Towards Verifying Logic Programs in the Input Language of CLINGO

Vladimir Lifschitz[1][0000−0001−6051−7907], Patrick Lühne[2][0000−0001−5902−4152], and Torsten Schaub[2][0000−0002−7456−041X]

[1] University of Texas at Austin, USA
vl@cs.utexas.edu
[2] University of Potsdam, Germany
{patrick.luehne, torsten}@cs.uni-potsdam.de

Abstract. We would like to develop methods for verifying programs in the input language of the answer set solver CLINGO using software created for the automation of reasoning in first-order theories. As a step in this direction, we extend Clark's completion to a class of CLINGO programs that contain arithmetic operations as well as intervals and prove that every stable model is a model of generalized completion. The translator ANTHEM calculates the completion of a program and represents it in a format that can be processed by first-order theorem provers. Some properties of programs can be verified by ANTHEM and the theorem prover VAMPIRE, working together.

1 Introduction

Rules in a logic program and axioms in a first-order theory can serve, in many cases, as alternative mechanisms for expressing the same mathematical idea. For instance, the rule

$$q(X) \leftarrow p(X, Y) \tag{1}$$

defines a unary predicate q in terms of a binary predicate p in the language of logic programs. The same relationship between p and q can be expressed by the first-order formula

$$\forall X(q(X) \leftrightarrow \exists Y p(X, Y)). \tag{2}$$

Software used by practitioners of logic programming performs reasoning about predicates that are defined by rules. Consider, for instance, the file

```
q(X) :- p(X, Y).
p(a, b). p(b, c).
```
(3)

Its first line represents rule (1); the second line expresses that p is the set $\{\langle a, b\rangle, \langle b, c\rangle\}$. If we feed this file into an answer set solver,[3] then the atoms

```
q(a) q(b)
```

[3] Answer set solvers are logic programming systems that calculate stable models (answer sets) of logic programs. Some of the best-known systems of this kind are CLINGO (https://potassco.org/) and DLV (http://www.dlvsystem.com/).

are generated: The solver tells us that q is the set $\{a, b\}$.

The efficiency of answer set solvers and the expressive possibilities of their input languages make them valuable tools for many applications [2, 6]. On the other hand, the class of reasoning tasks that they can perform is rather limited. For instance, it is clear that, under the assumption that q is defined by rule (1), one of the sets p, q is nonempty if and only if the other is nonempty. But there seems to be no way to instruct a logic programming system to verify this assertion. Its syntactic form is not suitable for the forms of automated reasoning implemented in these systems.

In the world of first-order theorem proving, the situation is different: The fact that the formula

$$\exists X q(X) \leftrightarrow \exists XY p(X, Y)$$

is entailed by (2) can be easily verified by a theorem prover.

We would like to develop methods for verifying properties of logic programs using software created for the automation of reasoning in first-order theories. In particular, we would like to verify the correctness of logic programs with respect to specifications expressed in traditional logical and mathematical notation. This paper describes initial steps towards that goal.

As an example, consider the CLINGO rule

```
q(X + Y) :- p(X), p(Y).
```
(4)

It describes an operation that transforms a set p of integers into another set q. The translator ANTHEM [11, 12], implemented as part of this project, turns this rule into a first-order formula describing the same transformation. The output of ANTHEM, along with a property of this transformation that we would like to verify, can be fed into a proof assistant or a theorem prover. In Section 5, we will see, for instance, that the theorem prover VAMPIRE [9] can use the output of ANTHEM to prove that, for every integer n, if all elements of p are less than or equal to n, then all elements of q are less than or equal to $2n$.

To take another example, the CLINGO program[4]

```
p(X) :- X = 0..n, X * X <= n.
q(X) :- p(X), not p(X + 1).
```
(5)

calculates the floor of $\sqrt{n}$, in the sense that the set q that it defines is the singleton $\{\lfloor\sqrt{n}\rfloor\}$. When CLINGO is called to find the stable model of this program, the value of the placeholder `n` is specified on the command line. In Section 5, we will see how the tandem of ANTHEM and VAMPIRE can be used to verify the claim about the relationship between n and q.

The translation performed by ANTHEM is a generalization of the well-known process of program completion [1, 14]. When applied to rule (1), for instance, it

[4] As discussed in Section 2.1, the "interval term" `0..n` in the first rule of this program is an arithmetic expression that has multiple values—all integers from 0 to n. The equality between a variable and an interval term in the body of a rule expresses that the value of the variable is equal to one of the values of the term.

gives a formula equivalent to (2). This generalization is applicable to programs containing arithmetic operations and intervals, such as (4) and (5). According to the theorem stated in Section 4.4 and proved in Section 6, this generalized completion formula is satisfied by all stable models of the program. It follows that any assertion that can be derived from it is a common property of all stable models. This fact is at the root of the verification method proposed in this paper.

2 Programs

2.1 Terms and Their Values

We assume that three countably infinite sets of symbols are selected: *numerals*, *symbolic constants*, and *program variables*.[5] We assume that a 1-to-1 correspondence between numerals and integers is chosen; the numeral corresponding to an integer n is denoted by $\overline{n}$. *Program terms* are defined recursively:

- Numerals, symbolic constants, program variables, and the symbols *inf* and *sup* are program terms;
- if t_1, t_2 are program terms and *op* is one of the *operation names*

$$+ \quad - \quad \times \quad / \quad \backslash \quad .. \tag{6}$$

  then $(t_1\ op\ t_2)$ is a program term.

The expression $-t$ is shorthand for $\overline{0} - t$.

A program term, or another syntactic expression, is *ground* if it does not contain variables. A ground expression is *precomputed* if it does not contain operation names.

For every ground program term t, the set $[t]$ of its *values* is defined as follows:

- if t is a numeral, a symbolic constant, *inf*, or *sup*, then $[t]$ is $\{t\}$;
- if t is $(t_1 + t_2)$, then $[t]$ is the set of numerals $\overline{i+j}$ for all integers i, j such that $\overline{i} \in [t_1]$ and $\overline{j} \in [t_2]$; 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 $\overline{\lfloor i/j \rfloor}$ for all integers i, j such that $\overline{i} \in [t_1]$, $\overline{j} \in [t_2]$, and $j \neq 0$;
- if t is $(t_1 \backslash t_2)$, then $[t]$ is the set of numerals $\overline{i - j \cdot \lfloor i/j \rfloor}$ for all integers i, j such that $\overline{i} \in [t_1]$, $\overline{j} \in [t_2]$, and $j \neq 0$;
- if t is $(t_1 .. t_2)$, then $[t]$ is the set of numerals $\overline{k}$ for all integers k such that, for some integers i, j,

$$\overline{i} \in [t_1], \quad \overline{j} \in [t_2], \quad \text{and } i \leq k \leq j.$$

[5] We talk about *program* variables and *program* terms to distinguish them from the variables and terms that are allowed in formulas (Section 3) and thus can be found in the output of ANTHEM.

It is clear that values of a ground program term are precomputed program terms and that the set of values of any term is finite. It can be empty; for instance, if c is a symbolic constant, then

$$[c + \overline{1}] = [\overline{1}/\overline{0}] = [\overline{1}\,..\,\overline{0}] = \emptyset.$$

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

2.2 Rules and Programs

The programming language defined in this section is a subset of the input language of CLINGO. We write programs in abstract syntax [7] that disregards details related to representing rules by strings of ASCII characters. For example, expression (1) is the first rule of program (3) written in abstract syntax.

We assume a total order on precomputed program terms such that

(i) *inf* is its least element and *sup* is its greatest element,
(ii) for any integers m and n, $\overline{m} < \overline{n}$ iff $m < n$,
(iii) for any integer n and any symbolic constant c, $\overline{n} < c$.

An *atom* is an expression of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of program terms. The parentheses can be dropped if $\mathbf{t}$ is empty. A *literal* is an atom possibly preceded by one or two occurrences of *not*. A *comparison* is an expression of the form $(t_1 \prec t_2)$, where t_1, t_2 are program terms and $\prec$ is one of the *relation names*

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{7}$$

A *rule* is an expression of the form

$$\mathit{Head} \leftarrow \mathit{Body}, \tag{8}$$

where

- *Body* is a conjunction (possibly empty) of literals and comparisons and
- *Head* is either an atom (then, we say that (8) is a *basic rule*), or an atom in braces (then, (8) is a *choice rule*), or empty (then, (8) is a *constraint*).

The arrow can be dropped if *Body* is empty.

A *program* is a finite set of rules.

An *interpretation* is a set of precomputed atoms. The semantics of programs [7], reviewed in Section 6.1, defines which interpretations are *stable models* of a program. We describe here a few features of the semantics that are related to the topic of this paper.

Terms with multiple values in the head of a rule. A rule of the form $p(t) \leftarrow \mathit{Body}$, where t is a ground term, has the same meaning as the collection of rules $p(r) \leftarrow \mathit{Body}$ over all values r of t. For instance, the one-rule program

$p(\overline{1}\,..\,\overline{3})$ has the same meaning as the collection of facts $p(\overline{1})$, $p(\overline{2})$, $p(\overline{3})$. A rule of the form $p(\overline{1}/\overline{0}) \leftarrow \mathit{Body}$ has the same meaning as the empty program: The set of stable models of any program would not be affected by adding such a rule. Using the expression $\overline{1}/\overline{0}$ in a program is not considered an error.

Terms with multiple values in the body of a rule. Similarly, a rule of the form $\mathit{Head} \leftarrow p(t)$, where t is a ground term, has the same meaning as the set of rules $\mathit{Head} \leftarrow p(r)$ over all values r of t. A rule of the form $\mathit{Head} \leftarrow p(\overline{1}/\overline{0})$ has the same meaning as the empty program.

Instances of rules. An *instance* of a rule R is a ground rule obtained from R by substituting precomputed program terms for program variables. The semantics of the language defines the stable models of a program in terms of the set of instances of its rules. In this sense, program variables are used as variables for arbitrary precomputed terms. For example, instances of the rule

$$q(X + \overline{1}) \leftarrow p(X) \tag{9}$$

are rules of the form

$$q(r + \overline{1}) \leftarrow p(r), \tag{10}$$

where r is an arbitrary precomputed term—a numeral, a symbolic constant, *inf*, or *sup*. But if r is not a numeral, then rule (10) is equivalent to the empty program, as discussed above. In this sense, the possibility of substituting terms other than numerals for r in the process of constructing instances of rule (9) is not essential. With the rule $q(X) \leftarrow p(X + \overline{1})$, the situation is similar.

2.3 Programs with Input

An input can be given to a CLINGO program in two ways: by specifying the predicates corresponding to some of the predicate symbols, as in programs (1) and (4), and by specifying the values of placeholders, as in program (5). The definition of a program with input [10, Section 3], reproduced below, makes this idea precise.

A *predicate symbol* is a pair p/n, where p is a symbolic constant and n is a nonnegative integer. 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 $p(t_1, \ldots, t_n)$.

A *program with input* is a pair (Π, P), where Π is a program and P is a finite set such that each of its elements is

- a symbolic constant or
- a predicate symbol that does not occur in the heads of the rules of Π.

The elements of P are the *input symbols* of (Π, P).

An *input* for (Π, P) is a function $\mathbf{i}$ defined on P such that

- for every symbolic constant c in P, $\mathbf{i}(c)$ is a precomputed term, and

– for every predicate symbol p/n in P, $\mathbf{i}(p/n)$ is a finite set of precomputed atoms containing p/n.

A *stable model* of (Π, P) for an input $\mathbf{i}$ is a stable model of the program consisting of

– all atoms in $\mathbf{i}(p/n)$ for all predicate symbols p/n in P and
– the rules obtained from the rules of Π by substituting simultaneously the precomputed terms $\mathbf{i}(c)$ for all occurrences of symbolic constants c in P.

We identify $(\Pi, \emptyset)$ with Π.

Example 1. The claim about rule (4) made in the introduction can be stated using this terminology as follows. Consider the program with input that consists of the rule

$$q(X + Y) \leftarrow p(X) \wedge p(Y) \tag{11}$$

(which is (4) written in abstract notation) and the input symbol $p/1$. We want to verify that

$$\begin{array}{l} \text{for every integer } n, \\ \text{if } \mathcal{I} \text{ is a stable model of this program for an input } \mathbf{i} \\ \quad \text{and every term } t \text{ such that } p(t) \in \mathbf{i}(p/1) \text{ is } \overline{m} \\ \quad\quad \text{for some integer } m \text{ such that } m \leq n, \\ \text{then every term } t \text{ such that } q(t) \in \mathcal{I} \text{ is } \overline{m} \\ \quad \text{for some integer } m \text{ such that } m \leq 2n. \end{array} \tag{12}$$

(The program in question has actually a unique stable model for every input $\mathbf{i}$. Verifying properties of this kind automatically goes beyond the scope of this paper.) Since $p/1$ does not occur in the head of (11), the condition $p(t) \in \mathbf{i}(p/1)$ in this statement can be replaced by $p(t) \in \mathcal{I}$.

Example 2. To reformulate the claim about program (5), consider the program with input that consists of the rules

$$\begin{array}{l} p(X) \leftarrow X = \overline{0}\,..\,n \wedge X \times X \leq n, \\ q(X) \leftarrow p(X) \wedge \mathit{not}\, p(X + \overline{1}) \end{array} \tag{13}$$

(program (5) written in abstract notation) and the input symbol n. We want to verify that

$$\begin{array}{l} \text{if } \mathcal{I} \text{ is a stable model of this program for an input } \mathbf{i} \\ \quad \text{and } \mathbf{i}(n) \text{ is a nonnegative integer,} \\ \text{then the set } \{t : q(t) \in \mathcal{I}\} \text{ is the singleton } \{\lfloor \sqrt{\mathbf{i}(n)} \rfloor\}. \end{array} \tag{14}$$

3 Formulas

In formula (2), the implication left-to-right tells us that X does not belong to q unless p contains a pair of the form $\langle X, Y \rangle$. In program (1), this property of q

is not stated explicitly. Rather, we draw this conclusion from the fact that in the absence of rules other than (1), an element of p that has the form $\langle X, Y\rangle$ is the only possible evidence for the claim that X belongs to q. The completion process, which turns (1) into (2), encodes this form of reasoning, specific for logic programs, in the language of first-order logic.

The generalization of completion defined in this paper also takes into account another difference between CLINGO programs and first-order formulas. In CLINGO, a ground term may have several values or no values. In first-order logic, the value of every ground term is unique. Among the operation names (6) allowed in programs, not a single one represents a total function on the set of precomputed terms. Consequently, these operation names cannot be used as function symbols in a first-order language with variables for precomputed terms.

The formulas introduced in this section are first-order formulas with variables of two sorts: program variables (the same as in Section 2.1) that range over precomputed program terms and new *integer variables* that range over integers. In the semantics of the language, integers are identified with the corresponding numerals. The first three of symbols (6) correspond to total functions on integers, and they are allowed in terms with integer values. The last three are banned from formulas altogether.

The definitions below follow [12, Section 5]. *Arithmetic terms* are formed from numerals and integer variables using the operation symbols $+$, $-$, and $\times$. Arithmetic terms, symbolic constants, program variables, and the symbols *inf* and *sup* are collectively called *formula terms*. It is clear that precomputed formula terms are identical to precomputed program terms. For every ground formula term, its *value* is the precomputed term defined recursively in a natural way.

Atomic formulas are expressions of two forms: $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of formula terms, and $(t_1 \prec t_2)$, where t_1 and t_2 are formula terms and $\prec$ is one of relation names (7). *Formulas* are formed from atomic formulas using propositional connectives and quantifiers as usual in first-order logic.

An interpretation $\mathcal{I}$ *satisfies* a closed atomic formula $p(t_1, \dots, t_n)$ if the formula $p(v_1, \dots, v_n)$, where each v_i is the value of t_i, belongs to $\mathcal{I}$. This relation is extended to arbitrary closed formulas as usual in first-order logic.

A formula is *universally valid* if its universal closure is satisfied by all interpretations. For instance, if X is a program variable and I is an integer variable, then the formula $\exists X(X = I)$ is universally valid because so is its universal closure $\forall I \exists X(X = I)$. The formula $\exists I(X = I)$ expresses that X is an integer. We denote it by $is\_int(X)$.

Formulas F and G are *equivalent* to each other if $F \leftrightarrow G$ is universally valid. For instance, $p(I + J)$, where I and J are integer variables, is equivalent to $p(J + I)$.

4 Completion

4.1 Transforming Program Terms into Formulas

For any program term t, the formula $val_t(Z)$, where Z is a program variable that does not occur in t, expresses, informally speaking, that Z is one of the values of t [12, Section 6]. It is defined recursively:

- if t is a numeral, a symbolic constant, a program variable, *inf*, or *sup*, then $val_t(Z)$ is $Z = t$;
- if t is $(t_1\ op\ t_2)$, where op is $+$, $-$, or $\times$, then $val_t(Z)$ is

$$\exists IJ(Z = I\ op\ J \wedge val_{t_1}(I) \wedge val_{t_2}(J))$$

  where I, J are fresh integer variables;
- if t is (t_1/t_2), then $val_t(Z)$ is

$$\exists IJQR(I = J \times Q + R \wedge val_{t_1}(I) \wedge val_{t_2}(J) \\ \wedge J \neq \overline{0} \wedge R \geq \overline{0} \wedge R < Q \wedge Z = Q),$$

  where I, J, Q, R are fresh integer variables;
- if t is $(t_1 \backslash t_2)$, then $val_t(Z)$ is

$$\exists IJQR(I = J \times Q + R \wedge val_{t_1}(I) \wedge val_{t_2}(J) \\ \wedge J \neq \overline{0} \wedge R \geq \overline{0} \wedge R < Q \wedge Z = R),$$

  where I, J, Q, R are fresh integer variables;
- if t is $(t_1 \mathinner{..} t_2)$, then $val_t(Z)$ is

$$\exists IJK(val_{t_1}(I) \wedge val_{t_2}(J) \wedge I \leq K \wedge K \leq J \wedge Z = K),$$

  where I, J, K are fresh integer variables.

For example, $val_{X+\overline{1}}(Z)$ is

$$\exists IJ(Z = I + J \wedge I = X \wedge J = \overline{1}),$$

which is equivalent to

$$\exists I(Z = I + \overline{1} \wedge I = X).$$

Another example: $val_{\overline{1}..X}(Z)$ is

$$\exists IJK(I = \overline{1} \wedge J = X \wedge I \leq K \wedge K \leq J \wedge Z = K),$$

which is equivalent to

$$is\_int(X) \wedge \overline{1} \leq Z \wedge Z \leq X.$$

4.2 Transforming Bodies of Rules into Formulas

The translation τ^b transforms bodies of rules into formulas.[6] For any atom $p(t_1, \dots, t_n)$, each of $\tau^b(p(t_1, \dots, t_n))$, $\tau^b(not\ not\ p(t_1, \dots, t_n))$ is defined as

$$\exists Z_1 \dots Z_n(val_{t_1}(Z_1) \wedge \dots \wedge val_{t_n}(Z_n) \wedge p(Z_1, \dots, Z_n))$$

(where each Z_i is a fresh program variable), and $\tau^b(not\ p(t_1, \dots, t_n))$ is

$$\exists Z_1 \dots Z_n(val_{t_1}(Z_1) \wedge \dots \wedge val_{t_n}(Z_n) \wedge \neg p(Z_1, \dots, Z_n)).$$

For any comparison $t_1 \prec t_2$, $\tau^b(t_1 \prec t_2)$ is

$$\exists Z_1 Z_2(val_{t_1}(Z_1) \wedge val_{t_2}(Z_2) \wedge Z_1 \prec Z_2).$$

If each of $E_1, \dots, E_m$ is a literal or a comparison, then $\tau^b(E_1 \wedge \dots \wedge E_m)$ stands for $\tau^b(E_1) \wedge \dots \wedge \tau^b(E_m)$.

For instance, τ^b transforms $p(\overline{1}\,..\,X)$ into a formula equivalent to

$$is\_int(X) \wedge \exists Z(\overline{1} \leq Z \wedge Z \leq X \wedge p(Z)).$$

The expression $Y = \overline{1}\,..\,X$ is transformed into a formula equivalent to

$$\exists Z_1 Z_2(Z_1 = Y \wedge is\_int(X) \wedge \overline{1} \leq Z_2 \wedge Z_2 \leq X \wedge Z_1 = Z_2)$$

and consequently to

$$is\_int(X) \wedge \overline{1} \leq Y \wedge Y \leq X.$$

4.3 Completed Definitions

Given a program with input (Π, P) and a predicate symbol p/n that occurs in Π and does not belong to P, we describe a formula called the *completed definition* of p/n in (Π, P).

In completed definitions, the symbolic constants $c_1, \dots, c_l$ from P are represented by program variables $C_1, \dots, C_l$, which are assumed to be pairwise distinct and different from the variables occurring in Π.

The *definition* of a predicate symbol p/n in (Π, P) is the set of all rules of Π that have the forms

$$p(t_1, \dots, t_n) \leftarrow Body \tag{15}$$

and

$$\{p(t_1, \dots, t_n)\} \leftarrow Body\,. \tag{16}$$

If the definition of p/n in (Π, P) consists of the rules $R_1, \dots, R_k$, then the *formula representations* $F_1, \dots, F_k$ of these rules are constructed as follows. Take fresh program variables $V_1, \dots, V_n$. If R_i is (15), then F_i is the formula

$$\tau^b(Body) \wedge val_{t_1}(V_1) \wedge \dots \wedge val_{t_n}(V_n).$$

[6] It differs from the translation τ^B [12, Section 6] in that it disregards the combination *not not*. Double negations are essential in the formulas that characterize stable models but not in completion formulas.

If R_i is (16), then F_i is the formula

$$\tau^b(Body) \wedge p(V_1, \dots, V_n) \wedge val_{t_1}(V_1) \wedge \cdots \wedge val_{t_n}(V_n). \tag{17}$$

The *completed definition* of p/n in Π is obtained from the formula

$$\forall V_1 \cdots V_n \left(p(V_1, \dots, V_n) \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{U}_i F_i \right),$$

where $\mathbf{U}_i$ is the list of all variables occurring in rule R_i, by substituting $C_1, \dots, C_l$ for $c_1, \dots, c_l$.

Example 1, continued. The completed definition of $q/1$ in this program is

$$\forall V(q(V) \leftrightarrow \exists XY(\tau^b(p(X)) \wedge \tau^b(p(Y)) \wedge val_{X+Y}(V))). \tag{18}$$

It is equivalent to

$$\forall V(q(V) \leftrightarrow \exists IJ(p(I) \wedge p(J) \wedge V = I + J)). \tag{19}$$

Example 2, continued. The completed definition of $p/1$ in this program is

$$\forall V(p(V) \leftrightarrow \exists X(\tau^b(X = \overline{0}\,..\,C) \wedge \tau^b(X \times X \leq C) \wedge V = X)).$$

It is equivalent to

$$\forall V(p(V) \leftrightarrow \exists I(I = V \wedge 0 \leq I \wedge I \leq C \wedge I \times I \leq C) \wedge is\_int(C)). \tag{20}$$

The completed definition of $q/1$ is

$$\forall V(q(V) \leftrightarrow \exists X(\tau^b(p(X)) \wedge \tau^b(not\, p(X + \overline{1})) \wedge V = X)).$$

It is equivalent to

$$\forall V(q(V) \leftrightarrow \exists I(I = V \wedge p(I) \wedge \neg p(I + \overline{1}))). \tag{21}$$

To clarify the role of substituting variables for input symbols in the process of forming a completed definition, consider the modification of the program from Example 2 in which n is not treated as an input symbol, so that the set of input symbols is empty. The completed definition of $p/1$ is equivalent, in this case, to

$$\forall V(p(V) \leftrightarrow \exists I(I = V \wedge 0 \leq I \wedge I \leq n \wedge I \times I \leq n) \wedge is\_int(n)). \tag{22}$$

The subformula $is\_int(n)$ is false because the symbolic constant n is not an integer. It follows that formula (22) is equivalent to $\forall V \neg p(V)$.

4.4 Soundness of Completion

The *completion* of a program with input (Π, P) is the conjunction of the following formulas:

- for every predicate symbol that occurs in Π and does not belong to P, its completed definition;
- for every constraint $\leftarrow Body$ in Π, the formula obtained from the universal closure of $\neg\tau^b(Body)$ by substituting the variables $C_1, \dots, C_l$ for $c_1, \dots, c_l$.

It is clear that the completion has no free variables other than $C_1, \dots, C_l$.

In the statement of the theorem, (Π, P) is a program with input; the expression $Comp(C_1, \dots, C_l)$ stands for its completion.

Theorem. *Every stable model of (Π, P) for an input* $\mathbf{i}$ *satisfies the sentence $Comp(\mathbf{i}(c_1), \dots, \mathbf{i}(c_l))$.*

Corollary. *For any formula $F(C_1, \dots, C_l)$ with all free variables explicitly shown, if the formula*

$$Comp(C_1, \dots, C_l) \to F(C_1, \dots, C_l) \tag{23}$$

is universally valid, then every stable model of (Π, P) for an input $\mathbf{i}$ *satisfies $F(\mathbf{i}(c_1), \dots, \mathbf{i}(c_l))$.*

The corollary shows that properties of stable models of a program with input can be established by proving formulas of form (23) in a first-order theory with universally valid axioms.

Example 1, continued. Let *Comp* be formula (19). To establish claim (12), it is sufficient to prove the universal validity of the formula

$$\begin{aligned} Comp \to \forall N(&\forall X(p(X) \to \exists I(I = X \wedge I \leq N)) \\ &\to \forall X(q(X) \to \exists I(I = X \wedge I \leq 2 \times N))), \end{aligned} \tag{24}$$

where N and I are integer variables and X is a program variable.

Example 2, continued. Let $Comp(C)$ be the conjunction of formulas (20) and (21). To establish claim (14), it is sufficient to prove the universal validity of the formula

$$\begin{aligned} Comp(C) \to \forall N(&N = C \wedge N \geq \overline{0} \\ &\to \exists M(\forall X(q(X) \leftrightarrow X = M) \wedge M \geq \overline{0} \\ &\qquad \wedge M \times M \leq N \wedge (M + \overline{1}) \times (M + \overline{1}) > N)), \end{aligned} \tag{25}$$

where M and N are integer variables and X is a program variable.

```
forall X1
(
    q(X1)
    <-> exists X2, X3
    (
        exists X4 (X4 = X2 and p(X4))
        and exists X5 (X5 = X3 and p(X5))
        and exists N1, N2 (X1 = N1 + N2 and N1 = X2 and N2 = X3)
    )
)
```

Fig. 1. Completed definition (18) of $q/1$ from Example 1 generated by ANTHEM. The output of ANTHEM is reformatted to improve readability

5 Verifying Properties of Programs

5.1 Generating Completed Definitions

Recall that our goal is to use a reasoning system, such as VAMPIRE, for verifying properties of CLINGO programs, and that this can be accomplished by proving formulas of form (23). To prepare an input for such a system, we need to generate the completed definitions of predicate symbols that occur in the antecedent of (23). This calculation can be performed by version 0.3 of ANTHEM.[7] When instructed, for instance, to calculate the completed definition of $q/1$ in program (4), ANTHEM generates formula (18) as shown in Figure 1. ANTHEM internally converts the output to the TFF ("typed first-order formula") format of the TPTP language [15] and passes it on to VAMPIRE as an axiom. Formulas in this format can be processed by many automated reasoners.

In the following experiments, we used VAMPIRE 4.4 with the options `--mode casc` and `--cores 8`.

5.2 Verification of Example 1

The set of axioms that were available to VAMPIRE in the experiment with Example 1 consists of two parts. One is the collection of properties of predicates and functions on integers that VAMPIRE treats as standard. The other includes several properties of the set of precomputed terms and of the correspondence between numerals and integers, such as those expressed by conditions (i)–(iii) in Section 2.2. All these axioms are universally valid in the sense of Section 3.

The claim about the relationship between $p/1$ and $q/1$ that we wanted to verify in this example is expressed as shown in Figure 2. ANTHEM transformed this formula into the TFF format, and VAMPIRE derived it from the axioms and the completed definition generated by ANTHEM (Figure 1) in a fraction of a second.

[7] https://github.com/potassco/anthem/releases

```
forall N
(
    forall X (p(X) -> exists I (I = X and I <= N))
    -> forall X (q(X) -> exists I (I = X and I <= 2 * N))
)
```

Fig. 2. The claim from Example 1

```
p(0) and forall N (N >= 0 and p(N) -> p(N + 1))
-> forall N(N >= 0 -> p(N))
```

Fig. 3. The induction axiom for $p/1$ from Example 2

5.3 Verification of Example 2

Example 2 was more of a challenge to us as the users of VAMPIRE, in two ways. First, we were unable to prove its claim using only the axiom set described in Section 5.2. Two more axioms, both expressing properties of numbers, had to be added. One axiom says that an inequality can be multiplied by a positive integer. Surprisingly, VAMPIRE 4.4, the version we worked with, was not able to prove this fact. The other axiom expresses induction for the predicate $p/1$ (Figure 3).

Second, VAMPIRE could not prove the conjecture that we are interested in "with one blow," at least in reasonable time. We used it as a proof assistant in a sense that we gave it a sequence of auxiliary conjectures, one by one. As soon as one of these "lemmas" was verified, we added it to the list of axioms.

Such interactive use of automated reasoners will be necessary, of course, when working on verifying more complex programs.

6 Proof of the Theorem

6.1 Review: Definition of a Stable Model

For any ground atom $p(\mathbf{t})$,

- $\tau(p(\mathbf{t}))$ stands for $\bigvee_{\mathbf{r}\in[\mathbf{t}]} p(\mathbf{r})$;
- $\tau(not\, p(\mathbf{t}))$ stands for $\bigvee_{\mathbf{r}\in[\mathbf{t}]} \neg p(\mathbf{r})$;
- $\tau(not\, not\, p(\mathbf{t}))$ stands for $\bigvee_{\mathbf{r}\in[\mathbf{t}]} \neg\neg p(\mathbf{r})$.

For any ground comparison $t_1 \prec t_2$, $\tau(t_1 \prec t_2)$ is

- $\top$ if the relation $\prec$ holds between some r_1 from $[t_1]$ and some r_2 from $[t_2]$;
- $\bot$ otherwise.

If each of $E_1, \ldots, E_m$ is a ground literal or a ground comparison, then $\tau(E_1 \wedge \cdots \wedge E_m)$ stands for $\tau E_1 \wedge \cdots \wedge \tau E_m$.

The *propositional image* of a ground rule R is the formula formed as follows. If R is a ground basic rule $p(\mathbf{t}) \leftarrow \mathit{Body}$, then its propositional image is

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

If R is a ground choice rule $\{p(\mathbf{t})\} \leftarrow \mathit{Body}$, then its propositional image is

$$\tau(\mathit{Body}) \rightarrow \bigwedge_{\mathbf{r} \in [\mathbf{t}]} (p(\mathbf{r}) \vee \neg p(\mathbf{r})).$$

If R is a ground constraint $\leftarrow \mathit{Body}$, then its propositional image is $\neg\tau(\mathit{Body})$.

For any program Π, its *propositional image* is the set of the propositional images of all instances R of its rules. An interpretation is a *stable model* (or *answer set*) of a program Π if it is an answer set of the propositional image of Π [13].

6.2 Leading Special Case

It is sufficient to prove the theorem for the case when P is empty. To derive the general case, we can reason as follows. If $\mathcal{I}$ is a stable model of (Π, P), then $\mathcal{I}$ is a stable model of the program Π' obtained from Π as described in Section 2.3—by adding some rules of the form $p(\mathbf{t})$ for predicate symbols p/n from P and by substituting the terms $\mathbf{i}(c)$ for all symbolic constants c in P. By the special case of the theorem with the empty P, $\mathcal{I}$ satisfies the completion Comp' of Π'. It remains to observe that every conjunctive term of $\mathit{Comp}(\mathbf{i}(c_1), \dots, \mathbf{i}(c_l))$ is a conjunctive term of Comp'.

To prove the special case when P is empty, we need to justify three claims:

Claim 1. *If a program Π contains a constraint $\leftarrow \mathit{Body}$, then every stable model of Π satisfies the universal closure of $\neg\tau^b(\mathit{Body})$.*

In the statements of Claim 2 and Claim 3, the symbols F_i, $\mathbf{U}_i$, and $V_1, \dots, V_n$ are understood as in Section 4.3.

Claim 2. *For every predicate symbol p/n occurring in Π, every stable model of Π satisfies the universal closure of the formula*

$$\bigvee_{i=1}^{k} \exists \mathbf{U}_i F_i \rightarrow p(V_1, \dots, V_n).$$

Claim 3. *For every predicate symbol p/n occurring in Π, every stable model of Π satisfies the universal closure of the formula*

$$p(V_1, \dots, V_n) \rightarrow \bigvee_{i=1}^{k} \exists \mathbf{U}_i F_i.$$

6.3 Two Lemmas

The two lemmas below are similar to Propositions 1 and 2 from [12].

Lemma 1. *For any ground program term t and any precomputed term r, the formula $val_t(r)$ is equivalent to $\top$ if $r \in [t]$ and to $\bot$ otherwise.*

Proof. The proof is by induction on t. If t is a numeral, a symbolic constant, *inf*, or *sup*, then $r \in [t]$ iff r is t. On the other hand, $val_t(r)$ is $r = t$; this formula is equivalent to $\top$ if r is t and to $\bot$ otherwise.

Assume that the assertion of the lemma holds for t_1 and t_2.

If t is $(t_1\ op\ t_2)$, where op is $+$, $-$, or $\times$, then $val_t(r)$ is

$$\exists IJ(r = I\ op\ J \wedge val_{t_1}(I) \wedge val_{t_2}(J)).$$

An arbitrary interpretation satisfies this formula iff there exist integers i, j such that

$$r \text{ is } \overline{i\ op\ j}, \quad \overline{i} \in [t_1], \quad \text{and } \overline{j} \in [t_2].$$

This condition holds iff $r \in [t]$.

If t is (t_1/t_2), then $val_t(r)$ is

$$\exists IJQR(I = J \times Q + R \wedge val_{t_1}(I) \wedge val_{t_2}(J) \\ \wedge J \neq \overline{0} \wedge R \geq \overline{0} \wedge R < Q \wedge r = Q).$$

An arbitrary interpretation satisfies this formula iff there exist integers i, j, q, *rem* such that

$$i = jq + rem, \quad \overline{i} \in [t_1], \quad \overline{j} \in [t_2], \quad j \neq 0, \quad 0 \leq rem < q, \quad \text{and } r \text{ is } \overline{q}.$$

Equivalently: Iff there exist integers i and j such that

$$\overline{i} \in [t_1], \quad \overline{j} \in [t_2], \quad j \neq 0, \quad \text{and } r \text{ is } \overline{\lfloor i/j \rfloor}.$$

This condition holds iff $r \in [t]$.

If t is $(t_1 \backslash t_2)$, then the proof is similar.

If t is $(t_1\,..\,t_2)$, then $val_t(r)$ is

$$\exists IJK(val_{t_1}(I) \wedge val_{t_2}(J) \wedge I \leq K \wedge K \leq J \wedge r = K).$$

An arbitrary interpretation satisfies this formula iff there exist integers i, j, k such that

$$\overline{i} \in [t_1], \quad \overline{j} \in [t_2], \quad i \leq k \leq j, \quad \text{and } r \text{ is } \overline{k}.$$

This condition holds iff $r \in [t]$.

Lemma 2. *If E is a ground literal or ground comparison, then $\tau^b(E)$ is equivalent to τE.*

Proof. This is immediate from Lemma 1.

6.4 Proof of Claim 1

Let $\mathcal{I}$ be a stable model of a program Π, let $\leftarrow E_1 \wedge \cdots \wedge E_m$ be a constraint from Π, let $\mathbf{x}$ be the list of variables occurring in this constraint, and let $\mathbf{r}$ be a list of precomputed terms of the same length as $\mathbf{x}$. Since the rule

$$\leftarrow (E_1)^{\mathbf{x}}_{\mathbf{r}} \wedge \cdots \wedge (E_m)^{\mathbf{x}}_{\mathbf{r}}$$

is an instance of a rule of Π, the propositional image of Π includes the formula

$$\neg(\tau((E_1)^{\mathbf{x}}_{\mathbf{r}}) \wedge \cdots \wedge \tau((E_m)^{\mathbf{x}}_{\mathbf{r}})). \tag{26}$$

Consequently, $\mathcal{I}$ satisfies (26). By Lemma 2, it follows that $\mathcal{I}$ satisfies the formula

$$\neg(\tau^b((E_1)^{\mathbf{x}}_{\mathbf{r}}) \wedge \cdots \wedge \tau^b((E_m)^{\mathbf{x}}_{\mathbf{r}})),$$

which can be also represented as

$$\neg(\tau^b(E_1)^{\mathbf{x}}_{\mathbf{r}} \wedge \cdots \wedge \tau^b(E_m)^{\mathbf{x}}_{\mathbf{r}})$$

and as

$$(\neg\tau^b(E_1 \wedge \cdots \wedge E_m))^{\mathbf{x}}_{\mathbf{r}}.$$

Since $\mathbf{r}$ here is an arbitrary tuple of precomputed terms, it follows that $\mathcal{I}$ satisfies the universal closure of $\neg\tau^b(E_1 \wedge \cdots \wedge E_m)$.

6.5 Proof of Claim 2

Let $\mathcal{I}$ be a stable model of a program Π, and let p/n be a predicate symbol occurring in Π. We need to show that $\mathcal{I}$ satisfies the universal closure of each of the formulas

$$F_i \rightarrow p(V_1, \ldots, V_n) \tag{27}$$

($i = 1, \ldots, k$). If F_i is a formula of form (17), corresponding to a choice rule, then $p(V_1, \ldots, V_n)$ is one of its conjunctive terms so that (27) is universally valid. Otherwise, F_i is the formula

$$\tau^b(E_1) \wedge \cdots \wedge \tau^b(E_m) \wedge val_{t_1}(V_1) \wedge \cdots \wedge val_{t_n}(V_n),$$

corresponding to a basic rule

$$p(t_1, \ldots, t_n) \leftarrow E_1 \wedge \cdots \wedge E_m. \tag{28}$$

The set of free variables of (27) consists of the variables $\mathbf{U}_i$ that occur in rule (28) and the variables $V_1, \ldots, V_n$. We need to prove that for every tuple $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{U}_i$ and every tuple $s_1, \ldots, s_n$ of precomputed terms, the formula

$$(\tau^b(E_1))^{\mathbf{U}_i}_{\mathbf{r}} \wedge \cdots \wedge (\tau^b(E_m))^{\mathbf{U}_i}_{\mathbf{r}} \wedge val_{t'_1}(s_1) \wedge \cdots \wedge val_{t'_n}(s_n) \rightarrow p(s_1, \ldots, s_n), \tag{29}$$

where t'_j $(j = 1, \ldots, n)$ stands for $(t_j)^{\mathbf{U}_i}_{\mathbf{r}}$, is universally valid. By Lemma 1, the formula $val_{t'_j}(s_j)$ is equivalent to $\top$ if $s_j \in [t'_j]$ and to $\bot$ otherwise. Consequently, it is sufficient to consider the case when

$$s_1 \in [t'_1], \ldots, s_n \in [t'_n] \tag{30}$$

(otherwise, (29) is universally valid). It remains to show that, under condition (30), $\mathcal{I}$ satisfies the formula

$$(\tau^b(E_1))^{\mathbf{U}_i}_{\mathbf{r}} \wedge \cdots \wedge (\tau^b(E_m))^{\mathbf{U}_i}_{\mathbf{r}} \rightarrow p(s_1, \ldots, s_n).$$

This formula can be represented also as

$$\tau^b\left((E_1)^{\mathbf{U}_i}_{\mathbf{r}}\right) \wedge \cdots \wedge \tau^b\left((E_m)^{\mathbf{U}_i}_{\mathbf{r}}\right) \rightarrow p(s_1, \ldots, s_n).$$

By Lemma 2, it is equivalent to

$$\tau\left((E_1)^{\mathbf{U}_i}_{\mathbf{r}}\right) \wedge \cdots \wedge \tau\left((E_m)^{\mathbf{U}_i}_{\mathbf{r}}\right) \rightarrow p(s_1, \ldots, s_n). \tag{31}$$

To show that $\mathcal{I}$ satisfies (31), consider the instance

$$p(t'_1, \ldots, t'_n) \leftarrow (E_1)^{\mathbf{U}_i}_{\mathbf{r}} \wedge \cdots \wedge (E_m)^{\mathbf{U}_i}_{\mathbf{r}}$$

of rule (28). The propositional image of that instance has the form

$$\tau\left((E_1)^{\mathbf{U}_i}_{\mathbf{r}}\right) \wedge \cdots \wedge \tau\left((E_m)^{\mathbf{U}_i}_{\mathbf{r}}\right) \rightarrow C, \tag{32}$$

where C is a conjunction containing the conjunctive term $p(s_1, \ldots, s_n)$. Since the interpretation $\mathcal{I}$ is a stable model of Π, it satisfies (32) and, consequently, (31).

6.6 Proof of Claim 3

Consider the set Γ of formulas that includes

- for every instance $p(\mathbf{t}) \leftarrow \mathit{Body}$ of a basic rule of Π, the formulas

  $$\tau(\mathit{Body}) \rightarrow p(\mathbf{r})$$

  for all $\mathbf{r}$ in $[\mathbf{t}]$, and
- for every instance $\{p(\mathbf{t})\} \leftarrow \mathit{Body}$ of a choice rule of Π, the formulas

  $$\tau(\mathit{Body}) \wedge \neg\neg p(\mathbf{r}) \rightarrow p(\mathbf{r})$$

  for all $\mathbf{r}$ in $[\mathbf{t}]$.

This set is strongly equivalent [13] to the propositional image of the program obtained from Π by removing all constraints. It follows that every stable model $\mathcal{I}$ of Π is a stable model of Γ and, consequently, is supported by Γ [3, Proposition 2].

In other words, every element A of $\mathcal{I}$ is the consequent of an implication from Γ such that its antecedent is satisfied by $\mathcal{I}$.

To prove Claim 3, we need to show that for every predicate symbol p/n occurring in Π and every tuple $s_1, \dots, s_n$ of precomputed terms, every stable model $\mathcal{I}$ of Π satisfies the formula

$$p(s_1, \dots, s_n) \to \bigvee_{i=1}^{k} \exists \mathbf{U}_i (F_i)_{s_1,\dots,s_n}^{V_1,\dots,V_n}. \tag{33}$$

Assume that $p(s_1, \dots, s_n)$ is an element of $\mathcal{I}$; we need to show that $\mathcal{I}$ satisfies the consequent of (33). Consider an implication from Γ with the consequent $p(s_1, \dots, s_n)$ such that its antecedent is satisfied by $\mathcal{I}$.

Case 1. This implication is the formula

$$\tau\left((E_1)_{\mathbf{r}}^{\mathbf{U}_i}\right) \wedge \dots \wedge \tau\left((E_m)_{\mathbf{r}}^{\mathbf{U}_i}\right) \to p(s_1, \dots, s_n) \tag{34}$$

corresponding to an instance

$$p\left((t_1)_{\mathbf{r}}^{\mathbf{U}_i}, \dots, (t_n)_{\mathbf{r}}^{\mathbf{U}_i}\right) \leftarrow (E_1)_{\mathbf{r}}^{\mathbf{U}_i} \wedge \dots \wedge (E_m)_{\mathbf{r}}^{\mathbf{U}_i}$$

of a basic rule

$$p(t_1, \dots, t_n) \leftarrow E_1 \wedge \dots \wedge E_m$$

such that

$$s_1 \in \left[(t_1)_{\mathbf{r}}^{\mathbf{U}_i}\right], \dots, s_n \in \left[(t_n)_{\mathbf{r}}^{\mathbf{U}_i}\right]. \tag{35}$$

By Lemma 2, the antecedent of (34) is equivalent to the formula

$$\tau^b\left((E_1)_{\mathbf{r}}^{\mathbf{U}_i}\right) \wedge \dots \wedge \tau^b\left((E_m)_{\mathbf{r}}^{\mathbf{U}_i}\right),$$

which can be also represented as

$$(\tau^b(E_1))_{\mathbf{r}}^{\mathbf{U}_i} \wedge \dots \wedge \tau^b((E_m))_{\mathbf{r}}^{\mathbf{U}_i}.$$

On the other hand, from conditions (35) and Lemma 1, we conclude that each of the formulas

$$val_{t_1}(s_1)_{\mathbf{r}}^{\mathbf{U}_i}, \dots, val_{t_n}(s_n)_{\mathbf{r}}^{\mathbf{U}_i}$$

is equivalent to $\top$. It follows that $\mathcal{I}$ satisfies the conjunction

$$(\tau^b(E_1))_{\mathbf{r}}^{\mathbf{U}_i} \wedge \dots \wedge \tau^b((E_m))_{\mathbf{r}}^{\mathbf{U}_i} \wedge val_{t_1}(s_1)_{\mathbf{r}}^{\mathbf{U}_i} \wedge \dots \wedge val_{t_1}(s_1)_{\mathbf{r}}^{\mathbf{U}_i},$$

which can be written as

$$(F_i)_{s_1,\dots,s_n,\mathbf{r}}^{V_1,\dots,V_n,\mathbf{U}_i}.$$

It follows that I satisfies the consequent of (33).

Case 2. The implication from Γ with the consequent $p(s_1, \ldots, s_n)$ such that its antecedent is satisfied by $\mathcal{I}$ is the formula

$$\tau\left((E_1)_{\mathbf{r}}^{\mathbf{U}_i}\right) \wedge \cdots \wedge \tau\left((E_m)_{\mathbf{r}}^{\mathbf{U}_i}\right) \wedge \neg\neg p(s_1, \ldots, s_n) \rightarrow p(s_1, \ldots, s_n)$$

corresponding to an instance

$$\left\{p\left((t_1)_{\mathbf{r}}^{\mathbf{U}_i}, \ldots, (t_n)_{\mathbf{r}}^{\mathbf{U}_i}\right)\right\} \leftarrow (E_1)_{\mathbf{r}}^{\mathbf{U}_i} \wedge \cdots \wedge (E_m)_{\mathbf{r}}^{\mathbf{U}_i}$$

of a choice rule

$$\{p(t_1, \ldots, t_n)\} \leftarrow E_1 \wedge \cdots \wedge E_m$$

such that the terms $t_1, \ldots, t_n$ satisfy condition (35). The proof is similar.

7 Related Work

From early research on the relationship between stable models and completion for programs without arithmetic [5], we know that every stable model of such a program is a model of its completion and that the converse holds under a syntactic condition that is now called *tightness* [3, 4]. That work has been extended to CLINGO programs with arithmetic [8], but completed definitions as defined in that paper are not expressed in a standard first-first order language and, consequently, cannot be processed by existing theorem provers.

The definition of a formula proposed in earlier work on ANTHEM [12] does not suffer from that defect. VAMPIRE is used there to verify the strong equivalence relation between logic programs.

Acknowledgements. We are grateful to Laura Kovács, Giles Reger, and Martin Suda for taking the time to answer our questions about the use of VAMPIRE. Also, we would like to thank the anonymous referee for giving us useful suggestions.

References

1. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
2. Erdem, E., Gelfond, M., Leone, N.: Applications of ASP. AI Magazine **37**(3), 53–68 (2016)
3. Erdem, E., Lifschitz, V.: Fages' theorem for programs with nested expressions. In: Codognet, P. (ed.) Proceedings of the Seventeenth International Conference on Logic Programming (ICLP'01). Lecture Notes in Computer Science, vol. 2237, pp. 242–254. Springer-Verlag (2001)
4. Erdem, E., Lifschitz, V.: Tight logic programs. Theory and Practice of Logic Programming **3**(4-5), 499–518 (2003)
5. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1**, 51–60 (1994)

6. Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R., Teppan, E.: Industrial applications of answer set programming. Künstliche Intelligenz **32**(2-3), 165–176 (2018)
7. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015), http://arxiv.org/abs/1507.06576
8. Harrison, A., Lifschitz, V., Raju, D.: Program completion in the input language of GRINGO. Theory and Practice of Logic Programming **17**(5-6), 855–871 (2017)
9. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) Proceedings of the Twenty-fifth International Conference on Computer Aided Verification (CAV'13). Lecture Notes in Computer Science, vol. 8044, pp. 1–35. Springer-Verlag (2013)
10. Lifschitz, V.: Achievements in answer set programming. Theory and Practice of Logic Programming **17**(5-6), 961–973 (2017)
11. Lifschitz, V., Lühne, P., Schaub, T.: anthem: Transforming gringo programs into first-order theories (preliminary report). In: Fandinno, J., Fichte, J. (eds.) Proceedings of the Eleventh Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'18) (2018)
12. Lifschitz, V., Lühne, P., Schaub, T.: Verifying strong equivalence of programs in the input language of GRINGO. In: Balduccini, M., Lierler, Y., Woltran, S. (eds.) Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19). Lecture Notes in Artificial Intelligence, vol. 11481, pp. 270–283. Springer-Verlag (2019)
13. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2**(4), 526–541 (2001)
14. Lloyd, J., Topor, R.: Making Prolog more expressive. Journal of Logic Programming **1**(3), 225–240 (1984)
15. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning **59**(4), 483–502 (2017)