7-18-2023

# External Behavior of a Logic Program and Verification of Refactoring

Jorge Fandinno

Zachary Hansen

Yuliya Lierler

Vladimir Lifschitz

Nathan temple

doi:10.1017/S1471068423000200

# *External Behavior of a Logic Program and Verification of Refactoring*

JORGE FANDINNO, ZACHARY HANSEN, YULIYA LIERLER
*University of Nebraska Omaha, NE, USA*
(*e-mails:* jfandinno@unomaha.edu, zachhansen@unomaha.edu, ylierler@unomaha.edu)

VLADIMIR LIFSCHITZ, NATHAN TEMPLE
*University of Texas at Austin, TX, USA*
(*e-mails:* lifschitzv@gmail.com, nathan-temple@live.com)

*submitted 29 May 2023; accepted 12 June 2023*

**Abstract**

Refactoring is modifying a program without changing its external behavior. In this paper, we make the concept of external behavior precise for a simple answer set programming language. Then we describe a proof assistant for the task of verifying that refactoring a program in that language is performed correctly.

*KEYWORDS*: answer set programming, software verification, proof assistant, automated reasoning

## 1 Introduction

This paper is about the process of refactoring in the context of answer set programming (ASP), that is, about modifying an ASP program without changing its external behavior. Examples of refactoring logic programs can be found in papers by Serebrenik and Demoen (2003), Gebser *et al.* (2011, Section 3.1) and Buddenhagen and Lierler (2015, Section 3). In this paper we propose, for a simple ASP language, a precise definition of external behavior and a method for verifying that two programs exhibit the same external behavior.

Refactoring a program usually involves a series of small changes that improve its structure or performance. The example below shows that, in ASP, refactoring may serve also another purpose: to transform a program that a grounder classifies as unsafe into an equivalent program that it is able to ground. The program

```
composite(I*J) :- I > 1, J > 1.
      prime(I) :- I = a..b, not composite(I).
```

defines the set of primes in the interval $\{a, \ldots, b\}$, assuming that $a > 1$. The grounder GRINGO (Gebser *et al.* 2019) tells us that the program is unsafe. A safe program defining the same set can be obtained by replacing the first rule with

```
composite(I*J) :- I = 2..b, J = 2..b.
```

This is an example of refactoring, because the extent of `prime/1` did not change.

We can also refactor the program to improve its performance using the fact that every composite number in $\{a, \dots, b\}$ has a divisor in the interval $\{2, \dots \lfloor\sqrt{b}\rfloor\}$:

```
     sqrt_b(M) :- M = 1..b, M*M <= b, (M+1)*(M+1) > b.
composite(I*J) :- sqrt_b(M), I = 2..M, J = 2..b.
      prime(I) :- I = a..b, not composite(I).
```

In the Abstract Gringo language (Gebser *et al.* 2015), a program is defined as a set of rules, so that a program includes neither directives nor comments. Under this narrow definition, the program itself does not tell us which predicate symbols are meant to represent the output, and which symbols are auxiliary. But this difference is essential, because changing auxiliary predicates does not indicate a mistake in the process of refactoring.

Furthermore, the rules of a program do not show what kind of input is supposed to be provided for it. Generally, an input for an ASP program can be specified in two ways. First, some symbolic constants, such as `a` and `b` in the programs above, may be meant to serve as placeholders for elements of the input. Second, some predicate symbols occurring in the program may occur in the bodies of rules only, not in the heads. The extents of such predicates may be specified as part of input when we run the program. Some inputs may not conform to the programmer's assumptions about the intended use of the program. For instance, when we run the prime number programs above, the placeholders `a` and `b` are expected to be replaced by integers; the cases when they are replaced by symbolic constants are not related to external behavior if the programs are used as intended.

To sum up, *what we consider external behavior of a set of rules depends on how these rules are meant to be used.* In Sections 3–5, we make this idea precise for the subset of Abstract Gringo called mini-GRINGO (Fandinno *et al.* 2020, Section 2; Lifschitz 2022, Sections 2, 3). After that, we describe the proof assistant ANTHEM-P2P,[1] which uses the theorem prover VAMPIRE (Kovaćs and Voronkov 2013) to verify that two mini-GRINGO programs have the same external behavior. This proof assistant is built on top of the system ANTHEM (Fandinno *et al.* 2020), whose focus is on the related and yet different task of confirming that an ASP program adheres to its specification. The prime number programs above are used as a running example. To make the paper more self-contained, we have reviewed some background material in Appendices A–C.

## 2 On the syntax of mini-GRINGO

There are minor syntactic differences between mini-GRINGO and the input language of the grounder GRINGO, explained by the fact that the former is designed for theoretical studies, and the latter for actual programming. For example, the definition of `sqrt_b/1` in the introduction, rewritten in the syntax of mini-GRINGO, becomes

$$sqrt\_b(M) \leftarrow M = \overline{1}\,..\,b \wedge M \times M \leq b \wedge (M + \overline{1}) \times (M + \overline{1}) > b.$$

Overlined symbols, such as $\overline{1}$, are "numerals" – syntactic objects representing integers. In examples of rules and programs, we will freely switch between the two styles.

[1] Available at `https://github.com/ZachJHansen/anthem-p2p`.

In mini-GRINGO, *precomputed terms* are numerals, symbolic constants, and the symbols *inf*, *sup*. We assume that a total order on precomputed terms is chosen, such that *inf* is its least element, *sup* is its greatest element, and, for all integers $m$ and $n$, $\overline{m} < \overline{n}$ iff $m < n$. A *precomputed atom* is an expression of the form $p(\mathbf{t})$, where $p$ is a symbolic constant and $\mathbf{t}$ is a tuple of precomputed terms. A *predicate symbol* is a pair $p/n$, where $p$ is a symbolic constant and $n$ is a nonnegative integer. About a rule or another syntactic expression we say that it *contains* $p/n$ if it contains an atom of the form $p(t_1, \dots, t_n)$.

## 3 External behavior

*Definition* 1. A *user guide* is a quadruple

$$(\mathit{PH}, \mathit{In}, \mathit{Out}, \mathit{Dom}), \tag{1}$$

where

- $\mathit{PH}$ is a finite set of symbolic constants, called *placeholders*,
- $\mathit{In}$ and $\mathit{Out}$ are disjoint finite sets of predicate symbols, called *input symbols* and *output symbols*, and
- $\mathit{Dom}$ is a set such that each of its elements is a pair $(v, \mathscr{I})$, where
  - (i) $v$ is a function that maps elements of $\mathit{PH}$ to precomputed terms that do not belong to $\mathit{PH}$, and
  - (ii) $\mathscr{I}$ is a subset of the set of precomputed atoms that contain an input symbol and do not contain placeholders.

The set $\mathit{Dom}$ is the *domain* of the user guide, and pairs $(v, \mathscr{I})$ satisfying conditions (i) and (ii) are called *inputs*. An input $(v, \mathscr{I})$ represents a way to choose the values of placeholders and the extents of input predicates: for every placeholder $c$, specify $v(c)$ as its value, and add the atoms $\mathscr{I}$ to the rules of the program as facts. If $\Pi$ is a mini-GRINGO program then $v(\Pi)$ stands for the program obtained from $\Pi$ by replacing every occurrence of every constant $c$ in the domain of $v$ by $v(c)$. Using this notation, we can say that choosing $(v, \mathscr{I})$ as input for $\Pi$ amounts to replacing $\Pi$ by the program $v(\Pi) \cup \mathscr{I}$.

To use a program in accordance with user guide (1) means to run it for inputs that belong to $\mathit{Dom}$. The inputs that do not belong to $\mathit{Dom}$ are not related to the external behavior of the program when it is used as intended.

*Example* 1. The intended use of the programs discussed in the introduction can be described by user guide (1) with $\mathit{PH} = \{a, b\}$, $\mathit{In} = \emptyset$, $\mathit{Out} = \{\mathit{prime}/1\}$, and with the domain consisting of the inputs $(v, \emptyset)$ such that $v(a)$, $v(b)$ are numerals. (We could choose also to include the condition $v(b) \geq v(a) > \overline{1}$.) This user guide will be denoted by $\mathit{UG}_p$.

*Example* 2. We would like to describe the meaning of the word *orphan* by a logic program (Gelfond and Kahl 2014, Section 4.1.2). The intended use of such a program can be described by user guide (1) with

$$\mathit{PH} = \emptyset, \;\; \mathit{In} = \{\mathit{father/2}, \mathit{mother/2}, \mathit{living/1}\}, \;\; \mathit{Out} = \{\mathit{orphan/1}\},$$

and with the domain consisting of all inputs. We will denote this user guide by $\mathit{UG}_o$. In the next two sections, we examine two possible definitions of *orphan/1* and consider the question of their equivalence with respect to $\mathit{UG}_o$.

User guides are closely related to lp-functions (Gelfond 2002, Section 2), and also to io-programs (Fandinno *et al.* 2020, Section 5), reviewed in Appendix C.

An *output atom* of a user guide $UG$ is a precomputed atom that contains an output symbol of $UG$.

*Definition* 2. Let $(v, \mathscr{I})$ be an input in the domain of a user guide $UG$, and let $\Pi$ be a mini-GRINGO program such that the heads of its rules do not contain input symbols of $UG$. The *external behavior* of $\Pi$ for the user guide $UG$ and the input $(v, \mathscr{I})$ is the collection of all sets that can be represented as the intersection of a stable model of $v(\Pi) \cup \mathscr{I}$ with the set of output atoms of $UG$.

*Example* 1, continued If $\Pi$ is one of the three prime number programs from the introduction, and $(v, \mathscr{I})$ is an input in the domain of $UG_p$, then the program $v(\Pi) \cup \mathscr{I}$ is $v(\Pi)$, and it has a unique stable model. If $v$ is defined by the conditions $v(a) = \overline{10}$, $v(b) = \overline{15}$, then that stable model includes the atoms $prime(\overline{11})$, $prime(\overline{13})$, and some atoms containing $composite/1$. The external behavior of each of the programs for this input is $\{\{prime(\overline{11}), prime(\overline{13})\}\}$. For the safe and optimized versions, this external behavior can be calculated by instructing CLINGO to find all answers for the file obtained from the program by appending the directives

```
#const a = 10.  #const b = 15.  #show prime/1.
```

*Example* 2, continued If $\Pi$ is the program

```
        parent_living(X) :- father(Y,X), living(Y).
        parent_living(X) :- mother(Y,X), living(Y).                    (2)
               orphan(X) :- living(X), not parent_living(X).
```

and $(v, \mathscr{I})$ is an input in the domain of $UG_o$, then the program $v(\Pi) \cup \mathscr{I}$ is $\Pi \cup \mathscr{I}$, and it has a unique stable model. If $\mathscr{I}$ is

$$\begin{gathered}\{father(jacob, joseph), mother(rachel, joseph), \\ living(jacob), living(rachel), living(joseph)\},\end{gathered} \tag{3}$$

then that stable model includes the atoms $orphan(jacob)$, $orphan(rachel)$, and some atoms containing predicate symbols other than $orphan/1$. The external behavior of this program for $UG_o$ and input (3) is

$$\{\{orphan(jacob), orphan(rachel)\}\}. \tag{4}$$

It can be calculated by instructing CLINGO to find all answers for the file obtained from program (2) by appending the facts

```
father(jacob,joseph).
mother(rachel,joseph).
living(jacob). living(rachel). living(joseph).
```

and the directive `#show orphan/1`.

In the special case when $UG$ has neither placeholders nor input symbols, and its set of output symbols includes all predicate symbols occurring in $\Pi$, the external behavior of $\Pi$ with respect to $UG$ and $(\emptyset, \emptyset)$ is the set of stable models of $\Pi$. In this sense, the concept of external behavior is a generalization of the stable model semantics.

## 4 Equivalence

*Definition* 3. Let $UG$ be a user guide, and let $\Pi_1$, $\Pi_2$ be mini-GRINGO programs such that the heads of their rules do not contain input symbols of $UG$. We say that $\Pi_1$ is *equivalent to* $\Pi_2$ *with respect to* $UG$ if, for every input $(v, \mathscr{I})$ in the domain of $UG$, the external behavior of $\Pi_1$ for $UG$ and $(v, \mathscr{I})$ is the same as the external behavior of $\Pi_2$.

*Example* 1, continued The three programs from the introduction are equivalent to each other with respect to $UG_p$. As discussed in Section 6, this claim can be verified using the automated reasoning tools ANTHEM-P2P and VAMPIRE.

*Example* 2, continued Perhaps surprisingly, the one-rule program

```
orphan(X) :- living(X), father(Y,X), mother(Z,X),
             not living(Y), not living(Z).
```
(5)

is not equivalent to (2) with respect to $UG_o$. Indeed, the external behavior of this program with respect to $UG_o$ and input (3) is $\{\emptyset\}$, which is different from (4). We will see that ANTHEM-P2P can help us clarify the relationship between programs (2) and (5).

We understand *refactoring* a mini-GRINGO program with respect to a user guide $UG$ as replacing it by a program that is equivalent to it with respect to $UG$.

This equivalence relation is essentially an example of relativized uniform equivalence with projection (Oetsch and Tompits 2008), except that the language discussed by Oetsch and Tompits includes neither arithmetic operations nor placeholders. It is *uniform* equivalence, because the programs are extended by adding facts, rather than more complex rules; *relativized*, because these facts $\mathscr{I}$ are assumed to be atoms containing input symbols, not arbitrary atoms; *with projection*, because we look at the output atoms in the stable model, not the entire model.

## 5 Formal notation for user guides

To design software for verifying the equivalence of programs with respect to a user guide, we need to represent user guides in formal notation. The format that we chose for user guide files is similar to the format of specification files, defined by Fandinno *et al.* (2020) within their work on the system ANTHEM. Placeholders and input symbols are represented by `input` statements, for instance:

```
input: n.
input: living/1, father/2, mother/2.
```

Output symbols are represented by `output` statements:

```
output: prime/1.
```

There can be several statements of both kinds in a user guide file, in any order.

The question of representing the domain *Dom* by a string of characters is more difficult, because the domain is a set of inputs, which is generally infinite. Our approach is to define "assumptions" as sentences of an appropriate first-order language, and characterize the domain by a list of assumptions; an input belongs to the domain iff it satisfies all assumptions on that list.

For any set $\mathscr{P}$ of predicate symbols, by $\sigma_0(\mathscr{P})$ we denote the subsignature of the two-sorted signature $\sigma_0$, described in Appendix A, in which the set of predicate symbols is limited to the comparison symbols and the symbols from $\mathscr{P}$. In this paper, an *assumption* is a sentence over the signature $\sigma_0(In)$. Besides `input` and `output` statements, a user guide file may include one or more statements consisting of the word `assume` followed by an assumption.

To use assumptions as conditions on an input, we need to relate inputs to interpretations in the sense of first-order logic. If $v$ is a function that maps elements of some set $PH$ of symbolic constants to symbolic constants, and $\mathscr{I}$ is a subset of the set of precomputed atoms that contain a predicate symbol from $\mathscr{P}$, then there exists a unique interpretation $I$ of $\sigma_0(\mathscr{P})$ such that

(a) the domain of the sort *general* in $I$ is the set of all precomputed terms;
(b) the domain of the sort *integer* in $I$ is the set of all numerals;
(c) $I$ interprets every symbolic constant $c$ in $PH$ as $v(c)$;
(d) $I$ interprets every precomputed term $t$ that does not belong to $PH$ as $t$;
(e) $I$ interprets the symbols for arithmetic operations as usual in arithmetic;
(f) if $p/n$ is a predicate constant from $\mathscr{P}$, and $\mathbf{c}$ is an $n$-tuple of precomputed atoms, then $I$ interprets $p(\mathbf{c})$ as true iff $p(\mathbf{c}) \in \mathscr{I}$;
(g) $I$ interprets the comparison symbols as in the definition of mini-GRINGO.

We will denote that interpretation by $I(v, \mathscr{I})$. The domain of the user guide defined by a set of assumptions is the set of inputs $(v, \mathscr{I})$ such that the interpretation $I(v, \mathscr{I})$ of $\sigma_0(In)$ satisfies all assumptions in that set.

*Example* 1, continued The user guide $UG_p$ can be described by the statements

```
input: a, b.
assume: exists N (a = N) and exists N (b = N).
output: prime/1.
```

The first two lines can be written more concisely as

```
input: a -> integer, b -> integer.
```

*Example* 2, continued The user guide $UG_o$ can be described by the statements

```
input: living/1, father/2, mother/2.
output: orphan/1.
```
(6)

The absence of `assume` statements here shows that the domain is the set of all inputs.

## 6 Functionality of ANTHEM-P2P

The proof assistant ANTHEM-P2P uses the theorem prover VAMPIRE to verify that two mini-GRINGO programs have the same external behavior with respect to a given user guide. We can verify, for instance, that the first two versions of the prime number program from the introduction are equivalent with respect to the user guide $UG_p$ by running ANTHEM-P2P on three files: the unsafe program

```
composite(I*J) :- I > 1, J > 1.
      prime(I) :- I = a..b, not composite(I).
```
(7)

the safe program

```
composite(I*J) :- I = 2..b, J = 2..b.
      prime(I) :- I = a..b, not composite(I).
```
(8)

and the user guide

```
input: a -> integer, b -> integer.
output: prime/1.
```
(9)

The system ANTHEM-P2P transforms the task of verifying equivalence with respect to a user guide (1) into the problem of verifying the provability of a formula in a first-order theory over the signature $\sigma_0(In \cup Out)$, and submits that problem to VAMPIRE; see Sections 7–9 for details.

The user can help VAMPIRE organize search more efficiently by supplying ANTHEM-P2P with "helper" files. Such a file may instruct VAMPIRE to prove a series of lemmas before trying to prove the goal formula. A helper file can suggest also instances of the induction schema that may be useful for the job in hand. This kind of help is needed, for instance, for verifying the equivalence of the optimized prime number program to the other two.

The use of ANTHEM-P2P for proving equivalence of programs is, generally, an interactive process. If VAMPIRE does not prove the goal formula in the allotted time then one of the options is to provide more lemmas and run ANTHEM-P2P again. Alternatively, the user can look for a counterexample that refutes the equivalence claim, as in Example 2 above.

Sometimes, ANTHEM-P2P can help us clarify the source of a puzzling discrepancy between two versions of a program if we run it in the presence of additional `assume` statements. If adding an assumption to the user guide makes the programs equivalent then it is possible that perceiving that assumption as self-evident is the reason why the discrepancy is puzzling. For instance, we can observe that the ANTHEM-P2P/VAMPIRE combination proves the equivalence of program (2) to program (5) if we extend user guide (6) by two existence and uniqueness assumptions:

```
assume: forall X exists Y forall Z (father(Z,X) <-> Y=Z).
assume: forall X exists Y forall Z (mother(Z,X) <-> Y=Z).
```

The limitations of the ANTHEM-P2P algorithm are inherited from the limitations of ANTHEM and can be described as follows. The *predicate dependency graph* of a mini-GRINGO program $\Pi$ (Fandinno *et al.* 2020, Section 6.3) is the directed graph that

- has the predicate symbols contained in $\Pi$ as its vertices, and
- has an edge from $p/n$ to $q/m$ if some rule of $\Pi$ contains $p/n$ in the head and $q/m$ in the body.

The edge from $p/n$ to $q/m$ is *positive* if there is a rule $R$ in $\Pi$ such that $p/n$ is contained in the head of $R$, and $q/m$ is contained in an atom in the body of $R$ that is not in the scope of negation. For example, the predicate dependency graph of program (2) has 6 edges; all of them except for the edge from `parent_living/1` to `orphan/1` are positive. We say that $\Pi$ is *tight* if this graph has no cycles consisting of positive edges.

A vertex $p/n$ of the graph is *private* for a user guide $UG$ if it is neither an input symbol nor an output symbol of $UG$. We say that $\Pi$ *uses private recursion* for $UG$ if

- the predicate dependency graph of $\Pi$ has a cycle such that every vertex in it is a private symbol, or
- $\Pi$ includes a choice rule with the head containing a private symbol.

As discussed in the next two sections, the applicability of the algorithm implemented in ANTHEM-P2P to a pair of mini-GRINGO programs and a user guide $UG$ is guaranteed whenever the programs are tight and do not use private recursion with respect to $UG$. We expect that it will be possible to replace the tightness requirement by a significantly weaker condition using the ideas of a recent paper on "locally tight" programs (Fandinno and Lifschitz 2021); this is a topic for future work.

## 7 Equivalence of tight programs

The theorem stated below relates equivalence of tight programs to the satisfaction relation of second-order logic. Its statement refers to the concept of second-order completion, reviewed in Appendix B, and also to the concept of standard interpretations, defined as follows. An interpretation $I$ of $\sigma_0(\mathscr{P})$ is *standard* for a set $PH$ of symbolic constants if it satisfies conditions (a), (b), (d), (e), (g) from Section 5 and the condition

(c′) $I$ interprets every symbolic constant in $PH$ as a term that does not belong to $PH$.

*Theorem* Let UG be a user guide (PH,In,Out,Dom) such that its domain is described by a finite set of assumptions, and let Asm be the conjunction of these assumptions. For any tight mini-GRINGO programs $\Pi_1$, $\Pi_2$ such that the heads of their rules do not contain the input symbols of UG, $\Pi_1$ is equivalent to $\Pi_2$ with respect to UG iff the sentence

$$Asm \to (\mathrm{COMP}(\Pi_1, In, Out) \leftrightarrow \mathrm{COMP}(\Pi_2, In, Out)), \qquad (10)$$

*is satisfied by all interpretations of the signature $\sigma_0(In \cup Out)$ that are standard for PH.*

This theorem shows that the equivalence of tight programs may be established by choosing a first-order theory $T$ over the signature $\sigma_0(In \cup Out)$ such that its axioms are satisfied by all interpretations that are standard for $PH$, and then exhibiting a derivation of formula (10) from the axioms of $T$ in classical second-order logic. For programs that do not use private recursion, the problem of constructing such a derivation can be reduced to proof search in first-order logic (see Section 8 below), for which many automated reasoning tools are available. This is the core of the procedure used by ANTHEM-P2P.

The proof of the theorem, including the lemma below, uses terminology related to io-programs, which is reviewed in C.

*Lemma.* Let $\Pi$ be a mini-GRINGO program such that the heads of its rules do not contain input symbols of a user guide (PH,In,Out,Dom). For any input $(v, \mathscr{I})$, a set $\mathscr{J}$ of output atoms is an element of the external behavior of $\Pi$ for (PH,In,Out,Dom) and $(v, \mathscr{I})$ iff $\mathscr{I} \cup \mathscr{J}$ is an io-model of the io-program ($\Pi$,PH,In,Out) for $(v, \mathscr{I})$.

*Proof* For every set $\mathscr{J}$ of output atoms, the conditions

- $\mathscr{J}$ is the set of all output atoms in some stable model $\mathscr{M}$ of $v(\Pi) \cup \mathscr{I}$;
- $\mathscr{I} \cup \mathscr{J}$ is the set of all public atoms in some stable model $\mathscr{M}$ of $v(\Pi) \cup \mathscr{I}$

are equivalent to each other. Indeed, since the heads of rules of $v(\Pi)$ do not contain input atoms, the set of input atoms in $\mathscr{M}$ is $\mathscr{I}$.

*Proof of the Theorem.* The condition

$$\Pi_1 \text{ is equivalent to } \Pi_2 \text{ with respect to } UG, \tag{11}$$

means that for any input $(v, \mathscr{I})$ such that $I(v, \mathscr{I}) \models Asm$ and any set $\mathscr{J}$ of output atoms,

$$\begin{gathered} \mathscr{J} \text{ is an element of the external behavior of } \Pi_1 \text{ for } UG \text{ and } (v, \mathscr{I}) \\ \text{iff} \\ \mathscr{J} \text{ is an element of the external behavior of } \Pi_2 \text{ for } UG \text{ and } (v, \mathscr{I}). \end{gathered} \tag{12}$$

By the lemma, condition (12) can be reformulated as follows:

$$\begin{gathered} \mathscr{I} \cup \mathscr{J} \text{ is an io-model of the io-program } (\Pi_1, PH, In, Out) \text{ for } (v, \mathscr{I}) \\ \text{iff} \\ \mathscr{I} \cup \mathscr{J} \text{ is an io-model of the io-program } (\Pi_2, PH, In, Out) \text{ for } (v, \mathscr{I}). \end{gathered}$$

By the theorem quoted at the end of Appendix C, this can be further reformulated as

$$I(v, \mathscr{I} \cup \mathscr{J}) \models \text{COMP}(\Pi_1, In, Out) \leftrightarrow \text{COMP}(\Pi_2, In, Out). \tag{13}$$

Hence condition (11) is equivalent to requiring that (13) hold for all inputs $(v, \mathscr{I})$ such that $I(v, \mathscr{I}) \models Asm$ and all set $\mathscr{J}$ of output atoms.

Since assumptions do not contain output symbols, $I(v, \mathscr{I}) \models Asm$ is equivalent to $I(v, \mathscr{I} \cup \mathscr{J}) \models Asm$. It follows that (11) is equivalent to asserting that implication (10) is satisfied by $I(v, \mathscr{I} \cup \mathscr{J})$ for all inputs $(v, \mathscr{I})$ and all sets $\mathscr{J}$ of output atoms. It remains to observe that an interpretation of the signature $\sigma_0(In \cup Out)$ can be represented in the form $I(v, \mathscr{I} \cup \mathscr{J})$ if and only if it is standard for $PH$.

## 8 Reduction to first-order logic

If $\Pi_1$ and $\Pi_2$ do not use private recursion then the reference to second-order consequences of the axioms of $T$ in Section 7 can be eliminated in the following way. Represent the formula $\text{COMP}(\Pi_1, In, Out)$ in the form

$$\exists \mathbf{P} \left( \bigwedge_i F_i(\mathbf{P}) \wedge F'(\mathbf{P}) \right),$$

where $\mathbf{P}$ is a list of distinct predicate variables corresponding to the private symbols $p_1, p_2, \ldots$ of $\Pi_1$, and $F_i(\mathbf{P})$ is the formula obtained from the completed definition of $p_i$ in $\Pi_1$ by replacing each of $p_1, p_2, \ldots$ by the corresponding member of $\mathbf{P}$. (Thus the conjunctive members of $F'(\mathbf{P})$ correspond to the completed definitions of the output symbols and to the constraints of $\Pi_1$.) Similarly, write $\text{COMP}(\Pi_2, In, Out)$ as

$$\exists \mathbf{Q} \left( \bigwedge_j G_j(\mathbf{Q}) \wedge G'(\mathbf{Q}) \right), \tag{14}$$

where $\mathbf{Q}$ is a list of distinct predicate variables corresponding to the private symbols $q_1, q_2, \ldots$ of $\Pi_2$, and the formulas $G_j(\mathbf{Q})$ are obtained from the completed definitions of these symbols in $\Pi_2$ by replacing them with corresponding variables. Take one half

$$Asm \to (\mathrm{COMP}(\Pi_1, In, Out) \to \mathrm{COMP}(\Pi_2, In, Out)), \tag{15}$$

of condition (10). Since $\Pi_2$ does not use private recursion, formula (14) is equivalent to

$$\forall \mathbf{Q} \left( \bigwedge_j G_j(\mathbf{Q}) \to G'(\mathbf{Q}) \right).$$

(Fandinno *et al.* 2020, Theorem 3). It follows that formula (15) is equivalent to

$$Asm \to \left( \exists \mathbf{P} \left( \bigwedge_i F_i(\mathbf{P}) \wedge F'(\mathbf{P}) \right) \to \forall \mathbf{Q} \left( \bigwedge_j G_j(\mathbf{Q}) \to G'(\mathbf{Q}) \right) \right),$$

and consequently to

$$\forall \mathbf{PQ} \left( \left( Asm \wedge \bigwedge_i F_i(\mathbf{P}) \wedge \bigwedge_j G_j(\mathbf{Q}) \right) \to (F'(\mathbf{P}) \to G'(\mathbf{Q})) \right), \tag{16}$$

(with the bound variables in $\mathbf{P}$, $\mathbf{Q}$ renamed, if necessary, to ensure that they are pairwise disjoint). Similarly, the second half

$$Asm \to (\mathrm{COMP}(\Pi_2, In, Out) \to \mathrm{COMP}(\Pi_1, In, Out)),$$

of condition (10) is equivalent to the formula obtained from (16) by swapping $F'(\mathbf{P})$ with $G'(\mathbf{Q})$. Thus (10) can be rewritten as

$$\forall \mathbf{PQ} \left( \left( Asm \wedge \bigwedge_i F_i(\mathbf{P}) \wedge \bigwedge_j G_j(\mathbf{Q}) \right) \to (F'(\mathbf{P}) \leftrightarrow G'(\mathbf{Q})) \right).$$

Finally, observe that this formula is entailed by the axioms of $T$ if and only if the axioms entail the first-order formula

$$\left( Asm \wedge \bigwedge_i F_i(\mathbf{p}) \wedge \bigwedge_j G_j(\mathbf{q}) \right) \to (F'(\mathbf{p}) \leftrightarrow G'(\mathbf{q})), \tag{17}$$

where $\mathbf{p}$, $\mathbf{q}$ are lists of fresh predicate constants.

We return to this formula in the description of the design of ANTHEM-P2P below. Note that its subformulas $F_i(\mathbf{p})$, $G_j(\mathbf{q})$, $F'(\mathbf{p})$, $G'(\mathbf{q})$ are parts of the first-order completion formulas of $\Pi_1$ and $\Pi_2$, modified by replacing their private symbols $p_1, p_2, \dots, q_1, q_2, \dots$ by members of the lists $\mathbf{p}$ and $\mathbf{q}$.

## 9 Design of ANTHEM-P2P

The system ANTHEM-P2P is a Python program that operates by converting a claim about the equivalence of two mini-GRINGO programs into an input for ANTHEM. The system ANTHEM verifies the correctness of an io-program with respect to a formal specification. The file describing a specification includes lists of placeholders, input symbols, output symbols, and assumptions, and also a list of "specs" that describe the intended behavior of the future program by sentences over the signature $\sigma_0(In \cup Out)$.

Given programs $\Pi_1$ and $\Pi_2$ and a user guide $(PH, In, Out, Dom)$ with the domain described by assumptions $Asm$, ANTHEM-P2P constructs the following specification $Sp$:

(i) the placeholders of *Sp* are the placeholders *PH* of the given user guide;
(ii) the input symbols of *Sp* are the input symbols *In* of the user guide and the predicate symbols $\mathbf{p}$ corresponding to the private symbols $p_1, p_2, \ldots$ of the program $\Pi_1$;
(iii) the output symbols of *Sp* are the output symbols *Out* of the user guide;
(iv) the assumptions of *Sp* are the assumptions *Asm* of the user guide and the modified completed definitions $F_i(\mathbf{p})$ of the private symbols of $\Pi_1$;
(v) the specs of *Sp* are the remaining conjunctive terms $F'(\mathbf{p})$ of the modified first-order completion formula of $\Pi_1$.

Then ANTHEM-P2P instructs ANTHEM to prove the claim that the io-program $(\Pi_2, \mathit{PH}, \mathit{In}, \mathit{Out})$ implements *Sp*. Providing ANTHEM with such an instruction makes it look for a derivation of the formula

$$\left(\mathit{Asm} \wedge \bigwedge_i F_i(\mathbf{p}) \wedge \bigwedge_j G_j(\mathbf{q})\right) \rightarrow (G'(\mathbf{q}) \leftrightarrow F'(\mathbf{p})), \tag{18}$$

from the axioms of $T$ by invoking the theorem prover VAMPIRE (Fandinno *et al.* 2020, Section 6.4). This formula is equivalent to (17). Thus instructing ANTHEM to verify that the io-program $(\Pi_2, \mathit{PH}, \mathit{In}, \mathit{Out})$ implements the specification *Sp* amounts to verifying the provability of formula (17) in $T$.

As an example, consider the operation of the ANTHEM-P2P algorithm on programs (7) and (8) and user guide (9). In each of the programs, the only private predicate is `composite/1`; it corresponds to both $p_1$ and $q_1$ in the notation of Section 8. The symbols `composite_1/1` and `composite_2/1`, generated by ANTHEM-P2P, play the parts of $\mathbf{p}$ and $\mathbf{q}$ in formula (17). The file describing the specification *Sp* is obtained in this case from user guide (9) by adding three statements. First, in accordance with clause (ii) in the description of *Sp* above, ANTHEM-P2P adds the statement

```
input: composite_1/1.
```

Second, in accordance with clause (iv), a definition of `composite_1/1` is assumed:

```
assume: forall X (composite_1(X) <->
                  exists N1, N2 (N1 > 1 and N2 > 1 and X = N1 * N2)).
```

Finally, in accordance with clause (v), a definition of `prime/1` in terms of `composite_1/1` is added as a spec:

```
spec: forall X (prime(X) <->
                exists N1 (not composite_1(N1) and
                    exists N2, N3 (N2 = a and N3 = b and
                                   N2 <= N1 and N1 <= N3) and X = N1)).
```

Once *Sp* is generated, ANTHEM calls VAMPIRE to prove formula (18) in the theory $T$, first by deriving the specs $F'(\mathbf{p})$ from the antecedent of (18) and $G'(\mathbf{q})$ ("verification of specification from translated program"), and then by deriving $G'(\mathbf{q})$ from the antecedent of (18) and the specs $F'(\mathbf{p})$ ("verification of translated program from specification"). In this example, the runtime of VAMPIRE will be significantly reduced (a few seconds instead of a few minutes) if we instruct it to start by proving two lemmas:

```
lemma: forall I, J (I > 1 and J > 1 -> I < I*J).
lemma: forall X (prime(X) ->
                 exists N1 (not composite_1(N1) and
                     exists N2, N3 (N2 = a and N3 = b and
                                    N2 <= N1 and N1 <= N3) and X = N1)).
```

The ANTHEM-P2P website[2] allows users to experiment with the system in their web browser. The proof search is conducted on a University of Nebraska Omaha server (Oracle Linux 8, 4 Intel(R) Xeon(R) Gold 6248 CPUs, 4 GB RAM) subject to a 10 minute timeout. For smaller problems, this is the recommended introduction to the system.

## 10 Conclusion

This paper contributes to the theory of logic programming by defining user guides, external behaviors, and equivalence with respect to a user guide. The theorem proved in Section 7 relates equivalence of tight programs to program completion.

The problem of checking equivalence between programs arises in many areas of computer science. For example, verifying the correctness of the translation performed by an optimizing compiler is a problem of this kind. What is special about the verification of refactoring is that it involves a pair of similar programs written in the same programming language. MEDIATOR (Wang *et al.* 2018) is a tool that uses an SMT solver for the verification of database refactoring.

The proof assistant ANTHEM-P2P can be used for verifying the correctness of refactoring an ASP program, and also for comparing alternative solutions to the same programming problem (for instance, in classroom teaching and in ASP programming contests). To make this tool more versatile, we plan to make it applicable to programs with aggregates, along the lines of recent publications (Fandinno *et al.* 2022; Lifschitz 2022).

## Acknowledgements

We are grateful to Isil Dillig, Warren Hunt and Jayadev Misra for valuable comments on the problems discussed in this paper. Thanks also to Michael Gelfond and to the anonymous referees for advice on improving the previous versions. This work was partially supported by the University of Nebraska Omaha, ORCA Grant 26215.

## References

Buddenhagen, M. and Lierler, Y. 2015. Performance tuning in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning.*

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

Fandinno, J., Hansen, Z. and Lierler, Y. 2022. Axiomatization of aggregates in answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence.*

[2] Available at https://ap2p.unomaha.edu/.

Fandinno, J. and Lifschitz, V. 2021. Verification of locally tight programs (extended abstract). In *Technical Communications of the 37th International Conference on Logic Programming (ICLP).*

Fandinno, J., Lifschitz, V., Lühne, P. and Schaub, T. 2020. Verifying tight logic programs with Anthem and Vampire. *Theory and Practice of Logic Programming 20.*

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

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. and Thiele, S. 2019. Potassco User Guide. Available at https://github.com/potassco/guide/releases/.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011. Challenges in answer set solving. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday.* Springer, 74–90.

Gelfond, M. 2002. *Representing Knowledge in A-Prolog.* Lecture Notes in Computer Science vol. 2408, 413–451.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press.

Kovaćs, L. and Voronkov, A. 2013. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, 1–35.

Lifschitz, V. 2022. Strong equivalence of logic programs with counting. *Theory and Practice of Logic Programming 22.*

Oetsch, J. and Tompits, H. 2008. Program correspondence under the answer-set semantics: The non-ground case. In *Proceedings of International Conference on Logic Programming.*

Serebrenik, A. and Demoen, B. 2003. Refactoring logic programs: Extended abstract. In *International Conference on Logic Programming.*

Wang, Y., Dillig, I., Lahiri, S. and Cook, W. 2018. Verifying equivalence of database-driven applications. In *Proceedings of the ACM Symposium on Programming Languages.*

## Appendix A Two-sorted formulas

The signature $\sigma_0$ has two sorts: the sort *general* and its subsort *integer*. Variables of the first sort are meant to range over arbitrary precomputed terms, and we will identify them with variables used in mini-GRINGO rules. Variables of the second sort are meant to range over numerals – or, equivalently, integers. The signature includes

- all precomputed terms as object constants; an object constant is assigned the sort *integer* iff it is a numeral;
- the symbols $+$, $-$ and $\times$ as binary function constants; their arguments and values have the sort *integer*;
- all predicate symbols $p/n$ as $n$-ary predicate constants; their arguments have the sort *general*;
- the comparison symbols $\neq < > \leq \geq$ as binary predicate constants; their arguments have the sort *general.*

An atomic formula $(p/n)(\mathbf{t})$ can be abbreviated as $p(\mathbf{t})$. An atomic formula $\prec(t_1, t_2)$, where $\prec$ is a comparison symbol, can be written as $t_1 \prec t_2$.

In this paper, we adopt the convention that general variables start with $U$, $V$, $W$, $X$, $Y$, and $Z$; integer variables start with $I$, $J$, $K$, $L$, $M$, and $N$. For example, the formula $\exists X(N = X)$ expresses that the value of $N$ is an object of the sort *general*; it is universally true, because *integer* is a subsort of *general*. The formula $\exists N(N = X)$ expresses that the value of $X$ is an object of the sort *integer*; it is generally not true.

## Appendix B Second-order completion

Second-order completion (Fandinno *et al.* 2020, Sections 6.1, 6.2) is a generalization of Clark's completion (Clark 1978) that uses bound predicate variables to model auxiliary ("private") predicates, such as `composite/1` in our prime number programs. The definition covering the full syntax of mini-GRINGO is rather lengthy, and in this appendix we only give an outline and an example.

Let *In* and *Out* be disjoint sets of predicate symbols, and let $\Pi$ be a mini-GRINGO program such that atoms in the heads of its rules do not contain predicate symbols from *In*. If a predicate symbol $p/n$

- is contained in an atom that occurs in a rule of $\Pi$, and
- belongs neither to *In* nor to *Out*,

then $p/n$ is a *private symbol* of $\Pi$. We denote the set of private symbols of $\Pi$ by *Prv*.

The *first-order completion* of $\Pi$ is the conjunction of the following first-order sentences over the signature $\sigma_0(In \cup Out \cup Prv)$:

- the completed definitions of the predicate symbols from $Out \cup Prv$ in $\Pi$;
- the constraints of $\Pi$ rewritten in the syntax of first-order logic.

The *second-order completion* of $\Pi$ is the sentence over the signature $\sigma_0(In \cup Out)$ obtained from the first-order completion of $\Pi$ by replacing all private symbols by predicate variables and binding these variables by an existential quantifier. We will denote the second-order completion of $\Pi$ by COMP$(\Pi, In, Out)$.

If, for instance, $In = \emptyset$, $Out = \{q/2\}$, and $\Pi$ is the program

$$\begin{array}{l} p(a), \\ p(b), \\ q(X,Y) \leftarrow p(X) \wedge p(Y), \end{array}$$

then $Prv = \{p/1\}$, the first-order completion of $\Pi$ is

$$\begin{array}{l} \forall V(p(V) \leftrightarrow V = a \vee V = b) \wedge \\ \forall V_1 V_2 (q(V_1,V_2) \leftrightarrow \exists XY(q(V_1,V_2) \wedge p(X) \wedge p(Y) \wedge V_1 = X \wedge V_2 = Y)), \end{array}$$

and COMP$(\Pi, In, Out)$ is

$$\begin{array}{l} \exists P(\forall V(P(V) \leftrightarrow V = a \vee V = b) \wedge \\ \forall V_1 V_2 (q(V_1,V_2) \leftrightarrow \exists XY(q(V_1,V_2) \wedge P(X) \wedge P(Y) \wedge V_1 = X \wedge V_2 = Y))). \end{array}$$

This formula is equivalent to the first-order sentence

$$\forall V_1 V_2 (q(V_1,V_2) \leftrightarrow (V_1 = a \vee V_1 = b) \wedge (V_2 = a \vee V_2 = b)).$$

## Appendix C Programs with input and output

A *program with input and output*, or an *io-program*, is a quadruple

$$(\Pi, PH, In, Out), \tag{C1}$$

where $PH$, $In$ and $Out$ are as in the definition of a user guide (Section 3), and $\Pi$ is a mini-GRINGO program such that the heads of its rules do not contain symbols from $In$ (Fandinno *et al.* 2020, Section 5.1). Inputs for an io-program are defined in the same way as inputs for a user guide in Section 3.

A *public atom* of an io-program (C1) is a precomputed atom that contains a predicate symbol from $In \cup Out$.

An *io-model* of an io-program (C1) for an input $(v, \mathscr{I})$ is a set that can be represented as the intersection of a stable model of $v(\Pi) \cup \mathscr{I}$ with the set of public atoms of (C1).

If $(v, \mathscr{I})$ is an input for an io-program (C1), and the program $\Pi$ is tight, then, for any set $\mathscr{J}$ of output atoms, $\mathscr{I} \cup \mathscr{J}$ is an io-model of (C1) iff the interpretation $I(v, \mathscr{I} \cup \mathscr{J})$ of the signature $\sigma_0(In \cup Out)$ satisfies the second-order completion sentence $\mathrm{COMP}(\Pi, In, Out)$ (Fandinno *et al.* 2020, Theorem 2).