Worst-Case Optimal Reasoning with Forest Logic Programs*

Cristina Feier
Institute of Information Systems
Vienna University of Technology
Favoritenstrasse 11, A-1040 Vienna, Austria
feier@kr.tuwien.ac.at

Abstract

The paper introduces a worst-case optimal tableau algorithm for reasoning with Forest Logic Programs, a decidable fragment of Open Answer Set Programming. FoLPs are a useful device for tight integration of the Description Logic and the Logic Programming worlds: reasoning with the DL $\mathcal{SHOQ}$ can be simulated within the fragment. The algorithm reuses a knowledge compilation technique previously introduced, but improves on previous results by decreasing the worst-case running time with one exponential level. The decrease in complexity is due to the usage in conjunction of a new redundancy and of a new caching rule.

There has been lots of work recently combining logical rules and ontologies: starting with approaches like *Description Logic Programs* (Grosof et al. 2003), *DL-safe rules* (Motik, Sattler, and Studer 2005), $\mathcal{DL}$+*log* (Rosati 2006), *dl-programs* (Eiter et al. 2008), *Description Logic Rules* (Krötzsch, Rudolph, and Hitzler 2008), and considering more recent approaches like Datalog± (Calì, Gottlob, and Lukasiewicz) and its fragments.

Most approaches impose safety restrictions on the rule component: any deducible fact has as arguments elements of the domain of the discourse, the Herbrand universe.One rule-based formalism which allows to express facts about unknown individuals is Forest Logic Programs (FoLPs) (Heymans, Van Nieuwenborgh, and Vermeir 2007). FoLPs is a decidable fragment of Open Answer Set Programming (OASP) (Heymans, Van Nieuwenborgh, and Vermeir 2008), an extension of Answer Set Programming (ASP) with open domains. The syntax of OASP is identical to ASP syntax, while its semantics (and implicitly, of FoLPs) is a hybrid between the standard ASP semantics and the classical FOL semantics: interpretations are relative to 'open' domains, i.e. non-empty supersets of the set of constants in the program. Syntactically, FoLPs is a 2-variable fragment of ASP which has the *forest model property*: every satisfiable unary predicate is satisfied by a *forest model*. One can simulate within FoLPs reasoning with the expressive DL $\mathcal{SHOQ}$: as such, FoLPs serve as an integrative device for *f-hybrid knowledge bases*, a tightly-coupled combination of FoLPs themselves and $\mathcal{SHOQ}$ KBs (Feier and Heymans 2011).

The contribution of this paper is a worst-case optimal algorithm for checking satisfiability of unary predicates w.r.t. FoLPs. The algorithm runs in the worst-case in exponential time, and it establishes a tight complexity characterization for FoLPs. It improves on the worst-case running time of previous approaches for reasoning with FoLPs with one exponential level. Like previous approaches, it is tableau-based: it constructs a so-called *completion structure*, which eventually can be unraveled to a model. To this purpose, it reuses a knowledge compilation technique introduced in (Feier and Heymans 2010). However, the new algorithm employs different termination techniques: a new version of the so-called *redundancy* rule which stops the unsuccessful expansion of a branch in the tableau is introduced along with a new *caching rule* for reusing computation across branches.

Section 1 contains some preliminaries, including the OASP semantics, while Section 2 introduces FoLPs. Section 3 describes the notion of completion structure and how this can be evolved using the knowledge compilation technique previously introduced. The termination rules of the new algorithm are described in Section 4. Section 5 describes when the expansion of the completion structure is complete, while section 6 shows that the algorithm terminates, is sound, and complete. The formal complexity analysis can be found in section 7. Finally, section 8 draws some conclusions and presents some related work.

1 Preliminaries

We recall the open answer set semantics (Heymans, Van Nieuwenborgh, and Vermeir 2008). *Constants* $a, b, c, \ldots$, *variables* $X, Y, \ldots$, *terms* $s, t, \ldots$, and *atoms* $p(t_1, \ldots, t_n)$ 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 is a *regular literal*.

For a set S of literals or (possibly negated) predicates, $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid not\ a \in S\}$. For a set S of atoms, $not\ S = \{not\ a \mid a \in S\}$. For a set of (possibly negated) unary predicates S: $S(X) = \{a(X) \mid a \in S\}$, and for a set of (possibly negated) binary predicates S: $S(X, Y) = \{a(X, Y) \mid a \in S\}$.

*This work is partially supported by the Austrian Science Fund (FWF) under the project P20840, and by the European Commission under the project OntoRule (IST2009231875).

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. Rules can also be named, as in $r : \alpha \leftarrow \beta$, where r is the name of the rule. 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. A *universe* U for P is a non-empty countable superset of the constants in P: $cts(P) \subseteq U$. P_U is obtained from P by substituting every variable in P by every element in U. Let $\mathcal{B}_P$ be the set of regular atoms 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 $I \models p(t_1, \ldots, t_n)$ if $p(t_1, \ldots, t_n) \in I$ and $I \models not\ p(t_1, \ldots, t_n)$ if $I \not\models p(t_1, \ldots, t_n)$. 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* (Gelfond and Lifschitz 1988) w.r.t. I is defined as P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P, $I \models not\ \beta^-$ 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 a finite set of rules; infinite programs may appear by 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. $p(x_1, \ldots, x_n) \in M$, for some $x_1, \ldots, x_n \in U$.

We introduce notation for trees which extend those in (Vardi 1998). Let $\cdot$ be a concatenation operator between sequences of constants or natural numbers. A *tree* T with root c (also denoted as T_c), where c is a specially designated constant, is a set of nodes, where each node is a sequence of the form $c \cdot s$, where s is a (possibly empty) sequence of positive integers formed with the help of the concatenation operator (we denote the set of all such sequences with $\langle \mathbb{N}^* \rangle$, where $\mathbb{N}^*$ is the set of positive integers); for $x \cdot d \in T$, $d \in \mathbb{N}^*$, we must 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 there exists $s \in \langle \mathbb{N}^* \rangle$ s.t. $y = x \cdot s$ and $x \neq y$. For nodes $x, y \in T$, let $common_T(x, y)$ be the node z s. t. $z <_T x$, $z <_T y$, and there is no $z' \in T$ s. t. $z' >_T z$, $z' <_T x$, and $z' <_T y$. A node $x \in T$ is said to be to the right of a node $y \in T$ and denoted with $right_T(x, y)$ iff there exists a node $z \in T$, $i, j \in \mathbb{N}^*$, and $s_1, s_2 \in \langle \mathbb{N}^* \rangle$, s. t. $x = z \cdot i \cdot s_1$, $y = z \cdot j \cdot s_2$, and $i > j$. The subtree of T_c at y, denoted $T_c[y]$, is the set $\{x \mid x \in T_c, x = y \cdot s, s \in \langle \mathbb{N}^* \rangle\}$.

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. 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.

Finally, a directed graph G is defined as usual by its sets of nodes V and arcs A. $paths_G$ denotes the set of paths in G, where each path is a tuple of nodes from V: $paths_G = \{(x_1, \ldots, x_n) \mid ((x_i, x_{i+1}) \in A)_{1 \leq i < n}\}$.

2 Forest Logic Programs

Definition 2.1. *A* FoLP *is a program with only unary and binary predicates, and s. t. a rule is either:*

- *a* free rule*: $a(s) \vee not\ a(s) \leftarrow$, or $f(s,t) \vee not\ f(s,t) \leftarrow$, where s and t are terms;*
- *a* unary rule*: $a(s) \leftarrow \beta(s), (\gamma_m(s, t_m), \delta_m(t_m))_{1 \leq m \leq k}, \psi$, with $\psi \subseteq \bigcup_{1 \leq i \neq j \leq k} \{t_i \neq t_j\}$ and $k \in \mathbb{N}$, or*
  a binary rule*: $f(s,t) \leftarrow \beta(s), \gamma(s,t), \delta(t)$, where: $a \in upreds(P)$ and $f \in bpreds(P)$, s, t, and $(t_m)_{1 \leq m \leq k}$ are terms, β, δ, $(\delta_m)_{1 \leq m \leq k} \subseteq upreds(P) \cup not\ (upreds(P))$ (sets of (possibly negated) unary predicates), γ,$(\gamma_m)_{1 \leq m \leq k} \subseteq bpreds(P) \cup not\ (bpreds(P))$ (sets of (possibly negated) binary predicates), and*
  1. *inequality does not appear in any γ: $\{\neq\} \cap \gamma_m = \emptyset$, for $1 \leq m \leq k$, and $\{\neq\} \cap \gamma = \emptyset$;*
  2. *there is a positive atom that connects the head term s with any successor term which is a variable: $\gamma_m^+ \neq \emptyset$, if t_m is a variable, for $1 \leq m \leq k$, and $\gamma^+ \neq \emptyset$, if t is a variable;*
- *a constraint: $\leftarrow a(s)$ or $\leftarrow f(s,t)$, with s and t terms.*

In every rule, all terms which are variables are distinct.

A *forest model* of a FoLP is a model whose universe can be seen as the set of nodes of a forest, with every node and arc of the forest being labeled with unary and binary predicates, resp.: an atom is in the model iff it can be formed using the unary/binary predicate in the label of some node/arc and the respective node/arc as argument(s).

Definition 2.2. *Let P be a program. A unary predicate p is* forest satisfiable *w.r.t. P iff there is an open answer set (U, M) of P and an extended forest $EF \equiv (\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES)$, with ε a constant, possibly one of the constants appearing in P,and a labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \rightarrow 2^{preds(P)}$ s. t.:*

- *$p \in \mathcal{L}(\varepsilon)$,*
- *$L(x) \in 2^{upreds(P)} / 2^{bpreds(P)}$ for $x \in N_{EF} / A_{EF}$,*
- *$U = N_{EF}$, $M = \{\mathcal{L}(x)(x) \mid x \in N_{EF} \cup A_{EF}\}$, and*
- *for every $(z, z \cdot i) \in A_{EF}$: $\mathcal{L}(z, z \cdot i)^+ \neq \emptyset$.*

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

Proposition 2.3 ((Heymans, Van Nieuwenborgh, and Vermeir 2007)). *FoLPs have the forest model property.*

3 Tableau Reasoning with FoLPs

The algorithm we present in this paper ($\mathcal{A}_3$) builds on previous algorithms introduced in (Feier and Heymans 2009) ($\mathcal{A}_1$) and (Feier and Heymans 2010) ($\mathcal{A}_2$). All three algorithms for checking satisfiability of a unary predicate p w.r.t. a FoLP P construct a forest model of P which satisfies p by evolving a data structure, whose main element is an extended forest EF as in Definition 2.1. This data structure is called *completion structure*. The set of nodes of EF is the universe of the model, and every node and arc of EF is labeled using a 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 completion is evolved by progressively justifying the presence/absence of certain atoms in the model. To this purpose, $\mathcal{A}_1$ introduced so-called expansion rules which justify the presence/absence in the model of one atom at a time. They enforce that the body of a ground rule which has the atom in the head is true (in case the atom is in the model) or that the bodies of all ground rules which have the atom in the head are false (in case the atom is not in the model). Rules which guess the presence/absence of atoms in the model are also part of the algorithm. $\mathcal{A}_2$ is an optimization of $\mathcal{A}_1$ consisting in a knowledge compilation technique: all possible building blocks of a model in the form of trees of depth 1 are pre-computed using $\mathcal{A}_1$ and the completion is evolved by matching and appending these blocks using similar conditions for termination as $\mathcal{A}_1$. Such a building block is called *unit completion structure* (UCS). The current algorithm reuses this technique. In this setting, a completion structure contains a 'status' function ST which assigns one the values exp or $unexp$ to nodes of the forest, depending whether their content has been justified or not.

Finally, the model has to be well-supported (Fages 1991): no atom in the model may depend on itself and there should be no infinite chain of dependencies between atoms. To check this property, a graph G which keeps track of dependencies between atoms in the model is also maintained.

Definition 3.1. *A completion structure for a FoLP P is a tuple $\langle EF, \text{CT}, \text{ST}, G\rangle$, with EF an extended forest,* $\text{CT} : N_{EF} \cup A_{EF} \rightarrow 2^{preds(P) \cup not\ (preds(P))}$ *the 'content' function,* $\text{ST} : N_{EF} \rightarrow \{exp, unexp\}$ *the 'status' function, and $G = \langle V, A\rangle$ a directed graph with $V \subseteq \mathcal{B}_{P_{N_{EF}}}$.*

The algorithm starts with an *initial completion structure for checking satisfiability of p w.r.t. P*: it enforces that all constants in P are part of the initial universe of the constructed model and that p is satisfied by some element ε of the universe (either one of $cts(P)$ or an anonymous individual) by asserting p to be part of $\text{CT}(\varepsilon)$.

Definition 3.2. *An* initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P *is a completion structure $\langle EF, \text{CT}, \text{ST}, G\rangle$, where: $EF = (F, \emptyset)$; $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, with ε a constant, possibly in $cts(P)$; $T_x = \{x\}$, for $x \in \{\varepsilon\} \cup cts(P)$;* $\text{ST}(x) = unexp$, *for $x \in \{\varepsilon\} \cup cts(P)$; $G = \langle V, \emptyset\rangle$, with $V = \{p(\varepsilon)\}$, and* $\text{CT}(\varepsilon) = \{p\}$.

As previously mentioned, a completion structure is evolved by considering some unexpanded node in the extended forest and trying to justify all constraints associated with that node by finding a UCS which 'matches' that node and replacing the node with the UCS. A UCS matches a node, if the content of the node is included in the content of the expanded node in the UCS. The result of replacing an unexpanded node x in a completion structure CS with a UCS UC which matches x is denoted as $expand_{CS}(x, UC)$. All formal definitions can be found in (Feier and Heymans 2010). Next rule captures the basic step of the algorithm:

Match. *For a node $x \in N_{EF}$: if* $\text{ST}(x) = unexp$, *nondeterministically choose a unit completion structure UC which matches x and perform $expand_{CS}(x, UC)$.*

4 Blocking, Redundancy, Caching

This section describes the rules which stop the expansion of a completion structure: a previously-introduced blocking rule and a new redundancy and caching rule.

4.1 Blocking

All three algorithms employ a blocking technique which is a generalization of the well-known *subset blocking* technique (Baader et al. 2003): if the label of a an unexpanded node x is included in the label of one of its ancestors y, x is said to be 'blocked' and it is no longer expanded as it can expanded similarly to y; either x is replaced by the subtree $T[y]$ (1) or the successors of y are reused as successors of x (2). In our case, the subset blocking condition is not enough as by applying either (1) or (2) one may introduce infinite paths or cycles in G and thus, violate the well-supportedness property. To avoid this, the blocking rule checks also that there is no path in G from an atom $p(y)$ to an atom $q(x)$.

Blocking. *A node $x \in T \in N_{EF}$ is* blocked *if there is a node $y <_T x$, $y \notin cts(P)$, s. t.* $\text{CT}(x) \subseteq \text{CT}(y)$, *and $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G\} = \emptyset$. We call (y, x) a* blocking pair. *No expansions can be performed on a blocked node.*

4.2 Redundancy

While the content inclusion condition in the blocking rule is eventually fulfilled when expanding a path in a completion structure, the extra condition regarding the nonexistence of paths in G might never be fulfilled. (Feier and Heymans 2009) introduced a new rule, called *redundancy* to ensure termination: if k nodes with equal content, where k is exponential in the size of the program, had already been explored on a branch, the algorithm aborts.

We introduce here a more refined strategy for aborting expansion of a branch. First, one has to keep track of the oldest path in G from which every atom in the constructed model makes part, where 'oldest' is defined w.r.t. the shallowest atom (as regards its depth in the extended forest) which is part of the branch: the smaller the depth of such an atom, the older the path in G it makes part from is. Formally, let the *rank* of an atom a be the shallowest depth of a node y s.t. there exists a unary/binary p/f with

$(p(y)/f(x,y),a) \in paths_G$, or the depth of its deepest argument, otherwise: $rank(p(x)/f(x,y)) = min(\{|x|/|y|\} \cup \{rank(a)|(a,p(x)/f(x,y)) \in A_G\})$. The rank of a node is the minimum of the ranks of the unary atoms having that node as an argument: $rank(x) = \min_{p \in \mathrm{CT}(x)} rank(p(x))$.

Next, the idea is to minimize the set of oldest paths in G running along a branch of the completion structure by keeping track of the intersections of such paths in G with nodes. By $in(k,x)$, one understands the intersection of the set of paths starting with an atom with rank k with node x (presuming $|x| \geq k$): $in(k,x) = \{p|rank(p,x) = k\}$. Nodes with identical content are allowed on the same branch only if every subsequent occurrence of such a node shrinks the intersection of the nodes with the set of oldest paths.

Redundancy. *A node $x \in N_{EF}$ is* redundant *if there is an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, s. t.:* $\mathrm{CT}(x) \subseteq \mathrm{CT}(y)$, $rank(x) = rank(y) = r$, *and* $in(r,x) \supseteq in(r,y)$.

The advantage of this new technique is that while before failure was detected only when reaching a node with exponential depth, the new strategy can potentially identify failure much earlier.

4.3 Caching

Blocking can be generalized to the so-called 'anywhere blocking' or 'caching', where a *cached* node is justified similarly to a *caching* node which belongs to a different branch of the same tree in the extended forest. Again, the content of the cached node has to be included in the content of the caching node. Additionally, a condition regarding sets of paths in G has to be fulfilled.

Caching. *A node $y \in T \in F$ is said to be* cached *if there is a node $x \in T \in F$, $y \not<_T x$, $x \not<_T y$, $x \notin cts(P)$, s. t. $right_T(y,x)$,* $\mathrm{CT}(y) \subseteq \mathrm{CT}(x)$, *and $connpr_G(z,y) \subseteq connpr_G(z,x)$, where $z = common_T(x,y)$. We call (y,x) a* caching pair *and y a* caching node. *A cached node is no longer expanded.*

Like in the case of blocking, when unraveling a completion structure containing a caching pair (x,y), the content of y is justified similarly to the content of x by copying the subtree T_x at y or by reusing the successors of x as successors of y. In order to maintain well-supportedness of the constructed model, it is essential that, by performing one of the two operations, one does not introduce cycles or infinite paths in G: a necessary and sufficient condition for this is $connpr_G(z,y) \subseteq connpr_G(z,x)$, where $z = common_T(x,y)$.

For a cached node to not reuse its own justification in case a successor of its caching node is at its turn a cached node, we impose that a cached node is always 'to the right' of the corresponding caching node in their common tree. Together with this requirement, we enforce the following expansion strategy for the completion structure: *a node $x \in T \in F$ can be expanded iff every node y s.t.: $right_T(y,x)$ is expanded.* The *Match* rule becomes:

Match'. *For a node $x \in N_{EF}$: if* $\mathrm{ST}(x) = unexp$ *and for every node y s.t. $right_T(y,x)$:* $\mathrm{ST}(x) = exp$, *non-deterministically choose a unit completion structure UC which matches x and perform $expand_{CS}(x,UC)$.*

5 Complete/Clash-free Expansion

This section specifies when the expansion is finished, i.e. the structure is 'complete' (no rules can be further applied), and when the obtained completion is a good one, i.e. 'clash-free' (no node is redundant or unexpanded).

Definition 5.1. *A* complete completion structure *for a FoLP P and a $p \in upreds(P)$, is a completion structure that results from the repeated application of the rule* Match' *to an initial completion structure for p and P, taking into account the applicability rules* Blocking, Redundancy, *and* Caching *s. t. no rules can be further applied.*

Definition 5.2. *A complete completion structure $CS = \langle EF, \mathrm{CT}, \mathrm{ST}, G\rangle$ is* clash-free *if for every $x \in N_{EF}$: x is not redundant, and it is either blocked, cached, or $st(x) = exp$.*

6 Termination, Soundness, and Completeness

We show that the algorithm terminates by computing exponential bounds on the branch length in any completion structure, and on the total number of nodes in a completion structure. The first bound is a direct consequence of using the redundancy and the blocking rules. For computing the second bound we use the first bound, and the atom ranks introduced in Section 4.3, and the caching and the redundancy rule.

Proposition 6.1. *Every path in a completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

Proposition 6.2. *A complete completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

The algorithm is sound and complete:

Proposition 6.3. *Let P be a FoLP and $p \in upreds(P)$. There exists a complete clash-free completion structure for p w.r.t. P iff p is satisfiable w.r.t. P.*

Proof Sketch. "$\Rightarrow$": From a complete clash-free completion structure for p w.r.t. P, one constructs an open interpretation by expanding blocked/cached similarly to corresponding blocking/caching nodes by reusing their successors. It is not difficult to show that this interpretation is a model, but one has to also prove minimality: the conditions regarding paths in G in the termination rules are paramount for this.

"$\Leftarrow$": If a unary p is satisfiable, then it is forest satisfiable. A procedure is given to construct a complete clash-free completion structure from any arbitrary (possibly infinite size) forest model of p. To this purpose, UCSs which are parts of the forest model are appended to build a completion structure. For termination, the blocking and caching rules are employed as usually. In the case of redundant nodes, the path between the redundant node and its redundancy witness is eliminated from the completion structure in construction and the process continues.

For complete proofs, see (Feier 2011).

7 Complexity

From Proposition 6.2 it follows that:

Proposition 7.1. *$\mathcal{A}_3$ runs in the worst case in non-deterministic exponential time.*

One can transform $\mathcal{A}_3$ to a deterministic procedure DET$\mathcal{A}_3$ which can be executed in exponential time. The procedure constructs an AND/OR extended forest with depth double the size of the largest depth encountered when running $\mathcal{A}_3$. At odd levels, there are OR nodes with unexpanded content (they contain just the constraints imposed by their predecessor), while at even levels, there are AND nodes which are 'realizations' of their predecessor, i.e., they (together with their outgoing arcs and direct successors) describe a possible way to expand the predecessor node. For more details regarding the construction, please see (Feier 2011).

Proposition 7.2. *DET$\mathcal{A}_3$ is sound, complete, and runs in the worst case in deterministic exponential time.*

Thus, satisfiability checking of a unary predicate p w.r.t. a FoLP P can be evaluated in exponential time in the size of P. This, together with the fact that the same task is EXPTIME-hard (see (Feier and Heymans 2009)), implies that the problem is EXPTIME-complete. With this we close an existing gap regarding the complexity of reasoning with FoLPs.

Proposition 7.3. *Satisfiability checking of a unary predicate p w.r.t. a FoLP P is* EXPTIME*-complete.*

Finally, this translates in a similar result concerning f-hybrid knowledge bases (Feier and Heymans 2011) which are a tight combination of FoLPs and the DL $\mathcal{SHOQ}$. As reasoning with f-hybrid knowledge bases (satisfiability checking) can be reduced to reasoning with FoLPs, it follows that the task is EXPTIME-complete as well.

Proposition 7.4. *Satisfiability checking w.r.t. f-hybrid knowledge bases is* EXPTIME*-complete.*

8 Conclusions and Related Work

We presented a worst-case optimal tableau algorithm for reasoning with FoLPs which improves the performance of its predecessors by employing a new redundancy rule which deals with eliminating *redundant computation along a branch* and a new caching rule which deals with eliminating *redundant computation across branches*.

Among rule-based formalisms, one which allows for unsafe rules is Datalog$^{\pm}$ (Calì, Gottlob, and Lukasiewicz 2009;): it extends Datalog with a special type of rules with existentially quantified variables in the head, called tuple generating dependencies (TGDs). The formalism is undecidable in the general case. Like in the case of OASP, several syntactical restrictions have been imposed on the shape of TGDs in order to regain decidability. These restrictions are orthogonal to the ones we imposed for achieving decidability on FoLPs.

Another decidable fragment of OASP which can simulate the description logic $\mathcal{SHIQ}$ is *Conceptual Logic Programs under the Inverted World Assumption*. The fragment has the *tree model property* and in (Heymans, Van Nieuwenborgh, and Vermeir 2006) reasoning with the fragment has been reduced to checking emptiness of a two-way alternating tree automata. However, there is no practical algorithm for dealing with such programs. Next step is to extend the algorithm introduced in this paper to reason with CoLPs under IWA.

References

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and (eds), P. F. P.-S. 2003. The description logic handbook: Theory, implementation, and applications. In *Description Logic Handbook*. Cambridge University Press.

Calì, A.; Gottlob, G.; and Lukasiewicz, T. Datalog : A Unified Approach to Ontologies and Integrity Constraints. In *Proc. ICDT 2009*, volume 9, 14–30.

Calì, A.; Gottlob, G.; and Lukasiewicz, T. 2009. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. In *Proc. PODS2009*, 77–86. ACM Press.

Eiter, T.; Ianni, G.; Lukasiewicz, T.; Schindlauer, R.; and Tompits, H. 2008. Combining answer set programming with description logics for the Semantic Web. *Artificial Intelligence* 172(12-13):1495–1539.

Fages, F. 1991. A new fix point semantics for generalized logic programs compared with the wellfounded and the stable model semantics. *New Generation Computing* 9(4).

Feier, C., and Heymans, S. 2009. Hybrid Reasoning with Forest Logic Programs. In *Proc. of 6th European Semantic Web Conference*, volume 5554, 338–352. Springer.

Feier, C., and Heymans, S. 2010. An optimization for reasoning with forest logic programs. In *Proc. of Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*. CoRR.

Feier, C., and Heymans, S. 2011. Reasoning with forest logic programs and f-hybrid knowledge bases. *TPLP*. Accepted for publication. Preliminary version at: http://arxiv.org/abs/1110.2773.

Feier, C. 2011. Worst case optimal reasoning with forest logic programs. Technical report, Technical University of Vienna. Available at: http://www.kr.tuwien.ac.at/research/reports/rr1107.pdf.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, 1070–1080.

Grosof, B. N.; Horrocks, I.; Volz, R.; and Decker, S. 2003. Description logic programs: Combining logic programs with description logic. In *Proc. WWW 2003*, 48–57. ACM.

Heymans, S.; Van Nieuwenborgh, D.; and Vermeir, D. 2006. Conceptual Logic Programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)* 47(1–2):103–137.

Heymans, S.; Van Nieuwenborgh, D.; and Vermeir, D. 2007. Open Answer Set Programming for the Semantic Web. *J. of Applied Logic* 5(1):144–169.

Heymans, S.; Van Nieuwenborgh, D.; and Vermeir, D. 2008. Open answer set programming with guarded programs. *Transactions on Computational Logic* 9(4):1–53.

Krötzsch, M.; Rudolph, S.; and Hitzler, P. 2008. Description logic rules. In *Proc. ECAI*, 80–84. IOS Press.

Motik, B.; Sattler, U.; and Studer, R. 2005. Query answering for OWL-DL with rules. *Journal of Web Semantics* 3(1):41–60.

Rosati, R. 2006. DL+log: Tight Integration of Description Logics and Disjunctive Datalog. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 68–78.

Vardi, M. Y. 1998. Reasoning about the Past with Two-way Automata. In *Proc. 25th Int. Colloquium on Automata, Languages and Programming*, 628–641. Springer.