TPLP: Page –. © The Author(s), 20XX. Published by Cambridge University Press 20XX
doi:10.1017/S147106840100xxxx

FOLASP: *FO(·) as Input Language for Answer Set Solvers*

KYLIAN VAN DESSEL

JO DEVRIENDT

JOOST VENNEKENS

KU Leuven, Dept. of Computer Science, De Nayer Campus, Sint-Katelijne-Waver, Belgium
Leuven.AI – KU Leuven Institute for AI, Leuven, Belgium

Abstract

Over the past decades, Answer Set Programming (ASP) has emerged as an important paradigm for declarative problem solving. Technological progress in this area has been stimulated by the use of common standards, such as the ASP-Core-2 language. While ASP has its roots in non-monotonic reasoning, efforts have also been made to reconcile ASP with classical first-order logic (FO). This has resulted in the development of FO(·), an expressive extension of FO, which allows ASP-like problem solving in a purely classical setting. This language may be more accessible to domain experts already familiar with FO, and may be easier to combine with other formalisms that are based on classical logic. It is supported by the IDP inference system, which has successfully competed in a number of ASP competitions. Here, however, technological progress has been hampered by the limited number of systems that are available for FO(·). In this paper, we aim to address this gap by means of a translation tool that transforms an FO(·) specification into ASP-Core-2, thereby allowing ASP-Core-2 solvers to be used as solvers for FO(·) as well. We present experimental results to show that the resulting combination of our translation with an off-the-shelf ASP solver is competitive with the IDP system as a way of solving problems formulated in FO(·).

1 Introduction

Answer set programming (ASP) is a knowledge representation (KR) paradigm in which a declarative language is used to model and solve combinatorial (optimization) problems (Marek and Truszczyński 1999). It is supported by performant solvers (Gebser et al. 2020), such as Clingo (Gebser et al. 2016) and the DLV system (Leone et al. 2006). Development and use of these solvers has been simplified and encouraged by the emergence of the unified ASP-Core-2 standard (Calimeri et al. 2020).

The roots of ASP lie in the area of non-monotonic reasoning and its semantics is defined in a non-classical way. Work by Denecker et al. (Denecker and Ternovska 2008) has attempted to integrate key ideas from ASP with classical first-order logic (FO), in an effort to clarify ASP's contributions from a knowledge representation perspective. This has resulted in the development of the language FO(·), pronounced "ef-oh-dot", which is a conservative extension of FO. This language may be easier to use for domain experts who are already familiar with FO than ASP, and can seamlessly be combined with other monotonic logics. A number of systems, such as the IDP system (De Cat et al. 2016) and Enfragmo (Aavani 2014), already support the FO(·) language. However, when compared to the variety of solvers for ASP-Core-2, the support for FO(·) is still rather limited. This hinders technological progress, both in terms of solver development and the development of applications.

In this paper we present FOLASP, a tool that translates FO(·) to ASP-Core-2, thereby allowing each solver that supports ASP-Core-2 to handle FO(·) as well. In this way, we significantly extend the range of solvers that is available for FO(·).

We believe that this tool will make the FO(·) language more accessible and useful for practical applications, while also helping to drive technological progress. To develop FOLASP, we build on fundamental results about the relation between ASP and FO(·) (Mariën et al. 2004; Denecker et al. 2012), which we have for the first time combined into a working tool.

2 Preliminaries

2.1 FO(·)

FO(·) is an extension of classical typed first-order logic with aggregates, arithmetic, and (inductive) definitions. To maximize clarity, we will consider only a core subset of FO(·): typed FO extended with definitions, cardinality aggregates, and comparison operators.

A vocabulary V consists of a set of types T, predicates P and function symbols F. Each predicate P/n with arity n has an associated typing $\tau(P) = (T_1, \ldots, T_n)$, as has each function symbol F/n: $\tau(F) = (T_1, \ldots, T_{n+1})$.

A structure S for a vocabulary V (also known as a V-structure) consists of a *domain* D and an appropriate *interpretation* σ^S for each symbol $\sigma \in V$. The interpretation T^S of a type T is a subset of D, the interpretation P^S of a predicate P with $\tau(P) = (T_1, \ldots, T_n)$ is a relation $P^S \in T_1^S \times \cdots \times T_n^S$, and the interpretation F^S of a function symbol F with $\tau(F) = (T_1, \ldots, T_n, T_{n+1})$ is a function from $T_1^S \times \cdots \times T_n^S$ to T_{n+1}^S. The interpretations T_i^S of the types $T_i \in V$ partition the domain D, i.e., $\bigcup_i T_i^S = D$ and $T_i^S \cap T_j^S = \emptyset$ for $i \neq j$.

A *term* is either a variable, an integer, a function $f(\vec{t})$ applied to a tuple of terms $\vec{t}$, or a cardinality expression of the form $\#\{\vec{x}\colon \varphi(\vec{x})\}$, which intuitively represents the number of $\vec{x}$'s for which $\varphi(\vec{x})$ holds. Note that cardinality expressions are a special case of a aggregate expressions, which sometimes are introduced as generalized quantifiers working as atoms. Here, we use the FO terminology, where a cardinality expression is a term.

We use the notion of a *simple term* to refer to a variable or an integer. An *atom* is either a predicate $P(\vec{t})$ applied to a tuple of terms or a comparison $t_1 \bowtie t_2$ between two terms, with $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. As usual, *formulas* are constructed by means of the standard FO connectives $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \exists, \forall$. Only well-typed formulas are allowed. A *sentence* is a formula without free variables. A *positive literal* is an atom, a *negative literal* a negated atom.

As in FO, an FO(·) theory can be a set of sentences. However, in addition to sentences, FO(·) also allows *definitions*. Such a definition is a set of rules of the form:

$$\forall x_1, \ldots, x_n \colon P(x_1, \ldots, x_n) \leftarrow \varphi(x_1, \ldots, x_n).$$

where P/n is a predicate symbol, $x_1, \ldots, x_n$ variables, and φ a formula. The atom $P(x_1, \ldots, x_n)$ is the *head* of the rule, while φ is the *body*. The purpose of such a definition is to define the predicates that appear in the heads of the rules in terms of the predicates that appear only in the body. The first kind of predicates are called the *defined predicates* $Def(\Delta)$ of the definition Δ, while the second are called its *open predicates* $Open(\Delta)$.

The formal semantics of these definitions is given by a parametrized variant of the well-founded semantics (Van Gelder et al. 1988). In order for a definition to be valid in FO(·), it must be such that it uniquely determines a single interpretation for the defined predicates, given any interpretation for the open predicates. Formally, the condition is imposed on definitions that their well-founded model must exist and always be two-valued, no matter what the interpretation for their open predicates might be.

Different logical inference tasks can be considered for FO(·). In this paper, we focus on the most common task, namely that of *model expansion*.

Definition 1. *Let Th be a theory over vocabulary V and S a structure for a subvocabulary $Voc(S) \subseteq V$. The* model expansion *problem $MX(V, S, Th)$ is the problem of computing a V-structure $S' \supseteq S$ such that $S' \models Th$.*

Example 1. *The following example models the well-known graph coloring problem as a model expansion problem $MX(V, S, Th)$, with an illustration of a definition for the symmetric closure of the border relation.*

$$\begin{aligned}
V\colon & \textit{type Country},\ \textit{type Color},\\
& \textit{predicate Border with } \tau(\textit{Border}) = (\textit{Country}, \textit{Country}),\\
& \textit{predicate SymBorder with } \tau(\textit{SymBorder}) = (\textit{Country}, \textit{Country}),\\
& \textit{function symbol ColorOf with typing } \tau(\textit{ColorOf}) = (\textit{Country}, \textit{Color})\\
S\colon & \textit{Country}^S = \{be, nl, lux\}\\
& \textit{Color}^S = \{red, blue\}\\
& \textit{Border}^S = \{(nl, be), (be, lux)\}\\
Th\colon & \forall c_1, c_2\colon \textit{Border}(c_1, c_2) \Rightarrow \textit{ColorOf}(c_1) \neq \textit{ColorOf}(c_2)\\
& \left\{ \begin{array}{l} \forall c_1, c_2\colon \textit{SymBorder}(c_1, c_2) \leftarrow \textit{Border}(c_1, c_2).\\ \forall c_1, c_2\colon \textit{SymBorder}(c_1, c_2) \leftarrow \textit{SymBorder}(c_2, c_1). \end{array} \right\}
\end{aligned}$$

One solution to $MX(V, S, Th)$ is

$$\begin{aligned}
S'\colon & \textit{Country}^{S'} = \textit{Country}^S, \textit{Color}^{S'} = \textit{Color}^S, \textit{Border}^{S'} = \textit{Border}^S\\
& \textit{SymBorder}^{S'} = \{(nl, be), (be, nl), (be, lux), (lux, be)\}\\
& \textit{ColorOf}^{S'} = \{be \mapsto red, nl \mapsto blue, lux \mapsto blue\}
\end{aligned}$$

2.2 ASP

A normal logic program is a set of rules of the form:

$$H \text{ :- } B_1, \ldots, B_m, \textit{not } B_{m+1}, \ldots, \textit{not } B_n. \tag{1}$$

Here, H and all B_i are atoms. Corresponding with definitions, H is called the *head* of the rule, while the conjunction $B_1, \ldots, \textit{not } B_n$ is called the *body*. Both the rule head and body can be empty. A rule with empty head (= false) is called a *constraint*, a rule with empty body (= true) a *fact*.

The semantics of a program is defined in terms of its *grounding* which is an equivalent program without any variables, so all atoms are *ground atoms*. An interpretation I is a set of ground atoms. A rule of form (1) is satisfied in I if $H \in I$ whenever $B_1, \ldots, B_m \in I$ and $B_{m+1}, \ldots, B_n \notin I$. An interpretation is a model of a program if it is a model of each rule. The reduct of a program P w.r.t. interpretation I, denoted P^I, contains the ground rule $H \text{ :- } B_1, \ldots, B_m$ for each rule of form (1) for which none of the atoms $B_{m+1}, \ldots, B_n$ belong to I. An interpretation I is a *stable model* or *answer set* of program P if it is a minimal model of P^I.

The ASP-Core-2 language extends this basic formalism in a number of ways. For instance, it includes *choice rules* that can be used to express that a certain atom H *may* be true:

$$\{H\} \text{ :- } B_1, \ldots, B_m, \textit{not } B_{m+1}, \ldots, \textit{not } B_n.$$

Choice rules allow to generate a search space of candidate answer sets, from which the desired solutions can be filtered out with constraints.

A second extension is the cardinality aggregate, which, as in FO($\cdot$), counts the size of the set of free variable instantiations for which a conjunction of atoms holds. Throughout this paper, we make use of cardinality aggregates in body atoms, which have the form:

$$\#\textit{count}\{\vec{X}\colon B_1, \ldots, B_m, \textit{not } B_{m+1}, \ldots, \textit{not } B_n\} \bowtie t$$

with $\bowtie\ \in \{=, \neq, \leq, \geq, <, >\}$ and t a simple term.

Example 2. *The following ASP program is the counterpart of the FO(·) graph coloring model expansion problem from the previous section:*

$$\begin{aligned}
& \mathit{country}(be).\ \mathit{country}(nl).\ \mathit{country}(lux). \\
& \mathit{border}(nl, be).\ \mathit{border}(be, lux).\ \mathit{color}(red).\ \mathit{color}(blue). && (2)\\
\{\mathit{colorOf}(C, X)\} \text{ :- } & \mathit{country}(C), \mathit{color}(X). && (3)\\
\text{:- } & \#\mathit{count}\{C, X \colon \mathit{colorOf}(C, X), \mathit{color}(X)\} \neq 1, \mathit{country}(C). && (4)\\
\text{:- } & \mathit{border}(C_1, C_2), \mathit{colorOf}(C_1, X), \mathit{colorOf}(C_2, X). && (5)\\
\mathit{symBorder}(C_1, C_2) \text{ :- } & \mathit{border}(C_1, C_2). \\
\mathit{symBorder}(C_1, C_2) \text{ :- } & \mathit{symBorder}(C_2, C_1). && (6)
\end{aligned}$$

3 Translation of FO(·) to ASP

In this section, we define a translation α from an FO(·) model expansion problem $M = MX(V, S, Th)$ to an ASP program $\alpha(M)$. The translation α consists of four components $\alpha_1, \alpha_2, \alpha_3$ and α_4. α_1 and α_2 translate V and S, respectively (discussed in Section 3.2). α_3 and α_4 translate the FO(·) sentences and definitions that belong to Th, respectively (Section 3.3 and 3.4). Prior to translation we normalize the specification in order to make it compatible for translation to ASP (Section 3.1).

3.1 Normalization of an FO(·) specification

As a first step, we normalize the specification of the model expansion problem $MX(V, S, Th)$. Firstly, we convert all formulas in Th to negation normal form (NNF), i.e., the Boolean operators are restricted to negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$), and the negation operator is only applied directly to atoms, as described in (Enderton 2001). We also assume that the type T of any variable x is known, either by automated type derivation or by an explicit annotation, e.g., $\forall x[T] \colon \varphi$.

In FO(·), a function symbol can be interpreted by any function of the appropriate arity and type. By contrast, in ASP, each function symbol F is interpreted by the Herbrand function that maps each tuple of arguments $\vec{t}$ to the syntactic term $F(\vec{t})$. Hence, we eliminate function symbols from the FO(·) specification. For this, we first rewrite the theory such that function symbols only appear in atoms of the form $F(\vec{x}) = y$ with $\vec{x}$ and y simple terms. This is done by recursively replacing a (negated) atom $(\neg)A$ with subterm $F(\vec{t})$ by the NNF equivalent of

$$\forall x \colon F(\vec{t}) = x \Rightarrow (\neg)A[F(\vec{t})/x]$$

In a similar way, we *unnest* cardinality terms, such that these only occur in a comparison atom with simple terms, i.e., $\#\{\vec{x} \colon \varphi(\vec{x})\} \bowtie y$ with y a simple term. After this, each term t that is not simple occurs only in equality atoms $t = x$ (if t is a function symbol application) or comparison atoms $t \bowtie x$ (if t is a cardinality expression), with x a simple term.

Each function symbol F/n is then transformed into a predicate $P_F/n{+}1$ with the same typing, i.e., $\tau(F) = (T_1, \ldots, T_n, T_{n+1}) = \tau(P_F)$. We replace the atoms $F(\vec{x}) = y$ in the theory with $P_F(\vec{x}, y)$. We also add the constraint implied by using a function symbol, i.e., that each tuple of arguments has exactly one image, to the theory:

$$\forall x_1 \ldots\ x_n \colon \#\{x_{n+1} \colon P_F(x_1, \ldots, x_n, x_{n+1})\} = 1.$$

If the structure S has an interpretation F^S, then we replace it by

$$P_F^S = \{(a_1, \ldots, a_{n+1}) \mid (a_1, \ldots, a_n) \mapsto a_{n+1} \in F^S\}.$$

Finally, after eliminating function symbols, we push negations through atoms of the form $t \bowtie x$ by adjusting the $\bowtie$ operator. After this, the only negated atoms remaining have the form $\neg P(\vec{x})$ for some predicate P and simple terms $\vec{x}$.

Example 3. *The normalized FO(·) graph coloring theory from Section 2.1 is:*

$$\begin{aligned} Th' \colon & \forall x[Country] \colon \#\{y[Color] \colon ColorOf(x, y)\} = 1 \\ & \forall c_1[Country] \colon \forall c_2[Country] \colon \neg Border(c_1, c_2) \vee \\ & \quad \forall x[Color] \colon \neg ColorOf(c_1, x) \vee \neg ColorOf(c_2, x) \\ & \left\{ \begin{array}{l} \forall c_1, c_2 \colon SymBorder(c_1, c_2) \leftarrow Border(c_1, c_2). \\ \forall c_1, c_2 \colon SymBorder(c_1, c_2) \leftarrow SymBorder(c_2, c_1). \end{array} \right\} \end{aligned}$$

3.2 Generating the search space

The solutions to a model expansion problem $MX(V, S, Th)$ are to be found among the set of all structures for V that expand S – the *search space*. In this section, we translate vocabulary V and structure S to generate precisely this search space.

Unlike FO(·), ASP-Core-2 imposes strict naming conventions: variable names must start with a capital, while the names of all other kinds of symbols must start with a lower case letter. For an FO(·) type, predicate, variable or domain element σ, we denote by $\dot{\sigma}$ a corresponding ASP symbol of the right kind. Naturally, we enforce that $\dot{\sigma} \neq \dot{\sigma}'$ whenever $\sigma \neq \sigma'$.

In a model expansion problem $M = MX(V, S, Th)$, some of the symbols in V are interpreted by S (we denote these by $Voc(S)$), while others (i.e., $V \setminus Voc(S)$) are not. Because types must always be interpreted by S and our normalization step transforms all function symbols into predicates, the uninterpreted vocabulary $V \setminus Voc(S)$ consists entirely of predicates.

We translate each $P \in V \setminus Voc(S)$ with associated typing $\tau(P) = (T_1, \ldots, T_n)$ into the following ASP choice rule $\alpha_1(P)$:

$$\{\dot{P}(X_1, \ldots, X_n)\} \text{ :- } \dot{T}_1(X_1), \ldots, \dot{T}_n(X_n).$$

This provides a first component $\alpha_1(M)$ of our translation of the model expansion problem. The second component $\alpha_2(M)$ translates the structure S.

We translate an interpretation $P^S = \{(a_1^1, \ldots, a_n^1), \ldots, (a_1^m, \ldots, a_n^m)\}$ of an n-ary predicate P into the following set $\alpha_2(P^S)$ of m ASP facts:

$$\dot{P}(\dot{a}_1^1, \ldots, \dot{a}_n^1). \quad \ldots \quad \dot{P}(\dot{a}_1^m, \ldots, \dot{a}_n^m).$$

The interpretation of a type is translated as though it were a unary predicate. The second component $\alpha_2(MX(V, S, Th))$ of our translation now consists of all $\alpha_2(P^S)$ for which $P \in Voc(S)$. Together, α_1 and α_2 allow us to generate the correct search space in ASP, as the following theorem shows.

Theorem 1. *For each structure S for a subvocabulary $Voc(S) \subseteq V$, $MX(V, S, \{\}) = AnswerSets(A)$ with A the ASP program $\alpha_1(V \setminus Voc(S)) \cup \alpha_2(S)$).*

Here, the equality between structures and answer sets is of course modulo a straightforward "syntactic" transformation: we can transform each structure S to the answer set $f(S)$ that consists of all atoms $P(\vec{d})$ for which $\vec{d} \in P^S$. Because we consider a typed logic, in which each element of the domain of S must belong to the interpretation T^S of at least one type T, this transformation is an isomorphism, which we omit from our notation for simplicity.

Proof. The set $MX(V, S, \{\})$ consists of all V-structures S' that can be constructed by starting from the structure S and then adding, for each predicate $P \in V \setminus Voc(S)$ with type $\tau(P) = (T_1, \ldots, T_n)$, any set of tuples $\subseteq T_1^S \times \cdots \times T_n^S$ as interpretation $P^{S'}$ of P in S'. For each predicate P interpreted by S, $\alpha_2(S)$ contains precisely all facts $P(\vec{d})$ for which $\vec{d} \in P^S$. Moreover, P does not appear in $\alpha_1(V \setminus Voc(S))$. This ensures that each answer set in $AnswerSets(A)$ contains precisely all atoms $P(\vec{d})$ for which $\vec{d} \in P^S$. For each predicate $P \in V \setminus Voc(S)$, S' may have any set of tuples $\subseteq T_1^S \times \cdots \times T_n^S$ in its interpretation. The choice rules in $\alpha_2(S)$ ensure that precisely these tuples also make up the possible interpretations for P in $AnswerSets(A)$. $\square$

3.3 Translating formulas

We now define a third component $\alpha_3(MX(V, S, Th))$ of our translation to transform the formulas $\varphi \in Th$ to ASP. We start by the base case: the translation $\alpha_3(A)$ for an atom A. Due to our normalization step, the only atoms A that appear in Th are of the form $P(\vec{x})$ (with P/n a predicate and $\vec{x}$ a tuple of simple terms) or $t \bowtie x$ (with t a simple term or a cardinality expression, $\bowtie$ a comparison operator, and x a simple term). The translation $\alpha_3(P(x_1, \dots, x_n))$ of a predicate atom is the ASP conjunction

$$\dot{P}(\dot{x_1}, \dots, \dot{x_n}), \dot{T_1}(\dot{x_1}), \dots, \dot{T_n}(\dot{x_n}) \tag{7}$$

with typing $\tau(P) = (T_1, \dots, T_n)$. Hence, the type information implicit in the typing of a predicate is added explicitly by means of the additional conjuncts $\dot{T_i}(X_i)$. With slight abuse of notation, we shorten such a conjunction of type atoms to $\dot{T}(\vec{x})$.

With x and y simple terms, the translation $\alpha_3(x \bowtie y)$ is the ASP conjunction

$$\dot{x} \bowtie \dot{y}, \dot{T}(\dot{x}), \dot{T}(\dot{y}). \tag{8}$$

The translation $\alpha_3(\#\{\vec{x}[\vec{T}]\colon \varphi(\vec{x}, \vec{y})\} \bowtie z)$ of a normalized cardinality atom is

$$\#count\{\dot{\vec{x}}\colon \delta(\dot{\vec{x}}, \dot{\vec{y}}), \dot{T}(\dot{\vec{x}})\} \bowtie \dot{z}, \dot{T}(\dot{\vec{y}}), \dot{T}(\dot{z}) \tag{9}$$

with δ a fresh auxiliary predicate representing the subformula φ. Hence, we also add the rule:

$$\delta(\dot{\vec{x}}, \dot{\vec{y}}) \text{ :- } \alpha_3(\varphi(\vec{x}, \vec{y})), \dot{T}(\dot{\vec{x}}), \dot{T}(\dot{\vec{y}}). \tag{10}$$

with a recursive application of α_3.

After normalization, a negation occurs only in literals of the form $\neg P(\vec{x})$, whose translation $\alpha_3(\neg P(\vec{x}))$ simply is

$$not\ \dot{P}(\dot{\vec{x}}), \dot{T}(\dot{\vec{x}}). \tag{11}$$

Note that the *not* is only added to the first atom and not to the type atoms.

Having defined how each (negated) atom $(\neg)A$ is translated into a corresponding ASP expression $\alpha_3((\neg)A)$, we now inductively define how more complex formulas are translated.

The translation $\alpha_3(\varphi \wedge \psi)$ of a conjunction is the ASP conjunction $\alpha_3(\varphi), \alpha_3(\psi)$.

The translation $\alpha_3(\varphi(\vec{x}) \vee \psi(\vec{y}))$ of a disjunction is the ASP atom $\delta(\dot{\vec{x}}, \dot{\vec{y}})$, with δ a fresh auxiliary predicate. Additionally, for each such auxiliary predicate, we add the following ASP rules:

$$\begin{aligned} \delta(\dot{\vec{x}}, \dot{\vec{y}}) &\text{ :- } \alpha_3(\varphi(\vec{x})), \dot{T}(\dot{\vec{x}}), \dot{T}(\dot{\vec{y}}). \\ \delta(\dot{\vec{x}}, \dot{\vec{y}}) &\text{ :- } \alpha_3(\psi(\vec{y})), \dot{T}(\dot{\vec{x}}), \dot{T}(\dot{\vec{y}}). \end{aligned} \tag{12}$$

to ensure that δ indeed corresponds to the disjunction of φ and ψ.

Since a variable that appears in the body of an ASP rule but not in its head is implicitly existentially quantified, the translation $\alpha_3(\exists x[T]\colon \varphi(x))$ of an existential quantification is the ASP conjunction $\alpha_3(\varphi(x)), \dot{T}(\dot{x})$.

The translation $\alpha_3(\forall x[T]\colon \varphi(x))$ of a universally quantified formula is the ASP cardinality atom $\alpha_3(\#\{x[T]\colon \varphi(x)\}) = n$, with $n = |T^S|$ the number of elements in type T. Note that, because of this step, the translation α_3 not only depends on the theory Th of our model expansion problem, but also on the structure S. A first way to avoid this dependence is to translate $\forall x[T]\colon \varphi(x)$ as $\alpha_3(\#\{x[T]\colon \neg\varphi(x)\}) = 0$. However, this would introduce an additional negation, which might lead to the introduction of loops over negation in Section 3.4. A second way introduces an aggregate term representing n, e.g., $\alpha_3(\#\{x[T]\colon \varphi(x)\} = \#\{x[T] : true\})$, but we expect this to be less efficient.

For any formula $\varphi(\vec{x})$, we can now use the transformation α_3 to define a fresh ASP symbol $\delta_\varphi(\dot{\vec{x}})$ such that the set of all $\vec{x}$ for which $\varphi(x)$ holds in the FO($\cdot$) theory Th coincides with the set of all $\dot{\vec{x}}$ for which $\delta_\varphi(\dot{\vec{x}})$ holds in the ASP program. We do this by adding the following *reification rule* r_φ:

$$\delta_\varphi(\dot{\vec{x}}) \mathrel{:\!-} \alpha_3(\varphi(\vec{x})), \dot{T}(\dot{\vec{x}}).$$

Theorem 2. *Let φ be a formula in vocabulary V and let S be a structure for V. Consider the ASP program R_φ that consists of all reification rules r_ψ for which ψ is a subformula of φ, together with all additional rules produced by the translation α_3 (see Eq. 10 and Eq. 12). Let $R_S = \alpha_2(S)$ be the translation of the structure S. Then $R_S \cup R_\varphi$ has a unique answer set $\mathbb{A}$ and for each subformula ψ the set of all $\vec{d}$ for which $S \models \psi(\vec{d})$ is equal to the set of all $\vec{d}$ for which $\delta_\psi(\vec{d}) \in \mathbb{A}$.*

Proof. R_S is a set of facts over V. R_φ is a strictly stratified set of rules with non-empty heads. Therefore, it is clear that the answer set of $R_S \cup R_\varphi$ is indeed unique. We now prove the theorem by induction over the subformula order. The base cases are atoms as translated in Eq. 7 and Eq. 8. Here, it is obvious from the translation that the correspondence holds. For an aggregate (Eq. 10), we can apply the induction hypothesis to obtain a correspondence between the tuples for which φ in the original aggregate holds and the tuples for which the fresh predicate δ in its translation holds; from this, the result follows. Similarly, the case for disjunction follows from applying the induction hypothesis to the fresh predicates in Eq. 12. The cases for negation, conjunction and existential quantification are trivial. The case for universal quantification follows immediately from the correctness of the translation of aggregates. □

Once we have the reification rules r_φ as defined above, we can eliminate answer sets in which the formula φ is not satisfied by adding a constraint :- *not* δ_φ. Denoting such a constraint by C_φ, the third component of our translation – the translation of sentences – now is

$$\alpha_3(MX(V, S, Th)) = \{r_\varphi \mid \varphi \in Th\} \cup R \cup \{C_\varphi \mid \varphi \in Th\},$$

where the rules r_φ and constraints C_φ are as above and R are all of the additional rules (see Eq. 10 and Eq. 12) generated by producing the r_φ.

Theorem 3. *Let M be a model expansion problem $MX(V, S, Th)$ in which Th is a set of FO sentences. The solutions to M coincide with the answer sets of $\alpha_1(M) \cup \alpha_2(M) \cup \alpha_3(M)$.*

Proof. By induction on the size of Th. The base case in which $|Th| = 0$ and therefore $Th = \{\}$ is covered by Theorem 1. Once the induction hypothesis gives us the correspondence between a theory Th of size $n-1$ and an ASP program A_{n-1}, we can add an additional formula φ_n and prove the correspondence between $Th \cup \{\varphi_n\}$ and $A_n = A_{n-1} \cup \{r_{\varphi_n}, C_{\varphi_n}\} \cup R$, with R the additional rules for producing $\alpha_3(\varphi_n)$. The atoms in the head of the new rules $\{r_{\varphi_n}, C_{\varphi_n}\} \cup R$ are all fresh atoms that do not appear in A_{n-1}. Therefore there can be no interference between the new rules and the old ones, and the result follows from Theorem 2. □

We now have a translation for theories that consists entirely of FO sentences. The next section examines how we can extend this to FO(·) theories that contain also definitions.

3.4 Translating definitions

In general, a theory in FO(·) can contain multiple definitions. However, it is well-known that each such theory can be transformed into a theory that contains just a single definition Van Gelder et al. (1991). This involves merging the different definitions and possibly renaming predicates to avoid the introduction of new loops. The necessity for this renaming step can be seen by comparing the following two theories: Th consists of two separate definitions (one defining p in terms of q and the other defining q in terms of p) and Th', which consists of a single definition that jointly defines both p and q:

$$Th = \left\{ \begin{array}{c} \{p \leftarrow q.\} \\ \{q \leftarrow p.\} \end{array} \right\} \qquad Th' = \left\{ \left\{ \begin{array}{c} p \leftarrow q. \\ q \leftarrow p. \end{array} \right\} \right\}$$

The theory *Th* has two models, namely $\{\}$ and $\{p, q\}$, while Th' has $\{\}$ as its unique model. We therefore cannot simply merge the two definitions in *Th*. The solution is to rename the predicates that are defined in (at least one of) these definitions, and then assert the equivalence between the old and the new predicates.

More formally, for each definition Δ, for each defined predicate P in Δ, we replace all occurrences of P in Δ with a fresh unique predicate P_Δ and add the equivalence constraint $P \Leftrightarrow P_\Delta$. Applying this merge procedure to *Th* yields the following theory:

$$Th'' = \left\{ p \Leftrightarrow p', q \Leftrightarrow q', \left\{ \begin{array}{c} p' \leftarrow q. \\ q' \leftarrow p. \end{array} \right\} \right\}$$

This avoids the introduction of additional loops and ensures Th'' equivalent to the original *Th*.

We therefore from now on assume that the theory of the model expansion problem contains only a single definition Δ. Each rule $r \in \Delta$ is of the form

$$\forall x_1[T_1] : \ldots : \forall x_n[T_n] : P(x_1, \ldots, x_n) \leftarrow \varphi.$$

We translate it to the following ASP rule $\alpha_4(r)$:

$$\dot{P}(\dot{x_1}, \ldots, \dot{x_n}) \text{ :- } \alpha_3(\varphi), \dot{T_1}(\dot{x_1}), \ldots, \dot{T_n}(\dot{x_n}).$$

We then define $\alpha_4(\Delta)$ as $\{\alpha_4(r) \mid r \in \Delta\}$.

We now first show the correctness of this transformation in isolation, before combining it with previous results.

Theorem 4. *Let Δ be a definition in vocabulary V and let S be a structure for Open(Δ), i.e, the set of all symbols in V that do not appear in the head of any rule of Δ. Then $\alpha_2(S) \cup \alpha_4(\Delta)$ has a unique answer set which coincides with the unique solution to $MX(V, S, \{\Delta\})$.*

Proof. A valid definition in FO($\cdot$) must be such that its well-founded model is always two-valued. Because the transformation from Δ to $\alpha_4(\Delta)$ introduces no additional loops over negation (in fact, it introduces no additional negations at all), the set of ASP rules $\alpha_4(\Delta)$ also has a two-valued well-founded model. It is well known that a two-valued well-founded model is also the unique stable model. Given this uniqueness result, the theorem now follows from the correctness of α_3 (Theorem 2). □

Note that this theorem does not hold for structures S that interpret some of the defined symbols of Δ. Consider, for instance, the definition consisting only of the rule $p \leftarrow true$ and the structure S in which $p^S = false$. The problem $MX(V, S, \Delta)$ has no solutions, but $\alpha_2(S) = \{\}$ and $\alpha_4(\Delta) = \{p \text{ :- } \}$, which means that $\alpha_2(S) \cup \alpha_4(\Delta)$ has $\{p\}$ as an answer set.

For the same reason, we cannot simply combine $\alpha_4(\Delta)$ with the choice rules introduced by α_1. To solve this problem, we will use the same renaming trick that we use to merge separate definitions.

Definition 2. *Let $MX(V, S, Th)$ be a model expansion problem in which the theory Th contains only a single definition Δ, and let $Th' = Th \setminus \{\Delta\}$. Let Δ' be the result of replacing each defined predicate P of Δ by a fresh predicate P_Δ, and denote by Eq the set of all equivalence constraints $\forall \vec{x}\colon P(\vec{x}) \Leftrightarrow P_\Delta(\vec{x})$ for defined predicates P. We define $\alpha(MX(V, S, Th))$ as the following ASP program:*

$$\alpha_1(MX(V, S, \cdot)) \cup \alpha_2(MX(V, S, \cdot)) \cup \alpha_3(MX(\cdot, S, Th' \cup Eq)) \cup \alpha_4(MX(\cdot, S, \Delta')).$$

(For clarity, arguments have been replaced by $\cdot$ where they are irrelevant.)

Theorem 5. *For a model expansion problem $M = MX(V, S, Th)$ in which the theory Th contains only a single definition Δ, the solutions to M coincide with the answer sets of $\alpha(M)$.*

Proof. Theorem 2 already shows that all parts of the model expansion problem apart from the definition are correctly translated by α_1, α_2 and α_3. Theorem 4 shows that the definition Δ can be correctly translated by $\alpha_4(\Delta)$. The renaming of the defined predicates of Δ ensures that both can be combined without invalidating the correctness of either theorem. □

3.5 Translating the graph coloring example

We now show how $MX(V, S, Th')$, with Th' the normalized theory from Example 3, can be translated to ASP. This translation consists of four parts – α_1, α_2, α_3, α_4 – which correspond to the translation of the vocabulary V, the structure S, the constraints in Th', and the definitions in Th', respectively.

Example 4. *The following is a translation of $M = MX(V, S, Th')$ from Example 3 to ASP:*

$$
\begin{array}{lrl}
\alpha_1(M): & \{colorOf(C,X)\} \text{ :- } & country(C), color(X). \\
& \{symBorder(C_1,C_2)\} \text{ :- } & country(C_1), country(C_2). \\
& \delta_2(C) \text{ :- } & \#count\{C,X\colon colorOf(C,X), color(X)\} = 1, country(C). \\
& \delta_1 \text{ :- } & \#count\{C\colon \delta_2(C), country(C)\} = 3. \\
& \text{ :- } & not\ \delta_1. \\
\alpha_2(M): & & country(be).\ country(nl).\ country(lux). \\
& & border(nl,be).\ border(be,lux).\ color(red).\ color(blue). \\
\alpha_3(M): & \delta_5(C_1,C_2,X) \text{ :- } & not\ colorOf(C_1,X), country(C_1), country(C_2), color(X). \\
& \delta_5(C_1,C_2,X) \text{ :- } & not\ colorOf(C_2,X), country(C_1), country(C_2), color(X). \\
& \delta_4(C_1,C_2) \text{ :- } & \#count\{X\colon \delta_5(C_1,C_2,X), color(X)\} = 2, \\
& & country(C_1), country(C_2). \\
& \delta_4(C_1,C_2) \text{ :- } & not\ border(C_1,C_2), country(C_1), country(C_2). \\
& \delta_3 \text{ :- } & \#count\{C_1,C_2\colon \delta_4(C_1,C_2), country(C_1), country(C_2)\} = 9. \\
& \text{ :- } & not\ \delta_3. \\
\alpha_4(M): & symBorder_\Delta(C_1,C_2) \text{ :- } & border(C_1,C_2). \\
& symBorder_\Delta(C_1,C_2) \text{ :- } & symBorder_\Delta(C_2,C_1). \\
& \text{ :- } & symBorder(C_1,C_2), not\ symBorder_\Delta(C_1,C_2). \\
& \text{ :- } & not\ symBorder(C_1,C_2), symBorder_\Delta(C_1,C_2).
\end{array}
$$

Example 4 and Example 2 are both ASP programs representing the same graph coloring problem. $\alpha_1(M)$ in Example 4 corresponds to rules (3) and (4) in Example 2, $\alpha_2(M)$ is the same set of facts (2), $\alpha_3(M)$ corresponds to the constraint (5), and $\alpha_4(M)$ corresponds to the rules (6).

It is clear that the translation in Example 4 is a lot less succinct. Firstly, the translation introduces a significant number of auxiliary predicates, both reification predicates δ_i and a renaming predicate $symBorder_\Delta$ for the defined predicate of the definition. Secondly, the universal quantifications in the FO(·) specification lead to several cardinality aggregates not present in the original formulation. Thirdly, the FO(·) implication that represents the graph coloring constraint is normalized into a nested disjunction (see Example 3) and this leads to four translated rules, compared to the single rule (5) in Example 2.

4 Implementation

By implementing this translation, we created a new model expansion engine for FO(·), called FOLASP. It uses the syntax of the IDP system for its input and output, and uses Clingo (Gebser et al. 2019) as back-end ASP solver.

In addition to the subset of FO(·) discussed in this paper, FOLASP also supports minimum and maximum aggregates, arithmetic, function symbols in the head of a definition, partial interpretations, and partial functions. We thereby cover almost all language constructs supported by IDP, except for symbol overloading, chained (in)equalities and constructed types. Besides the model expansion inference, FOLASP also supports the optimization inference, which computes a model that minimizes the value of some integer objective function.

Where appropriate, FOLASP uses the FO(·) type information to add "guards" of the form *type*(X) for each variable X to the bodies of the generated ASP rules. In other words, FOLASP generates so-called *safe* rules, which allows the resulting programs to be handled by ASP solvers such as Clingo.

FOLASP is implemented in Python 3. Its source code is published on Gitlab.[1] We tested the correctness of the implementation by checking that the solutions produced by FOLASP are accepted as such by IDP, and that, for optimization problems, the optimal objective values produced by IDP and FOLASP were in agreement.

5 Experiments

In our experiments, we evaluate FOLASP (commit 82ec7edc on the development branch) using Clingo (version 5.4.0) as backend ASP solver. This configuration is compared to two other approaches. The first comparison approach runs IDP (commit 4be3c797) on the same FO(·) specifications as taken as input by FOLASP. The second comparison runs Clingo (again version 5.4.0) on native ASP encodings of the same problems.

As benchmark set, we use the problem instances from the *model-and-solve* track of the fourth ASP competition (Alviano et al. 2013). Both IDP and Clingo participated in this competition, which means that we have—for these same problems—both FO(·) and ASP specifications already available, written by experts in both languages. As such, we believe these benchmarks provide a good opportunity for a fair comparison.

We used the scripts and FO(·) specifications from the IDP submission to generate FO(·) instances that are accepted by both IDP and FOLASP. For the ASP system Clingo, we used the native ASP encodings provided by the organizers of the fifth ASP competition (Calimeri et al. 2016) (which uses the same problem set) since Clingo could not parse the specifications from the fourth ASP competition.

IDP solves problems in NP. This covers all problems in the benchmark set, apart from the *strategic companies* problem, which has a higher complexity. In the competition itself, the IDP team therefore solved this problem using a separate script to generate an exponentially sized search space, which was then given to IDP. Because this trick is not representative of how IDP is intended to be used in the real world, we decided to omit this benchmark from our experiments. Fourteen benchmark families remained with each (close to) thirty instances. They cover a wide range of applications, from a simple reachability query over a transportation planning problem to optimizing the location of valves in an urban hydraulic network.

The experimental hardware consisted of a dual-socket Intel® Xeon® E5-2698 system with 512 GiB of RAM memory, with twenty hyper-threaded cores for each of the two processors. To reduce resource competition, we run only twenty instances simultaneously, using twenty threads, for a total of ten per processor. We employ a high memory limit of 64 GiB for each instance, as we observed that Clingo requires a significant amount of memory to solve instances generated by FOLASP.[2] We run the instances with a 6000 second timeout limit, but, to avoid imprecision at the timeout limit, we consider an instance unsolved if it takes more than 5000 seconds. Optimization instances are considered solved when the last solution is proven to be optimal.

Runnable software, instance files, and detailed experimental results are made available at Zenodo.[3]

We compare the efficiency of the three approaches—FOLASP, IDP and Clingo. Figure 1 (best viewed in color) shows the time needed for both FOLASP and IDP to solve each instance. For benchmarks such as *nomystery*, *sokoban*, *ricochet_robot* and

[1] `https://gitlab.com/EAVISE/folasp`
[2] Note that even though twenty simultaneous instances utilizing 64 GiB of memory each is more than the total of 512 GiB of memory available, this worst-case scenario did not occur in practice and the machine did not run out of memory.
[3] `https://doi.org/10.5281/zenodo.4771774`

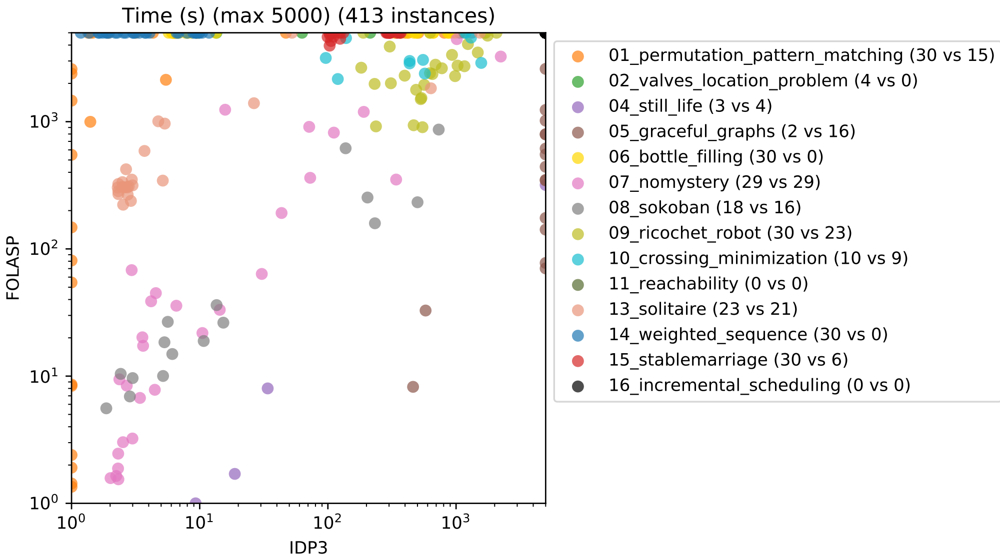

Fig. 1. Scatter plot comparing IDP and FOLASP runtime performance. "(x vs y)" denotes that IDP solved x instances within the family, and FOLASP y.

crossing_minimization, the performance of FOLASP and IDP is about equal. For benchmarks such as *permutation_pattern_matching*, *valves_location_problem*, *solitaire* and *weighted_sequence*, IDP clearly outperforms FOLASP. This suggests that the specifications that were hand-crafted by the IDP team are indeed well-suited to this solver's particular characteristics, and less to those of Clingo. In addition, our translation of course introduces a number of artifacts, such as reification predicates, renaming predicates and cardinality aggregates, that may adversely impact performance as well. Interestingly, however, on the *graceful_graphs* benchmark, FOLASP clearly outperforms IDP. Here, the inefficiencies of the translation are apparently overcome by the speed of Clingo. This highlights the usefulness of a translation such as ours: different benchmarks might be more suited for the architecture of different systems.

While the above experiments used different back-ends to handle precisely the same input, our next experiments (Figure 2) use the same Clingo back-end to solve the native ASP encodings as well as the translations that are automatically generated by FOLASP from the FO(·) specification for the same benchmark. Here, we see that performance is about equal for *sokoban*, *still_life* and *graceful_graphs*. For all other benchmarks apart from *nomystery*, the native ASP specification significantly outperforms the automatic translation. This further confirms our earlier remark that the native version is able to better take advantage of the particular properties of Clingo, and that our translation's performance may suffer from the introduction of artificial predicates.

In these experiments, the *nomystery* benchmark is the odd one out, since the FOLASP translation here significantly outperforms the native encodings. One possible explanation for this is that the modeling style encouraged by FO(·) has computational properties different to those of typical ASP programs, and that the FO(·) style happens to be particularly well-suited to *nomystery*. This would again point towards the value of a translation such as ours, but now from a different perspective: it is not only useful to be able to try out different back-ends with the same specification, but it is also useful to be able to run the same back-end with specifications that were written according to different paradigms.

In our discussion of the previous experiments, we have hypothesised that the artifacts of our translation may make FOLASP's specifications harder to solve. To further investigate this, Figures 3 and 4 compare the size of the ground programs for the three approaches. This ground

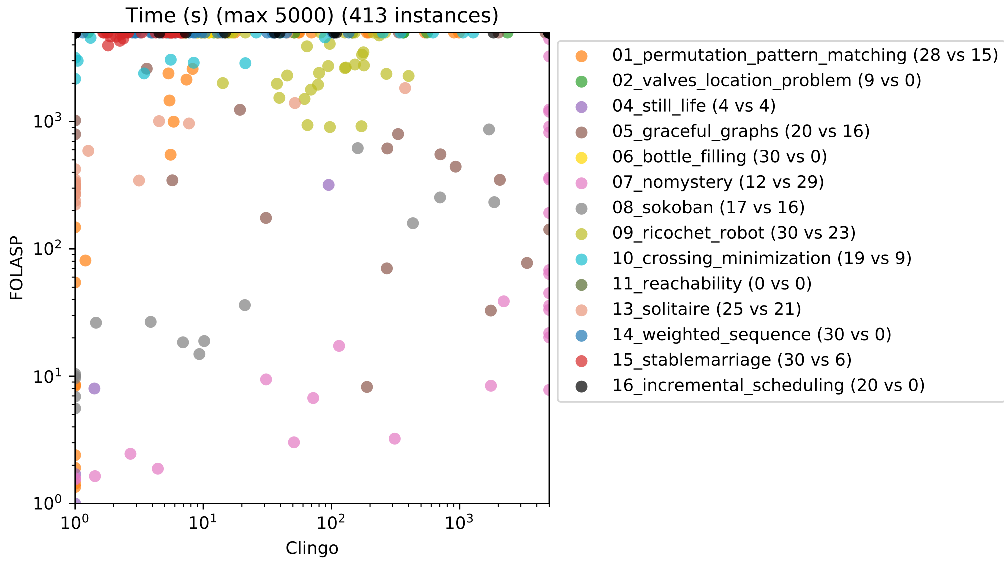

Fig. 2. Scatter plot comparing Clingo and FOLASP runtime performance. "(x vs y)" denotes that Clingo solved x instances within the family, and FOLASP y.

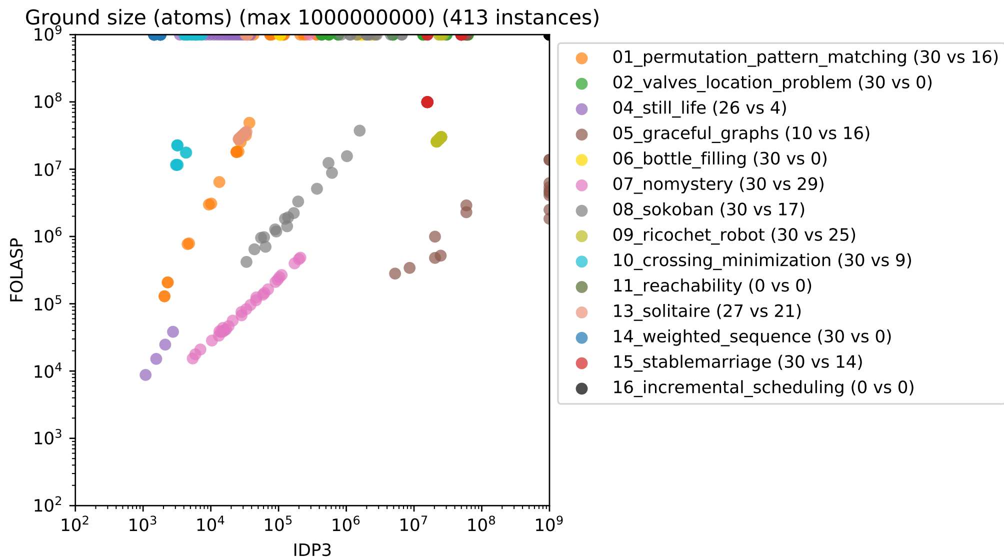

Fig. 3. Scatter plot comparing IDP and FOLASP ground size. "(x vs y)" denotes that IDP printed a ground size for x instances, and FOLASP for y.

size is measured as the number of atoms in the ground program, for both the ASP-solving approaches (FOLASP and Clingo) and IDP, with the caveat that a ground ASP program and a ground FO(·) specification may still be quite different. For instance, IDP retains non-Boolean *CP variables* (De Cat et al. 2013) in its grounding, though we did switch off *lazy grounding* (De Cat et al. 2015).

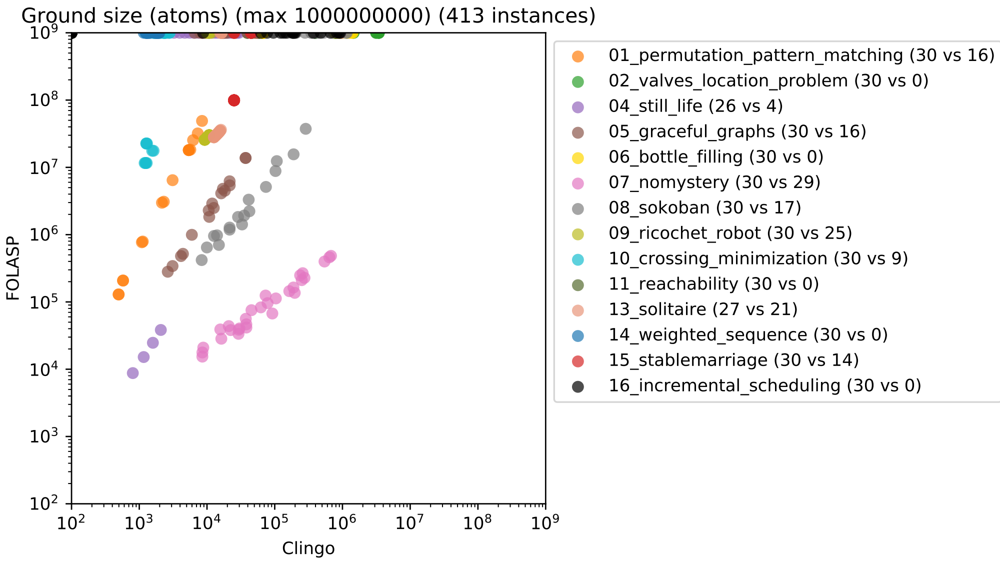

Fig. 4. Scatter plot comparing CLINGO and FOLASP ground size. "(x vs y)" denotes that CLINGO printed a ground size for x instances, and FOLASP for y.

For most benchmarks, the ground size for FOLASP is indeed significantly larger than the ground size for both IDP and CLINGO. Moreover, the ground size seems to correlate roughly with performance. For instance, FOLASP outperforms IDP and CLINGO on *graceful_graphs* and *nomystery*, respectively, and also has the smaller ground sizes on these benchmarks. On other benchmarks, such as *bottle_filling* or *weighted_sequence*, FOLASP actually hit the 64 GiB memory limit during grounding.

These observations appear to confirm our hypothesis that artifacts introduced by the translation, such as auxiliary predicates, cardinality aggregates and extra rules, are a main source of poor performance. Future work may focus on how to tweak the translation such that the ground size can be reduced.

6 Conclusion

To solve real-world problems using declarative methods, both a suitable modeling language and a suitable solver are needed. The Answer Set Programming community has converged on the ASP-Core-2 standard as a common modeling language. However, while such a common language is a great driver for technological progress, it is not necessarily well-suited for all applications.

The FO($\cdot$) language may provide an interesting alternative. It builds on classical first-order logic, which may make it easier to use for domain experts who are already familiar with FO, and which may make it easier to integrate with other FO-based languages. However, it is only supported by a few solvers, which restricts the applications for which FO($\cdot$) can be used in practice.

In this paper, we aim to provide more flexibility: by presenting a translation of FO($\cdot$) model expansion problems to ASP-Core-2, we both extend the range of solvers for FO($\cdot$) and enable the use of FO($\cdot$) as an alternative modeling language for these solvers. In this way, we stimulate technological progress in solver development and in the development of applications. We implemented our approach in the FOLASP tool, which, to the best of our knowledge, is the first tool to offer a full translation from FO($\cdot$) to ASP for both model expansion and optimization.

In our experimental evaluation, we used benchmarks from the ASP competition to verify that the results computed by FOLASP are indeed correct. We also compared the performance of running Clingo on the FOLASP translation of an FO(·) specification to two alternatives:

- directly running the IDP system on the FO(·) specification;
- running Clingo directly on a native ASP specification.

In general, our experiments confirmed what one would typically expect, namely that the best performance is obtained by running a specification that was native to a particular solver on that solver. However, the experiments also showed that, for a number of benchmarks, our translation-based approach is actually able to match or even, in rare cases, outperform the native approaches. This demonstrates the usefulness of our translation also from a computational perspective: a specification that performs poorly with one solver, may be more efficient when translated to the input language of another solver.

Our experiments also demonstrate that, in cases where the translation performs significantly worse than the native solutions, the grounding size often appears to play an important role. Future work will therefore focus on further optimising the translation to reduce the overhead it introduces.

In summary, the main contribution of our work is to provide increased flexibility, both in choice of specification language and in choice of solver. We believe that this will be useful to drive technological progress, to develop real-world applications using the best tools for the job, and to allow cross-fertilisation between different research groups.

References

Aavani, A. 2014. *Enfragmo: A System for Grounding Extended First-Order Logic to SAT*. PhD thesis, Faculty of Applied Sciences, Simon Fraser University, Vancouver, Canada.

Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L. K., Wallner, J. P., and Xiao, G. The fourth Answer Set Programming competition: Preliminary report. In Cabalar, P. and Son, T. C., editors, *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings* 2013, volume 8148 of *LNCS*, pp. 42–53. Springer.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. 2020. ASP-Core-2 input language format. *TPLP*, *20*, 2, 294–309.

Calimeri, F., Gebser, M., Maratea, M., and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artif. Intell.*, *231*, 151–181.

De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. 2016. Predicate logic as a modelling language: The IDP system. *CoRR*, *abs/1401.6312v2*.

De Cat, B., Bogaerts, B., Devriendt, J., and Denecker, M. Model expansion in the presence of function symbols using constraint programming. In *In ICTAI* 2013, pp. 1068–1075. IEEE.

De Cat, B., Denecker, M., Bruynooghe, M., and Stuckey, P. J. 2015. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, *52*, 235–286.

Denecker, M., Lierler, Y., Truszczynsky, M., and Vennekens, J. A Tarskian informal semantics for answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming* 2012, volume 17, pp. 227–289. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, *9*, 2, 14:1–14:52.

Enderton, H. B. Chapter two - first-order logic. In Enderton, H. B., editor, *A Mathematical*

Introduction to Logic (Second Edition) 2001, pp. 67 – 181. Academic Press, Boston, second edition edition.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. Theory solving made easy with Clingo 5. In Carro, M., King, A., De Vos, M., and Saeedloei, N., editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA* 2016, volume 52 of *OASIcs*, pp. 2:1–2:15. Schloss Dagstuhl.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2019. Multi-shot ASP solving with clingo. *TPLP*, *19*, 1, 27–82.

Gebser, M., Maratea, M., and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.*, *20*, 2, 176–204.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, *7*, 3, 499–562.

Marek, V. and Truszczyński, M. Stable models and an alternative logic programming paradigm. In Apt, K. R., Marek, V., Truszczyński, M., and Warren, D. S., editors, *The Logic Programming Paradigm: A 25-Year Perspective* 1999, pp. 375–398. Springer-Verlag.

Mariën, M., Gilis, D., and Denecker, M. On the relation between ID-Logic and answer set programming. In Alferes, J. J. and Leite, J. A., editors, *JELIA* 2004, volume 3229 of *LNCS*, pp. 108–120. Springer.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. Unfounded sets and well-founded semantics for general logic programs. In *PODS* 1988, pp. 221–230. ACM.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM*, *38*, 3, 619–649.