Abstract Gringo

MARTIN GEBSER∗
Aalto University, HIIT, Finland
University of Potsdam, Germany
gebser@cs.uni-potsdam.de

AMELIA HARRISON†
Univeristy of Texas at Austin, USA
ameliaj@cs.utexas.edu

ROLAND KAMINSKI∗
University of Potsdam, Germany
kaminski@cs.uni-potsdam.de

VLADIMIR LIFSCHITZ†
Univeristy of Texas at Austin, USA
vl@cs.utexas.edu

TORSTEN SCHAUB∗‡
University of Potsdam, Germany
INRIA Rennes, France
torsten@cs.uni-potsdam.de

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

This paper defines the syntax and semantics of the input language of the ASP grounder GRINGO. The definition covers several constructs that were not discussed in earlier work on the semantics of that language, including intervals, pools, division of integers, aggregates with non-numeric values, and lparse-style aggregate expressions. The definition is abstract in the sense that it disregards some details related to representing programs by strings of ASCII characters. It serves as a specification for GRINGO from Version 4.5 on.

To appear in Theory and Practice of Logic Programming (TPLP), Proceedings of ICLP 2015.

∗ Supported by AoF (grant 251170) and DFG (grants SCHA 550/8 and 550/9).
† Partially supported by the National Science Foundation under Grant IIS-1422455.
‡ Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

```
% place queens on the chess board
{ q(1..n,1..n) }.

% exactly 1 queen per row/column
:- X = 1..n, not #count{ Y : q(X,Y) } = 1.
:- Y = 1..n, not #count{ X : q(X,Y) } = 1.

% pre-calculate the diagonals
d1(X,Y,X-Y+n) :- X = 1..n, Y = 1..n.
d2(X,Y,X+Y-1) :- X = 1..n, Y = 1..n.

% at most one queen per diagonal
:- D = 1..n*2-1, 2 { q(X,Y) : d1(X,Y,D) }.
:- D = 1..n*2-1, 2 { q(X,Y) : d2(X,Y,D) }.
```

Table 1. *An ASP solution to the n-queens problem.*

1 Introduction

Version 4.0 of the ASP grounder GRINGO was released in March of 2013.[1] Harrison et al. (2014) defined the semantics of a subset of its input language in terms of stable models of infinitary propositional formulas (Truszczynski 2012).

That subset does not include, however, several constructs that are frequently used in ASP programs. One such construct is integer intervals. Take, for instance, the ASP solution to the n-queens problem shown in Table 1. (It is similar to one of the solutions in the language of Version 3 presented by Gebser et al. (2011).) Intervals are used in each rule of this program. To include intervals, we have to modify the semantics from Harrison et al. (2014) in two ways. First, we have to say that an arithmetic term denotes, generally, a finite set of integers, not a single integer. (And it is not necessarily a set of consecutive integers, because the language of GRINGO allows us to write `(1..3)*2`, for instance. This expression denotes the set $\{2, 4, 6\}$.) Second, in the presence of intervals we cannot treat a choice rule $\{\mathtt{A}\}$ as shorthand for the disjunctive rule `A ; not A` as proposed by Ferraris and Lifschitz (2005). Indeed, the first rule of the program in Table 1 has 2^{n^2} stable models; the rule

```
q(1..n,1..n) ; not q(1..n,1..n)
```

has only 2 stable models.

Another feature of GRINGO not covered by Harrison et al. (2014), which is somewhat similar to integer intervals, is pooling. Pooling is used, for instance, in the head of the rule

```
p(X;Y) :- q(X,Y).
```

(Note that a semicolon, not a comma, separates `X` from `Y` in the head.) This rule has the

[1] `http://potassco.sourceforge.net/`

same meaning as the pair of rules

```
p(X) :- q(X,Y).
p(Y) :- q(X,Y).
```

Pooling is often used to abbreviate a set of facts. For instance, instead of

```
p(a,5).  p(b,10).  p(c,12).
```

we can write `p(a,5;b,10;c,12).` In this paper, we talk about "pools"—groups of terms such as `a,5;b,10;c,12`.

Yet another limitation of the proposal from Harrison et al. (2014) is related to the difference between "dlv-style" aggregates, such as

```
not #count{ Y : q(X,Y) } = 1                (1)
```

in the second rule of the program shown in Table 1, and "lparse-style" aggregates, such as

```
2 { q(X,Y) : d2(X,Y,D) }                    (2)
```

in the last rule of the program. Both expressions have to do with counting. Syntactically, the difference is that in expression (1) both the name of the aggregate (`#count`) and the binary relation applied to the result of counting and a constant (=) are shown explicitly; in (2), the fact that the constant `2` occurs on the left, in the lower bound position, tells us that the relation $\leq$ is applied to that number and to the result of counting. More importantly, there is a difference between the kinds of objects that we count. In case of expression (1) we count, for a given value of `X`, the values of the variable `Y` such that `q(X,Y)` belongs to the stable model. In case of (2) we count, for a given value of `D`, the atoms `q(X,Y)` that belong to the stable model and satisfy an additional condition: `d2(X,Y,D)` belongs to the model as well. Thus the atom in front of the colon in (2) plays two roles: it tells us what to count, and it gives a condition on the stable model that needs to be checked.

The language studied by Harrison et al. (2014) does not include lparse-style aggregates. The easiest way to add such aggregates is to treat them as abbreviations. For instance, (2) can be viewed as shorthand for the dlv-style expression

```
2 <= #count{ q(X,Y) : q(X,Y), d2(X,Y,D) }.
```

(In this paper we adopt a more elaborate translation that allows us to accommodate negated atoms in front of the colon.) In this expression, the first occurrence of `q(X,Y)` is syntactically a term, and the second is an atom. Thus treating (2) as an abbreviation depends on the possibility of using the same symbol as a function and as a predicate. This is customary in Prolog, but not in first-order logic, and this was not allowed by Harrison et al. (2014).

Our goal is to define the syntax and semantics of the language AG (short for *Abstract Gringo*)—a large subset of the input language of GRINGO that includes the features mentioned previously and a few other constructs not described by Harrison et al. (2014). This

is similar to the work that has led to the definition of the ASP Core language (Calimeri et al. 2012).[2,3]

The semantics of AG defined in this paper serves a specification for GRINGO from Version 4.5 on. It can be used to prove the correctness of programs written in its input language. As an example, in the electronic appendix we prove the correctness of the program shown in Table 1.

AG is abstract in the sense that its definition disregards some details related to representing programs by strings of ASCII characters. For example, semicolons are used in the input language of GRINGO in at least three ways: to denote disjunction in the head of a rule, conjunction in the body, and pooling within an atom. In AG, three different symbols play these different roles. The richer alphabet of AG makes it easier to define the semantics of the language and to reason about ASP programs.

2 Syntax of AG

2.1 Symbols and Terms

We assume that five sets of symbols are selected: *numerals*, *symbolic constants*, *negated constants*, *variables*, and *aggregate names*. We assume that a 1–1 correspondence between the set of symbolic constants and the set of negated constants is chosen. For every symbolic constant p, the corresponding negated constant will be called its *strong negation* and denoted by $\tilde{p}$. Further, we assume that these sets do not contain the symbols

$$+ \quad - \quad \times \quad / \quad .. \tag{3}$$

$$inf \quad sup \tag{4}$$

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{5}$$

$$\bot \quad not \quad \wedge \quad \vee \quad \leftarrow \tag{6}$$

$$, \quad ; \quad : \quad (\quad) \quad \{ \quad \} \quad \langle \quad \rangle \tag{7}$$

and that they are pairwise disjoint. All these symbols together form the alphabet of AG, and AG rules will be defined as strings over this alphabet.

When a symbol is represented in ASCII, its type is determined by its first two characters. For instance, a numeral starts with a digit or – followed by a digit. A symbolic constant starts with a lower-case letter. A negated constant starts with – followed by a lower-case letter. A variable starts with an upper-case letter, and an aggregate name starts with #. (The

[2] Syntactically, AG is essentially an extension of ASP Core. But it does not include extra-logical constructs, such as weak constraints and queries. The semantics of aggregates in AG is based on the approach of Ferraris (2005) and thus is not equivalent to the semantics of aggregates in ASP Core when aggregates are used recursively in the presence of negation. Among the language constructs that are not in ASP Core, in AG we find pooling, intervals, and conditional literals. (These constructs originally appeared in the input language of LPARSE, but in AG they are more general; for instance, interval bounds may contain variables, and restrictions to "domain predicates" have disappeared.) Unlike ASP Core, AG supports aggregates in rule heads; see Section 3.

[3] The definition of ASP Core does not refer to infinitary objects, such as infinitary propositional formulas used in this paper. But it appears that infinitary objects of some kind will be required to correct the oversight in (Calimeri et al. 2012, Section 2.2)—the set inst($\{e_1; \ldots; e_n\}$), included in the body of a rule in the process of instantiation, can be infinite.

strings `#false`, `#inf`, and `#sup`, which represent $\bot$, *inf*, and *sup*, also start with `#`.) The symbols $\langle$ and $\rangle$ (which are used to indicate the boundaries of a tuple within a term) correspond to the ASCII characters `(` and `)`.[4] Each of the symbols (3)–(7) except for $\wedge$ and $\vee$ has a unique ASCII representation; the symbols $\wedge$ and $\vee$ can be represented by semicolons and in some cases also by commas.

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 $\overline{n}$.

Terms are defined recursively, as follows:

- all numerals, symbolic constants, and variables are terms;
- if f is a symbolic constant and $\mathbf{t}$ is a tuple[5] of terms then $f(\mathbf{t})$ is a term;
- if t_1 and t_2 are terms and $\star$ is one of the symbols (3) then $(t_1 \star t_2)$ is a term;
- if $\mathbf{t}$ is a tuple of terms then $\langle \mathbf{t} \rangle$ is a term.

In a term of the form $f()$ the parentheses can be dropped, so that every symbolic constant can be viewed as a term. In a term of the form $(t_1 \star t_2)$ we will drop the parentheses when it should not lead to confusion. A term of the form $(\overline{0} - t)$ can be abbreviated as $-t$.

A term, or a tuple of terms, is *precomputed* if it contains neither variables nor symbols (3). 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, $\overline{m} \leq \overline{n}$ iff $m \leq n$.

We assume that for each aggregate name α a function $\widehat{\alpha}$ is chosen that maps every set of tuples of precomputed terms to a precomputed term.[6] The AG counterparts of the aggregates implemented in Version 4.5 of GRINGO are defined below using the following terminology. If the first member of a tuple $\mathbf{t}$ of precomputed terms is a numeral $\overline{n}$ then we say that the integer n is the *weight* of $\mathbf{t}$; if $\mathbf{t}$ is empty or its first member is not an integer then the weight of $\mathbf{t}$ is 0. For any set T of tuples of precomputed terms,

- $\widehat{count}(T)$ is the numeral corresponding to the cardinality of T if T is finite, and sup otherwise;
- $\widehat{sum}(T)$ is the numeral corresponding to the sum of the weights of all tuples in T if T contains finitely many tuples with non-zero weights, and 0 otherwise;
- $\widehat{sum+}(T)$ is the numeral corresponding to the sum of the weights of all tuples in T whose weights are positive if T contains finitely many such tuples, and *sup* otherwise;
- $\widehat{min}(T)$ is sup if T is empty, the least element of the set consisting of the first elements of the tuples in T if T is a finite non-empty set, and *inf* if T is infinite;
- $\widehat{max}(T)$ is *inf* if T is empty, the greatest element of the set consisting of the first elements of the tuples in T if T is a finite non-empty set, and *sup* if T is infinite.

[4] When an AG term representing a tuple of length 1, such as $\langle a \rangle$, is represented in ASCII, a comma is appended to the tuple: `(a,)` .

[5] In this paper, when we refer to a *tuple* of syntactic objects, we mean that the tuple may be empty and that its members are separated by commas.

[6] This understanding of $\widehat{\alpha}$ is different from that given by Harrison et al. (2014, Section 3.3). There, $\widehat{\alpha}$ is understood as a function that maps tuples of precomputed terms to elements of $\mathbf{Z} \cup \{\infty, -\infty\}$.

2.2 Atoms, Literals, and Choice Expressions

A *pool* is an expression of the form $\mathbf{t}_1; \ldots; \mathbf{t}_n$ where $n \geq 1$ and each $\mathbf{t}_i$ is a tuple of terms.[7] In particular, every tuple of terms is a pool.

An *atom* is a string of one of the forms $p(P),\ \tilde{p}(P)$ where p is a symbolic constant and P is a pool. In an atom of the form $p()$ or $\tilde{p}()$ the parentheses can be dropped, so that all symbolic constants and all negated constants can be viewed as atoms.

For any atom A, the strings

$$A \qquad not\ A \qquad not\ not\ A \tag{8}$$

are *symbolic literals*.[8] An *arithmetic literal* is a string of the form $t_1 \prec t_2$ where t_1, t_2 are terms and $\prec$ is one of the symbols (5).

A *conditional literal* is a string of the form $H : \mathbf{L}$ where H is a symbolic or arithmetic literal or the symbol $\bot$ and $\mathbf{L}$ is a tuple of symbolic or arithmetic literals. If $\mathbf{L}$ is empty then we will drop the colon, so that every symbolic or arithmetic literal can be viewed as a conditional literal.[9]

An *aggregate atom* is a string of one of the forms

$$\alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \prec s \tag{9}$$

$$s \prec\ \alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \tag{10}$$

$$s_1 \prec_1 \alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \prec_2 s_2 \tag{11}$$

($n \geq 0$), where

- α is an aggregate name,
- each $\mathbf{t}_i$ is a tuple of terms,
- each $\mathbf{L}_i$ is a tuple of symbolic or arithmetic literals (if $\mathbf{L}_i$ is empty and $\mathbf{t}_i$ is nonempty then the preceding colon may be dropped),
- each of $\prec$, $\prec_1$, $\prec_2$ is one of the symbols (5),
- each of s, s_1, s_2 is a term.

For any aggregate atom A, the strings (8) are *aggregate literals*.

A *literal* is a conditional literal or an aggregate literal.

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

2.3 Rules and Programs

A *rule* is a string of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1 \wedge \cdots \wedge B_m \tag{12}$$

[7] This form of pooling is less general than what is allowed in the input language of GRINGO. For instance, $f(a; b)$ is neither a term nor a pool.

[8] Semantically, the status of "double negations" in AG is the same as in logic programs with nested expressions (Lifschitz et al. 1999), where conjunction, disjunction, and negation can be nested arbitrarily. Dropping a double negation may change the meaning of a rule. For instance, the one-rule program $p \leftarrow not\ not\ p$ has two stable models $\emptyset$, $\{p\}$ (see Section 4.1); the latter will disappear if we drop $not\ not$.

[9] In the input language of GRINGO, dropping the colon when $\mathbf{L}$ is empty is required.

or of the form

$$C \leftarrow B_1 \wedge \cdots \wedge B_m \tag{13}$$

($k, m \geq 0$), where each H_i is a symbolic or arithmetic literal,[10] C is a choice expression, and each B_j is a literal. The expression $B_1 \wedge \cdots \wedge B_m$ is the *body* of the rule; $H_1 \vee \cdots \vee H_k$ is the *head* of (12); C is the *head* of (13). If the body of a rule is empty and the head is not then the arrow can be dropped.

For instance, here are the first five rules of the program from Table 1 written in the syntax of AG:

$$\begin{array}{ll}
R_1: & \{\ q(\overline{1} \mathinner{..} \overline{n}, \overline{1} \mathinner{..} \overline{n})\ \}, \\
R_2: & \leftarrow\ X = \overline{1} \mathinner{..} \overline{n}\ \wedge\ not\ count\{Y : q(X,Y)\}\ =\ \overline{1}, \\
R_3: & \leftarrow\ Y = \overline{1} \mathinner{..} \overline{n}\ \wedge\ not\ count\{X : q(X,Y)\}\ =\ \overline{1}, \\
R_4: & d1(X, Y, X - Y + \overline{n})\ \leftarrow\ X = \overline{1} \mathinner{..} \overline{n}\ \wedge\ Y = \overline{1} \mathinner{..} \overline{n}, \\
R_5: & d2(X, Y, X + Y - \overline{1})\ \leftarrow\ X = \overline{1} \mathinner{..} \overline{n}\ \wedge\ Y = \overline{1} \mathinner{..} \overline{n}.
\end{array}$$

The other two rules use abbreviations introduced in the next section.

A *program* is a finite set of rules.

3 Abbreviations

Let C be an expression of the form

$$s_1 \prec_1 \alpha\{\mathbf{t}_1 : L_1 : \mathbf{L}_1 \,; \ldots ; \mathbf{t}_n : L_n : \mathbf{L}_n\} \prec_2 s_2 \tag{14}$$

($n \geq 0$), where each L_i is a symbolic literal of one of the forms

$$p(\mathbf{t}) \qquad not\ p(\mathbf{t}) \qquad not\ not\ p(\mathbf{t}) \tag{15}$$

(p is a symbolic or negated constant and $\mathbf{t}$ is a tuple of terms) and α, $\mathbf{t}_i$, $\mathbf{L}_i$, $\prec_1$, $\prec_2$, s_1, and s_2 are as in the definition of an aggregate atom. Then a string of the form

$$C \leftarrow B_1 \wedge \cdots \wedge B_m \tag{16}$$

($m \geq 0$), where each B_j is a literal, is shorthand for the set of rules consisting of the rule

$$\leftarrow B_1 \wedge \cdots \wedge B_m \wedge\ not\ s_1 \prec_1 \alpha\{\mathbf{t}_1 : L_1, \mathbf{L}_1 \,; \ldots ; \mathbf{t}_n : L_n, \mathbf{L}_n\} \prec_2 s_2 \tag{17}$$

and, for each L_i in (14) such that L_i is an atom, the rule

$$\{L_i\} \leftarrow B_1 \wedge \cdots \wedge B_m \wedge C_i \tag{18}$$

where C_i is the conjunction of the members of $\mathbf{L}_i$.

In both (17) and (18), the conjunction sign shown after B_m should be dropped if $m = 0$; in (18) it should also be dropped if C_i is empty. The parts $s_1\ \prec_1$ and $\prec_2\ s_2$ in (14) are optional; if one of them is missing then it is dropped from (17) as well; if both are missing then rule (17) is dropped from the set altogether. If $m = 0$ in (16) then the arrow can be dropped.

The *term representations* of literals (15) are the tuples

$$\overline{0}, p(\mathbf{t}) \qquad \overline{1}, p(\mathbf{t}) \qquad \overline{2}, p(\mathbf{t})$$

[10] In the input language of GRINGO, H_i may be any conditional literal.

of terms. (Each of them is indeed a tuple of terms, because $p(\mathbf{t})$ can be viewed as a term.)

Also viewed as an abbreviation is any expression of the form

$$s_1 \, \{L_1 : \mathbf{L}_1 \, ; \ldots ; \, L_n : \mathbf{L}_n\} \, s_2 \tag{19}$$

($n > 0$), where s_1, s_2 are terms, each L_i is a symbolic literal of one of the forms (15) that does not contain .. , and each $\mathbf{L}_i$ is a tuple of symbolic or arithmetic literals.[11] Such an expression is understood differently depending on whether it occurs in the head or the body of a rule. In the head of a rule, (19) is understood as shorthand for an expression of the form (14):

$$s_1 \leq \mathit{count}\{\mathbf{t}_1 : L_1 : \mathbf{L}_1 \, ; \ldots ; \, \mathbf{t}_n : L_n : \mathbf{L}_n\} \leq s_2 \tag{20}$$

where $\mathbf{t}_i$ is the term representation of L_i. If either or both of the terms s_1, s_2 are missing, the abbreviation is understood in a similar way. (Note that choice expressions that do not contain .. are expressions of the form (19) where both s_1 and s_2 are missing, $n = 1$, L_1 is of the form $p(\mathbf{t})$, and $\mathbf{L}_1$ is empty. In this case, we do not view (19) as an abbreviation.)

In the body of a rule (19) is understood as shorthand for the aggregate atom

$$s_1 \leq \mathit{count}\{\mathbf{t}_1 : L_1, \mathbf{L}_1 ; \ldots ; \, \mathbf{t}_n : L_n, \mathbf{L}_n\} \leq s_2$$

where $\mathbf{t}_i$ is the term representation of L_i.[12] If either of the terms s_1, s_2 in (19) is missing, the abbreviation is understood in a similar way.

These abbreviations can be used, for instance, to represent the last two rules of the program from Table 1 in the syntax of AG:

$$\begin{array}{l} \leftarrow \;\; D = \overline{1} \,..\, \overline{n} * \overline{2} - \overline{1} \;\wedge\; \overline{2} \, \{q(X,Y) : d1(X,Y,D)\}, \\ \leftarrow \;\; D = \overline{1} \,..\, \overline{n} * \overline{2} - \overline{1} \;\wedge\; \overline{2} \, \{q(X,Y) : d2(X,Y,D)\}. \end{array}$$

Written out in full, these expressions become

$$\begin{array}{ll} R_6: & \leftarrow \;\; D = \overline{1} \,..\, \overline{n} * \overline{2} - \overline{1} \;\;\wedge\;\; \overline{2} \leq \mathit{count}\{\overline{0}, q(X,Y) : q(X,Y), d1(X,Y,D)\}, \\ R_7: & \leftarrow \;\; D = \overline{1} \,..\, \overline{n} * \overline{2} - \overline{1} \;\;\wedge\;\; \overline{2} \leq \mathit{count}\{\overline{0}, q(X,Y) : q(X,Y), d2(X,Y,D)\}. \end{array}$$

4 Semantics of AG

We will define the semantics of AG using a syntactic transformation τ. The function τ converts rules into infinitary formulas formed from atoms of the form $p(\mathbf{t})$ or $\tilde{p}(\mathbf{t})$, where p is a symbolic constant, and $\mathbf{t}$ is a tuple of precomputed terms. Then the stable models of a program will be defined in terms of stable model semantics of infinitary formulas in the sense of Truszczynski (2012), which is reviewed below.

4.1 Review: Infinitary Propositional Formulas

Let σ be a propositional signature, that is, a set of propositional atoms. The sets $\mathcal{F}_0, \mathcal{F}_1, \ldots$ are defined as follows:

- $\mathcal{F}_0 = \sigma$,

[11] To be precise, if $\mathbf{L}_i$ is empty then the colon after L_i is dropped.
[12] If $\mathbf{L}_i$ is empty then the comma after L_i in this expression should be dropped.

- $\mathcal{F}_{i+1}$ is obtained from $\mathcal{F}_i$ by adding expressions $\mathcal{H}^\wedge$ and $\mathcal{H}^\vee$ for all subsets $\mathcal{H}$ of $\mathcal{F}_i$, and expressions $F \to G$ for all $F, G \in \mathcal{F}_i$.

The elements of $\bigcup_{i=0}^{\infty} \mathcal{F}_i$ are called *(infinitary) formulas* over σ.

In an infinitary formula, the symbols $\top$ and $\bot$ are understood as abbreviations for $\emptyset^\wedge$ and $\emptyset^\vee$ respectively; $\neg F$ stands for $F \to \bot$, and $F \leftrightarrow G$ stands for $(F \to G) \wedge (G \to F)$.

Subsets of a signature σ will also be called its *interpretations*. The satisfaction relation between an interpretation and a formula is defined recursively as follows:

- For every atom p from σ, $I \models p$ if $p \in I$.
- $I \models \mathcal{H}^\wedge$ if for every formula F in $\mathcal{H}$, $I \models F$.
- $I \models \mathcal{H}^\vee$ if there is a formula F in $\mathcal{H}$ such that $I \models F$.
- $I \models F \to G$ if $I \not\models F$ or $I \models G$.

We say that an interpretation satisfies a set $\mathcal{H}$ of formulas, or is a *model* of $\mathcal{H}$, if it satisfies every formula in $\mathcal{H}$. Two sets of formulas are *equivalent* if they have the same models.

The *reduct* F^I of a formula F w.r.t. an interpretation I is defined as follows:

- For $p \in \sigma$, $p^I = \bot$ if $I \not\models p$; otherwise $p^I = p$.
- $(\mathcal{H}^\wedge)^I = \{G^I \mid G \in \mathcal{H}\}^\wedge$.
- $(\mathcal{H}^\vee)^I = \{G^I \mid G \in \mathcal{H}\}^\vee$.
- $(G \to H)^I = \bot$ if $I \not\models G \to H$; otherwise $(G \to H)^I = G^I \to H^I$.

An interpretation I is a *stable model* of a set $\mathcal{H}$ of formulas if it is minimal w.r.t. set inclusion among the interpretations satisfying the reducts of all formulas from $\mathcal{H}$.

For instance, if $I = \emptyset$ then

$$(\neg\neg p \to p)^I = (\neg\neg p)^I \to p^I = \bot \to \bot;$$

if $I = \{p\}$ then

$$(\neg\neg p \to p)^I = (\neg\neg p)^I \to p^I = \neg(\neg p)^I \to p = \neg\bot \to p.$$

In both cases, I is a minimal model of the reduct. Consequently, both $\emptyset$ and $\{p\}$ are stable models of $\{\neg\neg p \to p\}$.

4.2 Semantics of Terms and Pools

A term is *ground* if it does not contain variables. The definition of "ground" for pools, symbolic literals, and arithmetic literals is the same. Semantically, every ground term t represents a finite set of precomputed terms $[t]$, which is defined recursively:

- if t is a numeral or a symbolic constant then $[t]$ is $\{t\}$;
- if t is $f(t_1, \ldots, t_n)$ then $[t]$ is the set of terms $f(r_1, \ldots, r_n)$ for all $r_1 \in [t_1], \ldots, r_n \in [t_n]$;
- if t is $(t_1 + t_2)$ then $[t]$ is the set of numerals $\overline{n_1 + n_2}$ for all integers n_1, n_2 such that $\overline{n_1} \in [t_1]$ and $\overline{n_2} \in [t_2]$; similarly when t is $(t_1 - t_2)$ or $(t_1 \times t_2)$;
- if t is (t_1/t_2) then $[t]$ is the set of numerals $\overline{\lfloor n_1/n_2 \rfloor}$ for all integers n_1, n_2 such that $\overline{n_1} \in [t_1]$, $\overline{n_2} \in [t_2]$, and $n_2 \neq 0$;

- if t is $(t_1 \mathinner{..} t_2)$ then $[t]$ is the set of numerals $\overline{m}$ for all integers m such that, for some integers n_1, n_2,

$$\overline{n_1} \in [t_1], \qquad \overline{n_2} \in [t_2], \qquad n_1 \leq m \leq n_2;$$

- if t is $\langle t_1, \ldots, t_n \rangle$ then $[t]$ is the set of terms $\langle r_1, \ldots, r_n \rangle$ for all $r_1 \in [t_1], \ldots, r_n \in [t_n]$.

This definition is extended to an arbitrary ground pool P; $[P]$ is a finite set of precomputed tuples:

- if P is a tuple $t_1, \ldots, t_n$ of terms $(n \neq 1)$ then $[P]$ is the set of tuples $r_1, \ldots, r_n$ for all $r_1 \in [t_1], \ldots, r_n \in [t_n]$;
- if P is a pool $\mathbf{t}_1; \ldots; \mathbf{t}_n$ $(n > 1)$ then $[P]$ is $[\mathbf{t}_1] \cup \cdots \cup [\mathbf{t}_n]$.

For instance, $[\overline{1}..\overline{n}, \overline{1}..\overline{n}]$ is the set $\{\overline{i}, \overline{j} : 1 \leq i, j \leq n\}$.

It is clear that if a ground term t contains neither symbolic constants nor the symbols $\langle$ and $\rangle$ then every element of $[t]$ is a numeral. If a tuple $\mathbf{t}$ of ground terms is precomputed then $[\mathbf{t}]$ is $\{\mathbf{t}\}$. The set $[t]$ can be empty. For example, $[1..0] = [1/0] = [1 + a] = \emptyset$.

About a tuple of terms that does not contain .. we say that it is *interval-free*. It is clear that if a tuple $\mathbf{t}$ of ground terms is interval-free then the cardinality of the set $[\mathbf{t}]$ is at most 1.

4.3 Semantics of Arithmetic and Symbolic Literals

For any ground (symbolic or arithmetic) literal L we will define two translations, $\tau_\wedge L$ and $\tau_\vee L$. The specific translation function applied to an occurrence of a symbolic or arithmetic literal in a rule depends on the context, as we will see in the following sections.

We will first consider symbolic literals. For any ground atom A,

- if A is $p(P)$ then $\tau_\wedge A$ is the conjunction of atoms $p(\mathbf{t})$ over all tuples $\mathbf{t}$ in $[P]$, and $\tau_\vee A$ is the disjunction of these atoms;
- if A is $\tilde{p}(P)$ then $\tau_\wedge A$ is the conjunction of atoms $\tilde{p}(\mathbf{t})$ over all tuples $\mathbf{t}$ in $[P]$, and $\tau_\vee A$ is the disjunction of these atoms;
- $\tau_\wedge(not\ A)$ is $\neg\tau_\vee A$, and $\tau_\vee(not\ A)$ is $\neg\tau_\wedge A$;
- $\tau_\wedge(not\ not\ A)$ is $\neg\neg\tau_\wedge A$, and $\tau_\vee(not\ not\ A)$ is $\neg\neg\tau_\vee A$.

The definitions of $\tau_\wedge$ and $\tau_\vee$ for arithmetic literals are as follows:

- $\tau_\wedge(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between the terms r_1 and r_2 for all $r_1 \in [t_1]$ and $r_2 \in [t_2]$, and $\bot$ otherwise;
- $\tau_\vee(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between the terms r_1 and r_2 for some r_1, r_2 such that $r_1 \in [t_1]$ and $r_2 \in [t_2]$, and $\bot$ otherwise.

For instance, $\tau_\vee p(\overline{2}..\overline{4})$ is $p(\overline{2}) \vee p(\overline{3}) \vee p(\overline{4})$, and $\tau_\vee(\overline{2} = \overline{2}..\overline{4})$ is $\top$.

For any tuple $\mathbf{L}$ of ground literals, $\tau_\vee \mathbf{L}$ stands for the conjunction of the formulas $\tau_\vee L$ for all members L of $\mathbf{L}$. The expressions $\tau_\wedge\bot$ and $\tau_\vee\bot$ both stand for $\bot$.

It is clear that if A has the form $p(\mathbf{t})$ or $\tilde{p}(\mathbf{t})$, where $\mathbf{t}$ is a tuple of precomputed terms, then each of the formulas $\tau_\wedge A$ and $\tau_\vee A$ is A.

4.4 Semantics of Choice Expressions

The result of applying τ to a choice expression $\{p(P)\}$ is the conjunction of the formulas $p(\mathbf{t}) \vee \neg p(\mathbf{t})$ over all tuples $\mathbf{t}$ in $[P]$. Similarly, the result of applying τ to a choice expression $\{\tilde{p}(P)\}$ is the conjunction of the formulas $\tilde{p}(\mathbf{t}) \vee \neg \tilde{p}(\mathbf{t})$ over all tuples $\mathbf{t}$ in $[P]$.

For instance, the result of applying τ to rule R_1 (see Section 2.3) is

$$\bigwedge_{1 \leq i,j \leq n} \left(q(\overline{i}, \overline{j}) \vee \neg q(\overline{i}, \overline{j}) \right). \tag{21}$$

4.5 Global Variables

About a variable we say that it is *global*

- in a conditional literal $H : \mathbf{L}$, if it occurs in H but does not occur in $\mathbf{L}$;
- in an aggregate literal A, $not\ A$, or $not\ not A$, where A is of one of the forms (9)–(11), if it occurs in s, s_1, or s_2;
- in a rule (12), if it is global in at least one of the expressions H_i, B_j;
- in a rule (13), if it occurs in C or is global in at least one of the expressions B_j.

An *instance* of a rule R is any rule that can be obtained from R by substituting precomputed terms for all global variables.[13] A literal or a rule is *closed* if it has no global variables. It is clear that any instance of a rule is closed.

For example, X is global in the rule R_2 from Section 2.3, so that the instances of R_2 are rules of the form

$$\leftarrow\ r = \overline{1} \mathinner{..} \overline{n}\ \wedge\ not\ count\{Y : q(r, Y)\} = \overline{1}$$

for all precomputed terms r. The variables X and Y are global in R_4; instances of R_4 are

$$d1(r, s, r - s + \overline{n})\ \leftarrow\ r = \overline{1} \mathinner{..} \overline{n}\ \wedge\ s = \overline{1} \mathinner{..} \overline{n}$$

for all precomputed terms r and s.

4.6 Semantics of Conditional Literals

If t is a term, $\mathbf{x}$ is a tuple of distinct variables, and $\mathbf{r}$ is a tuple of terms of the same length as $\mathbf{x}$, then the term obtained from t by substituting $\mathbf{r}$ for $\mathbf{x}$ will be denoted by $t^{\mathbf{x}}_{\mathbf{r}}$. Similar notation will be used for the result of substituting $\mathbf{r}$ for $\mathbf{x}$ in expressions of other kinds, such as literals and tuples of literals.

The result of applying τ to a closed conditional literal $H : \mathbf{L}$ is the conjunction of the formulas

$$\tau_\vee(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \rightarrow \tau_\vee(H^{\mathbf{x}}_{\mathbf{r}})$$

where $\mathbf{x}$ is the list of variables occurring in $H : \mathbf{L}$, over all tuples $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{x}$.

[13] This definition differs slightly from that given by Harrison et al. (2014, Section 3.3). There, substitutions that yield symbolic constants in the scope of arithmetical operators do not form instances. In a similar way, we treat variables in conditional literals and aggregate literals (Sections 4.6 and 4.7) differently than how they are treated by Harrison et al. (2014).

For instance, the result of applying τ to the arithmetic literal $r = \overline{1}..\overline{n}$, where r is a precomputed term, is $\tau_\vee(\epsilon) \rightarrow \tau_\vee(r = \overline{1}..\overline{n})$, where ϵ is the tuple of length 0. The antecedent of this implication is $\top$. The consequent is $\top$ if r is one of the numerals $\overline{1}, \ldots, \overline{n}$ and $\bot$ otherwise.

4.7 Semantics of Aggregate Literals

In this section, the semantics of ground aggregates proposed by Ferraris (2005, Section 4.1) is adapted to closed aggregate literals. Let E be a closed aggregate atom of one of the forms (9)–(11), and let $\mathbf{x}_i$ be the list of variables occurring in $\mathbf{t}_i : \mathbf{L}_i$ $(1 \leq i \leq n)$. By A_i we denote the set of tuples $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{x}_i$. By A we denote the set $\{(i, \mathbf{r}) : i \in \{1, \ldots, n\}, \mathbf{r} \in A_i\}$.

Let Δ be a subset of A. Then by $[\Delta]$ we denote the union of the sets $[(\mathbf{t}_i)^{\mathbf{x}_i}_{\mathbf{r}}]$ for all pairs $(i, \mathbf{r}) \in \Delta$. We say that Δ *justifies* E with respect to a precomputed term[14] t if

- E is of the form (9) and the relation $\prec$ holds between $\widehat{\alpha}[\Delta]$ and t, or
- E is of the form (10) and the relation $\prec$ holds between t and $\widehat{\alpha}[\Delta]$.

We say that Δ *justifies* E with respect to a pair t_1, t_2 of precomputed terms if E is of the form (11), the relation $\prec_1$ holds between t_1 and $\widehat{\alpha}[\Delta]$, and the relation $\prec_2$ holds between $\widehat{\alpha}[\Delta]$ and t_2.

If t is a precomputed term, and E is of form (9) or (10), we define $\tau_t E$ as the conjunction of the implications

$$\bigwedge_{(i,\mathbf{r})\in\Delta} \tau_\vee((\mathbf{L}_i)^{\mathbf{x}_i}_{\mathbf{r}}) \rightarrow \bigvee_{(i,\mathbf{r})\in A\setminus\Delta} \tau_\vee((\mathbf{L}_i)^{\mathbf{x}_i}_{\mathbf{r}}) \tag{22}$$

over all sets Δ that do not justify E with respect to t. If t_1, t_2 is a pair of precomputed terms, and E is of form (11), we define $\tau_{t_1,t_2} E$ as the conjunction (22) over all sets Δ that do not justify E with respect to t_1, t_2.

For instance, if E is $count\{p(X) : p(X)\} > 0$ then $\tau_{\overline{0}} E$ is the (conjunction containing the single) implication expressing that $p(r)$ holds for at least one precomputed term r:

$$\top \rightarrow \bigvee_r p(r).$$

For a closed aggregate atom E of form (9) or (10),

- by τE we denote the disjunction of formulas $\tau_t E$ over all terms t in $[s]$;
- by $\tau(not\ E)$ we denote the disjunction of formulas $\neg\tau_t E$ over all terms t in $[s]$; and
- by $\tau(not\ \ not\ \ E)$ we denote the disjunction of formulas $\neg\neg\tau_t E$ over all terms t in $[s]$.

It is clear that if $[s]$ is a singleton set $\{t\}$, then τE is (the disjunction containing only) $\tau_t E$. For a closed aggregate atom E of form (11),

[14] This definition of the semantics of aggregates is more complicated than that published in the original version of this document. There, a set Δ either justifies an aggregate atom or not, without reference to a particular precomputed term t. The version here corrects a discrepancy between the semantics and the behavior of GRINGO in the case when s represents a non-singleton set.

- by τE we denote the disjunction of formulas $\tau_{t_1,t_2} E$ over all pairs of precomputed terms t_1, t_2 such that t_1 in $[s_1]$ and t_2 in $[s_2]$;
- by $\tau(not\ E)$ we denote the disjunction of formulas $\neg\tau_{t_1,t_2} E$ over all pairs of precomputed terms t_1, t_2 such that t_1 in $[s_1]$ and t_2 in $[s_2]$; and
- by $\tau(not\ not\ E)$ we denote the disjunction of formulas $\neg\neg\tau_{t_1,t_2} E$ over all pairs of precomputed terms t_1, t_2 such that t_1 in $[s_1]$ and t_2 in $[s_2]$.

4.8 Semantics of Rules and Programs

For any rule R of form (12), τR stands for the set of the formulas

$$\tau B_1 \wedge \cdots \wedge \tau B_m \to \tau_\wedge H_1 \vee \cdots \vee \tau_\wedge H_k$$

for all instances (12) of R. For a rule of form (13), τR stands for the set of the formulas

$$\tau B_1 \wedge \cdots \wedge \tau B_m \to \tau C$$

for all instances (13) of R. For any program Π, $\tau\Pi$ stands for the union of the sets τR for all rules R of Π.

A *stable model* of a program Π is any stable model of $\tau\Pi$ (in the sense of Section 4.1) that does not contain any pair of atoms of the form $p(\mathbf{t})$, $\tilde{p}(\mathbf{t})$.

5 Simplifying $\tau\Pi$

When we investigate the stable models of an AG program, it is often useful to simplify the formulas obtained by applying transformation τ to its rules. By simplifying an infinitary propositional formula we mean turning it into a strongly equivalent formula that has simpler syntactic structure. The definition of strong equivalence, introduced by Lifschitz et al. (2001), is extended to infinitary formulas by Harrison et al. (2015). Corollary 1 from that paper shows that the stable models of an infinitary formula are not affected by simplifying its parts.

Proofs of the theorems stated in this section are outlined in the electronic appendix.

5.1 Monotone and Anti-Monotone Aggregate Atoms

When a rule contains aggregate atoms, we can sometimes simplify the implications (22) in the corresponding infinitary formula using the theorems on monotone and anti-monotone aggregates from Harrison et al. (2014, Section 6.1). The monotonicity or non-monotonicity of an aggregate atom (9) can sometimes be established simply by looking at its aggregate name α and its relation symbol $\prec$. If α is one of the symbols *count*, *sum+*, *max*, then (9) is monotone when $\prec$ is $<$ or $\leq$, and anti-monotone when $\prec$ is $>$ or $\geq$. It is the other way around if α is *min*.

Our semantics of aggregates is somewhat different from that adopted by Harrison et al. (2014, Section 3.5), as explained in Footnote 6 (and also in view of the difference in the treatment of variables discussed in Footnote 13, and the modification to the definition of "justifies" explained in Footnote 14). Nevertheless, the statements and proofs of the two

theorems mentioned above remain essentially the same in the framework of AG. The theorems show that the antecedent in (22) can be dropped if E is monotone, and that the consequent can be replaced by $\bot$ if E is anti-monotone. These simplifications produce strongly equivalent formulas.

5.2 Eliminating Equality from Aggregate Atoms

If $\prec$ in an aggregate atom (9) is $=$ then the following theorem[15] can be useful, in combination with the facts reviewed in Section 5.1:

Theorem 1
If E is a closed aggregate atom of the form

$$\alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} = s,$$

$E_{\leq}$ is

$$\alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \leq s,$$

and $E_{\geq}$ is

$$\alpha\{\mathbf{t}_1 : \mathbf{L}_1 ; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \geq s,$$

then for any precomputed term t, $\tau_t E$ is strongly equivalent to $\tau_t E_{\leq} \wedge \tau_t E_{\geq}$.

5.3 Properties of Counting

For any set S, by $|S|$ we denote the cardinality of S if S is finite, and ∞ otherwise.

Theorem 2
For any closed aggregate atom E of the form

$$\mathit{count}\{\mathbf{t}_1 : \mathbf{L}_1; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \geq \overline{m}$$

where m is an integer and each $\mathbf{t}_i$ is interval-free, τE is strongly equivalent to

$$\bigvee_{\substack{\Delta \subseteq A \\ |[\Delta]|=m}} \bigwedge_{(i,\mathbf{r}) \in \Delta} \tau_{\vee}((\mathbf{L}_i)^{\mathbf{x}_i}_{\mathbf{r}}). \tag{23}$$

Theorem 3
For any closed aggregate atom E of the form

$$\mathit{count}\{\mathbf{t}_1 : \mathbf{L}_1; \ldots ; \mathbf{t}_n : \mathbf{L}_n\} \leq \overline{m}$$

where m is an integer and each $\mathbf{t}_i$ is interval-free, τE is strongly equivalent to

$$\bigwedge_{\substack{\Delta \subseteq A \\ |[\Delta]|=m+1}} \neg \bigwedge_{(i,\mathbf{r}) \in \Delta} \tau_{\vee}((\mathbf{L}_i)^{\mathbf{x}_i}_{\mathbf{r}}). \tag{24}$$

[15] The statement and proof of this theorem have been modified with respect to the original version of this paper in accordance with the change in the definition of "justifies" described in Footnote 14.

Without the assumption that each $\mathbf{t}_i$ is interval-free the assertions of the theorems would be incorrect. For instance, if E is $count\{\overline{1}..\overline{2} : p\} \geq \overline{1}$ then τE is $\top \to p$, and (23) is $\bot$.

In the special (but common) case when E has the form $count\{\mathbf{x} : \mathbf{L}\} \geq \overline{m}$, where $\mathbf{x}$ is a tuple of variables and each variable occurring in $\mathbf{L}$ occurs also in $\mathbf{x}$, the condition $|[\Delta]| = m$ in (23) can be replaced by $|\Delta| = m$. Indeed, in this case Δ and $[\Delta]$ have the same cardinality because $[\Delta]$ is the set of tuples $\mathbf{r}$ of terms such that $(1, \mathbf{r}) \in \Delta$. Similarly, the condition $|[\Delta]| = m + 1$ in (24) can be replaced by $|\Delta| = m + 1$ if E has the form $count\{\mathbf{x} : \mathbf{L}\} \leq \overline{m}$.

6 Conclusion

We proposed a definition of stable models for programs in the language AG and stated a few theorems that facilitate reasoning about them. This definition can be viewed as a specification for the answer set system CLINGO (see Footnote 1) and other systems with the same input language. If such a system terminates given the ASCII representation of an AG program Π as input, and produces neither error messages nor warnings, then its output is expected to represent the stable models of Π.

Acknowledgements

We are grateful to the anonymous referees for useful comments.

References

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2012. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.0.pdf`.

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

FERRARIS, P. AND LIFSCHITZ, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5,* 1–2, 45–74.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2011. Challenges in answer set solving. In *Logic programming, knowledge representation, and nonmonotonic reasoning*. Springer, 74–90.

HARRISON, A., LIFSCHITZ, V., PEARCE, D., AND VALVERDE, A. 2015. Infinitary equilibrium logic and strong equivalence. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. `http://www.cs.utexas.edu/users/vl/papers/iel_lpnmr.pdf`; to appear.

HARRISON, A., LIFSCHITZ, V., AND YANG, F. 2014. The semantics of Gringo and infinitary propositional formulas. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

LIFSCHITZ, V., PEARCE, D., AND VALVERDE, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic 2*, 526–541.

LIFSCHITZ, V., TANG, L. R., AND TURNER, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25*, 369–389.

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.