Program Completion
in the Input Language of GRINGO

AMELIA HARRISON, VLADIMIR LIFSCHITZ
DHANANJAY RAJU
University of Texas
(e-mail: ameliaj,vl,draju@cs.utexas.edu)

submitted 2 May 2017; revised 20 June 2017; accepted 4 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 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.

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 for-
mal 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

\[
r(X) \leftarrow p(X), \\
r(X) \leftarrow q(X).
\]

The corresponding completed definition

\[
\forall X (r(X) \leftrightarrow p(X) \lor q(X))
\]

is the usual definition of union in set theory. Turning program (1) into a completed definition

gives us a plausible specification that could have led to this program in the first place. The stable
model semantics of program (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, \ldots, 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

\[ X \ldots X + 1 = Y \ldots Y + 1, \]

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 \(\text{sum}\{X \times Y : p(X,Y)\}\) 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.

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.

---

\(^1\) 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 \times Y : p(X,Y)\}.
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

= ≠ < > ≤ ≥

and the symbols

inf sup not ∧ ∨ ←

, :, ( ) { }

∈ ¬ ∧ ∨ → ↔ ∀ ∃

We assume that a 1–1 correspondence between the set of numerals and the set \( \mathbb{Z} \) of integers is chosen. For every integer \( n \), the corresponding numeral will be denoted by \( \bar{n} \). 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 \( \hat{op} \) from a subset of \( \mathbb{Z}^n \) to \( \mathbb{Z} \) is chosen, where \( n \) is the arity of \( op \). For instance, we can choose \( plus \) as a binary operation name, define \( \hat{plus} \) 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 \( t \) is a non-empty tuple of terms (separated by commas) then \( f(t) \) is a term,
- if \( op \) is an \( n \)-ary operation name and \( t \) is an \( n \)-tuple of terms then \( op(t) \) 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 Section A.1, every ground term \( t \) denotes a finite set \( [t] \) of precomputed terms, which are called the values of \( t \). For instance,

\[
[8] = \{8\}, \quad [1..8] = \{1,\ldots,8\}, \quad [abc + 1] = \emptyset
\]

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 \( n \), \( \bar{m} \leq \bar{n} \) iff \( m \leq n \).

Atoms are expressions of the form \( p(t) \), where \( p \) is a symbolic constant and \( t \) 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 \( (t_1 ≺ t_2) \) where \( t_1, t_2 \) are terms and \( ≺ \) 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

\[
\text{Head} \leftarrow \text{Body}
\]

where

- \( \text{Body} \) is a conjunction (possibly empty) of literals and comparisons, and
Head is either an atom (then we say that (2) is a basic rule), or a choice expression (then (2) is a choice rule), or empty (then (2) 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 $p(t)$ where $t$ 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. Formulas are defined recursively:

(a) if $p$ is a symbolic constant and $\text{arg}$ is a tuple of arguments then $p(\text{arg})$ is a formula,
(b) if $\text{arg}_1$ and $\text{arg}_2$ are arguments and $\prec$ is a relation symbol then $(\text{arg}_1 \prec \text{arg}_2)$ is a formula,
(c) if $\text{arg}$ is an argument and $t$ is a term then $\text{arg} \in t$ is a formula,
(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, $X \in \text{T..8}$ and $X \in Y + \text{T}$ are formulas, but $X = \text{T..8}$ and $X = Y + \text{T}$ are not. The reason why we do not allow $Y + \text{T}$ 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 $F^\mathcal{I}$, assigned by an interpretation $\mathcal{I}$ to a closed formula $F$, is defined as $t$ or $f$, in accordance with the following rules:

(a) $p(\text{arg})^\mathcal{I}$ is $t$ if $p(\text{arg}) \in \mathcal{I}$ (and $f$ otherwise),
(b) $(\text{arg}_1 \prec \text{arg}_2)^\mathcal{I}$ is $t$ if $\text{arg}_1 \prec \text{arg}_2$,
(c) $(\text{arg} \in t)^\mathcal{I}$ is $t$ if $\text{arg} \in [t]$,
(d) $\bot^\mathcal{I}$ is $f$,
(e) $(F \rightarrow G)^\mathcal{I}$ is $f$ if $F^\mathcal{I}$ is $t$ and $G^\mathcal{I}$ is $f$,
(f) $(\forall X F)^\mathcal{I}$ is $t$ if, for every precomputed term $r$, $(F^X_r)^\mathcal{I}$ is $t$.

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

For example, the interpretation $\{p(2), p(3), p(4)\}$ satisfies the formula $\exists X (p(X) \land X \in \text{T..8})$. Indeed, it satisfies $p(3)$, because it includes $p(3)$; it also satisfies $3 \in \text{T..8}$, because $[\text{T..8}]$ is

---

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.
(T,...,8), and 3 is an element of this set. Consequently, it satisfies the conjunction \( p(3) \land 3 \in \bar{T},\bar{8} \).

A formula is universally valid if its universal closure is satisfied by all interpretations. A formula \( F \) is equivalent to a formula \( G \) if \( F \equiv G \) 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 \( \text{arg} \) and any ground term \( t \), \( \text{arg} \in t \) is equivalent to \( \forall r \in [t](\text{arg} = r) \).

This is immediate from the definition of satisfaction.

For example, for any integers \( m \) and \( n \), \( \text{arg} \in [m..n] \) is equivalent to \( \forall i = m..n(\text{arg} = i) \).

**Observation 2.** For any arguments \( \text{arg}_1 \) and \( \text{arg}_2 \), \( \text{arg}_1 \in \text{arg}_2 \) is equivalent to \( \text{arg}_1 = \text{arg}_2 \).

It is sufficient to check this claim for the case when \( \text{arg}_1, \text{arg}_2 \) are ground. In this case, it follows from the fact that \( \{\text{arg}_2\} \) is the singleton \( \{\text{arg}_2\} \).

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

### 4 Representing Rules by Formulas

In this section we define a syntactic transformation \( \phi \) that turns rules and their subexpressions into formulas—their formula representations.

Formula representations of literals and comparisons are defined as follows:

- \( \phi p(t) \) is \( \exists X(X \in t \land p(X)) \).
- \( \phi \) (not \( p(t) \)) is \( \exists X(X \in t \land \neg p(X)) \).
- \( \phi (t_1 \prec t_2) \) is \( \exists X_1,X_2(X_1 \in t_1 \land X_2 \in t_2 \land X_1 \prec X_2) \);

here \( X \) is a tuple of new variables of the same length as \( t \), and \( X_1,X_2 \) are new variables.

For example, the transformation \( \phi \) turns \( p(X) \) into \( \exists Y(Y \in X \land p(Y)) \); this formula is equivalent to \( \exists Y(Y = X \land p(Y)) \), and consequently to \( p(X) \). The formula representation of \( p(\bar{T},X) \) is \( \exists Y(Y \in \bar{T},X \land p(Y)) \). The representation of \( X = \bar{T},\bar{8} \) is

\[
\exists X_1,X_2(X_1 \in X \land X_2 \in \bar{T},\bar{8} \land X_1 = X_2);
\]

this formula is equivalent to \( X \in \bar{T},\bar{8} \).

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

The formula representation of a basic rule

\[
p(t) \leftarrow \text{Body}
\]

is defined as the implication

\[
\forall V \in t \land \phi(Body) \rightarrow p(V),
\]

where \( V \) is a tuple of new variables of the same length as \( t \). For example, the formula representation of the rule

\[
q(X+T) \leftarrow p(X) \land X = \bar{T},\bar{8}
\]

is

\[
p(t) \leftarrow \text{Body}
\]

3 If \( X \) is \( X_1,...,X_n \), and \( t \) is \( t_1,...,t_n \), then \( X \in t \) stands for the conjunction \( \bigwedge_{i=1}^{n} X_i \in t_i \).
is
\[ V \in X + \bar{T} \land \phi p(X) \land \phi (X = \bar{T}..\bar{S}) \to q(V); \]

after applying equivalent transformations to the antecedent, this formula becomes
\[ V \in X + \bar{T} \land p(X) \land X \in \bar{T}..\bar{S} \to q(V). \quad (6) \]

The formula representation of a choice rule
\[ \{ p(t) \} \leftarrow \text{Body} \quad (7) \]
is defined as the (universally valid) formula
\[ V \in t \land \phi (\text{Body}) \land p(V) \to p(V), \quad (8) \]
where \( V \) is a tuple of new variables of the same length as \( t \).

For example, the formula representation of the rule \( \{ p(1..8) \} \) is
\[ V \in 1..8 \land p(V) \to p(V). \]

The formula representation of a constraint \( \leftarrow \text{Body} \) is the formula
\[ \neg \phi (\text{Body}). \quad (9) \]

5 Completion

A predicate symbol is a pair \( p/n \), where \( p \) is a symbolic constant and \( n \) is a nonnegative integer. The definition of a predicate symbol \( p/n \) in a program \( \Gamma \) consists of
- the basic rules of \( \Gamma \) with the head of the form \( p(t_1, \ldots, t_n) \), and
- the choice rules of \( \Gamma \) with the head of the form \( \{ p(t_1, \ldots, t_n) \} \).

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 \( \Gamma \) is \( \{ R_1, \ldots, R_k \} \) then each of the formulas \( \phi R_i \) has the form
\[ F_i \to p(V), \quad (10) \]
where \( V \) 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 \( \Gamma \) is the formula
\[ \forall V \left( p(V) \leftrightarrow \bigvee_{i=1}^k \exists U_i F_i \right), \quad (11) \]
where \( U_i \) is the list of all free variables of the formula \( F_i \) that do not belong to \( V \).

For example if the definition of \( p/1 \) in \( \Gamma \) is \( p(1..8) \) then \( k = 1 \), \( U_1 \) is empty, and \( F_1 \) is \( V \in 1..8 \), so that the completed definition of \( p/1 \) is
\[ \forall V (p(V) \leftrightarrow V \in 1..8). \quad (12) \]

If the definition of \( p/1 \) is the choice rule \( \{ p(1..8) \} \) then \( F_1 \) is
\[ V \in 1..8 \land p(V), \]
and the completed definition of \( p/1 \) is
\[ \forall V (p(V) \leftrightarrow V \in 1..8 \land p(V)). \]
This formula is equivalent to
\[ \forall V(p(V) \rightarrow V \in \bar{T} \bar{S}). \]  
(13)

It is clear that completed definitions are invariant with respect to equivalent transformations of the antecedents of implications \( \phi R_i \), in the sense that replacing an antecedent \( F_i \) in (11) by an equivalent formula is an equivalent transformation. Assume, for instance, that the definition of \( q/1 \) in \( \Gamma \) is (5). Formula (6) 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
\[ \forall V(q(V) \leftrightarrow \exists X(V \in X + T \land p(X) \land X \in \bar{T} \bar{S})). \]  
(14)

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 \( p(t_1, \ldots, t_n) \). The completion of a finite program \( \Gamma \) consists of
- the completed definitions of all predicate symbols occurring in \( \Gamma \), and
- the universal closures of the formula representations of all constraints in \( \Gamma \).

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 \( \Gamma \), by \( G_\Gamma \) we denote the directed graph that has the predicate symbols occurring in \( \Gamma \) as its vertices, and has an edge from \( q/m \) to \( p/n \) if \( \Gamma \) 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 \( G_\Gamma \) is acyclic then we will say that program \( \Gamma \) is **tight**.

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

\[ \{ \text{in}(1, \bar{1}, 1, \bar{r}) \}, \]  
(15)

\[ \text{covered}(X) \leftarrow \text{in}(X, S), \]  
(16)

\[ \leftarrow X = 1, \bar{1} \land \text{not covered}(X), \]  
(17)

\[ \leftarrow \text{in}(X, S) \land \text{in}(Y, S) \land \text{in}(X + Y, S). \]  
(18)

(The stable models of this program represent collections of \( r \) sum-free sets covering \( \{1, \ldots, n\} \); see [http://mathworld.wolfram.com/SchurNumber.html](http://mathworld.wolfram.com/SchurNumber.html). The graph \( G_{\Gamma_{r,n}} \) has one edge, from \( \text{covered}/1 \) to \( \text{in}/2 \), so that this program is tight. The **vocabulary** of a program \( \Gamma \), is the set of atoms \( p(r) \) such that \( r \) is a tuple of \( n \) precomputed terms, and \( p/n \) occurs in \( \Gamma \). For other syntactic expressions the vocabulary is defined in the same way.
Theorem 2
For any tight finite program $\Gamma$, an interpretation $\mathcal{I}$ is a stable model of $\Gamma$ iff $\mathcal{I}$ is contained in the vocabulary of $\Gamma$ and satisfies the completion of $\Gamma$.

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 (15) is

$$V_1 \notin T..r \land V_2 \in T..r \land \text{in}(V_1, V_2) \rightarrow \text{in}(V_1, V_2),$$

so that the completed definition of $\text{in}/2$ is

$$\forall V_1 V_2 (\text{in}(V_1, V_2) \leftrightarrow (V_1 \in T..r \land V_2 \in T..r \land \text{in}(V_1, V_2))).$$

This formula is equivalent to

$$\forall V_1 V_2 (\text{in}(V_1, V_2) \rightarrow (V_1 \in T..r \land V_2 \in T..r)).$$

(19)

The formula representation of rule (16) can be written as

$$V = X \land \text{in}(X, S) \rightarrow \text{covered}(V).$$

It follows that the completed definition of $\text{covered}/1$ is

$$\forall V (\text{covered}(V) \leftrightarrow \exists S (V = X \land \text{in}(X, S))),$$

which is equivalent to

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

(20)

The remaining two rules of the program are constraints. The universal closure of the formula representation of (17) is equivalent to

$$\forall X \neg (X \in T..r \land \neg \text{covered}(X)),$$

which can be further rewritten as

$$\forall X (X \in T..r \rightarrow \text{covered}(X)).$$

(21)

Finally, the universal closure of the formula representation of constraint (18) can be written as

$$\neg \exists X Y S (\text{in}(X, S) \land \text{in}(Y, S) \land \exists Z (Z \in X + Y \land \text{in}(Z, S))).$$

(22)

We showed that the completion of program $\Gamma_{r,n}$—its “engineered specification”—is equivalent to the conjunction of formulas (19)–(22).

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 $\alpha$ a function $\hat{\alpha}$ is chosen that maps every set of non-empty tuples of precomputed terms to a precomputed term. Examples:
• aggregate name \textit{count}; \textit{count}(T) is defined as the cardinality of \( T \) if \( T \) is finite, and \textit{sup} otherwise;

• aggregate name \textit{sum}; \textit{sum}(T) is the sum of the weights of all tuples in \( T \) if \( T \) contains finitely many tuples with non-zero weights, and \( 0 \) otherwise.

(The weight of a tuple \( t \) of precomputed terms is the first member of \( t \) if it is a numeral, and \( 0 \) otherwise.)

An aggregate expression is an expression of the form

\[ \alpha \{ t : C \} \prec s \]  

where \( \alpha \) is an aggregate name, \( t \) is a non-empty tuple of terms, \( C \) is a conjunction of literals and comparisons (in the case when \( C \) is empty the preceding colon can be dropped), \( \prec \) is a relation symbol, and \( s \) is a 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 \( \alpha \{ t : C \} \) of one of the aggregate expressions (23) in its body, and global otherwise. For instance, in the rule

\[ q(W) \leftarrow \text{sum}(X^2 : p(X)) = W \]  

\( X \) is local and \( W \) is global.

\section*{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 \textit{inf} and \textit{sup} are arguments;

(h) if \( f \) is a symbolic constant and \textit{arg} is a non-empty tuple of arguments then \( f(\textit{arg}) \) is an argument;

(i) if \( \alpha \) is an aggregate name, \( X \) is a non-empty tuple of distinct variables, and \( F \) is a formula, then \( \alpha \{ X \mid F \} \) 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 \( \alpha \{ X \mid F \} \) such that \( X \) is a member of the tuple \( X \). For example, in the argument

\[ \text{sum}(X \mid \exists Y p(X, Y, Z)) \]

\( X \) 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 \( \mathcal{I} \), the truth value \( F^\mathcal{I} \) that \( \mathcal{I} \) assigns to a
closed formula \( F \), and the precomputed term \( \text{arg}^F \) that \( \mathcal{F} \) assigns to a closed argument \( \text{arg} \), are described by a joint recursive definition:\(^4\)

(a) \( p(\text{arg}_1, \ldots, \text{arg}_k)^F \) is t if \( p(\text{arg}_1^F, \ldots, \text{arg}_k^F) \in \mathcal{F} \),
(b) \( \text{arg}_1 \prec \text{arg}_2)^F \) is t if \( \text{arg}_1^F \prec \text{arg}_2^F \),
(c) \( (\text{arg} \in t)^F \) is t if \( \text{arg}^F \in [t] \),
(d) \( \bot^F \) is f,
(e) \( (F \rightarrow G)^F \) is f if \( F^F \) is t and \( G^F \) is f,
(f) \( (\forall X F)^F \) is t if, for every precomputed term \( r, (F^X)^F \) is t.
(g) if \( \text{arg} \) is a numeral, or a symbolic constant, or \( \text{inf} \), or \( \text{sup} \), then \( \varphi^F \) is \( \text{arg} \);
(h) \( f(\text{arg}_1, \ldots, \text{arg}_k)^F \) is \( f(\text{arg}_1^F, \ldots, \text{arg}_k^F) \),
(i) \( \alpha(X_1, \ldots, X_k \{ F \})^F \) is \( \alpha(T) \), where \( T \) is the set of all tuples \( r_1, \ldots, r_k \) of precomputed terms such that \( (F^X_{r_1,\ldots,r_k})^F \) 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 \( \text{arg}_1 \) and \( \text{arg}_2 \) such that \( \text{arg}_2 \) does not contain aggregate names, \( \text{arg}_1 \in \text{arg}_2 \) is equivalent to \( \text{arg}_1 = \text{arg}_2 \).

### 8.3 Completion and Tightness in the Presence of Aggregates

How do we turn an aggregate expression (23) 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 \( \phi \) from Section 4 to aggregate expressions, we will define the transformation \( \phi^X \), where \( X \) is a list (possibly empty) of distinct variables—those that we treat as local. The result of applying \( \phi^X \) to an aggregate expression (23) is the formula

\[
\exists Y (\alpha(Z) \exists X (Z \in t \land \phi(C))) \prec Y \land Y \in s,
\]

where \( Z \) is a tuple of new variables of the same length as \( t \), and \( Y \) is a new variable.

Consider, for instance, the result of applying the transformation \( \phi^X \) (“treat \( X \) as local”) to the aggregate expression in the body of rule (24). It can be written as

\[
\exists Y (\text{sum}\{Z \mid \exists X (Z \in X^2 \land p(X))\} = Y \land Y = W),
\]

which is equivalent to

\[
\text{sum}\{Z \mid \exists X (Z \in X^2 \land p(X))\} = W.
\]

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

Now we are ready to state how the definitions (4), (8), and (9) of formula representations of rules are modified in the presence of aggregates. In all three definitions, we replace \( \phi(\text{Body}) \) by \( \phi^X(\text{Body}) \), where \( X \) is the list of local variables of the rule. For instance, the formula representation of rule (24) can be written as

\[
V = W \land \text{sum}\{Z \mid \exists X (Z \in X^2 \land p(X))\} = W \rightarrow q(V).
\]

\(^4\) 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.
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 (11), $\mathcal{U}_i$ is the list of global variables of rule $R_i$.

In the definition of $G_T$ (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 $\Gamma$ is the one-rule program (24) then $G_T$ 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 $8 \times 8$ chessboard so that no two queens attack each other.

\begin{align*}
\text{row} & (\overline{1}..\overline{8}), \quad (25) \\
\text{col} & (T..\overline{8}), \quad (26) \\
\{\text{queen}(X,Y)\} & \leftarrow \text{col}(X) \land \text{row}(Y), \quad (27) \\
\leftarrow & \text{count}(X,Y : \text{queen}(X,Y)) \notin \overline{8}, \quad (28) \\
\leftarrow & \text{queen}(X,Y) \land \text{queen}(X,Y) \land Y \neq YY, \quad (29) \\
\leftarrow & \text{queen}(X,Y) \land \text{queen}(XX,Y) \land X \neq XX, \quad (30) \\
\leftarrow & \text{queen}(X,Y) \land \text{queen}(XX,YY) \land X \neq XX \land |X - XX| = |Y - YY|. \quad (31)
\end{align*}

The formula representation of rule (25) is $V \in T..\overline{8} \rightarrow \text{row}(V)$, so that the completed definition of row/1 is

\[ \forall V (\text{row}(V) \leftrightarrow V \in T..\overline{8}). \quad (32) \]

Similarly, the completed definition of col/1 is

\[ \forall V (\text{col}(V) \leftrightarrow V \in T..\overline{8}). \quad (33) \]

The formula representation of (27) can be rewritten, after simplifying the antecedent, as

\[ V_1 = X \land V_2 = Y \land \text{col}(X) \land \text{row}(Y) \land \text{queen}(V_1,V_2) \rightarrow \text{queen}(V_1,V_2). \]

Consequently, the completed definition of queen/2 is

\[ \forall V_1V_2(\text{queen}(V_1,V_2) \leftrightarrow \exists XY(V_1 = X \land V_2 = Y \land \text{col}(X) \land \text{row}(Y) \land \text{queen}(V_1,V_2))). \]

This formula is equivalent to

\[ \forall V_1V_2(\text{queen}(V_1,V_2) \rightarrow \text{col}(V_1) \land \text{row}(V_2)). \quad (34) \]

Variables $X$ and $Y$ are local in constraint (28), so that its formula representation can be written as

\[ \exists Y_1(\text{count}[Z_1,Z_2 | \exists XY(Z_1 \in X \land Z_2 \in Y \land \text{queen}(X,Y))] \neq Y_1 \land Y_1 = \overline{8}) \rightarrow \bot, \]

or, equivalently,

\[ \text{count}[Z_1,Z_2 | \text{queen}(Z_1,Z_2)] = \overline{8}. \quad (35) \]

The formula representations of constraints (29)–(31) can be written as

\[ \text{queen}(X,Y) \land \text{queen}(X,YY) \rightarrow Y = YY, \]
\[ \text{queen}(X,Y) \land \text{queen}(XX,Y) \rightarrow X = XX, \]
\[ \text{queen}(X,Y) \land \text{queen}(XX,YY) \land |X - XX| = |Y - YY| \rightarrow X = XX. \quad (36) \]
The completion of program (25)–(31) consists of formulas (32)–(35) and the universal closures of formulas (36). 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 $\hat{op}$ is the whole set $\mathbb{Z}^n$, and $arg$ is an $n$-tuple of integer arguments, then $op(arg)$ is an integer argument.

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

$$(g) \text{ integer arguments, symbolic constants, general variables, inf, and sup are arguments.}$$

For example, since $N$ is an integer variable, the expression $N + 1$ is not only a term but also an argument, and both $p(N + 1)$ and $N + 1 = 4$ 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:

$$(f') (\forall NF) \hat{\neq}, \text{ where } N \text{ is an integer variable, is } t \text{ if, for every integer } n, (F_n^X) \hat{\neq} \text{ is } t;$$
$$(h') \text{ if } arg \text{ is } op(arg_1, \ldots, arg_k), \text{ arg}_1 = \overline{m_1}, \ldots, \text{arg}_k = \overline{m_k}, \text{ then arg}^{\hat{\neq}} \text{ is } \overline{op}(m_1, \ldots, m_k).$$

The following abbreviations will be useful. For any argument $arg$, by int$(arg)$ we denote the formula

$$\exists V \left( V \in arg + 1 \right),$$

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

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

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

Using integer variables, we can rewrite formula (12) as

$$\text{int}(p/1), \forall N(p(N) \leftrightarrow T \leq N \leq \overline{8}).$$

Formula (13) can be transformed in a similar way.

Formula (14) can be rewritten as

$$\text{int}(q/1), \forall N(q(N) \leftrightarrow \exists M(N + 1 = M \wedge p(M) \wedge T \leq M \leq \overline{8})).$$

The last formula can be simplified as follows:

$$\forall N(q(N) \leftrightarrow p(N - T) \wedge \overline{T} \leq N \leq \overline{8}).$$
Formula (19) is equivalent to
\[
\text{int}(\text{in}/2), \quad \forall IK(\text{in}(I,K) \rightarrow T \leq I \leq \pi \land T \leq K \leq \tau).
\]
Formula (21) is equivalent to
\[
\forall(I \leq I \leq \pi \rightarrow \text{covered}(I)).
\]
Formula (22) can be equivalently rewritten, in the presence of \text{int}(\text{in}/2), as
\[
\forall IJS(\text{in}(I,S) \land \text{in}(J,S) \land \exists K(K = I + J \land \text{in}(K,S))),
\]
and consequently as
\[
\forall IJS(\text{in}(I,S) \land \text{in}(J,S) \rightarrow \neg \text{in}(I + J,S)).
\]

In the presence of completed definitions (32)–(34), all variables in (35) and in the universal closures of (36) 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 \(G_T\) (condition (ii’)) in Section 8.3) may be unnecessary when the aggregates are known to be monotone or anti-monotone (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 at the University of Potsdam, the home of GRINGO.

Acknowledgements

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

\[5\] https://github.com/potassco/anthem/
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 $\tau$, defined below, transforms every EG program $\Gamma$ into an infinitary formula over the vocabulary of $\Gamma$. Stable models of $\Gamma$ 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 $[t] = \{t\}$;
- If $t = f(t_1, \ldots, t_n)$, where $f$ is a symbolic constant, then $[t]$ consists of the terms $f(r_1, \ldots, r_n)$ for all $r_1 \in [t_1], \ldots, r_n \in [t_n]$;
- If $t = op(t_1, \ldots, t_n)$ where $op$ is an operation name then $[t]$ consists of the numerals of the form $\hat{op}(k_1, \ldots, k_n)$ for all tuples $k_1, \ldots, k_n$ in the domain of $\hat{op}$ such that $\hat{k}_i \in [t_1], \ldots, \hat{k}_n \in [t_n]$;
- If $t = (t_1 \ldots t_2)$ then $[t]$ consists of the numerals $\hat{m}$ for all integers $m$ such that, for some integers $k_1, k_2$, \[\hat{k}_1 \in [t_1], \quad \hat{k}_2 \in [t_2], \quad 1 \leq m \leq k_2.\]

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

A.2 Transforming Programs into Infinitary Formulas

For any ground atom $p(t)$, $\tau p(t)$ stands for $\bigvee_{\tau(r) \in [t]} p(r)$, and $\tau (\neg p(t))$ stands for $\bigvee_{\tau(r) \in [t]} \neg p(r)$.

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 $C_1, \ldots, C_k$ 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 (23) is closed if the term $s$ is ground. Let $X$ be the list of variables occurring in a closed aggregate expression (23), and let $A$ be the set of tuples $r$ of precomputed terms of the same length as $X$. Let $\Delta$ be a subset of $A$. By $|\Delta|$ we denote the union of the sets $\{r \subseteq X : r \in A \}$ for all tuples of precomputed terms $r$ in $\Delta$. We say that $\Delta$ justifies the aggregate expression (23) if the relation $\prec$ holds between $\hat{\Delta} \hat{[\Delta]}$ and an element of the set $[s]$. We define the result of applying $\tau$ to (23) as the conjunction of the implications
\[
\bigwedge_{r \in \Delta} \tau(C^X_r) \rightarrow \bigvee_{r \in A \cap \Delta} \tau(C^X_r)
\]
over all subsets $\Delta$ of $A$ that do not justify (23).

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

\(^6\) The stable model semantics of infinitary formulas by Truszczyński (2012) and Gebser et al. (2015, Section 4.1) is a straightforward generalization of the definition due to Ferraris (2005).
A rule is closed if all its variables are local. If $R$ is a closed basic rule (3) then $\tau R$ is the formula

$$\tau(Body) \rightarrow \bigwedge_{r \in [t]} p(r). \quad (A2)$$

If $R$ is a closed choice rule (7) then $\tau R$ is the formula

$$\tau(Body) \rightarrow \bigwedge_{r \in [t]} (p(r) \lor \neg p(r)). \quad (A3)$$

If $R$ is a closed constraint $\leftarrow Body$ then $\tau R$ 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 $\Gamma$, $\tau \Gamma$ is the conjunction of the formulas $\tau R$ for all instances $R$ of the rules of $\Gamma$.

References


