

An Optimization for Reasoning with Forest Logic Programs ^{*}

CRISTINA FEIER, STIJN HEYMANS

Knowledge-Based Systems Group, Institute of Information Systems
Vienna University of Technology
Favoritenstrasse 9-11, A-1040 Vienna, Austria
{feier, heyman}@kr.tuwien.ac.at

Abstract. Open Answer Set Programming (OASP) is an attractive framework for integrating ontologies and rules. In general OASP is undecidable. In previous work we provided a tableau-based algorithm for satisfiability checking w.r.t. forest logic programs, a decidable fragment of OASP, which has the forest model property. In this paper we introduce an optimized version of that algorithm achieved by means of a knowledge compilation technique. So-called unit completion structures, which are possible building blocks of a forest model, in the form of trees of depth 1, are computed in an initial step of the algorithm. Repeated computations are avoided by using these structures in a pattern-matching style when constructing a model. Furthermore we identify and discard redundant unit completion structures: a structure is redundant if there is another structure which can always replace the original structure in a forest model.

1 Introduction

Integrating Description Logics (DLs) with rules for the Semantic Web has received considerable attention with approaches such as *Description Logic Programs* [6], *DL-safe rules* [11], *DL+log* [12], *dl-programs* [1], *Description Logic Rules* [10], and Open Answer Set Programming (OASP) [9]. OASP is a formalism which combines attractive features from the Logic Programming (LP) and the DL world. The syntax and semantics of OASP build upon the syntax and semantics of Answer Set Programming (ASP) [5]: there is a rule-based syntax with a *negation as failure* operator which is interpreted via a stable model semantics, but unlike the LP setting, an open domain semantics, like it is common in the DL world, is employed. This allows for stating generic knowledge, without the need to mention actual constants.

Several decidable fragments of OASP were identified by syntactically restricting the shape of logic programs, while carefully safe-guarding enough expressiveness for integrating rule- and ontology-based knowledge. A notable fragment is that of *Forest Logic Programs (FoLPs)* [8] that are able to simulate reasoning in the DL *SHOQ*. FoLPs allow for the presence of only unary and binary predicates in rules which have a

^{*} This work is partially supported by the Austrian Science Fund (FWF) under the projects P20305 and P20840, and by the European Commission under the project OntoRule (IST-2009-231875).

tree-like structure. A sound and complete algorithm for satisfiability checking of unary predicates w.r.t. FoLPs has been presented in [3]. The algorithm exploits the forest model property of the fragment: if a unary predicate is satisfiable, then it is satisfied by a forest-shaped model, with the predicate checked to be satisfiable being in the label of the root of one of the trees composing the forest. It is essentially a tableau-based procedure which builds such a forest model in a top-down fashion.

In this paper we describe an optimization for reasoning with FoLPs in the form of a knowledge compilation technique. The technique consists in pre-computing all possible building blocks of the tableau, in the form of trees of depth 1, blocks which we call *unit completion structures*. The original algorithm is used for computing the unit completion structures. The revised algorithm matches and appends such building blocks until a termination condition is met, like blocking or reaching a certain depth in the tableau expansion. In general, not all unit completion structures have to be considered: inherent redundancy in a FoLP, like rules which are less general than others gives rise to redundancy among completion structures. A unit completion structure is redundant iff there is another simpler (less constrained) unit completion structure. The latter can replace the former in any forest model. We formalize this notion, making it possible to identify such redundant structures and discard them.

The paper is structured as follows: Section 2 contains preliminaries, like the OASP semantics and some notation, and Section 3 introduces the FoLP fragment. An overview of the original algorithm for reasoning with FoLPs is given in Section 4. The main results of the paper concerning the computation of non-redundant unit completion structures, and the revised algorithm, are presented in Section 5. Finally, Section 6 draws some conclusions and discusses future work.

2 Preliminaries

We recall the open answer set semantics [9]. *Constants* a, b, c, \dots , *variables* X, Y, \dots , *terms* s, t, \dots , and *atoms* $p(t_1, \dots, t_n)$ are as usual. A *literal* is an atom L or a negated atom *not* L . We allow for *inequality literals* of the form $s \neq t$, where s and t are terms. A literal that is not an inequality literal will be called a *regular literal*. For a set S of literals or (possibly negated) predicates, $S^+ = \{l \mid l \in S\}$ and $S^- = \{l \mid \text{not } l \in S\}$. For a set S of atoms, *not* $S = \{\text{not } a \mid a \in S\}$. For a set of (possibly negated) predicates S , $S(X) = \{a(X) \mid a \in S\}$ and $S(X, Y) = \{a(X, Y) \mid a \in S\}$. For a predicate p , $\pm p$ denotes p or *not* p , whereby multiple occurrences of $\pm p$ in the same context will refer to the same symbol (either p or *not* p).

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α is a finite set of regular literals and β is a finite set of literals. The set α is the *head* and represents a disjunction, while β is the *body* and represents a conjunction. If $\alpha = \emptyset$, the rule is called a *constraint*. A special type of rules with empty bodies, are so-called *free rules* which are rules of the form: $q(t_1, \dots, t_n) \vee \text{not } q(t_1, \dots, t_n) \leftarrow$, for terms t_1, \dots, t_n ; these kind of rules enable a choice for the inclusion of atoms in the open answer sets. We call a predicate q *free* if there is a $q(X_1, \dots, X_n) \vee \text{not } q(X_1, \dots, X_n) \leftarrow$, with variables X_1, \dots, X_n . Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program R , let $cts(R)$ be the constants in R , $vars(R)$ its variables, and $preds(R)$

its predicates with $upreds(R)$ the unary and $bpreds(R)$ the binary predicates. For every non-free predicate q and a program P , P_q is the set of rules of P that have q as a head predicate. A *universe* U for P is a non-empty countable superset of the constants in P : $cts(P) \subseteq U$. We call P_U the ground program obtained from P by substituting every variable in P by every element in U . Let \mathcal{B}_P (\mathcal{L}_P) be the set of regular atoms (literals) that can be formed from a ground program P .

An *interpretation* I of a ground P is a subset of \mathcal{B}_P . We write $I \models p(t_1, \dots, t_n)$ if $p(t_1, \dots, t_n) \in I$ and $I \models \text{not } p(t_1, \dots, t_n)$ if $I \not\models p(t_1, \dots, t_n)$. Also, for ground terms s, t , we write $I \models s \neq t$ if $s \neq t$. For a set of ground literals L , $I \models L$ if $I \models l$ for every $l \in L$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I , denoted $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$. A ground constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$.

For a positive ground program P , i.e., a program without *not*, an interpretation I of P is a *model* of P if I satisfies every rule in P ; it is an *answer set* of P if it is a subset minimal model of P . For ground programs P containing *not*, the *GL-reduct* [5] w.r.t. I is defined as P^I , where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P , $I \models \text{not } \beta^-$ and $I \models \alpha^-$. I is an *answer set* of a ground P if I is an answer set of P^I .

A program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding with an infinite universe. An *open interpretation* of a program P is a pair (U, M) where U is a universe for P and M is an interpretation of P_U . An *open answer set* of P is an open interpretation (U, M) of P with M an answer set of P_U . An n -ary predicate p in P is *satisfiable* if there is an open answer set (U, M) of P s. t. $p(x_1, \dots, x_n) \in M$, for some $x_1, \dots, x_n \in U$.

We introduce notation for trees which extend those in [13]. Let \cdot be a concatenation operator between sequences of constants or natural numbers. A *tree* T with root c (T_c), where c is a specially designated constant, has as nodes sequences of the form $c \cdot s$, where s is a (possibly empty) sequence of positive integers formed with the concatenation operator; for $x \cdot d \in T$, $d \in \mathbb{N}^*$, we have that $x \in T$. The set $A_T = \{(x, y) \mid x, y \in T, \exists n \in \mathbb{N}^* : y = x \cdot n\}$ is the set of arcs of a tree T . For $x, y \in T$, we say that $x <_T y$ iff x is a prefix of y and $x \neq y$.

A *forest* F is a set of trees $\{T_c \mid c \in C\}$, where C is a set of distinguished constants. We denote with $N_F = \cup_{T \in F} T$ and $A_F = \cup_{T \in F} A_T$ the set of nodes and the set of arcs of a forest F , respectively. Let $<_F$ be a strict partial order relationship on the set of nodes N_F of a forest F where $x <_F y$ iff $x <_T y$ for some tree T in F . An extended forest EF is a tuple (F, ES) where $F = \{T_c \mid c \in C\}$ is a forest and $ES \subseteq N_F \times C$. We denote by $N_{EF} = N_F$ the nodes of EF and by $A_{EF} = A_F \cup ES$ its arcs. So unlike a normal forest, an extended forest can have arcs from any of its nodes to any root of some the tree in the forest.

Finally, for a directed graph G , $paths_G$ is the set of pairs of nodes for which there exists a path in G from the first node in the pair to the second one.

3 Forest Logic Programs

Forest Logic Programs (FoLPs) [8] are logic programs with tree-shaped rules which allow for constants and for which satisfiability checking under the open answer set semantics is decidable.

Definition 1. A forest logic program (FoLP) is a program with only unary and binary predicates, and such that a rule is either a free rule $a(s) \vee \text{not } a(s) \leftarrow$ or $f(s, t) \vee \text{not } f(s, t) \leftarrow$, where s and t are terms such that if s and t are both variables, they are different, a unary rule

$$r : a(s) \leftarrow \beta(s), (\gamma_m(s, t_m), \delta_m(t_m))_{1 \leq m \leq k}, \psi \quad (1)$$

where s and t_m , $1 \leq m \leq k$, are terms (again, if both s and t_m are variables, they are different; similarly for t_i and t_j), where

- $\psi \subseteq \bigcup_{1 \leq i \neq j \leq k} \{t_i \neq t_j\}$ and $\{\neq\} \cap \gamma_m = \emptyset$ for $1 \leq m \leq k$,
- $\forall t_i \in \text{vars}(r) : \gamma_i^+ \neq \emptyset$, i.e., for variables t_i there is a positive atom that connects s and t_i ,

or a binary rule

$$f(s, t) \leftarrow \beta(s), \gamma(s, t), \delta(t) \quad (2)$$

with $\{\neq\} \cap \gamma = \emptyset$ and $\gamma^+ \neq \emptyset$ if t is a variable (s and t are different if both are variables), or a constraint $\leftarrow a(s)$ or $\leftarrow f(s, t)$ where s and t are different if both are variables).

The following program P is a FoLP which says that an individual is a special member of an organization (*smember*) if it has the support of another special member: rule r_1 , or if it has the support of two regular members of the organization (*rmember*): rule r_2 . The binary predicate *support* which describes the ‘has support’ relationship is free. Nobody can be at the same time both a special member or a regular member: constraint r_4 . Two particular regular members are a and b : facts r_5 and r_6 .

Example 1.

$$\begin{aligned} r_1 : & \quad \text{smember}(X) \leftarrow \text{support}(X, Y), \text{smember}(Y) \\ r_2 : & \quad \text{smember}(X) \leftarrow \text{support}(X, Y), \text{rmember}(Y), \\ & \quad \text{support}(X, Z), \text{rmember}(Z), Y \neq Z \\ r_3 : & \quad \text{support}(X, Y) \vee \text{not } \text{support}(X, Y) \leftarrow \\ r_4 : & \quad \leftarrow \text{smember}(X), \text{rmember}(X) \\ r_5 : & \quad \text{rmember}(a) \leftarrow \\ r_6 : & \quad \text{rmember}(b) \leftarrow \end{aligned}$$

As their name suggests FoLPs have the *forest model property*:

Definition 2. Let P be a program. A predicate $p \in \text{upreds}(P)$ is forest satisfiable w.r.t. P if there is an open answer set (U, M) of P and there is an extended forest $EF \equiv (\{T_\varepsilon\} \cup \{T_a \mid a \in \text{cts}(P)\}, ES)$, where ε is a constant, possibly one of the constants appearing in P , and a labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in \text{cts}(P)\} \cup A_{EF} \rightarrow 2^{\text{preds}(P)}$ s. t.

- $U = N_{EF}$, and
- $p \in \mathcal{L}(\varepsilon)$,
- $z \cdot i \in T \in EF$, $i > 0$, iff there is some $f(z, z \cdot i) \in M$, $z \in T$, and
- for $y \in T \in EF$, $q \in \text{upreds}(P)$, $f \in \text{bpreds}(P)$, we have that

- $q(y) \in M$ iff $q \in \mathcal{L}(y)$, and
- $f(y, u) \in M$ iff $(u = y \cdot i \vee u \in \text{cts}(P)) \wedge f \in \mathcal{L}(y, u)$.

We call such a (U, M) a forest model and a program P has the forest model property if the following property holds: if $p \in \text{upreds}(P)$ is satisfiable w.r.t. P then p is forest satisfiable w.r.t. P .

Consider the FoLP P introduced in Example 1. The unary predicate $smember$ is forest satisfiable w.r.t. P : $(\{a, b, x\}, \{rmember(a), rmember(b), support(x, a), support(x, b), smember(x)\})$ is a forest model in which $smember$ appears in the label of the (anonymous) root of one of the trees in the forest (see Figure 1). Note that in the ordinary LP setting, where one restricts the universe to the Herbrand universe, $smember$ is not satisfiable.

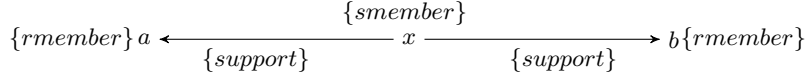


Fig. 1. A Forest Model for P

4 An Algorithm for Forest Logic Programs

In this section, we give an overview of the tableau algorithm for satisfiability checking for FoLPs introduced in [3]. For technical details we refer the reader to the original paper. We use as a running example the FoLP from Example 1. Constraints are not treated explicitly in the algorithm as they can be simulated using unary rules. As such, the constraint $r_i: \leftarrow smember(X), rmember(X)$ in Example 1 is replaced with $r'_i: co(X) \leftarrow not\ co(X), smember(X), rmember(X)$, with co a new predicate.

The basic data structure used by the algorithm to describe a forest model in construction is a so-called *completion structure*. Its main components are an extended forest EF , whose set of nodes constitutes the universe of the model, and a labeling function CT (*content*), which assigns to every node, resp. arc of EF , a set of possibly negated unary, resp. binary predicates. The presence of a predicate symbol $p/not\ p$ in the content of some node or arc x indicates the presence/absence of the atom $p(x)$ in the open answer set.

The presence (absence) of an atom in the open answer set is justified by imposing that the body of at least one ground rule which has the respective atom in the head is satisfied (no body of a rule which has the respective atom in the head is satisfied). In order to keep track which (possibly negated) predicate symbols in the content of some node or arc have already been expanded a so-called status function is introduced. Furthermore, in order to ensure that no atom in the partially constructed open answer set is circularly motivated, i.e. the atoms are well-supported [2], a graph G which keeps track of dependencies between atoms in the (partial) model is maintained.

Definition 3. An \mathcal{A}_1 -completion structure for a FoLP P^1 is a tuple $\langle EF, CT, ST, G \rangle$ where:

¹ We use the prefix \mathcal{A}_1 to denote completion structures computed using this original algorithm as opposed to completion structures computed using the optimised algorithm described in the next section for which we will use the prefix \mathcal{A}_2 .

- $EF = \langle F, ES \rangle$ is an extended forest,
- $CT : N_{EF} \cup A_{EF} \rightarrow 2^{preds(P) \cup not(preds(P))}$ is the ‘content’ function,
- $ST : \{(x, \pm q) \mid \pm q \in CT(x), x \in N_{EF} \cup A_{EF}\} \rightarrow \{exp, unexp\}$ is the ‘status’ function,
- $G = \langle V, A \rangle$ is a directed graph which has as vertices atoms in the answer set in construction: $V \subseteq \mathcal{B}_{P_{N_{EF}}}$.

An initial \mathcal{A}_1 -completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P is a completion structure $\langle EF, CT, ST, G \rangle$ with $EF = (F, \emptyset)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where ε is a constant, possibly in $cts(P)$, $T_x = \{x\}$, for $x \in \{\varepsilon\} \cup cts(P)$, $G = \langle V, \emptyset \rangle$, $V = \{p(\varepsilon)\}$, and $CT(\varepsilon) = \{p\}$, $ST(\varepsilon, p) = unexp$.

An extended forest is initialized with single-node trees with roots constants from P and, possibly, a new single-node tree with anonymous root. The forest model from Figure 1 has been evolved from an initial completion structure which has as ε , the root element where *smember* has to be satisfied, the anonymous individual, x . There are two other single-node trees: T_a and T_b . The predicate *smember* in the content of x is marked as unexpanded and G is a graph with a single vertex $smember(x)$.

$$\begin{array}{l}
 EF: \quad a\{ \qquad \qquad \qquad \{smember^u\} \qquad \qquad \qquad b\{ \\
 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad x \\
 V: \quad smember(x) \\
 A: \quad \emptyset
 \end{array}$$

An initial \mathcal{A}_1 -completion structure for checking the satisfiability of a unary predicate p w.r.t. a FoLP P is evolved by means of *expansion rules* to a complete clash-free structure that corresponds to a finite representation of an open answer set in case p is satisfiable w.r.t. P . *Applicability rules* govern the application of the expansion rules.

4.1 Expansion Rules

In the following, for a completion structure $\langle EF, ST, CT, G \rangle$, let $x \in N_{EF}$ and $(x, y) \in A_{EF}$ be the node, resp. arc, under consideration.

(i) **Expand unary positive.** For a unary positive (non-free) $p \in CT(x)$ s. t. $ST(x, p) = unexp$, choose a unary rule $r \in P_p$ for which s , the head term, unifies with x ; ground this rule by substituting s with x , and the successor terms t_m -s with successors of x in EF s. t. the inequalities in ψ are satisfied (if needed one can introduce new successors of x in EF , either as successors of x in T , where $x \in T$, or in the form of constants from P). We motivate the presence of $p(x)$ in the open answer set by enforcing its body to be satisfied by inserting appropriate (possibly negated) predicate symbols in the contents of nodes/arcs of the structure. The newly inserted predicate symbols are marked as unexpanded and G is updated, by adding arcs from $p(x)$ to every body atom.

In our example, *smember* is unexpanded in the initial completion structure. Rule r_2 is chosen to motivate the presence of $smember(x)$ in the open answer set. It is grounded by substituting X with x , and Y_1 and Y_2 with a and b , respectively: $smember(x) \leftarrow support(x, a), rmember(x, a), support(x, b), rmember(x, b)$. We enforce the body

of this ground rule to be true and obtain the following completion structure (note also that G has been updated):

$$EF: \quad \{rmember^u\}a \xleftarrow[\{support^u\}]{} x \xrightarrow[\{support^u\}]{} b\{rmember^u\}$$

$$V : smember(x), support(x, a), support(x, b), rmember(x, a), rmember(x, b)$$

$$A : smember(x) \rightarrow support(x, a), smember(x) \rightarrow support(x, b),$$

$$smember(x) \rightarrow rmember(x, a), smember(x) \rightarrow rmember(x, b)$$

All currently unexpanded predicates, i.e., $support$ in the content of arcs (x, a) and (x, b) , and $rmember$ in the content of nodes a and b , can be trivially expanded as $support$ is a free predicate and r_5 and r_6 are facts. However one still has to ensure that the structure constructed so far can be extended to an actual open answer set, i.e., it is consistent with the rest of the program. Next expansion rule takes care of this.

(ii) Choose a unary predicate. If all predicates in $CT(x)$ and in the contents of x 's outgoing edges are expanded and there are still unary predicates p which do not appear in $CT(x)$, pick such a p and inject either p or $not\ p$ in $CT(x)$. The intuition is that one has explore all unary/binary predicates at every node/arc as some predicate which is not reachable by dependency-directed expansion can render impossible the extension of the partially constructed model to a full model. Consider the simple case where there is a predicate p defined only by the rule: $p \leftarrow not\ p$ and $\pm p$ does not appear in the body of any other rule. The program is obviously inconsistent, but this cannot be detected without trying to prove that p is or is not in the open answer set.

In our example, one does not know whether co or $not\ co$ belongs to $CT(x)$. We choose to inject $not\ co$ in $CT(x)$ and mark it as unexpanded.

(iii) Expand unary negative. Justifying a negative unary predicate $not\ p \in CT(x)$ means refuting the body of every ground rule which defines $p(x)$, or in other words refuting at least a literal from the body of every ground rule which defines $p(x)$. For more technical details concerning this rule we refer the reader to [3].

In our example, the unexpanded predicate in $CT(x)$, $not\ co$, is defined by one rule, r'_4 , whose only possible grounding is $co(x) \leftarrow not\ co(x), smember(x), rmember(x)$. Refuting the body of this rule amounts to inserting $not\ rmember$ in $CT(x)$ ($smember$ and $not\ co$ are already part of the content of that node). At its turn, the presence of $not\ rmember$ in $CT(x)$ has to be motivated by using the expand unary negative rule, and the process goes on. Finally, we obtain a completion structure in which no expansion rule is further applicable and which represents exactly the forest model from Figure 1 ($smember$ and $rmember$ are abbreviated with sm and rm , respectively):

$$EF: \quad \{rm, not\ sm, not\ co\} a \xleftarrow[\{support\}]{} x \xrightarrow[\{support\}]{} b \{rm, not\ sm, not\ co\}$$

$$V : sm(x), support(x, a), support(x, b), rm(x, a), rm(x, b)$$

$$A : sm(x) \rightarrow support(x, a), sm(x) \rightarrow support(x, b), sm(x) \rightarrow rm(x, a), sm(x) \rightarrow rm(x, b)$$

Similarly to rules (i), (ii), and (iii) we define the expansion rules for binary predicates: (iv) *Expand binary positive*, (v) *Expand binary negative*, and (vi) *Choose binary*.

4.2 Applicability Rules

The applicability rules restrict the use of the expansion rules.

(vii) Saturation. A node $x \in N_{EF}$ is *saturated* if for all $p \in \text{upreds}(P)$, $p \in \text{CT}(x)$ or *not* $p \in \text{CT}(x)$, and no $\pm q \in \text{CT}(x)$ can be expanded with rules (i-iii), and for all $(x, y) \in A_{EF}$ and $p \in \text{bpreds}(P)$, $p \in \text{CT}(x, y)$ or *not* $p \in \text{CT}(x, y)$, and no $\pm f \in \text{CT}(x, y)$ can be expanded with (iv-vi). No expansions should be performed on a node from N_{EF} which does not belong to $\text{cts}(P)$ until its predecessor is saturated.

(viii) Blocking. A node $x \in N_{EF}$ is *blocked* if there is an ancestor y of x in F , $y <_F x$, $y \notin \text{cts}(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$ and the set $\text{paths}_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in \text{paths}_G \wedge q \text{ is not free}\}$ is empty. We call (y, x) a *blocking pair*. No expansions can be performed on a blocked node. One can notice that subset blocking is not enough for pruning the tableau expansion. Every atom in the open answer set has to be finitely motivated [7, Theorem 2]: in order to ensure that, one has to check that there is no dependency in G between an atom formed with the blocking node and an atom formed with the blocked node. The extra condition makes the blocking rule insufficient to ensure the termination of the algorithm. Next applicability rule ensures termination.

Example 2. Consider a restricted version of P from Example 1 which contains only rules r_1 , and r_3 . By checking satisfiability of smember w.r.t. the new program one obtains the following completion structure:

$$\begin{array}{l}
 EF: \quad V : \{\text{smember}(x), \text{smember}(y)\} \quad A : \{\text{smember}(x) \rightarrow \text{smember}(y)\} \\
 \begin{array}{c}
 x \{\text{smember}\} \\
 \downarrow \\
 \{\text{support}\} \\
 \downarrow \\
 y \{\text{smember}\}
 \end{array}
 \end{array}$$

While the contents of nodes x and y are identical, they do not form a blocking pair as there is an arc in G between $\text{smember}(x)$ and $\text{smember}(y)$: unfolding the structure (justifying y similarly as x) would lead to an infinite chain: $\text{smember}(x)$, $\text{smember}(y)$, $\text{smember}(z)$, \dots , in the atom dependency graph of the grounded program.

(ix) Redundancy. A node $x \in N_{EF}$ is *redundant* if it is saturated, it is not blocked, and there are k ancestors of x in F , $(y_i)_{1 \leq i \leq k}$, with $k = 2^p(2^{p^2} - 1) + 3$, and $p = |\text{upreds}(P)|$, s. t. $\text{CT}(x) = \text{CT}(y_i)$. In other words, a node is redundant if it is not blocked and it has k ancestors with content equal to its content. Any forest model of a FoLP P which satisfies p can be reduced to another forest model which satisfies p and has at most $k + 1$ nodes with equal content on any branch of a tree from the forest model, and furthermore the $(k + 1)$ st node, in case it exists, is blocked [3]. One can thus search for forest models only of the latter type. As such the detection of a redundant node constitutes a clash and stops the expansion process.

4.3 Termination, Soundness, Completeness, Complexity Results

An \mathcal{A}_1 -completion structure is *contradictory* if for some $x \in N_{EF}/A_{EF}$ and $p \in \text{upreds}(P)/\text{bpreds}(P)$, $\{p, \text{not } p\} \subseteq \text{CT}(x)$. An \mathcal{A}_1 -completion structure for a FoLP

P and a $p \in \text{upreds}(P)$ is *complete* if it is a result of applying the expansion rules to the initial completion structure for p and P , taking into account the applicability rules, s. t. no expansion rules can be further applied.

Also, a complete \mathcal{A}_1 -completion structure $CS = \langle EF, CT, ST, G \rangle$ is *\mathcal{A}_1 -clash-free* if: (1) CS is not contradictory (2) EF does not contain redundant nodes (3) G does not contain cycles (4) there is no $p \in \text{upreds}(P)/\text{bpreds}(P)$ and $x \in N_{EF}/A_{EF}$, x unblocked, s.t. $p \in CT(x)$, and $ST(x, p) = \text{unexp}$.

It has been shown that an initial \mathcal{A}_1 -completion structure for a unary predicate p and a FoLP P can always be expanded to a complete \mathcal{A}_1 -completion structure (*termination*), that, if p is satisfiable w.r.t. P , there is a complete clash-free \mathcal{A}_1 -completion structure (*soundness*), and, finally, that, if there is a complete clash-free \mathcal{A}_1 -completion structure, p is satisfiable w.r.t. P (*completeness*).

In the worst case the algorithm runs in nondeterministic double exponential time, and a complete completion structure has a double exponential number of nodes in the size of the program. The high complexity is mostly due to the fact that blocking is not enough to ensure termination, and that, in particular, anywhere blocking cannot be used as a termination technique. As already explained this peculiarity appears as a result of adopting a minimal model semantics.

5 Optimized Reasoning with FoLPs

This section presents a knowledge compilation technique for reasoning with FoLPs together with an algorithm which makes use of this pre-compiled knowledge. The main idea is to capture all possible local computations, which are typically performed over and over again in the process of saturating the content of a node, by pre-computing all possible completion structures of depth 1 using the original algorithm described in the previous section. In the new algorithm, saturating the content of a node reduces to picking up one of the pre-computed structures which satisfies the existing constraints regarding the content of that node and appending the structure to the completion in construction: such constraints are sets of unexpanded (possibly negated) predicates which are needed to motivate the presence/absence in the open answer set of atoms constructed with the current node and the node above them.

Picking up a certain unit completion structure to saturate a node can impose strictly more constraints on the resulted structure than picking another unit completion structure with the same root content. Such constraints refer to: (1) the contents of the successor (non-blocked) nodes in a unit completion structure; (2) the paths from an atom formed with the root node of a unit completion to an atom formed with a successor node of such a completion – the more paths there are the harder blocking becomes. We discard such structures which are strictly more constraining than others, as they can be seen as redundant building blocks for a model.

The rest of the section formalizes and exemplifies these notions.

5.1 Unit Completion Structures

As mentioned in the introduction of this section, the intention is to obtain all completion structures of depth 1 which can be used as building blocks in our algorithm. We call

such structures *unit completion structures*. The skeleton of such a structure, is a so-called *initial unit completion structure*. If they are to be used as building blocks in the algorithms, unit completion structures have to have as backbones trees of depth 1, and not forests. Hence, an initial unit completion structure is defined as a tree (unlike its counterpart notion from the previous section, initial completion structure, which is defined as a forest) with a single node, the root, which is either an anonymous constant or one of the constants already present in the program. The content of the root is empty.

Definition 4. An initial unit completion structure for a FoLP P is a completion structure $\langle EF, \text{CT}, \text{ST}, G \rangle$ with $EF = (F, ES)$, $F = \{T_\varepsilon\}$, where ε is a constant, possibly in $\text{cts}(P)$, $T_\varepsilon = \{\varepsilon\}$, $ES = \emptyset$, $G = \langle V, A \rangle$, $V = \emptyset$, $A = \emptyset$, and $\text{CT}(\varepsilon) = \emptyset$.

A unit completion structure captures a possible local computation: that is, it is obtained as an expansion of an initial unit completion structure, to a tree of depth 1.

Definition 5. A unit completion structure $\langle EF, \text{CT}, \text{ST}, G \rangle$ for a FoLP P , with $EF = (\{T_\varepsilon\}, ES)$, is an \mathcal{A}_1 -completion structure derived from an initial unit completion structure by application of the expansion rules (i)-(vi) described in Section 4.1, according to the applicability rules introduced in Section 4.2, which has been expanded such that ε is saturated and for all s such that $\varepsilon \cdot s \in T_\varepsilon$, and for all $\pm p \in \text{CT}(\varepsilon \cdot s)$, $\text{ST}(\pm p, \varepsilon \cdot s) = \text{unexp}$.²

Example 3. Consider the program Pr :

$$\begin{array}{ll}
r_1 : & p(X) \leftarrow \text{not } p(X) \\
r_2 : & p(X) \leftarrow f(X, Y), \text{not } q(Y) \\
r_3 : & p(X) \leftarrow f(X, Y), p(Y) \\
r_4 : & p(X) \leftarrow f(X, Y), \text{not } q(Y), p(Y) \\
r_5 : & q(X) \leftarrow f(X, Y), \text{not } p(Y) \\
r_6 : & f(X, Y) \vee \text{not } f(X, Y) \leftarrow
\end{array}$$

Figure 2 depicts three unit completion structures for Pr . They all have the same content for the root node: $\{p, \text{not } q\}$. The presence of p in the content of the root node has been motivated in the first structure by means of rule r_4 , in the second structure by means of rule r_3 , and in the third structure by means of rule r_2 . The different ways to motivate p lead to different sets of arcs in the dependency graphs belonging to each structure. On the other hand, to motivate that $\text{not } q$ is in the content of the root node, in each case it was shown that the body of r_5 grounded such that X is instantiated as the root node and Y as the successor node is not satisfied, or more concretely the presence of p in the content of the successor node was enforced in each case ($\text{not } f$ could not be used to invalidate the triggering of the rule as f was already present in the content of the arc from the root node to the successor node in each case).

One can notice that while the content of the successor node is included in the content of the root node in each of the cases, only for UC_3 , the two nodes form a blocking pair as $\text{paths}_{G_3}(c, c1) = \emptyset$.

² The status function is relevant only in the definition/construction of a unit completion structure, but not in the context of using such structures. As such, we will denote a unit complete structure in the following as a triple $\langle EF, \text{CT}, G \rangle$.

$UC_1 :$ $a\{p, not\ q\}$ \downarrow $\{f\}$ \downarrow $a1\{p, not\ q\}$	$UC_2 :$ $b\{p, not\ q\}$ \downarrow $\{f\}$ \downarrow $b1\{p\}$	$UC_3 :$ $c\{p, not\ q\}$ \downarrow $\{f\}$ \downarrow $c1\{p, not\ q\}$
$G_1 = (V_1, A_1)$	$G_2 = (V_2, A_2)$	$G_3 = (V_3, A_3)$
$V_1 : p(a), p(a1), f(a, a1)$	$V_2 : p(b), p(b1), f(d, d1)$	$V_3 : p(c), p(c1), f(c, c1)$
$A_1 : p(a) \rightarrow f(a, a1),$ $p(a) \rightarrow p(a1)$	$A_2 : p(b) \rightarrow f(b, b1),$ $p(b) \rightarrow p(b1)$	$A_3 : p(c) \rightarrow f(c, c1)$

Fig. 2. Three unit completion structures for $Pr: UC_1, UC_2,$ and UC_3 .

Definition 6. A unit completion structure is final iff all its successor nodes are blocked, or they have empty contents.

Proposition 1. A final unit completion structure is a complete clash-free \mathcal{A}_1 -completion structure.

In our example UC_3 is a final unit completion structure, and thus also a complete clash-free \mathcal{A}_1 -completion structure.

Proposition 2. There is a deterministic procedure which computes all unit completion structures for a FoLP P in the worst-case scenario in exponential time in the size of P .

Proof Sketch. We consider the transformation of the non-deterministic algorithm described in Definition 5 into a deterministic procedure. There are at most 2^p different values for the content of a saturated node, in this case for the content of the root of a unit completion structure, where $p = |upreds(P)|$. Justifying the presence of a predicate symbol p in the content of a node takes in the worst case polynomial time (choosing a possible grounding with successor nodes for some rule $r \in P_p$), but there is an exponential number of choices to do this (an exponential number of possible groundings for every rule). Justifying the presence of a negated predicate symbol $not\ p$ in the content of a node takes in the worst case exponential time (all possible groundings of every rule $r \in P_p$ have to be considered), while at every step of the computation there is a polynomial number of choices (for the ground rule in consideration, choosing a literal in its body to be refuted). Overall, such a deterministic procedure runs in exponential time in the worst case scenario. \square

5.2 Redundant Unit Completion Structures

As seen in Example 3, there are unit completion structures with roots with equal content, but possibly different topologies, contents of the successor nodes and/or possibly

different dependency graphs. As discussed in the introduction to this section it is worthwhile to identify structures which are strictly more constraining than others, in the sense that they impose more constraints on the content of the successor nodes of the structure and introduce more paths in the dependency graph as they can be discarded. The following definition singles out such redundant structures.

Definition 7. A unit completion structure $UC_1 = \langle EF_1, CT_1, G_1 \rangle$, with $EF_1 = (\{T_{\varepsilon_1}\}, ES_1)$, is said to be redundant iff there is another unit completion structure $UC_2 = \langle EF_2, CT_2, G_2 \rangle$, with $EF_2 = (\{T_{\varepsilon_2}\}, ES_2)$ s. t.:

- if $\varepsilon_2 \in cts(P)$, then $\varepsilon_2 = \varepsilon_1$;
- $CT(\varepsilon_1) = CT(\varepsilon_2)$;
- if $\varepsilon_2 \cdot s_1, \dots, \varepsilon_2 \cdot s_l$ are the non-blocked successors of ε_2 , there exist l distinct successors $\varepsilon_1 \cdot t_1, \dots, \varepsilon_1 \cdot t_l$ of ε_1 such that:
 - $CT(\varepsilon_2 \cdot s_i) \subseteq CT(\varepsilon_1 \cdot t_i)$, for every $1 \leq i \leq l$, and
 - $paths_{G_2}(\varepsilon_2, \varepsilon_2 \cdot s_i) \subseteq paths_{G_1}(\varepsilon_1, \varepsilon_1 \cdot t_i)$, for every $1 \leq i \leq l$,
with at least one inclusion being strict.

Considering the previous example, one can see that UC_1 , and UC_2 are redundant structures, while UC_3 is not, as UC_1 is more constraining than UC_2 , and UC_2 at its turn is more constraining than UC_3 .

Proposition 3. Computing the set of non-redundant unit completion structures for a FoLP P can be performed in the worst case in exponential time in the size of P .

Proof Sketch. The result follows from the fact that there is an exponential number of unit completion structures for a FoLP P in the worst case scenario. \square

5.3 Reasoning with FoLPs Using Unit Completion Structures

We define a new algorithm which uses the set of pre-computed non-redundant completion structures. We call this algorithm \mathcal{A}_2 . As in the case of the previous algorithm, \mathcal{A}_2 starts with an initial \mathcal{A}_2 -completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P and expands this to a so-called \mathcal{A}_2 -completion structure.

An \mathcal{A}_2 -completion structure $\langle EF, CT, ST, G \rangle$ is defined similarly as an \mathcal{A}_1 -completion structure, but the *status* function has a different domain, the set of nodes of the forest: $ST : N_{EF} \rightarrow \{exp, unexp\}$.

An initial \mathcal{A}_2 -completion structure for a unary predicate p and FoLP P is defined similarly as an initial \mathcal{A}_1 -completion structure for p and P , the only difference being that every node in the extended forest is marked as unexpanded: $ST(x) = unexp$, for every $x \in N_{EF}$.

The difference in the definition of an \mathcal{A}_2 -completion structure compared to its \mathcal{A}_2 homonym is due to the fact that in this scenario nodes are expanded by matching their content with existent unit completion structures, and not predicates in the content of nodes, like in the case of \mathcal{A}_1 . We make explicit the notion of matching the content of a node with a unit completion structure by introducing a notion of *local satisfiability*:

Definition 8. A unit completion structure UC for a FoLP P , $\langle EF, CT, G \rangle$, with $EF = (\{T_\varepsilon\}, ES)$, locally satisfies a (possibly negated) unary predicate p iff $p \in CT(\varepsilon)$. Similarly, UC locally satisfies a set S of (possibly) negated unary predicates iff $S \subseteq CT(\varepsilon)$.

All three unit completions in Figure 2 locally satisfy the set $\{a, \text{not } b\}$. It is easy to observe that if a unary predicate p is not locally satisfied by any unit completion structure UC for a FoLP P (or equivalently $\text{not } p$ is locally satisfied by every unit completion structure), p is unsatisfiable w.r.t. P . However, local satisfiability of a unary predicate p in every unit completion structure for a FoLP P does not guarantee 'global' satisfiability of p w.r.t. P (as in the case of the program in Example 2 whose only unit completion structure was the one depicted in that example).

In the process of building an \mathcal{A}_2 -completion structure $CS = \langle EF, CT, ST, G \rangle$, with $G = (V, A)$, for a FoLP P by using unit completion structures as building blocks an operation commonly appears: the expansion of a node $x \in N_{EF}$ by addition of a unit completion structure $UC = \langle EF', CT', G' \rangle$, with $EF' = (\{T_\varepsilon\}, ES')$ and $G' = (V', A')$, which locally satisfies $CT(x)$, at x , given that its root matches with x ³. We call this operation with $expand_{CS}(x, UC)$. Formally, its application updates CS as follows:

- $ST(x) = exp$,
- $N_{EF} = N_{EF} \cup \{x \cdot s \mid \varepsilon \cdot s \in T_\varepsilon\}$,
- $A_{EF} = A_{EF} \cup \{(x, x \cdot s) \mid (\varepsilon, \varepsilon \cdot s) \in A_{EF'}\}$,
- $CT(x) = CT(\varepsilon)$. For all s such that $\varepsilon \cdot s \in T_\varepsilon$, $CT(x \cdot s) = CT(\varepsilon \cdot s)$,
- $V = V \cup \{p(x) \mid p \in CT(\varepsilon)\} \cup \{p(x \cdot s) \mid p \in CT(\varepsilon \cdot s)\}$,
- $A = A \cup \{(p(\bar{z}), q(\bar{y})) \mid (p(z), q(y)) \in A'\}$, where $\bar{z} = x$, and $\bar{z} \cdot \bar{s} = x \cdot s$.

The algorithm has a new rule compared with the original algorithm which we call *Match*. This rule is meant to replace the expansion rules (i)-(vi) and the applicability rule (vii) from the original algorithm.

Match. For a node $x \in N_{EF}$: if $ST(x) = unexp$ non-deterministically choose a non-redundant unit completion structure UC with root matching x which satisfies $CT(x)$ and perform $expand_{CS}(x, UC)$.

In this variant of the algorithm we still employ rules (viii) *Blocking* and (ix) *Redundancy* described in Section 4.

Definition 9. A complete \mathcal{A}_2 -completion structure for a FoLP P and a $p \in upreds(P)$, is an \mathcal{A}_2 -completion structure that results from applying the rule *Match* to an initial \mathcal{A}_2 -completion structure for p and P , taking into account the applicability rules (viii) and (ix), s. t. no other rules can be further applied.

The local clash conditions regarding contradictory structures or structures which have cycles in the dependency graph G are no longer relevant:

³ An anonymous individual behaves like a variable: it matches with any term, while a constant matches only with itself; thus, unit completion structures with roots constants can only be used as initial building blocks for the trees with non-anonymous roots in the structure.

Definition 10. A complete \mathcal{A}_2 -completion structure $CS = \langle EF, CT, ST, G \rangle$ is clash-free if (1) EF does not contain redundant nodes (2) there is no node $x \in N_{EF}$, x unblocked, s.t. $st(x) = unexp$.

The termination of the algorithm follows immediately from the usage of the blocking and of the redundancy rule:

Proposition 4. An initial \mathcal{A}_2 -completion structure for a unary predicate p and a FoLP P can always be expanded to a complete \mathcal{A}_2 -completion structure.

The algorithm is sound and complete:

Proposition 5. A unary predicate p is satisfiable w.r.t. a FoLP P iff there is a complete clash-free \mathcal{A}_2 -completion structure.

Proof Sketch. The soundness of \mathcal{A}_2 follows from the soundness of \mathcal{A}_1 : any completion structure computed using \mathcal{A}_2 could have actually been computed using \mathcal{A}_1 by replacing every usage of the *Match* rule with the corresponding rule application sequence used by \mathcal{A}_1 to derive the unit completion structure which is currently appended to the structure.

The completeness of \mathcal{A}_2 derives from the completeness of \mathcal{A}_1 : any clash-free complete \mathcal{A}_1 -completion structure can actually be seen as a complete clash-free \mathcal{A}_2 -completion structure. It is essential here that the discarded unit completion structures were strictly more constraining than some other (preserved) unit completion structures. Whenever the expansion of a node in the complete clash-free \mathcal{A}_1 -completion structure has been performed by a sequence of rules captured by a redundant unit completion structure, it is possible to construct a complete clash-free \mathcal{A}_2 -completion structure by using the simpler non-redundant unit completion structure instead. \square

As we still employ the redundancy rule in this version of the algorithm, a complete \mathcal{A}_2 -completion structure has in the worst case a double exponential number of nodes in the size of the program. As such:

Proposition 6. \mathcal{A}_2 runs in the worst-case in nondeterministic double exponential time.

6 Discussion and Outlook

Our optimized algorithm runs in the worst case in non-deterministic double exponential time: this is not a surprise as the scope of the technique introduced here is saving time by avoiding redundant local computations. The worst-case running complexity of the algorithm depends on the depth of the trees which have to be explored in order to ensure completeness of the algorithm and on the fact that anywhere blocking is not feasible. Even with classical subset blocking one has to explore an exponential number of nodes across a branch in order for the algorithm to terminate. Thus, the only factor which would improve the worst-case performance is finding a termination condition which considers nodes in different branches. At the moment this seems highly unattainable.

The next step of our work is the evaluation of the new algorithm. We expect it will perform considerably better than the original algorithm in returning positive answers to satisfiability checking queries, while it might still take considerable time in the

cases where a predicate is not satisfiable. Especially problematic are cases like the one described in Example 2 where there exists a unit completion structure which locally satisfies the predicate checked to be satisfiable, but the predicate is actually unsatisfiable. An obvious strategy for implementation is to establish a limit on the depth of the explored structures: in practice it is highly improbable that if there exists a solution, it can be found only in an open answer set of a considerable size: actually, it is quite hard to come up with examples of such situations.

A knowledge compilation technique for reasoning with the Description Logic \mathcal{ALC} is described in [4]: there, TBoxes are pre-compiled in an expensive initial computation which then allows polynomial time satisfiability checking. We note that this initial step goes beyond compiling 'local' knowledge, as it follows role restrictions up to any depth. If we would want to pre-compute also non-local computations, we would basically generate all completion structures for a certain program, and then satisfiability checking becomes trivial.

References

1. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
2. F. Fages. A new fix point semantics for generalized logic programs compared with the wellfounded and the stable model semantics. *New Generation Computing*, 9(4), 1991.
3. C. Feier and S. Heymans. Hybrid Reasoning with Forest Logic Programs. In *Proc. of 6th European Semantic Web Conference*, volume 5554, pages 338–352. Springer, 2009.
4. U. Furbach, H. Günther, and C. Obermaier. A Knowledge Compilation Technique for ALC TBoxes. In *Proc. of the Twenty-Second International Florida Artificial Intelligence Research Society Conference, May 19-21, 2009, Sanibel Island, Florida, USA, 2009*.
5. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, 1988.
6. B. N. Groszof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. of the World Wide Web Conference (WWW)*, pages 48–57. ACM, 2003.
7. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Conceptual Logic Programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)*, 47(1-2):103–137, 2006.
8. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open Answer Set Programming for the Semantic Web. *J. of Applied Logic*, 5(1):144–169, 2007.
9. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *Transactions on Computational Logic*, 9(4):1–53, August 2008.
10. M. Krötzsch, S. Rudolph, and P. Hitzler. Description Logic Rules. In *Proc. 18th European Conf. on Artificial Intelligence (ECAI-08)*, pages 80–84. IOS Press, 2008.
11. B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics*, 3(1):41–60, 2005.
12. R. Rosati. DL+log: Tight Integration of Description Logics and Disjunctive Datalog. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 68–78, 2006.
13. M. Y. Vardi. Reasoning about the Past with Two-way Automata. In *Proc. 25th Int. Colloquium on Automata, Languages and Programming*, pages 628–641. Springer, 1998.