TPLP **20** (5): 735–750, 2020. © The Author(s), 2020. Published by Cambridge University Press
doi:10.1017/S1471068420000344

Verifying Tight Logic Programs with ANTHEM *and* VAMPIRE

JORGE FANDINNO
University of Potsdam, Germany

VLADIMIR LIFSCHITZ
University of Texas at Austin, USA

PATRICK LÜHNE and TORSTEN SCHAUB
University of Potsdam, Germany

submitted 9 August 2020; revised 10 August 2020; accepted 12 August 2020

Abstract

This paper continues the line of research aimed at investigating the relationship between logic programs and first-order theories. We extend the definition of program completion to programs with input and output in a subset of the input language of the ASP grounder GRINGO, study the relationship between stable models and completion in this context, and describe preliminary experiments with the use of two software tools, ANTHEM and VAMPIRE, for verifying the correctness of programs with input and output. Proofs of theorems are based on a lemma that relates the semantics of programs studied in this paper to stable models of first-order formulas.

1 Introduction

This paper continues the line of research aimed at investigating the relationship between logic programs and first-order theories, which goes back to Keith Clark's classical paper on program completion (Clark 1978). We are interested in extending that idea to input languages of answer set programming (ASP) (Brewka et al. 2011) and in using it for the verification of ASP programs.

These programs are different from logic programs studied by Clark in two ways. First, their semantics is based on the concept of a stable model (Gelfond and Lifschitz 1988), which is not equivalent to the completion semantics, unless some assumptions about the program are made, such as tightness (Fages 1994; Erdem and Lifschitz 2003). Second, ASP programs can include symbols for integers and arithmetic operations. Ways to extend completion to these constructs are proposed in two recent publications (Harrison et al. 2017; Lifschitz et al. 2020).

It is essential for our purposes that, in ASP, we often deal with programs that accept an input and employ auxiliary predicates. Consider, for instance, the program

$$\begin{array}{l} \{in_cover(1\,..\,n)\}, \\ \leftarrow I \neq J \wedge in_cover(I) \wedge in_cover(J) \wedge s(X,I) \wedge s(X,J), \\ covered(X) \leftarrow in_cover(I) \wedge s(X,I), \\ \leftarrow s(X,I) \wedge not\ covered(X), \end{array} \tag{1}$$

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press
CrossMark

which encodes exact covers[1] of a finite family of finite sets. It accepts an input, represented by a placeholder n and a predicate symbol $s/2$; n is the number of sets, and the formula $s(X, I)$ expresses that X belongs to the I-th set. The predicate symbol *in_cover*/1 represents the output, and *covered*/1 is auxiliary.

In this paper, we extend the definition of program completion to programs with input and output in a subset of the input language of the ASP grounder GRINGO, state a theorem describing the relationship between stable models and completion in this context, and describe preliminary experiments with the use of two software tools, ANTHEM (Lifschitz et al. 2018; Lifschitz et al. 2019) and VAMPIRE (Kovács and Voronkov 2013), for verifying the correctness of programs with input and output. A lemma that may be of independent interest relates the semantics of simple GRINGO programs to stable models of first-order formulas (Ferraris et al. 2011). It shows, in other words, that the approach to the semantics of logic programs advocated by Lee et al. (2008) is applicable to the subset of GRINGO studied here.

2 Review: Programs

The class of programs studied in this paper is the subset of the input language of GRINGO (Gebser et al. 2015) discussed in earlier publications by Lifschitz et al. (2019; 2020).

We assume that three countably infinite sets of symbols are selected: *numerals*, *symbolic constants*, and *program variables*. We assume a 1-to-1 correspondence between numerals and integers; 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 $\circ$ is one of the *operation names*

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

  then $(t_1 \circ t_2)$ 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 (in other words, if it is a numeral, a symbolic constant, or one of the symbols *inf* or *sup*).

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

- *inf* is its least element and *sup* 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

[1] An *exact cover* of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

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 *comparison symbols*

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{2}$$

A *rule* is an expression of the form

$$Head \leftarrow Body, \tag{3}$$

where

- *Body* is a conjunction (possibly empty) of literals and comparisons and
- *Head* is either an atom (then, we say that (3) is a *basic rule*) or an atom in braces (then, (3) is a *choice rule*), or empty (then, (3) is a *constraint*).

A *program* is a finite set of rules.

For example, (1) is a program in the sense of this definition, except that the expression $\{in_cover(1\,..\,n)\}$, strictly speaking, should be written as $\{in_cover(\overline{1}\,..\,n)\} \leftarrow$.

The semantics of programs uses a translation τ, which turns a program into a set of propositional combinations of precomputed atoms (Lifschitz et al. 2019, Section 3). For example, the result of applying τ to the program

$$\begin{aligned} &\{p(\overline{1}\,..\,\overline{3})\}, \\ &q(X + \overline{1}) \leftarrow p(X) \end{aligned} \tag{4}$$

consists of the ground formulas

$$\begin{aligned} &\top \rightarrow (p(\overline{1}) \vee \neg p(\overline{1})) \wedge (p(\overline{2}) \vee \neg p(\overline{2})) \wedge (p(\overline{3}) \vee \neg p(\overline{3})), \\ &p(\overline{n}) \rightarrow q(\overline{n+1}) \quad \text{for all integers } n, \text{ and} \\ &p(r) \rightarrow \top \quad \text{for all precomputed terms } r \text{ other than numerals.} \end{aligned}$$

Using the translation τ, we define the semantics of programs as follows: A set of precomputed atoms is a *stable model* of a program Π if it is a stable model (answer set) of $\tau\Pi$ in the sense of Ferraris (2005).

3 Review: Representing Programs by First-Order Theories

The translation τ^* (Lifschitz et al. 2019) converts a program into a finite set of first-order sentences with variables of two sorts. Besides program variables, which are meant to range over precomputed program terms, these sentences contain *integer variables*, which are meant to range over numerals. The second sort is a subsort of the first.

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 $p(t_1, \ldots, t_n)$.

For any program Π, $\tau^*\Pi$ consists of formulas over the two-sorted signature σ_Π including

- all precomputed terms as object constants; a precomputed 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*;

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

- all predicate symbols occurring in Π and the comparison symbols other than equality as predicate constants.

An atomic formula $(p/n)(t_1, \dots, t_n)$ can be abbreviated as $p(t_1, \dots, t_n)$. An atomic formula $\prec (t_1, t_2)$, where $\prec$ is a comparison symbol, can be written as $t_1 \prec t_2$. *Formulas* are formed from atomic formulas using the propositional connectives

$$\bot \text{ (“false”)} \quad \wedge \quad \vee \quad \rightarrow$$

and the quantifiers $\forall$ and $\exists$ as usual in first-order languages with equality. We use $\top$ as shorthand for $\bot \rightarrow \bot$, $\neg F$ as shorthand for $F \rightarrow \bot$, and $F \leftrightarrow G$ as shorthand for $(F \rightarrow G) \wedge (G \rightarrow F)$.

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. 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 $(t_1 \circ t_2)$, where $\circ$ is $+$, $-$, or $\times$, then $val_t(Z)$ is

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

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

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

  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 $(t_1 \,..\, t_2)$, 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, $val_{\overline{1}..\overline{3}}(Z)$ is

$$\exists IJK\,(I = \overline{1} \wedge J = \overline{3} \wedge I \leq K \wedge K \leq J \wedge Z = K),$$

and $val_{X+\overline{1}}(Z)$ is

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

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

- $\tau^B(p(t_1, \dots, t_k))$ is

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

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

- $\tau^B(not\, p(t_1,\dots,t_k))$ is
$$\exists Z_1 \dots Z_k\, (val_{t_1}(Z_1) \wedge \dots \wedge val_{t_k}(Z_k) \wedge \neg p(Z_1,\dots,Z_k));$$
- $\tau^B(not\, not\, p(t_1,\dots,t_k))$ is
$$\exists Z_1 \dots Z_k\, (val_{t_1}(Z_1) \wedge \dots \wedge val_{t_k}(Z_k) \wedge \neg\neg p(Z_1,\dots,Z_k));$$
- $\tau^B(t_1 \prec t_2)$ is
$$\exists Z_1 Z_2\, (val_{t_1}(Z_1) \wedge val_{t_2}(Z_2) \wedge Z_1 \prec Z_2);$$

where each Z_i is a fresh program variable.

Now, we define
$$\tau^*(Head \leftarrow B_1 \wedge \dots \wedge B_n)$$
as the universal closure of the formula
$$\tau^B(B_1) \wedge \dots \wedge \tau^B(B_n) \rightarrow H,$$
where H is

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

where $Z_1,\dots,Z_k$ are fresh program variables.

For any program Π, $\tau^*\Pi$ stands for the set of formulas $\tau^* R$ for all rules R of Π. For example, the result of applying τ^* to program (4) is the pair of sentences
$$\top \rightarrow \forall Z\, (val_{\overline{1}..\overline{3}}(Z) \rightarrow p(Z) \vee \neg p(Z)),$$
$$\forall X\, (\exists Z\, (val_{X+\overline{1}}(Z) \wedge p(Z)) \rightarrow \forall Z\, (Z = X \rightarrow p(Z))).$$

4 Main Lemma

The lemma stated in this section is used in the proofs of Theorems 1 and 2, which are stated later in Section 6. It relates the stable models of a program Π to the stable models of the first-order theory $\tau^*\Pi$.

Recall that, for any first-order sentence F and any list $\mathbf{p}$ of predicate symbols (other than equality), $\mathrm{SM}_{\mathbf{p}}[F]$ stands for the second-order sentence
$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u}))$$
(Ferraris et al. 2011, Section 2.3). The definition of $\mathrm{SM}_{\mathbf{p}}$ in that paper is given for formulas with one sort of variables, but its extension to many-sorted formulas is straightforward, and we will apply it here to the class of formulas defined in Section 3.

The models of $\mathrm{SM}_{\mathbf{p}}[F]$ are called the $\mathbf{p}$-*stable models* of F. For a set Γ of first-order sentences, the $\mathbf{p}$-*stable models* of Γ are the $\mathbf{p}$-stable models of the conjunction of all formulas in Γ.

For any program Π and any set $\mathcal{I}$ of precomputed atoms, $\mathcal{I}^\uparrow$ stands for the interpretation of the signature σ_Π such that

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

- the universe of the first sort of $\mathcal{I}^\uparrow$ is the set of all precomputed terms;
- the universe of the second sort of $\mathcal{I}^\uparrow$ is the set of all numerals;
- $\mathcal{I}^\uparrow$ interprets every precomputed term t as t;
- $\mathcal{I}^\uparrow$ interprets $\overline{m} + \overline{n}$ as $\overline{m+n}$, and similarly for subtraction and multiplication;
- $\mathcal{I}^\uparrow$ interprets every precomputed atom $p(t_1, \dots, t_n)$ as true iff it belongs to $\mathcal{I}$;
- $\mathcal{I}^\uparrow$ interprets every atomic sentence $t_1 \prec t_2$, where t_1 and t_2 are precomputed terms, as true iff the relation $\prec$ holds for the pair (t_1, t_2).

Main Lemma. *Let* Π *be a program and let* $\mathbf{p}$ *be the list of all predicate symbols occurring in* Π *other than the comparison symbols. A set* $\mathcal{I}$ *of precomputed atoms is a stable model of* Π *iff* $\mathcal{I}^\uparrow$ *is a* $\mathbf{p}$*-stable model of* $\tau^*\Pi$.

Proofs are relegated to Appendix A.[2]

5 Programs with Input and Output

5.1 Syntax

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

$$(\Pi, PH, In, Out), \tag{5}$$

where

- Π is a finite set of rules;
- PH is a finite set of symbolic constants, called the *placeholders* of (5);
- In is a finite set of predicate symbols that do not occur in the heads of rules of Π; they are called the *input symbols* of (5);
- Out is a finite set of predicate symbols that is disjoint from In; they are called the *output symbols* of (5).

The input and output symbols of an io-program are collectively called its *public symbols*; the predicate symbols occurring in the rules that are not public are called *private*. We also distinguish between public and private atoms, depending on whether their predicate symbols are public or private. Every set $\mathcal{I}$ of public atoms is the union of two disjoint sets—the set $\mathcal{I}^{in}$ of atoms in $\mathcal{I}$ that begin with an input symbol and the set $\mathcal{I}^{out}$ of atoms that begin with an output symbol.

As an example, consider the io-program Ω_1 with rules (1), the placeholder n, the input symbol $s/2$, and the output symbol $in_cover/1$. The only private symbol of Ω_1 is $covered/1$.

By σ_Ω, we denote the part of the signature σ_Π (Section 3) obtained from it by removing all private symbols. Formulas of that signature will be used in writing specifications.

Programs can be viewed as a special case of io-programs if we agree to identify a program Π with the io-program (5) in which PH and In are empty and Out is the set of all predicate symbols that occur in the rules of Π.

[2] Appendix available online at `https://arxiv.org/abs/2008.02025`.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

5.2 Semantics

A *valuation* for an io-program Ω is a function defined on the set of placeholders of Ω such that its values are precomputed terms different from placeholders. An *input* for Ω is a pair $(\mathbf{v}, \mathbf{i})$, where $\mathbf{v}$ is a valuation and $\mathbf{i}$ is a set of precomputed atoms that do not contain placeholders, such that the predicate symbol of each atom in $\mathbf{i}$ is an input symbol of Ω.

For an io-program Ω and an input $(\mathbf{v}, \mathbf{i})$, we denote by $\Omega(\mathbf{v}, \mathbf{i})$ the program consisting of

- the rules obtained from the rules of Ω by replacing every occurrence of every placeholder t by the term $\mathbf{v}(t)$ and
- the facts $\mathbf{i}$.

About a set of precomputed atoms, we say that it is an *io-model* of an io-program Ω for an input $(\mathbf{v}, \mathbf{i})$ if it is the set of all public atoms of some stable model of $\Omega(\mathbf{v}, \mathbf{i})$.

For example, we can define an input $(\mathbf{v}_1, \mathbf{i}_1)$ for Ω_1 by the conditions

$$\mathbf{v}_1(n) = \overline{3}, \quad \mathbf{i}_1 = \{s(a, \overline{1}), s(b, \overline{1}), s(b, \overline{2}), s(c, \overline{2}), s(c, \overline{3})\}.$$

With this input, Ω_1 encodes the exact covers of the collection $\{\{a, b\}, \{b, c\}, \{c\}\}$. The program $\Omega_1(\mathbf{v}_1, \mathbf{i}_1)$ consists of the rules

$$\begin{aligned}
&\{\mathit{in_cover}(\overline{1}\,..\,\overline{3})\},\\
&\leftarrow I \neq J \wedge \mathit{in_cover}(I) \wedge \mathit{in_cover}(J) \wedge s(X, I) \wedge s(X, J),\\
&\mathit{covered}(X) \leftarrow \mathit{in_cover}(I) \wedge s(X, I),\\
&\leftarrow s(X, I) \wedge \mathit{not}\ \mathit{covered}(X),\\
&s(a, \overline{1}),\ \ldots,\ s(c, \overline{3}).
\end{aligned}$$

It has one stable model,

$$\{s(a, \overline{1}), \ldots, s(c, \overline{3}), \mathit{in_cover}(\overline{1}), \mathit{in_cover}(\overline{3}), \mathit{covered}(a), \mathit{covered}(b), \mathit{covered}(c)\},$$

so that the only io-model of Ω is $\{s(a, \overline{1}), \ldots, s(c, \overline{3}), \mathit{in_cover}(\overline{1}), \mathit{in_cover}(\overline{3})\}$.

It is clear that atoms in an io-model do not contain private symbols. Furthermore, if $\mathcal{I}$ is an io-model of Ω for an input $(\mathbf{v}, \mathbf{i})$, then $\mathcal{I}^{in} = \mathbf{i}$. This is clear from the fact that the only rules of $\Omega(\mathbf{v}, \mathbf{i})$ containing input symbols in the head are the facts $\mathbf{i}$.

IO-models of io-programs also have a characterization in terms of first-order formulas similar to the characterization of stable models in the Main Lemma above (see Appendix A.2).[2] The proof of this characterization is based on the aforementioned Main Lemma and results from (Ferraris et al. 2009) and (Cabalar et al. 2020). This is later used in the proofs of Theorems 1 and 2 below.

5.3 Specifications

In this paper, we are interested in io-programs written in response to formal specifications. A specification, as we understand this term, includes a list *PH* of placeholders, a list *In* of input symbols, and a list *Out* of output symbols that a future io-program Ω is required to have. (The choice of private symbols of Ω is left to the discretion of the programmer.) That part of the specification determines the set of valuations and inputs of Ω as well as the signature σ_Ω. A specification also includes two lists of sentences in first-order logic over that signature: a list A of *assumptions* and a list S of *specs*. The sentences in A do

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

not contain output symbols. They are intended to describe the class of inputs that the author of the specification is interested in. The sentences in S are meant to describe the io-models that the future io-program is required to have for these inputs.

For example, a specification for the exact cover problem (see Footnote 1) may include the assumptions

$$\begin{aligned}&\exists N\,(n = N \wedge N \geq \overline{0}),\\&\forall XY\,(s(X,Y) \rightarrow \exists I\,(Y = I \wedge I \geq \overline{1} \wedge I \leq n)),\end{aligned} \tag{6}$$

where N and I are integer variables. They tell us that a "good" input describes the length and composition of a list of sets of precomputed atoms. Its specs may look like this:

$$\begin{aligned}&\forall Y\,(\mathit{in_cover}(Y) \rightarrow \exists I\,(Y = I \wedge I \geq \overline{1} \wedge I \leq n)),\\&\forall X\,(\exists Y\,s(X,Y) \rightarrow \exists Y(s(X,Y) \wedge \mathit{in_cover}(Y))),\\&\forall YZ\,(\exists X\,(s(X,Y) \wedge s(X,Z)) \wedge \mathit{in_cover}(Y) \wedge \mathit{in_cover}(Z) \rightarrow Y = Z).\end{aligned} \tag{7}$$

These formulas describe the concept of an exact cover.

To give a precise definition of the semantics of specifications, we need to generalize the process of transforming a set $\mathcal{I}$ of precomputed atoms into the interpretation $\mathcal{I}^{\uparrow}$ defined in Section 4. This generalization takes into account the fact that placeholders may be "reinterpreted" by a valuation. For any set $\mathcal{I}$ of precomputed public atoms that do not contain placeholders and any valuation $\mathbf{v}$, we denote by $\mathcal{I}^{\mathbf{v}}$ the interpretation of the signature σ_Ω such that

- the universe of the first sort of $\mathcal{I}^{\mathbf{v}}$ is the set of all precomputed terms;
- the universe of the second sort of $\mathcal{I}^{\mathbf{v}}$ is the set of all numerals;
- $\mathcal{I}^{\mathbf{v}}$ interprets every placeholder t as $\mathbf{v}(t)$ and every other precomputed term t as t;
- $\mathcal{I}^{\mathbf{v}}$ interprets $\overline{m} + \overline{n}$ as $\overline{m+n}$, and similarly for subtraction and multiplication;
- $\mathcal{I}^{\mathbf{v}}$ interprets every precomputed atom $p(t_1,\dots,t_n)$ that does not contain placeholders as true iff it belongs to $\mathcal{I}$;
- $\mathcal{I}^{\mathbf{v}}$ interprets every atomic sentence $t_1 \prec t_2$, where t_1 and t_2 are precomputed terms different from placeholders, as true iff the relation $\prec$ holds for the pair (t_1,t_2).

In the exact cover example, $\mathcal{I}^{\mathbf{v}}$ interprets the placeholder n as $\mathbf{v}(n)$. This interpretation satisfies the first of assumptions (6) iff $\mathbf{v}(n)$ is a nonnegative numeral.

We say that Ω *implements* the specification with assumptions A and specs S if, for every valuation $\mathbf{v}$ and every set $\mathcal{I}$ of precomputed public atoms without placeholders such that $\mathcal{I}^{\mathbf{v}}$ satisfies A,

$$\mathcal{I} \text{ is an io-model of } \Omega \text{ for the input } (\mathbf{v},\mathcal{I}^{in}) \text{ iff } \mathcal{I}^{\mathbf{v}} \text{ satisfies } S.$$

In the next section, we show that this condition can be sometimes established by proving a sentence in a first-order theory. This is how ANTHEM works.

6 Completion and Tightness

6.1 Completed Definitions

The *definition* of a predicate symbol p/n in an io-program Ω is the set of all rules of Ω that have the forms

$$p(t_1,\dots,t_n) \leftarrow \mathit{Body} \tag{8}$$

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

and

$$\{p(t_1, \dots, t_n)\} \leftarrow \mathit{Body}\,. \tag{9}$$

If the definition of p/n in Ω consists of 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 (8), then F_i is the formula

$$\tau^B(\mathit{Body}) \wedge \mathit{val}_{t_1}(V_1) \wedge \cdots \wedge \mathit{val}_{t_n}(V_n)$$

(see Section 3 for the definitions of τ^B and *val*). If R_i is (9), then F_i is the formula

$$\tau^B(\mathit{Body}) \wedge p(V_1, \dots, V_n) \wedge \mathit{val}_{t_1}(V_1) \wedge \cdots \wedge \mathit{val}_{t_n}(V_n). \tag{10}$$

We are ready to define "completed definitions" of predicate symbols. Completed definitions are generally second-order formulas—the private symbols of Ω are represented in them by predicate variables.

Let Ω be an io-program with private symbols $p_1/n_1, \dots, p_l/n_l$. Choose pairwise distinct predicate variables $P_1, \dots, P_l$ and take a predicate symbol p/n that occurs in Ω and is different from its input symbols. The *completed definition* of p/n in Ω is obtained from the formula

$$\forall V_1 \dots V_n \left(p(V_1, \dots, V_n) \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{U}_i\, F_i \right), \tag{11}$$

where $\mathbf{U}_i$ is the list of all variables occurring in rule R_i after substituting $P_1, \dots, P_l$ for $p_1/n_1, \dots, p_l/n_l$.

For example, the completed definition of $\mathit{in_cover}/1$ in Ω_1 is

$$\begin{aligned} &\forall V\,(\mathit{in_cover}(V) \\ &\quad \leftrightarrow \top \wedge \mathit{in_cover}(V) \wedge \exists IJK\,(I = \overline{1} \wedge J = n \wedge I \leq K \wedge K \leq J \wedge V = K)). \end{aligned} \tag{12}$$

It can be equivalently rewritten as

$$\forall V\,(\mathit{in_cover}(V) \rightarrow \exists IJK\,(I = \overline{1} \wedge J = n \wedge I \leq K \wedge K \leq J \wedge V = K)).$$

6.2 Completion

For a constraint $\leftarrow \mathit{Body}$, its *formula representation* is defined as the formula obtained from the universal closure of $\neg\tau^B(\mathit{Body})$ by substituting $P_1, \dots, P_l$ for $p_1/n_1, \dots, p_l/n_l$. For example, the formula representation of the second constraint of Ω_1 is obtained from the formula

$$\forall XI\, \neg\tau^B(s(X, I) \wedge \mathit{not\ covered}(X))$$

by substituting a predicate variable *Covered* for the predicate constant *covered*. It can be equivalently rewritten as

$$\forall XI\, \neg(s(X, I) \wedge \neg \mathit{Covered}(X)). \tag{13}$$

The *completion* $\mathrm{COMP}(\Omega)$ of an io-program Ω is the sentence $\exists P_1 \dots P_l\, F$, where $P_1, \dots, P_l$ are the predicate variables corresponding to the private symbols of Ω, as described above, and F is the conjunction of

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

- the completed definitions of all predicate symbols occurring in Ω other than its input symbols and
- the formula representations of all constraints in Ω.

Note that this sentence does not contain any of the private symbols of Ω. In other words, it is a second-order sentence over the signature σ_Ω (Section 5.1).

For example, the predicate completion of Ω_1 is

$$\exists \mathit{Covered}\,(F_1 \wedge F_2 \wedge F_3 \wedge F_4),$$

where F_1 is the completed definition (12) of *in_cover*, F_2 is the completed definition of *covered*, and F_3, F_4 are the formula representations of the two constraints.

Theorem 1. *If $\mathcal{I}$ is an io-model of an io-program Ω for an input $(\mathbf{v}, \mathbf{i})$, then the interpretation $\mathcal{I}^{\mathbf{v}}$ satisfies* COMP(Ω).

6.3 Tight Programs

The *predicate dependency graph* of an io-program Ω is the directed graph that

- has the predicate symbols occurring in rules of Ω as its vertices and
- has an edge from p/n to q/m if there is a rule R in Ω such that p/n occurs in the head of R and q/m occurs in the body of R.

The edge from p/n to q/m in the predicate dependency graph is *positive* if there is a rule R in Ω such that p/n occurs in the head of R and q/m occurs in an atom in the body of R that is not preceded by *not*.

An io-program is *tight* if its predicate dependency graph has no cycles consisting of positive edges.

Theorem 2. *For any tight io-program Ω and any set $\mathcal{I}$ of precomputed public atoms without placeholders, $\mathcal{I}$ is an io-model of Ω for an input $(\mathbf{v}, \mathbf{i})$ iff $\mathcal{I}^{\mathbf{v}}$ satisfies* COMP(Ω) *and $\mathcal{I}^{in} = \mathbf{i}$.*

This theorem shows that, in application to tight io-programs, the semantics of specifications (Section 5.3) can be characterized in terms of completion: A tight io-program Ω implements the specification with assumptions A and specs S iff, for every valuation $\mathbf{v}$ and every set $\mathcal{I}$ of precomputed public atoms without placeholders, the interpretation $\mathcal{I}^{\mathbf{v}}$ satisfies the sentence

$$A \to (\mathrm{COMP}(\Omega) \leftrightarrow S). \tag{14}$$

(We identify a list of sentences with the conjunction of its members.)

An interpretation of the signature σ_Ω is *standard* if it can be represented in the form $\mathcal{I}^{\mathbf{v}}$ for some set $\mathcal{I}$ of precomputed public atoms without placeholders and some valuation $\mathbf{v}$. We can then say that a tight io-program Ω implements the specification with assumptions A and specs S iff sentence (14) is satisfied by all standard interpretations.

The automation of reasoning about formulas of form (14) is difficult because COMP(Ω) has, generally, bound predicate variables. We will now discuss a special case when reasoning about such formulas can be reduced to first-order reasoning.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

6.4 Private Recursion

Recall that the completion of an io-program Ω is a sentence of the form $\exists \mathbf{P}\, F$, where $\mathbf{P}$ is a list $P_1, \dots, P_l$ of predicate variables (Section 6.2). That list has the same length l as the list $\mathbf{p}$ of the private symbols $p_1/n_1, \dots, p_l/n_l$ of Ω. Formula F is a conjunction that includes, among its conjunctive terms, the completed definitions $F_1(\mathbf{P}), \dots, F_l(\mathbf{P})$ of $p_1/n_1, \dots, p_l/n_l$. Denote the conjunction of all other conjunctive terms of F by $F'(\mathbf{P})$. Then, the completion of Ω can be written as

$$\exists \mathbf{P}\, (F_1(\mathbf{P}) \wedge \cdots \wedge F_l(\mathbf{P}) \wedge F'(\mathbf{P})). \tag{15}$$

About an io-program Ω, we say that it *uses private recursion* if

- its predicate dependency graph has a cycle such that every vertex in it is a private symbol or
- it includes a choice rule with a private symbol in the head.

Theorem 3. *If an io-program Ω does not use private recursion, then its completion is equivalent to*

$$\forall \mathbf{P}\, (F_1(\mathbf{P}) \wedge \cdots \wedge F_l(\mathbf{P}) \to F'(\mathbf{P})). \tag{16}$$

From this theorem, we can conclude that, if Ω does not use private recursion, sentence (14) is equivalent to

$$\begin{aligned}&(A \to (\exists \mathbf{P}\, (F_1(\mathbf{P}) \wedge \cdots \wedge F_l(\mathbf{P}) \wedge F'(\mathbf{P})) \to S))\\ \wedge\, &(A \to (S \to \forall \mathbf{P}\, (F_1(\mathbf{P}) \wedge \cdots \wedge F_l(\mathbf{P}) \to F'(\mathbf{P}))))\end{aligned}$$

and consequently to

$$\forall \mathbf{P}\, (A \wedge F_1(\mathbf{P}) \wedge \cdots \wedge F_l(\mathbf{P}) \to (F'(\mathbf{P}) \leftrightarrow S)). \tag{17}$$

The procedure used by ANTHEM for program verification is applicable to an io-program if it is tight and does not use private recursion. The procedure is based on the following idea. We choose a set of *axioms*—sentences over the signature Ω that are satisfied by all standard interpretations. This set may include, for instance, the formulas $\forall N\, (N + \overline{0} = N)$ and $\overline{2} \times \overline{2} = \overline{4}$. It may include also the formula $a \neq b$, where a and b are distinct symbolic constants, unless at least one of them is a placeholder. To establish that Ω implements the given specification, ANTHEM verifies that the axioms entail sentence (17). This can be accomplished by first-order reasoning—by verifying that the axioms entail the sentence obtained from (17) by replacing the bound predicate variables with distinct predicate constants that do not occur in the axioms. The predicate constants used by ANTHEM are actually the private predicate symbols from the rules of Ω, so that the formula derived by ANTHEM from the axioms is

$$A \wedge F_1(\mathbf{p}) \wedge \cdots \wedge F_l(\mathbf{p}) \to (F'(\mathbf{p}) \leftrightarrow S).$$

Proof search is performed by the resolution theorem prover VAMPIRE. The task is divided into two parts: deriving the specs S from the axioms, the assumptions A, and the formulas $F_1(\mathbf{p}), \dots, F_l(\mathbf{p}), F'$ ("forward direction") and deriving F' from the axioms, the assumptions A, and the formulas $F_1(\mathbf{p}), \dots, F_l(\mathbf{p}), S$ ("backward direction").

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

```
input: n -> integer, s/2.
output: in_cover/1.
assume: n >= 0.
assume: forall Y (exists X s(X, Y) -> exists I (Y = I and I >= 1 and I <= n)).
spec: forall Y (in_cover(Y) -> exists I (Y = I and I >= 1 and I <= n)).
spec: forall X (exists Y s(X, Y) -> exists Y (s(X, Y) and in_cover(Y))).
spec: forall Y, Z (exists X (s(X, Y) and s(X, Z)) and in_cover(Y) and in_cover(Z)
                   -> Y = Z).
```

Fig. 1. A specification file to verify that a program encoding the exact cover problem fulfills specification (7) under assumptions (6). First, n is declared a placeholder using an `input` statement. The `-> integer` notation instructs ANTHEM to represent n by an integer variable internally. Next, the input and output predicates of Ω_1 ($s/2$ and $in/1$, respectively) are declared. The `assume` and `spec` statements encode assumptions (6) and specs (7).

7 Verifying Tight Logic Programs with ANTHEM

Let us return to the exact cover problem, which we encoded in rules (1) and represented as io-program Ω_1 earlier. We would like to show that this program implements specification (7) under the assumptions stated in (6). This verification can be automated with ANTHEM 1.0.[3] To this effect, ANTHEM reads a logic program and a specification, checks that the program is tight and does not contain private recursion, generates the program's completion, and then performs automated proof searches trying to establish that the program actually implements the given specification.

As the input program, we can simply pass rules (1) to ANTHEM in the input language of GRINGO. For the specification, ANTHEM accepts files in a custom *specification file* format. Specification files may contain multiple types of statements. `input` and `output` statements specify input predicates and placeholders as opposed to output predicates. Axioms, assumptions and specs are expressed by `axiom`, `assume`, and `spec` statements, respectively. ANTHEM's specification file format has a simple syntax for such formulas. This format is a concise form of the segment of the TPTP language (Sutcliffe 2017) that is required for expressing specifications.

Similar to the input language of GRINGO, identifiers starting with a lowercase letter are interpreted as predicate and object constants, while variables are denoted by identifiers starting with an uppercase letter. An important distinction is that variables starting with letters `I`, `J`, `K`, `L`, `M`, or `N` are implicitly assumed to be integer variables—as opposed to program variables, which must start with `U`, `V`, `W`, `X`, `Y`, or `Z`. ANTHEM shows an error message if variables are used that start with letters not listed above. Terms may contain integer arithmetics with a syntax close to GRINGO. In addition to that, the keywords `not`, `and`, `or`, `->`, `<->`, `exists`, and `forall` denote the common operators of first-order logic.

We can encode specification (7) and assumptions (6) in this format, as shown in Figure 1. If we give this specification to ANTHEM along with the exact cover program encoded in (1), ANTHEM first generates the program's completion and shows it to the user (see Figure 2). This is done as described in Section 6.2. Afterward, ANTHEM verifies very quickly that

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

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

```
forall X (covered(X) <-> exists N (in_cover(N) and s(X, N)))
forall X (in_cover(X) <-> in_cover(X) and exists N (1 <= N and N <= n and X = N))
forall N1, N2, X not (N1 != N2 and in_cover(N1) and in_cover(N2)
                      and s(X, N1) and s(X, N2))
forall X, N not (s(X, N) and not covered(X))
```

Fig. 2. ANTHEM generates the completion of the input program—here, Ω_1, the exact cover problem encoded by rules (1). The first two lines contain the completed definitions of *covered*/1 and *in_cover*/1. They are followed by the two translated constraints. Note that ANTHEM applies basic equivalent transformations to simplify the completion formulas. For example, the completed definition of *in_cover* shown here corresponds to formula (12), but the integer variables I and J were eliminated by ANTHEM. The last line matches formula (13), the formula representation of the second constraint of Ω_1.

```
input: n -> integer.
assume: n >= 0.
output: q/1.
spec: exists N (forall X (q(X) <-> X = N) and N >= 0
                and N * N <= n and (N + 1) * (N + 1) > n).
```

Fig. 3. A specification for programs calculating the floor of the square root of an integer n.

the program indeed implements the specification in Figure 1 using the theorem prover VAMPIRE and informs the user accordingly.[4]

To accomplish this, ANTHEM splits the equivalence verification task into two parts, as mentioned at the end of Section 6.4. In either direction, ANTHEM starts with a set of presupposed formulas and performs one proof step for each formula to verify. For each proof step, ANTHEM devises a program in the TFF ("typed first-order formula") segment of the TPTP language. In this TPTP program, all presupposed formulas are represented as TPTP axioms, and the formula to verify in this proof step is used as the conjecture. ANTHEM then passes this TPTP program to VAMPIRE. If a proof step is successful within a predefined time limit, the formula is added to the list of presupposed formulas and the next proof step is conducted. The fact that ANTHEM uses the TPTP format to communicate with VAMPIRE is completely hidden from the user.

In some cases, VAMPIRE is incapable of verifying the equivalence of a specification to a program directly and within a reasonable time frame, for which there are two possible explanations. First, the set of axioms known to VAMPIRE might not be sufficient to perform the proof, and second, it might be necessary to further guide VAMPIRE's proof search by providing helper lemmas. To that end, ANTHEM allows users to supply additional axioms and helper lemmas with the `axiom` and `lemma` directives, respectively. ANTHEM verifies helper lemmas in the given order before the specs or the completion of the program. The usage of axioms and helper lemmas can be seen in a second example, a specification for programs that compute the floor of the square root of an integer n shown in Figure 3 and the implementation in Figure 4. ANTHEM and VAMPIRE verify the equivalence between

[4] When running VAMPIRE 4.4 with the options `--mode casc` and `--cores 8` on a computer with an Intel Core i7-7700K CPU and 16 GB RAM, the equivalence proof completes within a total of 0.37 seconds.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

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

Fig. 4. A program meant to implement the specification in Figure 3.

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

lemma(forward): forall X (p(X) <-> exists N (X = N and N >= 0 and N * N <= n)).
lemma(forward): forall X (q(X) <-> exists N2 (X = N2 and N2 >= 0 and N2 * N2 <= n
                                              and (N2 + 1) * (N2 + 1) > n)).
lemma(forward): forall N1, N2 (N1 >= 0 and N2 >= 0 and N1 < N2
                               -> N1 * N1 < N2 * N2).
lemma(forward): forall N (N >= 0 and p(N + 1) -> p(N)).
lemma(forward): not p(n + 1).
lemma(forward): forall N1, N2 (q(N1) and N2 > N1 -> not q(N2)).
lemma(forward): forall N (N >= 0 and not p(N + 1) -> (N + 1) * (N + 1) > n).

lemma(backward): forall N1, N2 (N1 >= 0 and N2 >= 0 and N1 * N1 <= N2 * N2
                                -> N1 <= N2).
lemma(backward): forall N (p(N) <-> 0 <= N and N <= n and N * N <= n).
lemma(backward): forall N (not p(N) and N >= 0 -> N * N > n).
lemma(backward): forall N (N >= 0 -> N * N < (N + 1) * (N + 1)).
lemma(backward): forall N1, N2 (p(N1) and not p(N2) and N2 >= 0 -> N1 < N2).
lemma(backward): forall N1, N2 (p(N1) and not p(N1 + 1) and p(N2)
                                and not p(N2 + 1) -> N1 = N2).
```

Fig. 5. VAMPIRE is unable to verify the equivalence of the specification in Figure 3 to the program in Figure 4 in a reasonable time frame. This can be addressed by supplying helper lemmas using the `lemma` keyword, where `forward` and `backward` instruct ANTHEM only to apply lemmas in the respective part of the equivalence proof. With these helper lemmas, VAMPIRE can perform the backward direction of the equivalence proof task but is still unable to verify the forward direction. To address this, the set of axioms is manually extended by a fact that is not known to VAMPIRE, an instance of induction. Such axioms can be added with `axiom` statements.

this specification and program only when supplying an axiom unknown to VAMPIRE and a list of helper lemmas (shown in Figure 5).

While ANTHEM 1.0 introduces the mode of operation described in the present paper, earlier versions of ANTHEM allowed users to try to prove that the stable models of a (not necessarily tight) program satisfy a specification (Lifschitz et al. 2020). This functionality is retained as a second mode of operation in ANTHEM 1.0.

8 Conclusions

This paper has four main contributions. First, we presented an extended definition of completion for programs with input and output in a subset of the input language of GRINGO. Such programs are similar to lp-functions, defined by Gelfond and Przymusinska (1996), except that we allow an input to include the values of placeholders. Also, they are closely related to SMODELS program modules (Oikarinen and Janhunen 2009). Second, we stated a result describing the relationship between stable models and completion in this context.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

It uses the theory of stable models of infinitary formulas by (Truszczyński 2012) and (Harrison et al. 2017). Third, we proposed a new theory of formal specifications for programs in the input language of GRINGO. Fourth, we presented preliminary experiments with the new version of ANTHEM, a software tool used in combination with a theorem prover to verify that programs with input and output implement a specification. Only few theorem provers support both the typed first-order form of TPTP and integer arithmetics, and VAMPIRE was chosen because its performance outclassed other provers in our early experiments. In later experiments, CVC4 emerged as a possible alternative with similar performance to VAMPIRE, and we consider using CVC4 in the future. As another direction of future work, we would like to extend the subset of the input language of GRINGO supported by ANTHEM, in particular, by simple forms of aggregates.

Acknowledgments

We thank Michael Gelfond and the anonymous referees for their valuable comments. Jorge Fandinno is funded by the Alexander von Humboldt Foundation.

References

Brewka, G., Eiter, T., and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Cabalar, P., Fandinno, J., and Lierler, Y. 2020. Modular answer set programming as a formal specification language. *Theory and Practice of Logic Programming,* this volume.

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*, Plenum Press, 293–322.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming 3,* 4-5, 499–518.

Fages, F. 1994. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer, 119–131.

Ferraris, P., Lee, J., and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence 175,* 1, 236–263.

Ferraris, P., Lee, J., Lifschitz, V., and Palla, R. 2009. Symmetric splitting in the general theory of stable models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, AAAI/MIT Press, 797–803.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming 15,* 4-5, 449–463.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, 1070–1080.

Gelfond, M. and Przymusinska, H. 1996. Towards a theory of elaboration tolerance: Logic programming approach. *International Journal of Software Engineering and Knowledge Engineering 6,* 1, 89–112.

Harrison, A., Lifschitz, V., Pearce, D., and Valverde, A. 2017. Infinitary equilibrium logic and strongly equivalent logic programs. *Artificial Intelligence 246*, 22–33.

Harrison, A., Lifschitz, V., and Raju, D. 2017. Program completion in the input language of GRINGO. *Theory and Practice of Logic Programming 17,* 5-6, 855–871.

Kovács, L. and Voronkov, A. 2013. First-order theorem proving and Vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, Springer, 1–35.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press

Lee, J., Lifschitz, V., and Palla, R. 2008. A reductive semantics for counting and choice in answer set programming. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, AAAI Press, 472–479.

Lifschitz, V., Lühne, P., and Schaub, T. 2018. anthem: Transforming gringo programs into first-order theories (preliminary report). In *Proceedings of the 11th Workshop on Answer Set Programming and Other Computing Paradigms*,

Lifschitz, V., Lühne, P., and Schaub, T. 2019. Verifying strong equivalence of programs in the input language of GRINGO. In *Proceedings of the 15th International Conference and Symposium of Logic Programming and Nonmonotonic Reasoning*, Springer, 270–283.

Lifschitz, V., Lühne, P., and Schaub, T. 2020. Towards verifying logic programs in the input language of CLINGO. In *Essays Dedicated to Yuri Gurevich on the Occasion of His 80th Birthday*, Springer, 190–209.

Oikarinen, E. and Janhunen, T. 2009. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation 19,* 4, 591–613.

Sutcliffe, G. 2017. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning 59,* 4, 483–502.

Truszczyński, M. 2012. Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Springer, 543–559.

https://doi.org/10.1017/S1471068420000344 Published online by Cambridge University Press