Verifying Strong Equivalence of Programs in the Input Language of GRINGO

Vladimir Lifschitz[1], Patrick Lühne[2], and Torsten Schaub[2]

[1] University of Texas at Austin, USA
[2] University of Potsdam, Germany

Abstract. The semantics of the input language of the ASP grounder GRINGO uses a translation that converts a logic program, which may contain variables and arithmetic operations, into a set of infinitary propositional formulas. In this note, we show that the result of that translation can be replaced in some cases by a finite set of first-order sentences. The translator ANTHEM constructs that set of sentences and converts it to a format that can be processed by automated reasoning tools. ANTHEM, in combination with the first-order theorem prover VAMPIRE, allows us to verify the strong equivalence of programs in the language of GRINGO.

1 Introduction

The semantics of the input language of the ASP grounder GRINGO [4] uses a translation τ that converts a logic program, which may contain variables and arithmetic operations, into a set of infinitary propositional formulas. In this note, we show that the set produced by τ can be replaced in some cases by a finite set of first-order sentences. The translator ANTHEM constructs that set of sentences and converts it to a format that can be processed by automated reasoning tools.

In combination with the first-order theorem prover VAMPIRE [8], ANTHEM allows us to verify strong equivalence of programs in the language of GRINGO. This relation between logic programs is important because it guarantees the possibility of replacing one program by the other in any context [10]. Earlier work on verifying strong equivalence [7, 2] was restricted to programs that do not contain arithmetic operations.

The definition of a program in Section 2 largely follows [4, 6], and it disregards some details of the syntax of GRINGO. For instance, about the set of symbolic constants we assume only that it is countably infinite and totally ordered; in GRINGO, symbolic constants are actually strings, and they are ordered lexicographically. Dropping the condition `abc < acb` from the body of a rule in a GRINGO program does not change the set of stable models, but this fact is not reflected in our more abstract theory of strong equivalence.

The class of programs studied in this note is more restricted than that in the papers quoted above, and the version of τ defined in Section 3 does not use infinite conjunctions and disjunctions. It produces, generally, an infinite set of finite formulas. Some of the theorems in this paper refer, however, to infinitary propositional formulas and to the strong equivalence relation between them [5].

2 Programs

We assume that three countably infinite sets of symbols are selected: *numerals*, *symbolic constants*, and *program variables*. (We talk about "program" variables to distinguish them from "integer" variables, introduced in Section 5 below. Integer variables are allowed in formulas but not in programs.) We assume that a 1-to-1 correspondence between numerals and integers is chosen; we denote the numeral corresponding to an integer n by $\overline{n}$.

Program terms are defined recursively as follows. (Program terms are to be distinguished from "formula terms," defined in Section 5.)

- 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*

$$+ \quad - \quad \times \quad / \quad \backslash \quad ..$$

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

If t is a term, then $-t$ 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.

We assume that a total order on precomputed program terms is chosen, where

- *inf* is its least element and *sup* is its greatest element,
- for any integers m and n, $\overline{m} < \overline{n}$ iff $m < n$, and
- 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. A *literal* is an atom possibly preceded by one or two occurrences of *not*. A *comparison* is an expression of the form $(t_1\ rel\ t_2)$, where t_1, t_2 are program terms and *rel* is one of the *relation names*

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{1}$$

A *rule* is an expression of the form

$$Head \leftarrow Body, \tag{2}$$

where

- *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 an atom in braces (then (2) is a *choice rule*), or empty (then (2) is a *constraint*).

A *program* is a finite set of rules.

3 Stable Models

An *interpretation* is a set of precomputed atoms. We define which interpretations are stable models of a program Π by first transforming Π into a set $\tau\Pi$ of propositional formulas formed from precomputed atoms and then referring to the definition of a stable model (answer set) [3] of a set of propositional formulas.

In propositional formulas, we consider the connectives

$$\bot \text{ (“false”)},\ \wedge,\ \vee,\ \rightarrow \tag{3}$$

primitives; $\top$ is shorthand for $\bot \rightarrow \bot$, $\neg F$ is shorthand for $F \rightarrow \bot$, and $F \leftrightarrow G$ is shorthand for $(F \rightarrow G) \wedge (G \rightarrow F)$.

Before defining τ, we define, for every ground program term t, the set $[t]$ of its *values*:

- 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 $\overline{n_1 + n_2}$ 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 $(t_1 \backslash t_2)$, then $[t]$ is the set of numerals $\overline{n_1 - n_2 \cdot \lfloor n_1/n_2 \rfloor}$ 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 $(t_1 \,..\, t_2)$, then $[t]$ is the set of numerals $\overline{m}$ for all integers m such that, for some integers n_1, n_2,

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

It is clear that values of a ground program term are precomputed program terms.

For example,

- the only value of $\overline{2} \times \overline{2}$ is $\overline{4}$;
- the values of $\overline{1} \,..\, \overline{3}$ are $\overline{1}$, $\overline{2}$, $\overline{3}$;
- $\overline{1}/\overline{0}$ has no values;
- $a + \overline{1}$, where a is a symbolic constant, has no values.

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

Now we can turn to the definition of τ. For any ground atom $p(\mathbf{t})$,

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

For example,

- $\tau p(\overline{1} \,..\, \overline{3})$ is $p(\overline{1}) \vee p(\overline{2}) \vee p(\overline{3})$,
- $\tau(\mathit{not}\, p(\overline{1} \,..\, \overline{3}))$ is $\neg p(\overline{1}) \vee \neg p(\overline{2}) \vee \neg p(\overline{3})$.

For any ground comparison $t_1\, \mathit{rel}\, t_2$, we define $\tau(t_1\, \mathit{rel}\, t_2)$ as

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

For example, $\tau(\overline{1} = \overline{1}\,..\,\overline{3})$ is $\top$.

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

If R is a ground basic rule $p(\mathbf{t}) \leftarrow \mathit{Body}$, then τR is the propositional formula

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

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

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

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

An *instance* of a rule is a ground rule obtained from it by substituting precomputed program terms for program variables. For any program Π, $\tau\Pi$ is the set of the propositional formulas τR for all instances R of the rules of Π.

For example, the instances of the rule

$$q(X + \overline{1}) \leftarrow p(X) \tag{4}$$

are the ground rules

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

for all precomputed program terms r. If r is a numeral $\overline{n}$, then the result of applying τ to this instance is

$$p(\overline{n}) \to q(\overline{n+1}). \tag{5}$$

If r is not a numeral, then the result is

$$p(r) \to \top \tag{6}$$

(because the empty conjunction is understood as $\top$). Consequently, the result of applying τ to rule (4) consists of propositional formulas (5) for all integers n and propositional formulas (6) for all precomputed program terms r other than numerals.

Similarly, the result of applying τ to the rule

$$q(X) \leftarrow p(X - \overline{1}) \tag{7}$$

consists of the propositional formulas

$$p(\overline{n-1}) \to q(\overline{n}) \tag{8}$$

for all integers n and the propositional formulas

$$\bot \to q(r) \tag{9}$$

for all precomputed program terms r other than numerals.

An interpretation is a *stable model* of a program Π if it is a stable model of $\tau\Pi$.

4 Strong Equivalence

Recall that sets Γ_1 and Γ_2 of propositional formulas are said to be *strongly equivalent* to each other if for every set Γ of propositional formulas, $\Gamma_1 \cup \Gamma$ has the same stable models as $\Gamma_2 \cup \Gamma$. Two sets of propositional formulas are strongly equivalent iff each of them can be derived from the other in the propositional logic of here-and-there, which is intermediate between classical and intuitionistic [10].

We extend the definition of strong equivalence to programs in the sense of Section 2 as follows: Programs Π_1 and Π_2 are *strongly equivalent* to each other if $\tau\Pi_1$ is strongly equivalent to $\tau\Pi_2$.

For example, one-rule program (4) is strongly equivalent to (7). To justify this claim, note that the sets of formulas obtained from these two rules by applying the transformation τ are intuitionistically equivalent. Indeed, the set of formulas (5) for all integers n is identical to the set of formulas (8) for all integers n; on the other hand, all formulas (6) and (9) are provable intuitionistically.

This argument is quite simple, but it involves reasoning about infinite sets of propositional formulas. It is not immediately clear how to automate generating proofs of this kind. This is the challenge that we are interested in. Our approach is to replace τ by a transformation τ^*, defined in Section 6 below, which produces a finite set of first-order sentences. Sets of that kind can be processed by automated reasoning tools. The transformation τ^* is somewhat similar to the transformations defined in [6] and implemented in an earlier version of ANTHEM [9].

To take another example, consider the rules

$$q(X) \leftarrow p(X) \tag{10}$$

and

$$q(X + \overline{1}) \leftarrow p(X + \overline{1}). \tag{11}$$

They are not strongly equivalent to each other. Indeed, adding the rule $p(a)$, where a is a symbolic constant, to the former gives a program with the stable model $\{p(a), q(a)\}$; adding that rule to the latter gives a program with the stable model $\{p(a)\}$.

We call a rule *trivial* if it is strongly equivalent to the empty program. It is clear that a rule R is trivial iff τR is provable in the logic of here-and-there. Removing a trivial rule from a program does not affect its stable models. For example, the rule

$$p(\overline{4}) \leftarrow p(\overline{2} \times \overline{2}) \tag{12}$$

is trivial because the result

$$p(\overline{4}) \rightarrow p(\overline{4})$$

of applying τ to it is intuitionistically provable. The rule

$$p(\overline{1}\,..\,\overline{3}) \leftarrow p(\overline{1}\,..\,\overline{3}) \tag{13}$$

is not trivial because the program obtained by adding it to the fact $p(\overline{1})$ has $p(\overline{2})$ and $p(\overline{3})$ in its stable model.

5 Formulas

In this section, we define the target language of the new translation τ^*. This is a first-order language with variables of two sorts. First, we include program variables, introduced in Section 2; they range over precomputed program terms. Second, *integer variables* range over numerals (or, equivalently, integers).

Arithmetic terms are formed from numerals and integer variables using the operation symbols $+$, $-$, and $\times$. Note that $/$ and $\backslash$ are not allowed in arithmetic terms. This is because division by 0 is undefined, and in first-order logic, a function symbol is expected to denote a total function. Intervals are not allowed either because an interval expression, generally, does not have a single value.

We collectively refer to arithmetic terms, symbolic constants, program variables, and the symbols *inf* and *sup* as *formula terms*. Thus, the set of program terms (defined in Section 2) and the set of formula terms partially overlap. In a program term, integer variables are not allowed; on the other hand, in a formula term, arithmetic operations cannot be applied to symbolic constants and program variables. It is clear that the only precomputed arithmetic terms are numerals. Precomputed formula terms are identical to precomputed program terms so that we can talk simply about "precomputed terms."

Atomic formulas are expressions of the forms

- $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of formula terms (separated by commas, possibly empty), and
- $(t_1\, rel\, t_2)$, where t_1 and t_2 are formula terms and *rel* is one of the relation names (1).

Formulas are formed from atomic formulas using propositional connectives (3) and the quantifiers $\forall$ and $\exists$ as usual in first-order logic. It is clear that every propositional formula in the sense of Section 3—a propositional combination of precomputed atoms—is a closed formula in the sense of this definition.

The satisfaction relation between interpretations and propositional formulas is extended to arbitrary closed formulas as usual in classical logic; program variables range over precomputed program terms, and integer variables range over numerals. Two closed formulas are *classically equivalent* to each other if they are satisfied by the same interpretations.

For describing the relationship between the translations τ and τ^*, we need a translation that converts closed formulas in this language into infinitary propositional formulas formed from precomputed atoms. The infinitary propositional formula F^{prop} corresponding to a closed formula F is defined as follows:

- if F is $p(\mathbf{t})$, then F^{prop} is obtained from F by replacing each member of $\mathbf{t}$ by its value;
- if F is $(t_1\, rel\, t_2)$, then F^{prop} is $\top$ if the values of t_1 and t_2 are in the relation *rel*, and $\bot$ otherwise;
- $\bot^{\mathrm{prop}}$ is $\bot$;
- $(F \odot G)^{\mathrm{prop}}$ is $(F^{\mathrm{prop}} \odot G^{\mathrm{prop}})$ for every binary connective $\odot$;

- $(\forall X F(X))^{\text{prop}}$ is the conjunction of the formulas $F(r)^{\text{prop}}$ over all precomputed terms r if X is a program variable, and over all numerals r if X is an integer variable;
- $(\exists X F(X))^{\text{prop}}$ is the disjunction of the formulas $F(r)^{\text{prop}}$ over all precomputed terms r if X is a program variable, and over all numerals r if X is an integer variable.

It is clear that a closed formula F is satisfied by the same interpretations as the corresponding infinitary propositional formula F^{prop}. Closed formulas F and G are classically equivalent iff the infinitary propositional formulas F^{prop} and F^{prop} are classically equivalent. If F^{prop} and G^{prop} are strongly equivalent, then F and G are classically equivalent.

For example, if F is

$$\forall X \exists N(N \geq 0 \wedge p(X, N)),$$

where X is a program variable and N is an integer variable, then F^{prop} is

$$\bigwedge_r \left(\bigvee_{n \geq 0} (\top \wedge p(r, \overline{n})) \vee \bigvee_{n < 0} (\bot \wedge p(r, \overline{n})) \right),$$

where r ranges over precomputed terms and n ranges over integers. This formula is strongly equivalent to

$$\bigwedge_r \bigvee_{n \geq 0} p(r, \overline{n}).$$

6 Transforming Programs into Formulas

Prior to defining τ^*, we define, for every program term t, a formula $val_t(Z)$, where Z is a program variable that does not occur in t. That formula expresses, informally speaking, that Z is one of the values of t. This property is made precise in Proposition 1 below.

The definition is recursive:

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

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

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

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

  where I, J, Q, R are fresh integer variables;

– if t is $(t_1 \backslash t_2)$, then $val_t(Z)$ is

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

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

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

where I, J are integer variables.

Proposition 1. *For any ground program term t and any precomputed term r, the formula $val_t(r)^{\mathrm{prop}}$ is strongly equivalent to $\top$ if $r \in [t]$ and to $\bot$ otherwise.*

This assertion can be proved by induction on t.

The last thing to do in preparation for defining τ^* is to define the translation τ^B that is applied to expressions in the body of the rule:

– $\tau^B(p(t_1, \ldots, t_k))$ is

$$\exists Z_1 \ldots Z_k(val_{t_1}(Z_1) \wedge \cdots \wedge val_{t_k}(Z_k) \wedge p(Z_1, \ldots, Z_k));$$

– $\tau^B(not\, p(t_1, \ldots, t_k))$ is

$$\exists Z_1 \ldots Z_k(val_{t_1}(Z_1) \wedge \cdots \wedge val_{t_k}(Z_k) \wedge \neg p(Z_1, \ldots, Z_k));$$

– $\tau^B(not\, not\, p(t_1, \ldots, t_k))$ is

$$\exists Z_1 \ldots Z_k(val_{t_1}(Z_1) \wedge \cdots \wedge val_{t_k}(Z_k) \wedge \neg\neg p(Z_1, \ldots, Z_k));$$

– $\tau^B(t_1\, rel\, t_2)$ is

$$\exists Z_1 Z_2(val_{t_1}(Z_1) \wedge val_{t_2}(Z_2) \wedge Z_1\, rel\, Z_2);$$

where each Z_i is a fresh program variable.

From Proposition 1, we conclude:

Proposition 2. *If L is a ground literal or ground comparison, then $(\tau^B L)^{\mathrm{prop}}$ is strongly equivalent to τL.*

Now we define

$$\tau^*(Head \leftarrow B_1 \wedge \cdots \wedge B_n)$$

as the universal closure of the formula

$$\tau^B(B_1) \wedge \cdots \wedge \tau^B(B_n) \rightarrow H,$$

where H is

- $\forall Z_1 \dots Z_k(val_{t_1}(Z_1) \wedge \dots \wedge val_{t_k}(Z_k) \to p(Z_1, \dots, Z_k))$
  if *Head* is $p(t_1, \dots, t_k)$;
- $\forall Z_1 \dots Z_k(val_{t_1}(Z_1) \wedge \dots \wedge val_{t_k}(Z_k) \to p(Z_1, \dots, Z_k) \vee \neg p(Z_1, \dots, Z_k))$
  if *Head* is $\{p(t_1, \dots, t_k)\}$;
- $\bot$ if *Head* is empty;

where each Z_i is a fresh program variable.

For example, the result of applying τ^* to rule (4) is

$$\forall X(\exists Z(Z = X \wedge p(Z)) \to \forall Z_1(\exists IJ(Z_1 = I + J \wedge I = X \wedge J = \overline{1}) \to q(Z_1))). \quad (14)$$

The result of applying τ^* to rule (7) is

$$\forall X(\exists Z(\exists IJ(Z = I - J \wedge I = X \wedge J = \overline{1}) \wedge p(Z)) \to \forall Z_1(Z_1 = X \to q(X))). \quad (15)$$

From Proposition 2, we conclude:

Proposition 3. *For any rule R, $(\tau^* R)^{\mathrm{prop}}$ is strongly equivalent to τR.*

For any program Π, $\tau^*\Pi$ stands for the set of formulas τR for all rules R of Π. From Proposition 3, we conclude:

Proposition 4. *A program Π_1 is strongly equivalent to a program Π_2 iff $(\tau^*\Pi_1)^{\mathrm{prop}}$ is strongly equivalent to $(\tau^*\Pi_2)^{\mathrm{prop}}$.*

For example, the question about the strong equivalence of rule (4) to rule (7), resolved in Section 4, can be reformulated as the question about the strong equivalence of the propositional counterparts of formulas (14) and (15).

With Proposition 4 available, our goal of verifying strong equivalence of programs using automated reasoning tools for classical logic is not yet within reach; what we need in addition is a way to use these tools to verify the condition

$$(\tau^*\Pi_1)^{\mathrm{prop}} \text{ is strongly equivalent to } (\tau^*\Pi_2)^{\mathrm{prop}}. \quad (16)$$

This can be achieved using an additional transformation that replaces each predicate symbol by two, corresponding to the two worlds of the logic of here-and-there, and thus reduces that logic to classical. A transformation of this kind is part of the design of SELP [2]. Implementing this idea in the context of ANTHEM is a topic for future work. In the next section, we show, however, that in the case of "definite" rules, such as (4), (7), and (10)–(13), condition (16) can be replaced by the condition

$$\tau^*\Pi_1 \text{ is classically equivalent to } \tau^*\Pi_2,$$

which can be verified by VAMPIRE and similar systems directly.

7 Definite Programs

A *definite rule* is a basic rule that does not contain the negation symbol *not*. In other words, a rule is definite if its head is an atom and its body is a conjunction of atoms and comparisons. A program is *definite* if all its rules are definite.

Proposition 5. *A definite program Π_1 is strongly equivalent to a definite program Π_2 iff $\tau\Pi_1$ is classically equivalent to $\tau\Pi_2$.*

This is immediate from the following lemma:

For any definite program Π and any definite ground rule R, if τR is derivable from $\tau\Pi$ classically, then τR is derivable from $\tau\Pi$ intuitionistically.

This lemma can be proved using results on Glivenko classes due to Orevkov [11].

Proposition 3 shows that for any program Π, $\tau\Pi$ is classically equivalent to $\tau^*\Pi$. In view of this fact, from Proposition 5, we can conclude:

Proposition 6. *A definite program Π_1 is strongly equivalent to a definite program Π_2 iff $\tau^*\Pi_1$ is classically equivalent to $\tau^*\Pi_2$.*

This theorem justifies the use of ANTHEM for verifying strong equivalence of definite programs described below.

8 ANTHEM

ANTHEM 0.2 implements τ^* as specified in Section 6. ANTHEM supports input programs in the input language of GRINGO of the form described in Section 2, including nondefinite programs (Section 7), and generates output formulas in human-readable form by default. For example, ANTHEM translates the simple definite program consisting of rule (7),

```
q(X) :- p(X - 1).
```

into the formula

```
forall X
   (exists X1
      (exists N1, N2 (X1 = N1 - N2 and N1 = X and N2 = 1)
       and p(X1))
    -> forall X2 (X2 = X -> q(X2)))
```

In the output language of ANTHEM, integer variables are denoted by `N1`, `N2`, etc., while all other variables are program variables. For this program, ANTHEM additionally prints a note that the input program was detected to be definite:

```
info: definite program
```

When instead passing a nondefinite program to ANTHEM, such as

```
q :- not p.
```

ANTHEM still performs the translation to

```
not p -> q
```

but issues the following note:

```
info: nondefinite program
```

ANTHEM's implementation takes advantage of GRINGO's library functionality for accessing the abstract syntax tree (AST) of a nonground program. The AST obtained from GRINGO is taken by ANTHEM and turned into the AST of the collection of formulas representing the rules of the program according to τ^*.

ANTHEM's source code and usage instructions are available at GitHub.[3]

9 Proving Strong Equivalence of Programs with VAMPIRE

When given two input programs, ANTHEM generates an output formula expressing that the collections of formulas obtained by applying τ^* to both programs are equivalent. For definite programs, it is sufficient to prove equivalence classically to conclude that both programs are strongly equivalent (because of Proposition 6).

In order to verify the strong equivalence of two definite programs programatically, ANTHEM is able to communicate with automated first-order theorem provers supporting integer arithmetic such as VAMPIRE. To that end, ANTHEM can be instructed to generate output in the syntax of TPTP [12], a standard input language for theorem provers. More precisely, ANTHEM leverages the *typed first-order form* (TFF) of TPTP with interpreted integer arithmetic.

In the output of ANTHEM, there are variables of two sorts—integer and program variables (see Sections 5 and 8), where the domain of integer variables is a subset of the domain of program variables. In TPTP, there may be variables of multiple sorts, but it is not clear how to express that one sort is a subsort of another. ANTHEM works around this limitation by applying an additional transformation to the output formulas when TPTP output is requested. For this purpose, a custom sort *object* is introduced; all integer and symbolic constants are then mapped to distinct values of type *object* through auxiliary functions *integer* and *symbolic*. Then, all variables in the output formulas are changed to the *object* domain; if a quantifier binds an integer variable `N`, ANTHEM restricts it to the condition that the value of `N` is in the range of the function *integer*. For example, the TPTP counterpart of

```
forall N (p(N))
```

is

```
![N: object]: ((?[X: $int]: N = integer(X)) => p(N))
```

With this transformation, ANTHEM can be used in combination with a first-order theorem prover to verify the strong equivalence of definite programs. The remainder of this section presents experimental results obtained with VAMPIRE.

[3] https://github.com/potassco/anthem

The experiments were conducted on a Linux system with an Intel Core i7-7700K (4 physical cores, 4.5 GHz) and 16 GB of RAM. VAMPIRE was invoked with the options `--mode casc --cores 4`.

Example 1: Predecessor/Successor

Program 1	**Program 2**
`q(X + 1) :- p(X).`	`q(X) :- p(X - 1).`

These two programs represent rules (4) and (7) from Section 3. VAMPIRE proves the strong equivalence of these programs in about 2.4 s.

Example 2: Multiplication by 2

Program 1	**Program 2**	**Program 3**
`q(X + X) :- p(X).`	`q(X + Y) :- p(X), X = Y.`	`q(2 * X) :- p(X).`

VAMPIRE proves the strong equivalence of Programs 1 and 2 in about 5 ms. The strong equivalence of the other combinations is proved in about 3.0 s.

Example 3: Integer Between 3 and 5

Program 1	**Program 2**
`p(X) :- X > 3, X < 5.`	`p(4).`

VAMPIRE proves the strong equivalence of these two programs in about 3.5 s. This result is particularly interesting because Program 1 contains an unsafe rule. While the program would be rejected by GRINGO, ANTHEM is able to prove the strong equivalence to Program 2.

Example 4: Trivial Rule

Program 1	**Program 2**
`p(X) :- X < 3, X > 5.`	`q :- q.`

VAMPIRE verifies that the rule in Program 1 is trivial by checking that Program 1 is strongly equivalent to Program 2. The trivial rule `q :- q.` is used here because ANTHEM does not support the empty program yet. VAMPIRE proves the strong equivalence in about 14 ms.

Example 5: Incorrect Refactoring

Program 1	**Program 2**
`q(X) :- p(X).`	`q(X + 1) :- p(X + 1).`

These programs contain rules (10) and (11) from Section 4, respectively. As explained earlier, these two programs aren't strongly equivalent, which might come as a surprise to a programmer rewriting Program 2 as Program 1. VAMPIRE refuses to prove the strong equivalence within 300 seconds.

Example 6: Infinite Stable Models

Program 1	**Program 2**	**Program 3**
`p(X + 0).`	`p(X + 1).`	`p(X).`

VAMPIRE proves the strong equivalence of Programs 1 and 2 in about 87 ms. The stable model of Program 3 is the infinite set of atoms `p(r)` for all precomputed terms `r`. In contrast, the stable models of Programs 1 and 2 include `p(r)` only if `r` is an integer. Consequently, VAMPIRE does not prove the strong equivalence of Program 3 to Programs 1 or 2.

10 Future Work

We plan to extend this research effort in several directions. First, investigate the possibility of using theorem provers other than VAMPIRE to verify strong equivalence. Second, enable ANTHEM to use theorem provers for verifying strong equivalence of nondefinite programs. Third, enable ANTHEM to use theorem provers for verifying the correctness of tight programs in the language of GRINGO by proving the equivalence of the given specification to the program's completion. Fourth, extend ANTHEM to cover a larger subset of the language of GRINGO, including symbolic functions.

References

1. Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (2005)
2. Chen, Y., Lin, F., Li, L.: SELP—a system for studying strong equivalence between logic programs. In: [1], pp. 442–446
3. Ferraris, P.: Answer sets for propositional theories. In: [1], pp. 119–131
4. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015).
5. Harrison, A., Lifschitz, V., Pearce, D., Valverde, A.: Infinitary equilibrium logic and strongly equivalent logic programs. Artificial Intelligence **246**, 22–33 (2017)
6. 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)
7. Janhunen, T., Oikarinen, E.: LPEQ and DLPEQ—translators for automated equivalence testing of logic programs. In: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning, pp. 336–340 (2004)
8. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Proceedings of the International Conference on Computer Aided Verification, pp. 1–35 (2013)
9. Lifschitz, V., Lühne, P., Schaub, T.: anthem: Transforming gringo programs into first-order theories (preliminary report). In: Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms (2018)
10. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. Transactions on Computational Logic **2**(4), 526–541 (2001)
11. Orevkov, V.: On Glivenko sequent classes. In: Proceedings of the Steklov Institute of Mathematics, vol. 98, pp. 147–173 (1971)
12. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning **59**(4), 483–502 (2017)