universally valid) formula

$$\mathbf{V} \in \mathbf{t} \wedge \phi(Body) \wedge p(\mathbf{V}) \rightarrow p(\mathbf{V}), \tag{4.6}$$

where $\mathbf{V}$ is a tuple of new variables of the same length as $\mathbf{t}$.

For example, the formula representation of the rule $\{p(\overline{1}..\overline{8})\}$ is

$$V \in \overline{1}..\overline{8} \wedge p(V) \rightarrow p(V).$$

The formula representation of a constraint $\leftarrow Body$ is the formula

$$\neg\phi(Body). \tag{4.7}$$

5 Completion

A *predicate symbol* is a pair p/n, where p is a symbolic constant and n is a non-negative integer. The *definition* of a predicate symbol p/n in a program Γ consists of

- the basic rules of Γ with the head of the form $p(t_1, \dots, t_n)$, and
- the choice rules of Γ with the head of the form $\{p(t_1, \dots, t_n)\}$.

It is clear that any program is the union of the definitions of predicate symbols and a set of constraints.

If the definition of p/n in a finite program Γ is $\{R_1, \dots, R_k\}$, then each of the formulas ϕR_i has the form

$$F_i \rightarrow p(\mathbf{V}), \tag{5.1}$$

where $\mathbf{V}$ is a tuple of distinct variables. We will assume that this tuple is chosen in the same way for all i. The *completed definition* of p/n in Γ is the formula

$$\forall \mathbf{V} \left(p(\mathbf{V}) \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{U}_i F_i \right), \tag{5.2}$$

where $\mathbf{U}_i$ is the list of all free variables of the formula F_i that do not belong to $\mathbf{V}$.

For example, if the definition of $p/1$ in Γ is $p(\overline{1}..\overline{8})$, then $k = 1$, $\mathbf{U}_1$ is empty, and F_1 is $V \in \overline{1}..\overline{8}$, so that the completed definition of $p/1$ is

$$\forall V (p(V) \leftrightarrow V \in \overline{1}..\overline{8}). \tag{5.3}$$

If the definition of $p/1$ is the choice rule $\{p(\overline{1}..\overline{8})\}$, then F_1 is

$$V \in \overline{1}..\overline{8} \wedge p(V),$$

and the completed definition of $p/1$ is

$$\forall V (p(V) \leftrightarrow V \in \overline{1}..\overline{8} \wedge p(V)).$$

This formula is equivalent to

$$\forall V (p(V) \rightarrow V \in \overline{1}..\overline{8}). \tag{5.4}$$

It is clear that completed definitions are invariant with respect to equivalent transformations of the antecedents of implications ϕR_i, in the sense that replacing an antecedent F_i in (5.2) by an equivalent formula is an equivalent transformation. Assume, for instance, that the definition of $q/1$ in Γ is (4.3). Formula (4.4) is the result of simplifying the antecedent of the formula representation of that rule, and the completed definition of $q/1$ can be written as

$$\forall V(q(V) \leftrightarrow \exists X(V \in X+\overline{1} \wedge p(X) \wedge X \in \overline{1}..\overline{8})). \tag{5.5}$$

About a program or another syntactic expression, we say that a predicate symbol p/n *occurs* in it if it contains an atom of the form $p(t_1,\dots,t_n)$. The *completion* of a finite program Γ consists of

- the completed definitions of all predicate symbols occurring in Γ, and
- the universal closures of the formula representations of all constraints in Γ.

The definition of completion matches the stable model semantics in the following sense:

Theorem 1
Every stable model of a finite program satisfies its completion.

In the next section, we define a class of programs for which the converse of Theorem 1 can be proved.

6 Tight programs

For any program Γ, by G_Γ we denote the directed graph that has the predicate symbols occurring in Γ as its vertices, and has an edge from q/m to p/n if Γ includes a rule R such that

(i) q/m occurs in the head of R, and
(ii) p/n occurs in a positive literal in the body of R.

If graph G_Γ is acyclic, then we will say that program Γ is *tight*.

Consider, for instance, the program $\Gamma_{r,n}$ (r and n are positive integers) that consists of the rules

$$\{in(1..\overline{n}, 1..\overline{r})\}, \tag{6.1}$$

$$covered(X) \leftarrow in(X,S), \tag{6.2}$$

$$\leftarrow X = 1..\overline{n} \wedge not\ \ covered(X), \tag{6.3}$$

$$\leftarrow in(X,S) \wedge in(Y,S) \wedge in(X+Y,S). \tag{6.4}$$

(The stable models of this program represent collections of r sum-free sets covering $\{1,\dots,n\}$; see `http://mathworld.wolfram.com/SchurNumber.html`.) The graph $G_{\Gamma_{r,n}}$ has one edge, from *covered*/1 to *in*/2, so that this program is tight.

The *vocabulary* of a program Γ is the set of atoms $p(\mathbf{r})$ such that $\mathbf{r}$ is a tuple of n precomputed terms, and p/n occurs in Γ. For other syntactic expressions, the vocabulary is defined in the same way.

Theorem 2
For any tight finite program Γ, an interpretation $\mathscr{I}$ is a stable model of Γ iff $\mathscr{I}$ is contained in the vocabulary of Γ and satisfies the completion of Γ.

The theorem shows, for instance, that the stable models of $\Gamma_{r,n}$ can be characterized as the subsets of its vocabulary that satisfy its completion.

7 Example

We will now calculate and simplify the completion of $\Gamma_{r,n}$. The formula representation of rule (6.1) is

$$V_1 \in \overline{1}..\overline{n} \wedge V_2 \in \overline{1}..\overline{r} \wedge in(V_1, V_2) \;\rightarrow\; in(V_1, V_2),$$

so that the completed definition of $in/2$ is

$$\forall V_1 V_2 (in(V_1, V_2) \leftrightarrow (V_1 \in \overline{1}..\overline{n} \wedge V_2 \in \overline{1}..\overline{r} \wedge in(V_1, V_2))).$$

This formula is equivalent to

$$\forall V_1 V_2 (in(V_1, V_2) \rightarrow (V_1 \in \overline{1}..\overline{n} \wedge V_2 \in \overline{1}..\overline{r})). \tag{7.1}$$

The formula representation of rule (6.2) can be written as

$$V = X \wedge in(X, S) \rightarrow covered(V).$$

It follows that the completed definition of $covered/1$ is

$$\forall V (covered(V) \leftrightarrow \exists X S (V = X \wedge in(X, S))),$$

which is equivalent to

$$\forall V (covered(V) \leftrightarrow \exists S\, in(V, S)). \tag{7.2}$$

The remaining two rules of the program are constraints. The universal closure of the formula representation of (6.3) is equivalent to

$$\forall X \neg (X \in \overline{1}..\overline{n} \wedge \neg covered(X)),$$

which can be further rewritten as

$$\forall X (X \in \overline{1}..\overline{n} \rightarrow covered(X)). \tag{7.3}$$

Finally, the universal closure of the formula representation of constraint (6.4) can be written as

$$\neg \exists X Y S (in(X, S) \wedge in(Y, S) \wedge \exists Z (Z \in X + Y \wedge in(Z, S))). \tag{7.4}$$

We showed that the completion of program $\Gamma_{r,n}$—its “engineered specification”—is equivalent to the conjunction of formulas (7.1)–(7.4).

8 Incorporating aggregates

8.1 Programs with aggregates

In addition to the four sets of symbols mentioned at the beginning of Section 2, we assume now that a set of *aggregate names* is selected, and for every aggregate name α

a function $\widehat{\alpha}$ is chosen that maps every set of non-empty tuples of precomputed terms to a precomputed term. Examples are as follows:

- aggregate name *count*; $\widehat{count}(T)$ is defined as the cardinality of T if T is finite, and *sup* otherwise;
- aggregate name *sum*; $\widehat{sum}(T)$ is the sum of the weights of all tuples in T if T contains finitely many tuples with non-zero weights, and $\overline{0}$ otherwise.

(The *weight* of a tuple $\mathbf{t}$ of precomputed terms is the first member of $\mathbf{t}$ if it is a numeral, and $\overline{0}$ otherwise.)

An *aggregate expression* is an expression of the form

$$\alpha\{\mathbf{t} : \mathbf{C}\} \prec s \tag{8.1}$$

where α is an aggregate name, $\mathbf{t}$ is a non-empty tuple of terms, $\mathbf{C}$ is a conjunction of literals and comparisons (in the case when $\mathbf{C}$ is empty, the preceding colon can be dropped), $\prec$ is a relation symbol, and s is a variable or precomputed term.

In the definition of a rule, the body is now allowed to have, among its conjunctive terms, not only literals and comparisons, but also aggregate expressions.

A variable V occurring in a rule R is *local* if every occurrence of V in R belongs to the left-hand side $\alpha\{\mathbf{t} : \mathbf{C}\}$ of one of the aggregate expressions (8.1) in its body, and *global* otherwise. For instance, in the rule

$$q(W) \leftarrow sum\{X^2 : p(X)\} = W \tag{8.2}$$

X is local and W is global.

8.2 Formulas with aggregates

The definitions of an argument and a formula in Section 3 are replaced now by a mutually recursive definition of both concepts. It includes clauses (a)–(f) from the old definition of a formula and three additional clauses:

(g) numerals, symbolic constants, variables, and the symbols *inf* and *sup* are arguments;
(h) if f is a symbolic constant and $\mathbf{arg}$ is a non-empty tuple of arguments, then $f(\mathbf{arg})$ is an argument;
(i) if α is an aggregate name, $\mathbf{X}$ is a non-empty tuple of distinct variables, and F is a formula, then $\alpha\{\mathbf{X} \mid F\}$ is an argument.

Clause (i) is what makes the new definition more general than the definitions from Section 3.

In this more general setting, the distinction between free and bound occurrences of variables applies not only to formulas, but also to arguments. An occurrence of a variable X in an argument or in a formula is *bound* if it belongs to a subformula of the form $\forall X F$, or if it belongs to a subargument $\alpha\{\mathbf{X} \mid F\}$ such that X is a member of the tuple $\mathbf{X}$. For example, in the argument

$$sum\{X \mid \exists Y\, p(X, Y, Z)\}$$

X and Y are bound, and Z is free. An argument or a formula is *closed* if all occurrences of variables in it are bound.

The substitution notation will be now applied not only to formulas, but also to arguments: arg^X_r is the argument obtained from an argument arg by substituting a precomputed term r for all free occurrences of a variable X. For every interpretation $\mathscr{I}$, the truth value $F^{\mathscr{I}}$ that $\mathscr{I}$ assigns to a closed formula F, and the precomputed term $arg^{\mathscr{I}}$ that $\mathscr{I}$ assigns to a closed argument arg, are described by a joint recursive definition[4]:

(a) $p(arg_1,\dots,arg_k)^{\mathscr{I}}$ is t if $p(arg_1^{\mathscr{I}},\dots,arg_k^{\mathscr{I}}) \in \mathscr{I}$,
(b) $(arg_1 \prec arg_2)^{\mathscr{I}}$ is t if $arg_1^{\mathscr{I}} \prec arg_2^{\mathscr{I}}$,
(c) $(arg \in t)^{\mathscr{I}}$ is t if $arg^{\mathscr{I}} \in [t]$,
(d) $\bot^{\mathscr{I}}$ is f,
(e) $(F \rightarrow G)^{\mathscr{I}}$ is f if $F^{\mathscr{I}}$ is t and $G^{\mathscr{I}}$ is f,
(f) $(\forall X F)^{\mathscr{I}}$ is t if, for every precomputed term r, $(F^X_r)^{\mathscr{I}}$ is t,
(g) if arg is a numeral, or a symbolic constant, or inf, or sup, then $arg^{\mathscr{I}}$ is arg,
(h) $f(arg_1,\dots,arg_k)^{\mathscr{I}}$ is $f(arg_1^{\mathscr{I}},\dots,arg_k^{\mathscr{I}})$,
(i) $\alpha\{X_1,\cdots,X_k \mid F\}^{\mathscr{I}}$ is $\widehat{\alpha}(T)$, where T is the set of all tuples $r_1,\dots,r_k$ of precomputed terms such that $(F^{X_1\cdots X_k}_{r_1\dots r_k})^{\mathscr{I}}$ is t.

Since an argument containing aggregate names is not a term, in this more general setting the statement of Observation 2 (Section 3) has to be modified:

Observation 2′. *For any arguments arg_1 and arg_2 such that arg_2 does not contain aggregate names, $arg_1 \in arg_2$ is equivalent to $arg_1 = arg_2$.*

8.3 Completion and tightness in the presence of aggregates

How do we turn an aggregate expression (8.1) into a formula? It depends on how we classify the variables occurring in this expression into local and global. For this reason, instead of extending the definition of ϕ from Section 4 to aggregate expressions, we will define the transformation $\phi^{\mathbf{X}}$, where $\mathbf{X}$ is a list (possibly empty) of distinct variables—those that we treat as local. The result of applying $\phi^{\mathbf{X}}$ to an aggregate expression (8.1) is the formula

$$\exists Y\,(\alpha\{\mathbf{Z} \mid \exists\mathbf{X}(\mathbf{Z} \in \mathbf{t} \wedge \phi\mathbf{C})\} \prec Y \wedge Y \in s),$$

where $\mathbf{Z}$ is a tuple of new variables of the same length as $\mathbf{t}$, and Y is a new variable.

Consider, for instance, the result of applying the transformation ϕ^X ("treat X as local") to the aggregate expression in the body of rule (8.2). It can be written as

$$\exists Y\,(sum\{Z \mid \exists X(Z \in X^2 \wedge p(X))\} = Y \wedge Y = W),$$

which is equivalent to

$$sum\{Z \mid \exists X(Z \in X^2 \wedge p(X))\} = W.$$

[4] This notation can be ambiguous, because some expressions can be viewed both as formulas and as arguments. But its meaning will be always clear from the context.

In application to literals and comparisons, $\phi^{\mathbf{X}}$ has the same meaning as ϕ. If each of the expressions $C_1, \dots, C_k$ is a literal, a comparison, or an aggregate expression, then $\phi^{\mathbf{X}}(C_1 \wedge \cdots \wedge C_k)$ stands for $\phi^{\mathbf{X}} C_1 \wedge \cdots \wedge \phi^{\mathbf{X}} C_k$.

Now we are ready to state how the definitions (4.2), (4.6), and (4.7) of formula representations of rules are modified in the presence of aggregates. In all three definitions, we replace $\phi(Body)$ by $\phi^{\mathbf{X}}(Body)$, where $\mathbf{X}$ is the list of local variables of the rule. For instance, the formula representation of rule (8.2) can be written as

$$V = W \wedge sum\{Z \mid \exists X(Z \in X^2 \wedge p(X))\} = W \rightarrow q(V).$$

All definitions from Section 5, including the definition of the completion of a finite program, remain the same. It is easy to see that in formula (5.2), $\mathbf{U}_i$ is the list of global variables of rule R_i.

In the definition of G_Γ (Section 6), clause (ii) is restated as follows:

(ii′) p/n occurs in a positive literal or in an aggregate expression in the body of R.

For example, if Γ is the one-rule program (8.2), then G_Γ has an edge from $q/1$ to $p/1$. Otherwise, the definition of a tight program remains the same.

8.4 *Example:* 8-*Queens*

The following program with aggregates encodes a solution to the problem of how to place 8 queens on an 8×8 chessboard so that no two queens attack each other.

$$row(\overline{1}..\overline{8}), \tag{8.3}$$

$$col(\overline{1}..\overline{8}), \tag{8.4}$$

$$\{queen(X, Y)\} \leftarrow col(X) \wedge row(Y), \tag{8.5}$$

$$\leftarrow count\{X, Y : queen(X, Y)\} \neq \overline{8}, \tag{8.6}$$

$$\leftarrow queen(X, Y) \wedge queen(X, YY) \wedge Y \neq YY, \tag{8.7}$$

$$\leftarrow queen(X, Y) \wedge queen(XX, Y) \wedge X \neq XX, \tag{8.8}$$

$$\leftarrow queen(X, Y) \wedge queen(XX, YY) \wedge X \neq XX \wedge |X - XX| = |Y - YY|. \tag{8.9}$$

The formula representation of rule (8.3) is $V \in \overline{1}..\overline{8} \rightarrow row(V)$, so that the completed definition of $row/1$ is

$$\forall V(row(V) \leftrightarrow V \in \overline{1}..\overline{8}). \tag{8.10}$$

Similarly, the completed definition of $col/1$ is

$$\forall V(col(V) \leftrightarrow V \in \overline{1}..\overline{8}). \tag{8.11}$$

The formula representation of (8.5) can be rewritten, after simplifying the antecedent, as

$$V_1 = X \wedge V_2 = Y \wedge col(X) \wedge row(Y) \wedge queen(V_1, V_2) \rightarrow queen(V_1, V_2).$$

Consequently, the completed definition of $queen/2$ is

$$\forall V_1 V_2(queen(V_1, V_2) \leftrightarrow \exists X Y(V_1 = X \wedge V_2 = Y \wedge col(X) \wedge row(Y) \wedge queen(V_1, V_2))).$$

This formula is equivalent to

$$\forall V_1 V_2(queen(V_1, V_2) \rightarrow col(V_1) \wedge row(V_2)). \tag{8.12}$$

Variables X and Y are local in constraint (8.6), so that its formula representation can be written as

$$\exists Y_1(count\{Z_1, Z_2 \mid \exists XY(Z_1 \in X \wedge Z_2 \in Y \wedge queen(X, Y))\} \neq Y_1 \wedge Y_1 = \overline{8}) \rightarrow \bot,$$

or, equivalently,

$$count\{Z_1, Z_2 \mid queen(Z_1, Z_2)\} = \overline{8}. \tag{8.13}$$

The formula representations of constraints (8.7)–(8.9) can be written as

$$\begin{aligned}&queen(X, Y) \wedge queen(X, YY) \rightarrow Y = YY,\\ &queen(X, Y) \wedge queen(XX, Y) \rightarrow X = XX,\\ &queen(X, Y) \wedge queen(XX, YY) \wedge |X - XX| = |Y - YY| \rightarrow X = XX.\end{aligned} \tag{8.14}$$

The completion of program (8.3)–(8.9) consists of formulas (8.10)–(8.13) and the universal closures of formulas (8.14). This set of formulas is an "engineered specification" for that program.

9 Integer variables

We will now make the definition of formulas and arguments more general. We assume here that the set of variables is partitioned into two classes, *general variables* and *integer variables*. General variables are variables for precomputed terms; integer variables are variables for numerals. Formulas without general variables are similar to formulas of first-order arithmetic. In examples, integer variables will be represented by identifiers that start with I, J, K, L, M, and N.

Integer arguments are defined recursively:

- numerals and integer variables are integer arguments,
- if op is an n-ary operation name such that the domain of the corresponding function $\widehat{op}$ is the whole set $\mathbf{Z}^n$, and $\mathbf{arg}$ is an n-tuple of integer arguments, then $op(\mathbf{arg})$ is an integer argument.

Clause (g) in the definition of formulas and arguments (Section 8.2) is reformulated as follows:

(g) integer arguments, symbolic constants, general variables, inf, and sup are arguments.

For example, since N is an integer variable, the expression $N+\overline{1}$ is not only a term, but also an argument, and both $p(N+\overline{1})$ and $N+\overline{1} = \overline{4}$ are formulas.

To extend the definition of the semantics of formulas and arguments given in Section 8.2, we restrict clause (f) in that definition to the case when X is a general variable, and add two clauses:

(f′) $(\forall N F)^{\mathscr{I}}$, where N is an integer variable, is t if, for every integer n, $(F^X_{\overline{n}})^{\mathscr{I}}$ is t;
(h′) if arg is $op(arg_1, \dots, arg_k)$, $arg_1^{\mathscr{I}} = \overline{n_1}, \dots, arg_k^{\mathscr{I}} = \overline{n_k}$, then $arg^{\mathscr{I}}$ is $\overline{\widehat{op}(n_1, \dots, n_k)}$.

The following abbreviations will be useful. For any argument *arg*, by int(*arg*) we denote the formula $\exists V(V \in arg+\overline{1})$, where V is a general variable that does not occur in *arg*. For any predicate symbol p/n, by int(p/n), we denote the formula

$$\forall X_1 \dots X_n(p(X_1,\dots,X_n) \rightarrow \text{int}(X_1) \wedge \dots \wedge \text{int}(X_n)),$$

where $X_1,\dots,X_n$ are distinct general variables. This formula expresses that the extent of the predicate p/n is a subset of $\mathbf{Z}^n$.

Using integer variables, we can rewrite formula (5.3) as

$$\begin{array}{l}\text{int}(p/1),\\ \forall N(p(N) \leftrightarrow \overline{1} \leqslant N \leqslant \overline{8}).\end{array}$$

Formula (5.4) can be transformed in a similar way.

Formula (5.5) can be rewritten as

$$\begin{array}{l}\text{int}(q/1),\\ \forall N(q(N) \leftrightarrow \exists M(N = M+\overline{1} \wedge p(M) \wedge \overline{1} \leqslant M \leqslant \overline{8})).\end{array}$$

The last formula can be simplified as follows:

$$\forall N(q(N) \leftrightarrow p(N-\overline{1}) \wedge \overline{2} \leqslant N \leqslant \overline{9}).$$

Formula (7.1) is equivalent to

$$\begin{array}{l}\text{int}(in/2),\\ \forall IK(in(I,K) \rightarrow \overline{1} \leqslant I \leqslant \overline{n} \wedge \overline{1} \leqslant K \leqslant \overline{r}).\end{array}$$

Formula (7.3) is equivalent to

$$\forall I(\overline{1} \leqslant I \leqslant \overline{n} \rightarrow covered(I)).$$

Formula (7.4) can be equivalently rewritten, in the presence of int($in/2$), as

$$\neg\exists IJS(in(I,S) \wedge in(J,S) \wedge \exists K(K = I+J \wedge in(K,S))),$$

and consequently as

$$\forall IJS(in(I,S) \wedge in(J,S) \rightarrow \neg in(I+J,S)).$$

In the presence of completed definitions (8.10)–(8.12), all variables in (8.13) and in the universal closures of (8.14) can be equivalently replaced by integer variables.

10 Conclusion

This paper extends familiar results on the relationship between stable models and program completion to a large class of programs in the input language of GRINGO, and we hope that this technical contribution will help us apply formal methods to answer set programming. Much still remains to be done.

First, we would like to extend the main result of this paper, Theorem 2 from Section 6, in several directions. Including edges from head to aggregate expressions in graph G_Γ [condition (ii′) in Section 8.3] may be unnecessary when the aggregates are known to be monotone or antimonotone (Harrison *et al.* 2014, Section 6.1).

Further, a dependency graph with atoms from the program's vocabulary as its vertices, rather than predicate symbols, may be useful. Finally, we would like to adapt the definition of completion to a class of "almost tight" programs that may contain simple recursive definitions (such as the definition of reachability in a graph). It may be possible to achieve this at the price of allowing the least fixed point operator (Gurevich and Shelah 1986) in completed definitions.

Second, the process of generating and simplifying completed definitions needs to be automated. In some cases, programmers may be able to convince themselves that a program is correct—or to decide that it is not—by examining its simplified completion. Sometimes automated reasoning tools may help them establish a correspondence between a given specification and the completion of the program. These are themes of an ongoing project[5] at the University of Potsdam, the home of GRINGO.

Acknowledgements

Many thanks to Michael Gelfond, Joohyung Lee, Patrick Lühne, and Torsten Schaub for useful discussions related to the topic of this paper, and to the anonymous referees for their comments.

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S1471068417000394

References

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming 3*, 499–518.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proc. 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. and Lifschitz, V. 2010. The stable model semantics for first-order formulas with aggregates. In *Proc. of International Workshop on Nonmonotonic Reasoning (NMR)* `http://www.kr.org/NMR/proceedings.html`.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming 15*, 449–463.

[5] `https://github.com/potassco/anthem/`

Gurevich, Y. and Shelah, S. 1986. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic 32*, 265–280.

Harrison, A., Lifschitz, V. and Yang, F. 2014. The semantics of Gringo and infinitary propositional formulas. In *Proc. of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 32–41.

Lee, J. and Meng, Y. 2009. On reductive semantics of aggregates in answer set programming. In *Proc. of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 182–195.

Lee, J. and Meng, Y. 2012. Stable models of formulas with generalized quantifiers. In *Working Notes of the 14th International Workshop on Non-Monotonic Reasoning (NMR)*.

Truszczynski, M. 2012. Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler and D. Pearce, Eds. Springer, 543–559.

Appendix A. Semantics of programs

Gebser *et al.* (2015) showed that stable models of many programs in the input language of GRINGO can be described in terms of stable models of infinitary propositional formulas. That approach is applied here to programs in the sense of Section 8.1; we will call them *EG programs* (for “Essential GRINGO”).

The translation τ, defined below, transforms every EG program Γ into an infinitary formula over the vocabulary of Γ. Stable models of Γ are defined as stable models of $\tau\Gamma$[6].

A.1 Semantics of ground terms

The set $[t]$ of precomputed terms denoted by a ground term t is defined recursively:

- if t is a numeral, a symbolic constant, or one of the symbols *inf*, *sup*, then $[t]$ is $\{t\}$;
- if t is $f(t_1,\dots,t_n)$, where f is a symbolic constant, then $[t]$ consists of the terms $f(r_1,\dots,r_n)$ for all $r_1 \in [t_1],\dots,r_n \in [t_n]$;
- if t is $op(t_1,\dots,t_n)$ where op is an operation name, then $[t]$ consists of the numerals of the form $\overline{\widehat{op}(k_1,\dots,k_n)}$ for all tuples $k_1,\dots,k_n$ in the domain of $\widehat{op}$ such that $\overline{k_1} \in [t_1],\dots, \overline{k_n} \in [t_n]$;
- if t is $(t_1\,..\,t_2)$, then $[t]$ consists of the numerals $\overline{m}$ for all integers m such that, for some integers k_1,k_2,

$$\overline{k_1} \in [t_1], \qquad \overline{k_2} \in [t_2], \qquad k_1 \leqslant m \leqslant k_2.$$

For any ground terms $t_1\dots,t_n$, $[t_1,\dots,t_n]$ is the set of tuples $r_1,\dots,r_n$ for all $r_1 \in [t_1],\dots,\ r_n \in [t_n]$.

[6] The stable model semantics of infinitary formulas by Truszczynski (2012) and Gebser *et al.* (2015, Section 4.1) is a straightforward generalization of the definition due to Ferraris (2005).

A.2 Transforming programs into infinitary formulas

For any ground atom $p(\mathbf{t})$, $\tau p(\mathbf{t})$ stands for $\bigvee_{\mathbf{r}\in[\mathbf{t}]} p(\mathbf{r})$, and $\tau(not\ \ p(\mathbf{t}))$ stands for $\bigvee_{\mathbf{r}\in[\mathbf{t}]} \neg p(\mathbf{r})$.

For any ground comparison $t_1 \prec t_2$, $\tau(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between some terms r_1, r_2 such that $r_1 \in [t_1]$ and $r_2 \in [t_2]$, and $\bot$ otherwise.

If each of $C_1, \dots, C_k$ is a ground literal or a ground comparison, then $\tau(C_1 \wedge \cdots \wedge C_k)$ stands for $\tau C_1 \wedge \cdots \wedge \tau C_k$.

An aggregate expression (8.1) is *closed* if the term s is ground. Let $\mathbf{X}$ be the list of variables occurring in a closed aggregate expression (8.1), and let A be the set of tuples $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{X}$. Let Δ be a subset of A. By $[\Delta]$, we denote the union of the sets $[\mathbf{t}^{\mathbf{X}}_{\mathbf{r}}]$ for all tuples of precomputed terms $\mathbf{r}$ in Δ. We say that Δ *justifies* the aggregate expression (8.1) if the relation $\prec$ holds between $\widehat{\alpha}[\Delta]$ and an element of the set s. We define the result of applying τ to (8.1) as the conjunction of the implications

$$\bigwedge_{\mathbf{r}\in\Delta} \tau(\mathbf{C}^{\mathbf{X}}_{\mathbf{r}}) \to \bigvee_{\mathbf{r}\in A\setminus\Delta} \tau(\mathbf{C}^{\mathbf{X}}_{\mathbf{r}}) \tag{A.1}$$

over all subsets Δ of A that do not justify (8.1).

The definition of τ for conjunctions of ground literals and ground comparisons extends in the obvious way to the case when some conjunctive terms are closed aggregate expressions.

A rule is *closed* if all its variables are local. If R is a closed basic rule (4.1), then τR is the formula

$$\tau(Body) \to \bigwedge_{\mathbf{r}\in[\mathbf{t}]} p(\mathbf{r}). \tag{A.2}$$

If R is a closed choice rule (4.5), then τR is the formula

$$\tau(Body) \to \bigwedge_{\mathbf{r}\in[\mathbf{t}]} (p(\mathbf{r}) \vee \neg p(\mathbf{r})). \tag{A.3}$$

If R is a closed constraint $\leftarrow Body$, then τR is $\neg\tau(Body)$.

An *instance* of a rule is a closed rule obtained from it by substituting precomputed terms for its global variables. For any EG program Γ, $\tau\Gamma$ is the conjunction of the formulas τR for all instances R of the rules of Γ.