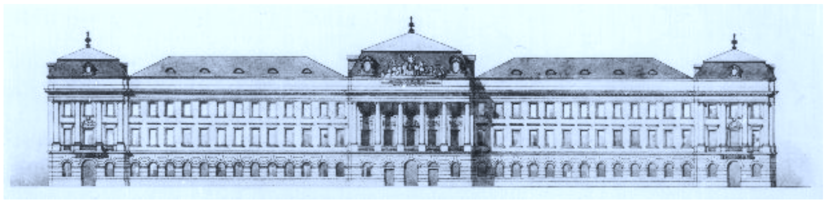
INFSYS

RESEARCH

REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

WORST-CASE OPTIMAL REASONING WITH FOREST LOGIC PROGRAMS

CRISTINA MARIA FEIER

INFSYS RESEARCH REPORT 1843-11-07

DECEMBER 2011

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 1843-11-07, DECEMBER 2011

WORST-CASE OPTIMAL REASONING WITH FOREST LOGIC PROGRAMS

Cristina Feier[1]

Abstract. This report describes a new worst-case optimal tableau algorithm for reasoning with Forest Logic Programs (FoLPs), 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 improves on previous results concerning reasoning with the fragment by decreasing the worst-case running time with one exponential level. The decrease in complexity is mainly due to the usage of a new caching rule, whose introduction is highly non-trivial: this has been made possible by employing a different strategy for reducing an infinite model to a model of finite bounded size. The algorithm also reuses a knowledge compilation technique introduced in previous work.

[1]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: {feier}@kr.tuwien.ac.at.

Acknowledgements: 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).

1 Introduction

Forest Logic Programs (FoLPs) is a decidable fragment of Open Answer Set Programming (OASP) which serves as a tight integration device for $\mathcal{SHOQ}$ ontologies and FoLPs themselves. The integrating formalism is called *f-hybrid* knowledge bases. One salient feature of OASP and thus also of FoLPs, is that while its syntax is typical ASP syntax (where unsafe rules are allowed), its semantics is a hybrid between the classical Answer Set Programming semantics and the classical FOL semantics. From the ASP world it retains a stable model based semantics, while from the FOL world it retains the possibility of having an infinite domain of interpretation: the universe is a non-empty superset of the set of constants in the program.

Example 1.1. *Consider the following program:*

$$\begin{array}{rcl} \mathit{fail}(X) & \leftarrow & \mathit{not\ pass}(X) \\ \mathit{pass}(\mathit{john}) & \leftarrow & \end{array}$$

Although the predicate $fail$ is not satisfiable under the ordinary answer set semantics – the only answer set being $\{pass(john)\}$ – it is satisfiable under the open answer set semantics. If one considers, for example, the universe $\{john, x\}$, with x some individual which does not belong to the Herbrand universe, there is an open answer set $\{pass(john), fail(x)\}$ which satisfies $fail$.

The fact that the universe of interpretation is not restricted to the Herbrand universe makes it possible to simulate within the formalism, General Inclusion Axioms with an *exists restriction* on their right-hand side: this is a feature which was identified as desirable in the Ontorule project[1] during the analysis of requirements from use cases. For a discussion about this, see the Analysis of the Steel Industry Use Case in the Appendix B of Ontorule deliverable D3.3 [Feier et al., 2010].

FoLPs allow for the presence of only unary and binary predicates in rules which have a tree-like structure. This makes the fragment decidable by ensuring that it has the forest model property: if a unary predicate is satisfiable, than it is satisfied by a forest-shaped model. A forest shaped model is a model in which the universe of interpretation can be seen as a forest, two nodes in the forest being connected iff there is a binary atom in the model having as arguments the respective nodes.

A sound and complete algorithm for satisfiability checking of unary predicates w.r.t. FoLPs has been described in [Feier and Heymans, 2009] and [Feier and Heymans, 2011]. The algorithm exploits the forest model property of the fragment: it is essentially a tableau-based procedure which builds such a forest model in a top-down fashion. It starts with a skeleton for a forest model which contains only one constraint: p, the unary predicate checked to be satisfiable, has to appear in the label of a node in the forest. Then, in order to satisfy existing constraints, it progressively introduces new ones by inserting (negated) predicates in the contents of nodes/arcs of the forest based on the rules of the program. The forest model is constructed by evolving a data structure called "completion structure" which contains a labeled forest, the model in construction, together with additional information needed for the construction. When certain conditions are met, like either there are no unsatisfied constraints, or the ones left can be satisfied similarly to previously met constraints, the algorithm terminates successfully. We refer to this algorithm as $\mathcal{A}_1$.

[Feier and Heymans, 2010] presents an optimization of the first algorithm by means of a knowledge compilation technique: the new algorithm computes in an initial step all possible building blocks of the model using $\mathcal{A}_1$, and then matches and appends these blocks using similar conditions for termination as the original algorithm. Such building blocks are restricted to completion structures, in which there is only one node fully

[1] http://ontorule-project.eu/

expanded (i.e. all constraints associated to that node are satisfied), and are called *unit completion structures*. We refer to this algorithm as $\mathcal{A}_2$.

Both $\mathcal{A}_1$ and $\mathcal{A}_2$ run in the worst case in double exponential time. The current report presents an algorithm with improved running time by dropping the worst-case complexity one exponential level: it runs in the worst case in exponential time in the size of the input program. This also settles a gap concerning the complexity of FoLPs: it was known that FoLPs are EXPTIME-hard, but not known whether they are EXPTIME-complete. The new algorithm shows that they are indeed EXPTIME-complete. We refer to the new algorithm as $\mathcal{A}_3$.

$\mathcal{A}_3$ takes over the idea of using unit complete structures from $\mathcal{A}_2$. Constraints regarding contents of nodes are satisfied by finding appropriate unit completion structures and appending them. However, unlike $\mathcal{A}_2$, it employs different termination techniques. In particular it employs a new technique for identifying redundancy across a path and a caching technique.

The section is organized as follows: Section 2 introduces some technical preliminaries. Section 3 introduces formally Forest Logic Programs and the notions of forest model and forest satisfiability. Section 4 describes a simplified version of $\mathcal{A}_2$. Finally, Section 5 describes the new algorithm, while Section 6 draws some conclusions and discusses future work.

2 Preliminaries

We recall the open answer set semantics [Heymans et al., 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 will be called a *regular literal*. For a regular literal L, $pred(L)$, and $args(L)$ denote the predicate, and the (tuple of) arguments of L[2], respectively. By $args_i(L)$, for a regular literal L, we understand the i-th argument of L.

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: $S(X,Y) = \{a(X,Y) \mid a \in S\}$. 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. Rules can also be named, as in $r : \alpha \leftarrow \beta$, where r is the name of the rule. 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: $q(t_1, \ldots, t_n) \vee not\ q(t_1, \ldots, t_n) \leftarrow$, for terms $t_1, \ldots, t_n$; this kind of rules enables a choice for the inclusion of atoms in the open answer sets. We call a predicate q *free* if there is a free rule: $q(X_1, \ldots, X_n) \vee not\ q(X_1, \ldots, X_n) \leftarrow$, with variables $X_1, \ldots, X_n$. 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.

For a term t, the *exact replacement* of ground term x with ground term y in t, denoted $t_{x|y}$, is defined as

[2]If the literal L has just one argument, $args(L)$ will return the argument itself.

follows: $t_{x|y}=\begin{cases} y, & \text{if } t = x; \\ t, & \text{otherwise} \end{cases}$. The notation extends to tuples of terms, literals, rules, and programs. For a tuple of terms $T = (t_1, \ldots, t_n)$, $T_{x|y} = ((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a regular literal $L = (not\ )p(t_1, \ldots, t_n)$, $L_{x|y} = (not\ )\ p((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a set of literals S, $S_{x|y} = \{L_{x|y} \mid L \in S\}$. For a named rule $r : \alpha \leftarrow \beta$, its image under the exact replacement of x with y is $r_{x|y} : \alpha_{x|y} \leftarrow \beta_{x|y}$ (where $r_{x|y}$ is the new name of the rule, and does not involve any term replacement). For a ground program P, its image under the exact replacement of x with y is $P_{x|y} = \{r_{x|y} \mid r \in P\}$.

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 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. $p(x_1, \ldots, x_n) \in M$, 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 $x \in T$. For example a tree with root c and 2 successors will be denoted as $\{c, c \cdot 1, c \cdot 2\}$ or $\{c, c1, c2\}$ [3]. By convention $x \cdot 0 = x$ and $(x \cdot c) \cdot -1 = x$ ($c \dot{-} 1$ is undefined). 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 $x \neq y$. The predecessor of a node x in a tree T is denoted with $prev_T(x)$ and it is the node y such that there exists $\imath \in \mathbb{N}_*$ such that $x = y \cdot i$. The deepest common ancestor of two nodes x and y in a tree T, denoted $common_T(x, y)$ is the node z such that $z <_T x$, $z <_T y$, and there is no node $z' \in T$ such that $z' >_T z$, $z' <_T x$, 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 $s_1, s_2 \in \langle\mathbb{N}^*\rangle$, such that $x = z \cdot i \cdot s_1$, $y = z \cdot j \cdot s_2$, 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 path in a tree T from x to y is denoted with $path_T(x, y) = \{z \mid x \leqslant z \leqslant y\}$.

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 tree in the forest.

In the following, all terms in ground programs which we operate with are nodes in some extended forest,

[3]By abuse of notation, we consider that there are at most 9 successors for every node, so we can abbreviate $a \cdot b$ with ab

and as such they are sequences formed with the help of the $\cdot$ operator. Taking into account the structure of such terms, we introduce a finer grain (ground) term replacement operator which replaces the prefix of a term with another term. This is simply called *replacement* of x with y in t, it is denoted with $t_{x||y}$, and it is defined as $t_{x||y} = \begin{cases} y \cdot z, & \text{if } t = x \cdot z; \\ t, & \text{otherwise} \end{cases}$. Similarly as for the exact replacement, the notion of replacement is extended to (sets of) literals, tuples, rules, and programs.

When an extended forest $EF = (F, ES)$, is such that F is a set of trees $\{T_c \mid c \in C\}$, for C a set of distinguished constants, and there exists $d \in C$ such that $T_c = \{c\}$, for every $c \in C \backslash \{d\}$, and $ES \subseteq T_d \times C$, we call the forest an *extended tree with root* d *w.r.t.* C: all trees but one are single-node trees and the nodes of the *distinguished tree* T_d can be interlinked with constants from C; no other links from elements of C are allowed. The depth of an extended tree is the depth of its distinguished tree.

Finally, a directed graph G is defined as usual by its sets of nodes V and arcs A. We introduce some graph-related notations: $paths_G$ denotes the set of paths in G, where each path is a tuple of nodes from V: $paths_G = \{(x_1, \ldots, x_n) \mid ((x_i, x_{i+1}) \in A)_{1 \leqslant i < n}\}$, $paths_G(x, y)$ denotes the set of paths in G from x to y: $paths_G(x, y) = \{(x_1 = x, \ldots, x_n = y) \mid ((x_i, x_{i+1}) \in A)_{1 \leqslant i < n}\}$, while $conn_G$ denotes the set of pairs of connected nodes from V: $conn_G = \{(x, y) \mid \exists Pt = (x_1, \ldots, x_n) \in paths_G : x_1 = x \wedge x_n = y\}$. As an extended forest is a particular type of graph, these notations apply also to extended forests. Cycles and elementary cycles in directed graphs are defined as usually. In order to operate with paths in directed graphs we also introduce some tuple operators: the concatenation of two tuples $T_1 = (x_1, \ldots, x_n)$, and $T_2 = (y_1, \ldots, y_m)$, denoted $T_1 \hat{\ } T_2$ is the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$. A tuple T_1 is part of another tuple T_2: $T_1 \subseteq T_2$, if there exists two (possibly empty) tuples T_3 and T_4 such that $T_2 = T_3 \hat{\ } T_1 \hat{\ } T_4$.

3 FoLPs

Forest Logic Programs are a subset of Open Answer Set Programming (OASP) which allows one to simulate the DL $\mathcal{SHOQ}$, underpinning the tightly-coupled combination of rules and ontologies *f-hybrid* knowledge bases.

Definition 3.1. *A* forest logic program (FoLP) *is a program with only unary and binary predicates, and such that a rule is either:*

- *a* free rule*:*

$$a(s) \vee not\ a(s) \leftarrow \ \ or\, f(s, t) \vee not\ f(s, t) \leftarrow \tag{1}$$

where s *and* t *are terms;*

- *a unary rule:*

$$a(s) \leftarrow \beta(s), (\gamma_m(s, t_m), \delta_m(t_m))_{1 \leqslant m \leqslant k}, \psi \tag{2}$$

with $\psi \subseteq \bigcup_{1 \leqslant i \neq j \leqslant k} \{t_i \neq t_j\}$ *and* $k \in \mathbb{N}$*, or a* binary rule*:*

$$f(s, t) \leftarrow \beta(s), \gamma(s, t), \delta(t) \tag{3}$$

where $a \in upreds(P)$ *and* $f \in bpreds(P)$*,* s*,* t*, and* $(t_m)_{1 \leqslant m \leqslant k}$ *are terms,* β*,* δ*,* $(\delta_m)_{1 \leqslant m \leqslant k} \subseteq upreds(P) \cup not\ (upreds(P))$ *(sets of (possibly negated) unary predicates),* $\gamma, (\gamma_m)_{1 \leqslant m \leqslant k} \subseteq bpreds(P) \cup not\ (bpreds(P))$ *(sets of (possibly negated) binary predicates), and*

1. *inequality does not appear in any γ: $\{\neq\} \cap \gamma_m = \emptyset$, for $1 \leqslant m \leqslant k$, 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 $1 \leqslant m \leqslant k$, and $\gamma^+ \neq \emptyset$, if t is a variable;*

- *a constraint: $\leftarrow a(s)$ or $\leftarrow f(s,t)$, where s and t are terms.*

In every rule, all terms which are variables are distinct[4].

Example 3.2. *The following program*[5] *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* supportedBy *which describes the 'has support' relationship is free:* rule r_3. *No individual can be at the same time both a special member and a regular member:* constraint r_4. *Somebody is a regular member if it is involved in some project:* rule r_5. *The binary predicate* involvedIn *which describes the 'involved in a project' relationship is free:* rule r_6. *There is a project j:* fact r_7.

$$\begin{array}{lrcl}
r_1: & smember(X) & \leftarrow & supportedBy(X,Y), smember(Y) \\
r_2: & smember(X) & \leftarrow & supportedBy(X,Y), rmember(Y), \\
 & & & supportedBy(X,Z), rmember(Z), \\
 & & & Y \neq Z \\
r_3: & supportedBy(X,Y) \vee not\ supportedBy(X,Y) & \leftarrow & \\
r_4: & & \leftarrow & smember(X), rmember(X) \\
r_5: & rmember(X) & \leftarrow & involvedIn(X,Y), project(Y) \\
r_6: & involvedIn(X,Y) \vee not\ involvedIn(X,Y) & \leftarrow & \\
r_7: & project(j) & \leftarrow &
\end{array}$$

As already mentioned, FoLPs have the *forest model property*.

Definition 3.3. *Let P be a program. A predicate $p \in upreds(P)$ is* forest satisfiable *w.r.t. P if there is an open answer set (U, M) of P and there is an extended forest $EF \equiv (\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES)$, where ε is a constant, possibly one of the constants appearing in P*[6]*, and a labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$ such that*

- *$p \in \mathcal{L}(\varepsilon)$,*
- *$U = N_{EF}$, and*
- *$M = \{\mathcal{L}(x)(x) \mid x \in N_{EF}\} \cup \{\mathcal{L}(x,y)(x,y) \mid (x,y) \in A_{EF}\}$, 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.

[4]This restriction precludes the presence in rules of literals of the form $f(X,X)$ or $not\ f(X,X)$ which would break the forest model property.

[5]The example is a variation of an example described in [Feier and Heymans, 2010].

[6]Note that in this case $T_\varepsilon \in \{T_a \mid a \in cts(P)\}$. Thus, the extended forest contains for every constant from P a tree which has as root that specific constant and possibly, but not necessarily, an extra tree with unidentified root node.

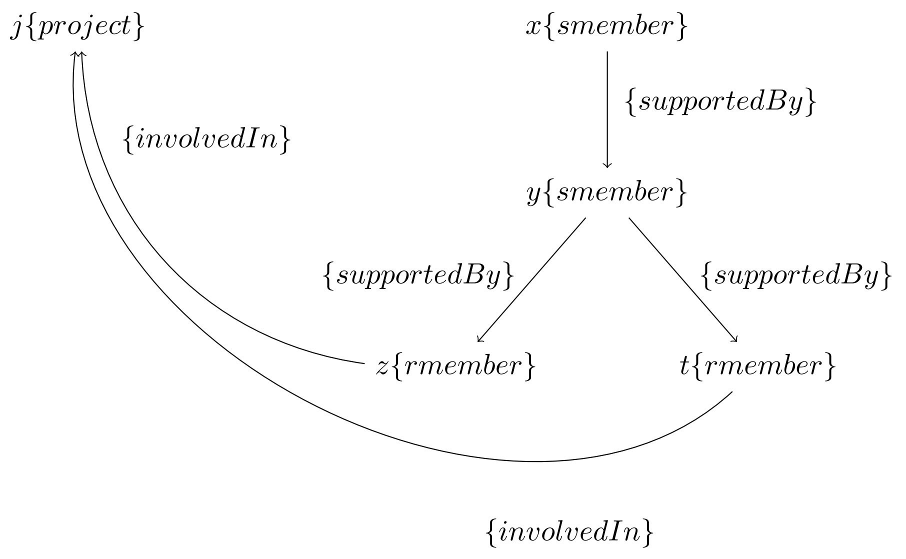

$$\begin{tikzcd}
j\{project\} & & x\{smember\} \arrow[d, "\{supportedBy\}"] & \\
& & y\{smember\} \arrow[dl, "\{supportedBy\}"'] \arrow[dr, "\{supportedBy\}"] & \\
& z\{rmember\} \arrow[ull, bend left, "\{involvedIn\}"] & & t\{rmember\} \arrow[ulll, bend left=60, "\{involvedIn\}"]
\end{tikzcd}$$

Figure 1: A forest model for P.

Proposition 3.4 ([Heymans et al., 2007]). *FoLPs have the forest model property.*

Example 3.5. *Consider the FoLP P introduced in Example 3.2.*

The unary predicate $smember$ is forest satisfiable w.r.t. P: there is a forest model $(\{j, x, y, z, t\}$, $\{smember(x), supportedBy(x, y), smember(y), rmember(z), rmember(t), supportedBy(y, z), supportedBy(y, t)$, $involvedIn(z, j)$, $involvedIn(t, j)$, $project(j)\})$ in which $smember$ appears in the label of the (anonymous) root of one of the trees in the forest (see Figure 1).

4 Previous Algorithm for Reasoning with FoLPs using Unit Completion Structures

As mentioned in the introduction, all algorithms developed so far for reasoning with FoLPs have the same underlying principle: they try to construct a forest model in a tableau-like fashion. All algorithms share the same data structure, called *completion structure*, which is a representation of a forest model in construction. In section 4.1 we describe this data structure and recall how it can be evolved using so-called expansion rules introduced in [Feier and Heymans, 2009] and described in more detail in [Feier and Heymans, 2011]. These expansion rules are used by $\mathcal{A}_2$, the algorithm which precomputes the set of unit completion structures [Feier and Heymans, 2010]. The new version of the algorithm, $\mathcal{A}_3$ reuses the knowledge compilation technique introduced by $\mathcal{A}_2$. As such, section 4.2 recalls $\mathcal{A}_2$.

4.1 Completion Structures

The main data structure used by all three algorithms is a so-called *completion structure*. A completion structure describes a forest model in construction. It contains an extended forest EF, whose set of nodes constitutes the universe of the model in construction, and a labeling function $\textsc{ct}$ (*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. Note that unlike the labeling function $\mathcal{L}$ in Definition 3.3, that describes which atoms are in the forest model, the labeling function $\textsc{ct}$ keeps track also of which atoms are not in the forest model. This is needed as the completion structure is updated by justifying both the presence or the absence of a certain atom in the model.

There is a difference in how a completion structure is updated by $\mathcal{A}_1$ as opposed to how it is updated by $\mathcal{A}_2$ and $\mathcal{A}_3$. The original algorithm $\mathcal{A}_1$ updates a completion structure by means of so-called expansion rules which justify or assert the presence/absence in the model of one atom at a time. The 'local status' function $\textsc{lst}$ assigns the value $unexp$ to pairs of nodes/arcs and possibly negated unary/binary predicates which have not yet been 'expanded', i.e. justified, and the value exp to such pairs which have already been considered. However, $\mathcal{A}_2$ and $\mathcal{A}_3$, update the structure by considering one node at a time and trying to satisfy all constraints imposed by that node in a single step. So, in this case, the local status function has been replaced by a 'status' function $\textsc{st}$ which assigns one the values exp or $unexp$ to nodes of the forest, depending whether their content has been justified or not. Based on this difference concerning the status function, we distinguish between $\mathcal{A}_1$- and $\mathcal{A}_2$- completion structures.

Furthermore, all algorithms have to ensure that the constructed forest model is a well-supported one [Fages, 1991], or in other words, no atom in the model is circularly justified (does not depend on itself) or infinitely justified (does not depend on an infinite chain of other atoms). As such, a graph G which keeps track of dependencies between atoms in the model is maintained both by a $\mathcal{A}_1$- and a $\mathcal{A}_2$- completion structure. The formal definition is given below.

Definition 4.1. *An $\mathcal{A}_1$-/$\mathcal{A}_2$- completion structure for a FoLP P is a tuple $\langle EF, \textsc{ct}, \textsc{lst}/\textsc{st}, G\rangle$ where:*

- *$EF = \langle F, ES\rangle$ is an extended forest,*
- *$\textsc{ct} : N_{EF} \cup A_{EF} \to 2^{preds(P) \cup not\ (preds(P))}$ is the 'content' function,*
- *$\textsc{lst} : N_{EF} \times 2^{upreds(P) \cup not\ upreds(P)} \cup A_{EF} \times 2^{bpreds(P) \cup not\ bpreds(P)} \to \{exp, unexp\}$/$\textsc{st} : N_{EF} \to \{exp, unexp\}$ is the 'local status'/'status' function,*
- *$G = \langle V, A\rangle$ is a directed graph which has as vertices atoms in the answer set in construction: $V \subseteq \mathcal{B}_{P_{N_{EF}}}$.*

$\mathcal{A}_1$-/$\mathcal{A}_2$- completion structures are constructed by starting with a skeleton for a model of a FoLP P which satisfies a unary predicate p, which is called $\mathcal{A}_1$-/$\mathcal{A}_2$- *initial completion structure for checking satisfiability of a unary predicate* p *w.r.t.* P and then progressively updating such a structure.

An $\mathcal{A}_1$-/$\mathcal{A}_2$- initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P imposes a single constraint on the model in construction: that some atom $p(\varepsilon)$ has to be part of the model, where ε is an anonymous individual or one of the constants in the program. As every model of P has as part of its universe the set $cts(P)$, the extended forest EF is initialized with a set of single-node trees, one tree for each constant appearing in P (having the respective constant as a root) and possibly a new single-node tree with an anonymous root (in case ε, the node where p is asserted to be satisfied, is anonymous)[7]. The content of ε is initialized with $\{p\}$, while the contents of the other nodes (roots) are initialized with $\emptyset$. G is initialized to the graph with a single vertex $p(\varepsilon)$. Formally:

[7]Note that this complies with the generic shape of a forest model described in section 3.

Definition 4.2. *An* $\mathcal{A}_1$-/$\mathcal{A}_2$- initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P *is a completion structure* $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$*, where:*

- $EF = (F, \emptyset)$*,* $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$*, where* ε *is a constant, possibly in* $cts(P)$*,*
- $T_x = \{x\}$*, for* $x \in \{\varepsilon\} \cup cts(P)$*,*
- $\textsc{lst}(\varepsilon, p) = unexp$*/*$\textsc{st}(x) = unexp$*, for* $x \in \{\varepsilon\} \cup cts(P)$*,*
- $G = \langle V, \emptyset\rangle$*,* $V = \{p(\varepsilon)\}$*, and*
- $\textsc{ct}(\varepsilon) = \{p\}$.

Note that the extended forest EF in an $\mathcal{A}_1$-/$\mathcal{A}_2$- *initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP* P is an extended tree.

The original algorithm, $\mathcal{A}_1$, expands an $\mathcal{A}_1$-completion structure by means of so-called expansion rules which justify or assert the presence/absence in the model of one atom at a time. Expansion rules satisfy current constraints in the structure, or in other words, they justify the presence/absence of certain atoms in the constructed model, by making true the body of a ground rule which has the atom in the head (in case the atom is in the model) or making false all bodies of ground rules which have the atom in the head (in case the atom is not in the model). Concretely, expansion rules may introduce new successors for the node under consideration in order to obtain successful groundings for unary rules, and may assert predicate symbols or their negation to the contents of nodes/arcs in the completion structure in order to make the body of the corresponding ground rule satisfiable/unsatisfiable. The expansion rules which take care of this are called *expand unary/binary positive/negative* rules and they are formally described in [Feier and Heymans, 2011] as the expansion rules (i), (iii) and (iv), (vi), respectively.

Newly introduced domain elements give rise to new ground atoms and rules and some of these rules might render the program inconsistent. In order to be sure that the partially constructed model is a complete one every ground atom which can be formed with unary/binary predicates from the program and nodes/arcs in the forest model in construction has to be proved to be either part or not part of the forest model. As such, if for a unary/binary atom, neither the atom nor its negation appear in the content of some node/arc in the forest, either the unary/binary atom or its negation is inserted in the content of such a node/arc. The expansion rules which take care of this are called *choose unary/binary* rules and they are formally described in in [Feier and Heymans, 2011] as the expansion rules (ii) and (v).

For an extended example regarding $\mathcal{A}_1$, and thus the application of the expansion rules, see [Feier and Heymans, 2011].

Before describing $\mathcal{A}_2$, the knowledge compilation method, we recall one more notation introduced in [Feier and Heymans, 2009] (as part of the applicability rule (vii) Saturation): a node $x \in N_{EF}$ is said to be *saturated* if every unary predicate or its negation appear in its content with status exp and every binary predicate or its negation appear in the content of each of its outgoing arcs with status exp.

4.2 Unit Completion Structures

A unit completion structure (UCS) is a completion structure in which EF is an extended tree of depth 1 having as roots of the trees the constants in the program and possibly an additional node standing for an anonymous individual: if there is such an anonymous individual, it is the root of the distinguished tree in the extended tree. The root of the distinguished tree, ε, is saturated: every unary predicate appears (either in a positive or a negated form) in $\textsc{ct}(\varepsilon)$ and every binary predicate appears (either positive or negated) in the content of every outgoing arc of ε. As the distinguished tree has depth 1, we call the nodes which are direct successors of ε in EF simply *successor nodes* (in the UCS). Note that successor nodes can be either anonymous individuals or constants from the program.

Unit completion structures can be used as building blocks of a forest model. A UCS describes how the literals formed with the (possibly negated) unary/binary predicates in the content of ε and its outgoing arcs are justified by the presence of some other (possibly negated) predicates in the contents of the nodes/arcs of the structure. No predicate in the contents of successor nodes is expanded. At an abstract level a UCS captures the process of justifying the constraints imposed by a node in a completion structure by introducing new constraints in the form of successors of that node.

4.2.1 Constructing the Set of Unit Completion Structures

In order to construct a unit completion structure one starts with a skeleton, an *initial unit completion structure* which is similar to an $\mathcal{A}_1$-initial completion structure for checking the satisfiability of a unary predicate p w.r.t. a FoLP P. An initial unit completion structure has the same extended tree skeleton like an $\mathcal{A}_1$-initial completion structure, but it does not impose any constraints regarding membership of predicates to nodes/arcs. This is because UCSs have to be more generic if they are to be reused as building blocks of the model. It also employs a 'local status function' as a UCS is constructed from an initial UCS by using the expansion rules previously mentioned. So, an initial unit completion structure is actually an $\mathcal{A}_1$-completion structure.

Definition 4.3. *An* initial unit completion structure with root ε *for a FoLP P is an $\mathcal{A}_1$-completion structure $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$ where:*

- $EF = (F, ES)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, *where ε is a constant, possibly in $cts(P)$, and* $ES = \emptyset$,
- $T_x = \{x\}$ *for every* $x \in \{\varepsilon\} \cup cts(P)$,
- $\textsc{ct}(x) = \emptyset$, *for every* $x \in \{\varepsilon\} \cup cts(P)$,
- $G = \langle V, A\rangle$, $V = \emptyset$, $A = \emptyset$.

Next we specify when a unit completion structure is 'fully' expanded.

Definition 4.4. *A* unit completion structure $\langle EF, \textsc{ct}, \textsc{lst}, G\rangle$ *with root ε for a FoLP P, with* $EF = (\{T_\varepsilon\}, ES)$, *is an $\mathcal{A}_1$-completion structure derived from an initial unit completion structure with root ε for P by application of the expansion rules (i)-(vi) from [Feier and Heymans, 2011], which has the following properties:*

- *for all* $p \in upreds(P)$, *either* $p \in \textsc{ct}(\varepsilon)$ *and* $\textsc{lst}(p, \varepsilon) = exp$, *or* $not\ p \in \textsc{ct}(\varepsilon)$ *and* $\textsc{lst}(not\ p, \varepsilon) = exp$;
- *for all* $c \in \mathbb{N}^*$ *s.t.* $\varepsilon \cdot c \in T$, *and for all* $f \in bpreds(P)$, *either* $f \in \textsc{ct}(\varepsilon, \varepsilon \cdot c)$ *and* $\textsc{lst}(p, (\varepsilon, \varepsilon \cdot c)) = exp$, *or* $not\ f \in \textsc{ct}(\varepsilon, \varepsilon \cdot c)$ *and* $\textsc{lst}(not\ p, (\varepsilon, \varepsilon \cdot c)) = exp$;
- *for all* $c \in cts(P)$ *s.t.* $(\varepsilon, c) \in ES$ *and for all* $f \in bpreds(P)$ *either* $f \in \textsc{ct}(\varepsilon, c)$ *and* $\textsc{lst}(p, (\varepsilon, c)) = exp$, *or* $not\ f \in \textsc{ct}(\varepsilon, c)$ *and* $\textsc{lst}(not\ p, (\varepsilon, c)) = exp$;
- *for all c s.t.* $\varepsilon \cdot c \in T$ *and for all* $\pm p \in \textsc{ct}(\varepsilon \cdot c)$, $\textsc{lst}(\pm p, \varepsilon \cdot c) = unexp$;

- *for all c s.t. $(\varepsilon, c) \in ES$ and for all $\pm p \in \text{CT}(c)$, $\text{LST}(\pm p, c) = unexp$.*

For examples of unit completion structures for a FoLP P, see [Feier and Heymans, 2010].

Proposition 4.5. *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. 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. □

Once a unit completion structure is constructed, the local status function is no longer relevant. As such, from now on we will refer to unit completion structures as triples $\langle EF, \text{CT}, G\rangle$, leaving the local status function apart.

4.2.2 Using Unit Completion Structures

As mentioned previously, in a unit completion structure, the contents of the root and of the arcs are fully justified while no constraint associated with one of the successor nodes is satisfied. An $\mathcal{A}_2$-completion structure is evolved by starting with an $\mathcal{A}_2$-initial completion structure and repeatedly appending new unit completion structures to the structure such that every new added UCS justifies the constraints imposed by some unexpanded node in the structure. Leaf nodes of the completion structure in construction (successor nodes of previously added UCS) are matched with new UCS-s and are eventually replaced by these. The notion of matching will be made clear later.

We introduce next the notion of *local satisfiability* for a unit completion structure.

Definition 4.6. *A unit completion structure UC for a FoLP P with root ε* locally satisfies *a (possibly negated) unary predicate p iff $p \in \text{CT}(\varepsilon)$. Similarly, UC locally satisfies a set S of (possibly) negated unary predicates iff $S \subseteq \text{CT}(\varepsilon)$.*

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 $not\ p$ 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.

A node of a completion structure can be matched with a unit completion structure if the unit completion structure locally satisfies the content of the node and the constraints imposed by the UCS on nodes which are constants from P are not in contradiction with the current contents of those nodes.

Definition 4.7. *Let $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$ be an $\mathcal{A}_2$-completion structure. A node $x \in N_{EF}$ is* matchable *with a unit completion structure $UC = \langle EF', \text{CT}', G'\rangle$ with root ε, with $EF' = (F', ES')$, iff:*

- $\text{ST}(x) = unexp$,
- *$x = \varepsilon$, if[8] $\varepsilon \in cts(P)$,*
- *UC locally satisfies $\text{CT}(x)$, and*
- *for every arc $(x, c) \in ES'$, and for every $\pm p \in \text{CT}'(c)$: $\mp p \notin \text{CT}(c)$.*

We say that UC matches *x.*

[8] Unit completion structures with roots constants can only be matched with the corresponding constant nodes.

Next we define the operation which expands an $\mathcal{A}_2$-completion structure by adding a new UCS which matches an unexpanded node in the structure. Suppose that x is such an unexpanded node in an $\mathcal{A}_2$-completion structure $CS = \langle EF, \text{CT}, \text{ST}, G \rangle$, with $G = (V, A)$, and that x is matchable with a unit completion structure $UC = \langle EF^{'}, \text{CT}^{'}, G' \rangle$, with root ε, $EF = (F', ES^{'})$, and $G^{'} = (V^{'}, A^{'})$. Suppose also that $x \in T_c \in EF$. Node x can then be expanded by replacing it with UC using an operation called $expand_{CS}(x, UC)$ which updates CS as follows:

- $\text{ST}(x) = exp$,
- $T_c = T_c \cup (T_\varepsilon)_{\varepsilon||x}$;
- $ES = ES \cup \{(x, v) \mid (\varepsilon, v) \in ES'\}$;
- if $u \in T_\varepsilon$ and $v \in succ_{EF'}(u)$: $\text{CT}(u_{\varepsilon||x}) = \text{CT}'(u)$ and $\text{CT}(u_{\varepsilon||x}, v_{\varepsilon||x}) = \text{CT}'(u, v)$;
- if $u \in T_c[x]$ (the new $T_c[x]$) and $v \in N_{EF}$: $\text{ST}(u_{\varepsilon||x}) = \text{ST}'(u)$ and $\text{ST}(u_{\varepsilon||x}, v_{\varepsilon||x}) = \text{ST}'(u, v)$;
- for all $c \in cts(P)$: $\text{CT}(c) = \text{CT}(c) \cup \text{CT}'(c)$;
- $V = V \cup \{a_{\varepsilon||x} \mid a \in V'\}$;
- $A = A \cup \{(a_{\varepsilon||x}, b_{\varepsilon||x}) \mid (a, b) \in A'\}$.

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

Now that we have a way to evolve a completion structure some conditions regarding termination are in order. The algorithm uses two rules for this: the first one, blocking, describes a condition for successful termination of expansion of a branch of a completion structure, while the other, redundancy, describes a condition for unsuccessful termination - if this condition is met, the algorithm backtracks.

Rule. (viii) Blocking. *A node $x \in N_{EF}$ is* blocked *if there is an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, s. t.:*

- $\text{CT}(x) \subseteq \text{CT}(y)$, *and*
- *the set $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty.*

We call (y, x) a blocking pair. *No expansions can be performed on a blocked node.*

Unlike the typical case for tableau algorithms for DLs [Baader et al., 2003], subset blocking is not enough for pruning the expansion of a path in the extended forest. To understand why, we recall the intuition behind using blocking techniques in tableau algorithms: the idea is that a completion structure which contains a blocking pair (y, x) is unfolded to a model by justifying the content of the blocked node x similarly to the way the content of its corresponding blocking node y has been already justified. This can be done either by copying the subtree T_y at x or by reusing the successors of y as successors of x. The first case is described by Figure 2: in this case one obtains an infinite forest shaped model, as the new copy of T_y contains a new copy of x, which again will be justified by copying there T_y, and so on. In the second case, the resulted model is no longer forest-shaped: this is the way we construct models from completion structures in our Soundness proof and it is depicted in Figure 7.

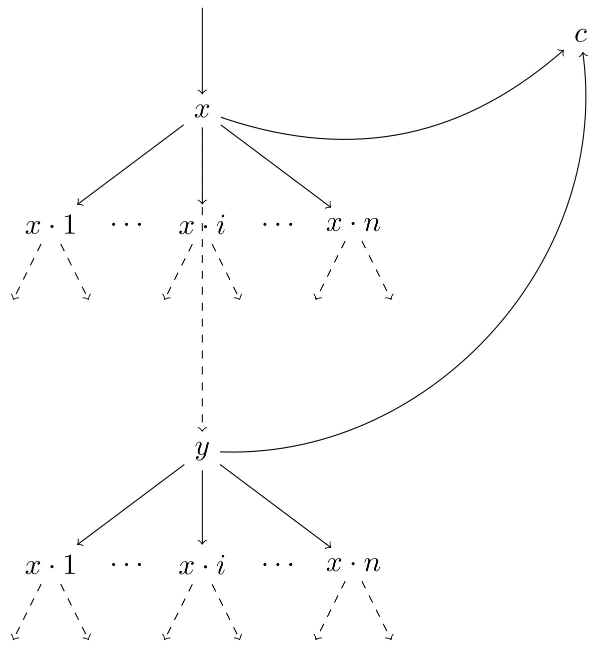

c
x
$x \cdot 1$ $\cdots$ $x \cdot i$ $\cdots$ $x \cdot n$
y
$x \cdot 1$ $\cdots$ $x \cdot i$ $\cdots$ $x \cdot n$

Figure 2: Justifying a blocked node y by replicating the justification of its corresponding blocking node x

The particularity in dealing with FoLPs consists in the fact that unraveling the completion by applying one of the two operations described above can potentially introduce infinite paths in G (in the first case) or cycles in G (in the second case). This would contradict the fact that every atom in the open answer set has to be finitely motivated [Heymans et al., 2006, Theorem 2]. In order to avoid this, the blocking rule verifies also that there is no path from a $p(y)$ to a $q(x)$. The extra condition makes the blocking rule insufficient to ensure the termination of the algorithm. The following applicability rule ensures termination.

Rule. (ix) Redundancy. *A node $x \in N_{EF}$ is* redundant *iff:*

- *x is saturated and not blocked, and*
- *there are k ancestors of x in F, $(y_i)_{1 \leqslant i \leqslant k}$, with $k = 2^p(2^{p^2} - 1) + 3$, and $p = |upreds(P)|$, 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 $(k + 1)th$ node, in case it exists, is blocked [Feier and Heymans, 2009]. One can thus search for forest models only of the latter type. As such the detection of a redundant node indicates a failure in the expansion process and stops the expansion.

Next we define when the expansion of a completion structure is *complete*, and when the completion structure is a 'good one', i.e. it is *clash-free*.

Definition 4.8. *An $\mathcal{A}_2$-*complete completion structure *for a FoLP P and a unary predicate $p \in upreds(P)$, is a completion structure that results from repeated applications of the rule* Match *to an initial completion* *structure for* p *and* P*, taking into account the applicability rules (viii) and (ix),s. t. no further 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: if among the first $k+1$ nodes on a path with equal content, there is no blocking node, the last node on the path is redundant.

Definition 4.9. *An* $\mathcal{A}_2$*-complete completion structure* $CS = \langle EF, \textsc{ct}, \textsc{st}, G \rangle$ *is* clash-free *if (1)* EF *does not contain redundant nodes (2) there is no node* $x \in N_{EF}$*,* x *unblocked, s.t.* $st(x) = unexp$*.*

4.2.3 Termination, Soundness, Completeness

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

Proposition 4.10. *An initial completion structure for a unary predicate* p *and a FoLP* P *can always be expanded to an* $\mathcal{A}_2$*-complete completion structure in a finite number of steps.*

The algorithm is sound and complete:

Proposition 4.11. *A unary predicate* p *is satisfiable w.r.t. a FoLP* P *iff there is an* $\mathcal{A}_2$*-complete clash-free 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.

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

Proposition 4.12. $\mathcal{A}_2$ *runs in the worst-case in double exponential time.*

5 Optimized/Optimal Reasoning with FoLPs

In this section we describe $\mathcal{A}_3$, th new worst-case optimal algorithm. Like in the case of $\mathcal{A}_2$, a structure is constructed by appending UCSs using the *Match* rule, but a different strategy is employed for termination.

Firstly, the algorithm employs a technique that identifies when some redundant computation has been performed during the expansion of a path and stops the expansion of that path, much earlier than the redundancy rule in $\mathcal{A}_1$ did. This led to the replacement of the redundancy rule with a new rule with the same name. This rule is described in Section 5.1.

Secondly, $\mathcal{A}_3$ is able to identify when some computation on a path can be reused during the expansion of another path: if a node which is currently selected for expansion is similar to a non-ancestor node which has been already been expanded, the justification of the latter is reused when dealing with the original node. The new rule which deals with this is called *caching*. This rule is described in Section 5.2.

Section 5.3 introduces the usual notions of complete and clash-free $\mathcal{A}_3$-completion structure, while Section 5.4 shows that the algorithm terminates by computing a bound on the size of an $\mathcal{A}_3$-completion structure: a structure has a maximum number of nodes which is exponential in the size of the input program.

Further on, Sections 5.5 and 5.6 show that the algorithm is sound and complete. While the two new applicability rules are at a first glance not that much different to previous applicability rules they rely on different proof strategies, especially on a different strategy to reduce an infinite model to a finite one (which is part of the completeness proof). As such we consider these proofs to be a main contribution of this work and reproduce them inline.

The usage of the caching rule has improved the worst case running time of the algorithm by one exponential level. The formal complexity analysis can be found in section 5.7.

5.1 Failure: Redundancy

As discussed in section 4.2 the blocking condition is complex enough to not always be fulfilled when exploring a finite number of nodes. The previous algorithm used an extra condition to ensure termination: if a certain number of nodes with equal content had already been explored on a path, there was a failure and the algorithm aborted. Now we introduce a more refined strategy for aborting expansion of a path which is based on the idea that the set of oldest paths running between two nodes with similar content should decrease. While before failure was detected only when reaching a node with exponential depth, the new strategy identifies failure much earlier.

The idea is to keep track of the oldest path in G ('oldest' refers to its starting level w.r.t. the forest) from which every atom makes part and to try to minimize the set of oldest paths running along a path of the forest. Nodes with identical content are allowed on the same path only if every subsequent occurrence of such a node shrinks the set of oldest paths.

A new notation is introduced: by *rank* of an atom a one understands the shallowest depth of a node x such that there exists a unary/binary p/f where $(p(x)/f(x,y), a) \in paths_G$.

Formally:

$$rank(p(x)) = min(\{|x|\} \cup \{rank(a)|(a, p(x)) \in A_G\})$$

$$rank(f(x,y)) = min(\{|y|\} \cup \{rank(a)|(a, f(x,y)) \in A_G\})$$

$$rank(x) = \min_{p \in \mathrm{CT}(x)} rank(p(x))$$

Example 5.1. *Consider again the forest model depicted in Figure 1. Every atom in the model can be reached by a path starting with $smember(x)$. As $smember(x)$ is not reached by any other atom, its rank is equal to its depth, 1. Thus, all atoms in the model have rank 1.*

Example 5.2. *Figure 3 shows an extract from a completion structure in which every predicate p in the content of a node x is augmented with the rank of $p(x)$. The arcs between predicates in the content of some node are arcs in the dependency graph: thus, G contains arcs from $b(x)$ to $a(y)$, $b(y)$, and $c(y)$, respectively. As $rank(b(x)) = 1$ we have that also: $rank(a(y)) = rank(b(y)) = rank(c(y)) = 1$.*

We will denote with $in(k, x)$ the set of incoming paths from level k to node x (presuming $|x| \geqslant k$):

$$in(k, x) = \{p|rank(p, x) = k\}$$

Example 5.3. *For the completion structure depicted in Figure 3 we have that: $in(x, 1) = \{a, b\}$, $in(x, 2) = \{c\}$, and $in(1, y) = \{a, b, c\}$.*

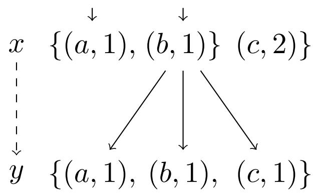


Figure 3: Redundancy: y is redundant as the set of paths coming from the ancestor at depth 1 increases

Formally:

Rule. (ix') 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.:*

- $\text{CT}(x) \subseteq \text{CT}(y)$,
- $rank(x) = rank(y) = r$, *and* $in(r, x) \supseteq in(r, y)$.

The expansion stops when a redundant node is identified.

Intuition: Both strategies for identifying redundant nodes are related to techniques for reducing an infinite forest model to a finite one (used in the completeness proof of the algorithm). In the general case this can be done by considering nodes in the infinite model which are on the same path and have equal content and collapsing the two nodes onto each other by deleting the path between the two nodes (together with all the paths which start with nodes on this path). However, nodes with equal content cannot be indiscriminately collapsed: some extra conditions have to be met in order for the remaining structure to still remain a model.

In the original algorithm, the technique used for reducing a model was to first identify blocking pairs (nodes with equal content with no path running between them) and then collapse nodes with equal content if the set of paths between a 'reference' node and the first node is included or equal within the set of paths between the reference node and the second node. Some extra conditions had to be met for collapsing the two nodes, like there is no blocking node between them. Such conditions can only be checked at 'proof time', but not at 'construction time'. That's why at construction time one could only use the bound established by this technique, but not the technique itself.

The new technique for reducing models using the set of oldest paths traversing a node does not use any reference point when comparing nodes with equal content. Also, except for checking subset inclusion of the set of oldest paths, no extra condition has to be met before collapsing a node into another. This is due to the fact that when reducing a model and scanning a path, first such redundant nodes are identified and collapsed, and then eventually a blocking pair is found. This is guaranteed by the fact that, for infinite paths, by always chasing the set of oldest paths (and exhausting them in a finite number of steps) we reach a point where there are no running paths between two nodes (within finite distance of each other), and due to the infinity of the path we reach two nodes with equal content with this property (within finite distance of each other).

Example 5.4. *Nodes x and y in Figure 3 are such that $\text{CT}(y) \subset \text{CT}(x)$, and the set of oldest paths is expanding when traversing y: $in(1, y) \supset in(1, x)$. Thus, y is redundant.*

Example 5.5. *Consider the FOLP P in example 3.2 and the open answer set depicted in Figure 6. That particular open answer would never be constructed by our algorithm: if one constructs a completion structure*

$$\begin{tikzcd}
x\{(smember, 1), not\ rmember, not\ project\} \arrow[d, "{\{(supportedBy, 1), not\ \ldots)\}}"] \\
y\{(smember, 1), not\ rmember, not\ project\}
\end{tikzcd}$$

Figure 4: x and y form a redundancy pair

for checking satisfiability of smember w.r.t. P, in the style of that particular forest model, one encounters a redundancy node, y. The situation is depicted in Figure 4 (negative predicates do not have ranks): $\text{CT}(x) = \text{CT}(y) = \{smember, not\ rmember, not\ project\}$*,* $rank\ (\ smember\ (x)) = rank(smember(y)) = 1$ *and* $in(1, x) = in(1, y) = \{smember\}$*, and thus, node y is redundant.*

5.2 Caching

Blocking can be generalized to the so-called anywhere blocking or caching where a node reuses the justification/expansion of another node which is not on the same path as itself. Again, the typical condition regarding subset inclusion of the contents of the nodes has to be fulfilled. Additionally, a condition regarding sets of paths running between the common ancestor of the nodes and the nodes themselves has to be fulfilled. Formally:

Rule. (x) Caching. *A node $y \in T \in N_{EF}$ is said to be* cached *if there is a node $x \in T \in N_{EF}$, $y \not<_T x$, $x \not<_T y$, $x \notin cts(P)$, s. t.:*

- $right_T(y, x)$,
- $\text{CT}(y) \subseteq \text{CT}(x)$*, and*
- $connpr_G(z, y) \subseteq connpr_G(z, x)$*, where z is the common ancestor of x and y:* $z = common_T(x, y)$*.*

We call (y, x) a caching pair *and y a* caching node. *A cached node is no longer expanded:* $\text{ST}(x) = exp$

Intuition. Similarly to dealing with blocking pairs, the cached node will be expanded similarly to the caching node. One prerequisite for this is that the content of the cached node is a subset of the content of the caching node.

Like in the case of blocking, the content of the cached node can be justified in two different ways: either by copying the subtree T_x at y or by reusing the successors of x as successors of y. In the first case (depicted in Figure 5), it has to hold that if (u, v) is a blocking pair, with u being a leaf node in T_x, and $v \geqslant_T z$, where $z = common_T(x, y)$, then (u, v') is still a blocking pair, where v' is the copy of v in the new subtree T_y (1). In the second case, the obtained model is no longer forest shaped and one has to check that no cycles are introduced in G (2): this is the approach we take in the Soundness proof and it is described in Figure 5 in Section 5.5. The extra condition $connpr_G(z, y) \subseteq connpr_G(z, x)$ ensures that (1) and (2) hold.

In order for the cached node to not have to reuse its own justification by having a successor of the caching node to be at its turn a cached node, we impose that cached nodes always have to be 'at the right' in the tree of the corresponding caching nodes. Together with this requirement, we enforce the following expansion

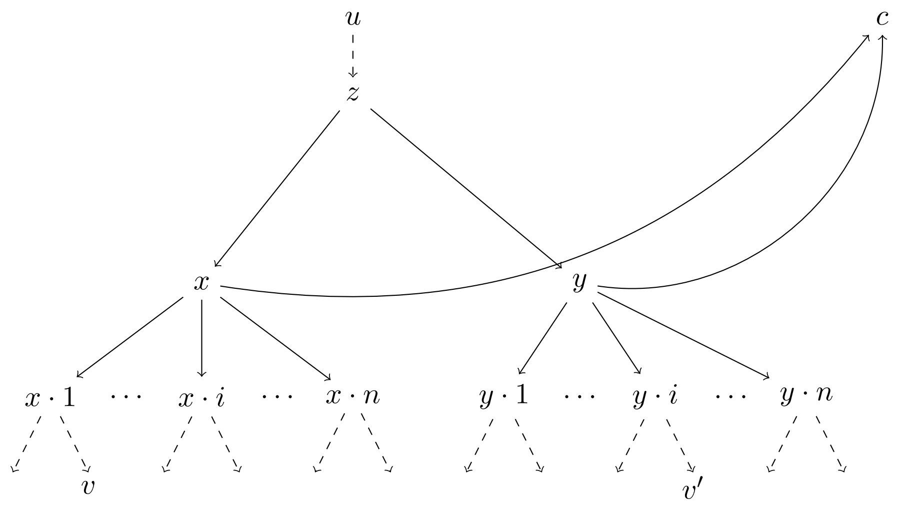


Figure 5: Justifying a cached node y by replicating the justification of its corresponding caching node x

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. $right_T(y, x)$: $\mathrm{ST}(y) = exp$, non-deterministically choose a unit completion structure UC which matches x and perform $expand_{CS}(x, UC)$.*

There is an exponential number of structures of the type $((x_1, r_1), (x_2, r_2), \ldots)$, where $x_1, x_2, \ldots \in upreds(P) \cup not\ upreds(P)$, $r_1, r_2, \ldots \in \overline{0, n}$, x_i-s are distinct, and n is the maximum length of a path (exponential in the size of P - see Proposition 5.9). Any node x can be annotated by such a structure, where for every $p/not\ p \in \mathrm{CT}(x)$: $(p, rank(p(x))/0)$ is a tuple in the structure.

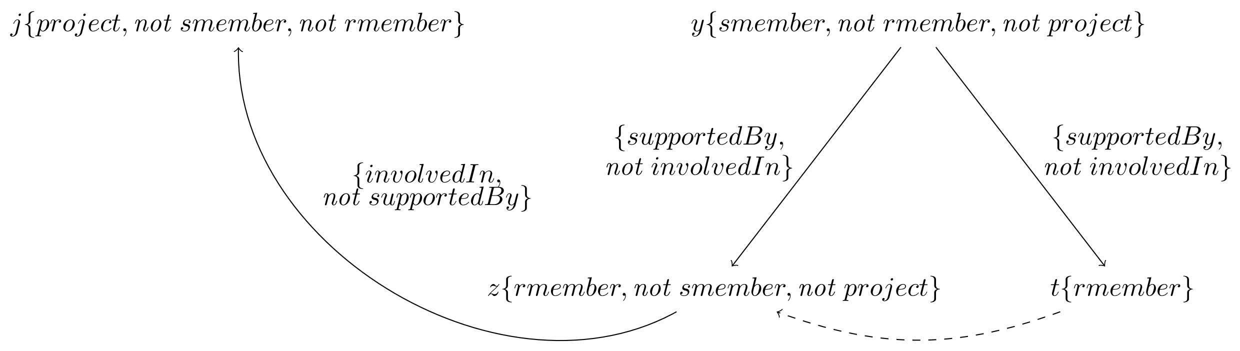
Example 5.6. *Figure 6 shows a completion structure for P from example 3.2 in which every node except t is expanded: note that the completion structure contains no redundancy pair. We have that $y = common_T(z, t)$, $\mathrm{CT}(t) \subset \mathrm{CT}(z)$, and $connpr_G(y, z) = connpr_G(y, t) = (smember, rmember)$, and thus z and t form a caching pair: t will be expanded similar to z either by reusing the successors of z or replicating the expansion of z: note that in this case the two types of justification give the same result as the only successor of z is a constant j (thus, also when replicating the expansion of z, a new successor is not introduced, but j is reused).*

5.3 Complete/Clash-free Completion structures

In this section we redefine the notions of complete completion structure and clash-free completion structure to reflect on the changes introduced by the new applicability rules.

Definition 5.7. *An* $\mathcal{A}_3$-complete completion structure *for a FoLP P and a $p \in upreds(P)$, is an $\mathcal{A}_3$-completion structure that results from the repeated application of the rule* Match *to an initial $\mathcal{A}_3$-completion structure for p and P, taking into account the applicability rules (viii)* Blocking, *(ix')* Redundancy, *and (x)* Caching *s. t. no rules can be further applied.*

As regards clash conditions, the presence of redundant nodes as defined by rule (ix') *Redundancy* constitutes also in this case a clash. Another clash condition is the impossibility to expand an unexpanded node (by finding an appropriate matchable unit completion structure):



$j\{project, not\ smember, not\ rmember\}$
$y\{smember, not\ rmember, not\ project\}$
$\{supportedBy, not\ involvedIn\}$
$\{supportedBy, not\ involvedIn\}$
$\{involvedIn, not\ supportedBy\}$
$z\{rmember, not\ smember, not\ project\}$
$t\{rmember\}$

Figure 6: A completion structure in which (z, t) is a caching pair

Definition 5.8. *A $\mathcal{A}_3$-completion structure $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$ is* clash-free *if there is no redundant node in EF and for every $x \in N_{EF}$: $\text{ST}(x) = exp$.*

An overview of the algoritm $\mathcal{A}_3$ for checking satisfiability of p w.r.t. a FoLP P is provided by Algorithm 1.

5.4 Termination

In this section we show that the algorithm $\mathcal{A}_3$ terminates: first we compute a bound on the path length in any $\mathcal{A}_3$-completion structure, and then, using this result, we compute a bound in the total number of nodes in any $\mathcal{A}_3$-completion structure. Both bounds are exponential in the size of the input FoLP P. The latter result is a direct consequence of employing the caching rule.

Proposition 5.9. *Every path in an $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

Proof. We show that any path has at most $n2^{2n}$ nodes, where $n = |upreds(P)|$.

There is a finite amount of nodes with different contents: 2^n, on any path in the completion structure and in the completion structure itself. As such, there are at least $n2^n$ nodes with equal content on any path which contains $n2^{2n}$ nodes. Let $x_1 < \ldots < x_{n2^n}$ be a sequence of such nodes and let $(r_l)_{1 \leqslant l \leqslant n}$ be the ordered sequence of ranks of unary predicates in $\text{CT}(x_1)$: $r_l \in \{k \mid p \in \text{CT}(x_1) \wedge rank(p, x_1) = k\}|$, for $1 \leqslant l \leqslant n$, and $r_l \geqslant r_{l+1}$, for $1 \leqslant l < n$. As some predicates might have equal ranks, and thus $r = |\{k \mid p \in \text{CT}(x_1) \wedge rank(p, x_1) = k\}| < n$, we take $r_i = |x_1|$, for every $i > r$. We show that $rank(x_{j2^n}) > r_j$, for every $1 \leqslant j \leqslant n$ by induction.

Base case: $j = 1$. We have that $rank(x_i) \geqslant rank(x_1) = r_1$, for $1 \leqslant i \leqslant 2^n$, and (x_i, x_k) is neither a blocking nor a caching pair, for any $1 \leqslant i < k \leqslant 2^n$. Assume that $rank(x_{2^n}) = r_1$. Then $rank(x_i) = r_1$, for $1 \leqslant i \leqslant 2^n$, and $in(x_1, r) \not\subseteq in(x_k, r)$, for any $1 \leqslant i < k \leqslant 2^n$. But $|\{S \mid S = in(x_i, r),$ for some $x_i \in N_{EF}$ and $r \in \mathbb{N}\}| = 2^n$, which contradicts with the previous statement. Thus, the original assumption was false and $rank(x_{2^n}) > r_1$.

Induction case: if $rank(x_{2^j}) > r_j$, for a certain $1 \leqslant j < n$, one can bring a similar argument to the one from the base case to show that $rank(x_{2^{j+1}}) > r_{j+1}$.

```
input  : FoLP $P$, unary predicate $p$;
output: checks satisfiability of $p$ w.r.t.$P$;
1) Construct the set of Unit Completion Structures (UCSs) for $P$ (if not constructed already):;
    To construct a UCS: a) Construct an initial unit completion structure for $p$ w.r.t. $P$ as in Definition
    4.3;
    b) Apply expansion rules (i)-(vi) introduced in [Feier and Heymans, 2009] until the conditions in
    Definition 4.4 are met.;
2) Construct an $\mathcal{A}_2$-initial completion structure for $p$ w.r.t. $P$ as in Definition 4.2;
3) For every $x \in N_{EF}$ apply one of the followings rules (in decreasing order of priority) (we assume
$EF$ is explored in a depth-first fashion): ;
    a) if there is an ancestor $y$ of $x$: $y <_F x$, $y \notin cts(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$, and
    $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty then
        $x$ is blocked;
    end
    b) if there is an ancestor $y$ of $x$ in $F$, $y <_F x$, $y \notin cts(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$,
    $rank(x) = rank(y) = r$, and $in(r, x) \supseteq in(r, y)$ then
        $x$ is redundant: return false;
    end
    c) if there is a node $y \in T \in N_{EF}$, $y \not<_T x$, $x \not<_T y$, $y \notin cts(P)$, s. t. $right_T(x, y)$, and
    $\text{CT}(x) \subseteq \text{CT}(y)$, and $connpr_G(z, x) \subseteq connpr_G(z, y)$, where $z = common_T(x, y)$ then
        $x$ is cached;
    end
    d) if $\text{ST}(x) = unexp$ and for every node $y$ s.t. $right_T(y, x)$: $\text{ST}(y) = exp$ then
        non-deterministically choose a unit completion structure $UC$ which matches $x$ and perform
        $expand_{CS}(x, UC)$ (Match');
    end
4) if for every node $x \in N_{EF}$: $st(x) = exp$ then
    return true
end
return false.
```

Algorithm 1: Overview of $\mathcal{A}_3$.

As such, $rank(x_{j2^n}) > r_j$, for every $1 \leqslant j \leqslant n$, and in particular, $rank(x_{n2^n}) > r_n$, for every $1 \leqslant j \leqslant n$. As $rank(p, x_1) \leqslant r_n$, for every $p \in \text{CT}(x_1)$, it results that $rank(p, x_1) < rank(x_{n2^n})$, for every $p \in \text{CT}(x_1)$. This translates in the fact that the set of oldest paths in G traversing x_{n2^n} started at a node below x_1, and thus there are no paths in G running between x_1 and x_{n2^n}. As $\text{CT}(x_1) = \text{CT}(x_{n2^n})$, this implies that (x_1, x_{n2^n}) is a blocking pair and thus x_{n2^n}, being a blocked node is the last node on the path. This reasoning applies to every possible content for a node, thus in case $n > 1$, we achieve that there have to be less than $n2^{2n}$ nodes on every path: otherwise, there is a blocking node for every possible type of content for a node, which contradicts the fact that a path has at most one blocking node.

Furthermore, one can show that after an exponential number of steps, one always reaches a complete completion structure. Note that in the previous version of the algorithm a complete completion structure had in the worst case a double exponential number of node in the size of the program. Now, due to caching, the complexity drops one exponential level.

Proposition 5.10. *A complete $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

Proof. Interestingly, the additional condition concerning paths running between the common ancestor and the two nodes in a caching pair can be reformulated in a condition regarding inclusion of the intersections of sets of paths running through the tree with the two nodes ordered by their ranking.

$$connpr_G(z, x) \subseteq connpr_G(z, y) \text{ iff } in(r, x) \subseteq in(r, y), \text{for every } r \leqslant rank(z)$$

This property enables us to obtain an exponential bound on the number of nodes in any complete completion structure using the three applicability rules. To do this we overestimate the number of nodes, by making caching even harder by imposing an even stricter condition: $in(r, x) \subseteq in(r, y)$, for every $r \geqslant 0$.

We count how many structures of the type $((x_1, r_1), (x_2, r_2), \ldots)$ are, where $x_1, x_2, \ldots \in upreds(P) \cup not\ upreds(P)$, $r_1, r_2, \ldots \in \overline{0, n}$, x_i-s are distinct, and n is a natural number exponential in the size of P (the maximum length of a path in a completion structure - see Proposition 5.9), or, in other words, the number of possible node contents annotated with the rank of every predicate in the content (predicates which appear negated in the content of some node are annotated with 0). This is equal to the number of functions $f : 2^{upreds}(P) \rightarrow \overline{0, n} \cup \overline{0, n}^2 \cup \ldots \overline{0, n}^{|upredsP|}$ such that $f(x) \in \overline{0, n}^{|x|}$, which at its turn is exponential in the size of P.

Assume there are two distinct nodes with identical annotation structures as described above. If they are on the same path, they form a redundant pair, otherwise they form a caching pair.

5.5 Soundness

Proposition 5.11 (soundness)**.** *Let P be a FoLP and $p \in upreds(P)$. If there exists a complete clash-free completion structure for p w.r.t. P, then p is satisfiable w.r.t. P.*

Proof. From a clash-free complete completion structure for p w.r.t. P, we construct an open interpretation, and show that this interpretation is an open answer set of P that satisfies p. Let $\langle EF, \text{CT}, \text{ST}, G\rangle$ be such a clash-free complete completion structure with $EF = \langle F, ES\rangle$ the extended forest and $G = (V, A)$ the corresponding dependency graph and let bl and ch be the sets of blocking pairs and caching pairs corresponding to the completion. Let $blocked$ and $cached$ be the sets of blocked and cached nodes respectively: $blocked = \{y \mid (x, y) \in bl\}$ and $cached = \{y \mid (x, y) \in ch\}$.

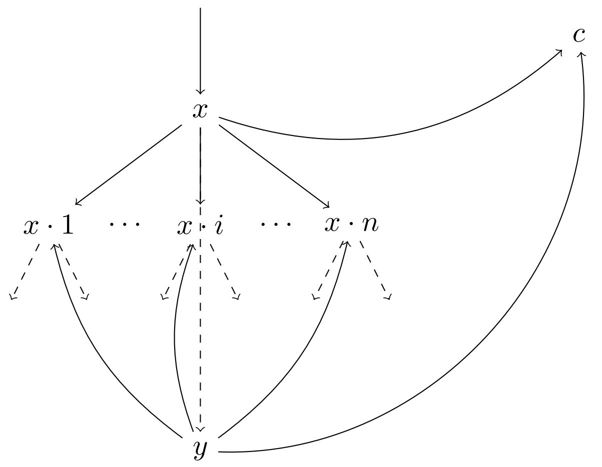


Figure 7: Justifying a blocked node y by reusing the successors of its corresponding blocking node x

1. *Construction of open interpretation.*

   We construct a new graph $G_{ext} = (V_{ext}, A_{ext})$ by extending G in the following way: for every pair of blocking/caching nodes, the content of the blocking/caching node is copied into the content of the blocked/cached node, and all connections from the blocking/caching node to its successors or within itself are replicated by connections from the blocked/cached node to the successors of the blocking/caching node or within itself (or, in other words, the content of the blocked/cached node is identical with the content of the blocking/caching node and it is motivated in a similar way). The underlying forest is also extended with arcs from the blocked/cached node to all successors of the blocking/caching node. Formally:

   - $V_{ext} = V \cup \{a_{x|y} \mid a \in V \wedge args_1(a) = x \wedge (x, y) \in bl \cup ch\}$;
   - $A_{ext} = A \cup \{(a_{x|y}, b_{x|y}) \mid (a, b) \in A \wedge args_1(a) = x \wedge (x, y) \in bl \cup ch\}$;
   - $A_{EF}^{ext} = A_{EF} \cup \{(y, z) \mid (x, y) \in bl \cup ch \wedge (x, z) \in A_{EF}\}$.

Lemma 5.12. *Let $(x, y) \in bl \cup ch$ and $G_{ext} = (V_{ext}, A_{ext})$ constructed as described above. Then, for any ground rule $r \in P_{N_{EF}}$: $V_{ext} \models r$ iff $V_{ext} \models r_{x|y}$ iff $V_{ext} \models r_{y|x}$.*

Proof. By construction of V_{ext}. $\square$

Lemma 5.13. *Let $UC = \langle EF, \text{CT}, G\rangle$ be a unit completion structure for a FoLP P with $EF = (\{T_\varepsilon\}, ES)$, and $G = (V, A)$. Then, the open interpretation induced by UC: (N_{EF}, V), is an open answer set of the program: $\cup_{r \in P} r_{args_1(head(r))|\varepsilon}$. This is equivalent to $V \models \cup_{r \in P_{N_{EF}}} r_{args_1(head(r))||\varepsilon}$, or, in other words, the set of atoms induced by UC satisfies the grounding of P with elements from N_{EF} s.t. the first term in the head of each ground rule is ε.*

Proof. By construction of a unit completion structure. $\square$

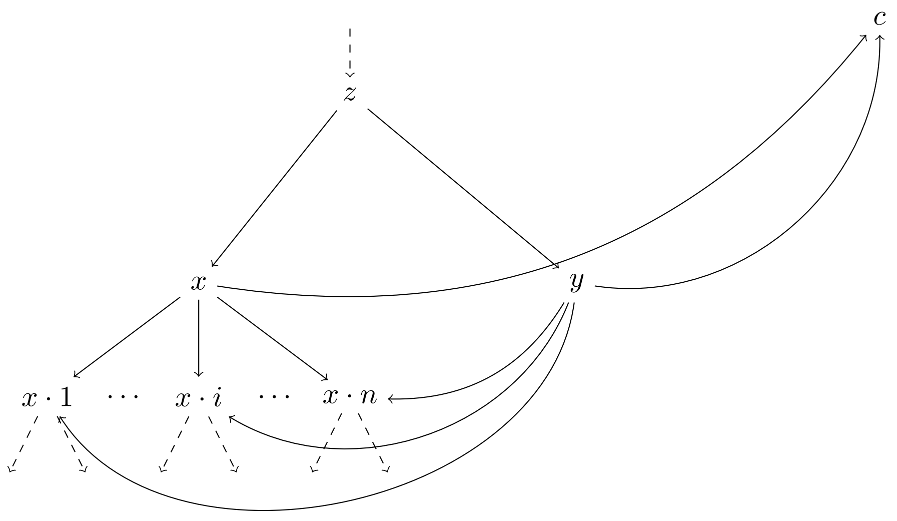


Figure 8: Justifying a cached node y by reusing the successors of its corresponding caching node x

Let there be an open interpretation (U, M), with $U = N_{EF}$, i.e., the universe is the set of nodes in the extended forest, and $M = V_{ext}$, i.e., the interpretation corresponds to the set of nodes in the extended graph.

2. *M is a model of P_U^M.* First of all let's note that $M \models P_U^M$ iff $M \models P_U$. We will show that $M \models P_U$.

   Let's note that $P_U = \cup_{x \in U} \cup_{r \in P_U} r_{args_1(head(r))||x}$.

   For every node $x \in U$ we will show that $M \models \cup_{r \in P_U} r_{args_1(head(r))|x}$:

   - (i) suppose $x \notin \mathit{blocked} \cup \mathit{cached}$. Then, at some point in the construction of CS, x has been expanded by replacing it with a unit completion structure $UC = \langle EF', \text{CT}', G' \rangle$, where $G' = (V', A')$. According to Lemma 5.13, $V' \models \cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||\varepsilon}$. Let $V'' = \{a_{\varepsilon||x} \mid a \in V'\}$. Then $V'' \models \cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||x}$. As $V'' \subseteq V$, $V \subseteq M$, and $\cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||x} = \cup_{r \in P_U} r_{args_1(head(r))||x}$, so $M \models \cup_{r \in P_U} r_{args_1(head(r))||x}$.
   - (ii) suppose $x \in \mathit{blocked} \cup \mathit{cached}$. Then, according to Lemma 5.12, for every $r \in P_U$: $M \models r$ iff $M \models r_{x|y}$, where y is the corresponding blocking or caching node. That $M \models r_{x|y}$ follows from case (i).

3. *M is a minimal model of P_U^M.* Before proceeding with the actual proof we introduce a notation and a lemma which will prove useful in the following. Let $EF^{'}$ be the directed graph $(N_{EF}, A^{'})$ which has as nodes all the nodes from EF and as arcs all the arcs of EF plus some 'extra' arcs which point from blocked/cached nodes to successors of corresponding blocking/caching nodes: $A^{'} = A_{EF} \cup \{(y, z) \mid \exists x \text{ s. t. } (x, y) \in \mathit{bl} \cup \mathit{ch} \wedge z \in \mathit{succ}_{EF}(x)\}$. The new graph captures in a more accurate way the structure of M: blocked/cached nodes are connected to successors of the corresponding blocking nodes, as their contents is justified similarly as the content of the blocking/caching nodes.

The following lemma associates paths in the dependency graphs G/G_{ext} to paths in the underlying extended forest: $EF/EF^{'}$. It basically says that by projecting a path in the dependency graph on the arguments of every atom in the path and eliminating all binary arguments and redundant unary arguments one obtains a path in the extended forest.

Lemma 5.14. *Let $Pt = (a_1, \ldots, a_n) \in paths_G/paths_{G_{ext}}$, with $pred(a_1) \in upreds(P)$, and $T_1 = (args(a_{i_1}), \ldots, args(a_{i_m}))$ be a tuple obtained by selecting all and only the unary atoms in Pt_1 in the order they appear in Pt and retaining only their argument: $1 \leqslant i_j \leqslant n$, $i_j < i_{j+1}$, for every $1 \leqslant j \leqslant m$, and $pred(a_k) \in upreds(P)$ iff there exists $1 \leqslant j \leqslant m$ such that $k = i_j$, for every $1 \leqslant k \leqslant n$. Then, the tuple obtained by eliminating consecutive duplicates in T_1, $T_2 = (b_1, \ldots, b_p)$, where for every $1 \leqslant j \leqslant p$, there exists $1 \leqslant k \leqslant m$ such that $b_j = args(a_{i_k})$ and $args(a_{i_k}) \neq args(a_{i_{k-1}})$ is a path in $EF/EF^{'}$: $T_2 \in paths_{EF}/paths_{EF^{'}}$. We will also call T_2, the* argument path *of Pt and denote it with $argpath(Pt)$.*

Furthermore, if Pt_1 is a cycle in G/G_{ext}, than T_2 is a cycle in $EF/EF^{'}$.

Proof. We construct a sequence of pairs of indexes $((k_1, q_1), \ldots, (k_{p-1}, q_{p-1}))$ such that k_i is the greatest index for which $args(a_{k_i}) = b_i$ and q_i is the smallest index for which $args(a_{k_i}) = b_{i+1}$, for every $1 \leqslant i < p$.

Then, we consider subpaths of Pt of the form $(a_{k_i}, \ldots, a_{q_i})$, for $1 \leqslant i < p$. Every such subpath has the form: $(p(b_i), f_1(b_i, b_{i+1}), \ldots, f_s(b_i, b_{i+1}), q(b_{i+1}))$, with $p, q \in upreds(P)$, $f_1, \ldots, f_s \in bpreds(P)$, and $s \geqslant 1$. Thus: $(b_i, b_{i+1}) \in A/A^{'}$ for every $1 \leqslant i < p$: T_2 is a path in $EF/EF^{'}$.

If Pt is a cycle then $a_1 = a_n$. By construction of T_2, $b_1 = b_n = args(a_1)$. $\Box$

Now we can proceed to the actual proof of statement. Assume there is a model $M' \subset M$ of $Q = P_U^M$. Then $\exists l_1 \in M : l_1 \notin M'$. Take a rule $r_1 \in Q$ of the form $l_1 \leftarrow \beta_1$ with $M \models \beta_1$; note that such a rule always exists by construction of M and expansion rule (i) . If $M' \models \beta_1$, then $M' \models l_1$ (as M' is a model), a contradiction. Thus, $M' \not\models \beta_1$ such that $\exists l_2 \in \beta_1 : l_2 \notin M'$. Continuing with the same line of reasoning, one obtains an infinite sequence $\{l_1, l_2, \ldots\}$ with $(l_i \in M)_{1 \leqslant i}$ and $(l_i \notin M')_{1 \leqslant i}$. M is finite (the complete clash-free completion structure has been constructed in a finite number of steps, and when constructing $M(V_{ext})$ we added only a finite number of atoms to the ones already existing in V), thus there must be $1 \leqslant i, j$, $i \neq j$, such that $l_i = l_j$. We observe that $(l_i, l_{i+1})_{1 \leqslant i} \in E_{ext}$ by construction of E_{ext} and expansion rule (i), so our assumption leads to the existence of a cycle in G_{ext}.

Assume G_{ext} contains a cycle $C = (a_1, \ldots, a_n = a_1)$. Then, potentially, the cycle falls into one of the following categories:

- 'local' cycles: cycles in which all unary atoms have identical arguments or there are no unary atoms.
- 'blocking' cycles: non-local cycles which do not contain unary atoms having as arguments cached nodes.
- 'caching' cycles: non-local cycles which have as arguments cached nodes.

Note that G does not contain any cycle (by construction), so every cycle in G_{ext} has to be a result of the introduction of new nodes/arcs in G: as such, each cycle should contain at least an atom having as one of its arguments a blocked or cached node. We will show by reductio ad absurdum that each of these types of cycles cannot appear in G_{ext}. In the following we consider only elementary cycles as in the absence of elementary cycles there cannot be any cycles whatsoever.

Lemma 5.15. *There are no (elementary) local cycles in G_{ext}.*

Proof. Assume $C = (a_1, \ldots, a_n = a_1)$ is an (elementary) local cycle in G_{ext}. Then C contains only atoms of the form $p(x)$, and/or $f(x, y)$, for $p \in upreds(P)$, $f \in bpreds(P)$, and $x, y \in N_{EF}$. Assume $x \in blocked/cached$. Then let $z \in N_{EF}$ be such that $(x, z) \in bl/ch$. Then $C_{x|z} = ((a_1)_{x|z}, \ldots, (a_n)_{x|z} = (a_1)_{x|z})$ is a cycle in G. Contradiction with the fact that there are no cycles in G. □

To show that there are no elementary blocking cycle in G_{ext} we employ a three step process: the first two steps restrict the set of possibly cycles by constraining the structure of the argument path of such a cycle (lemmas 3 and 3).

Lemma 5.16. *There is no elementary cycle C in G_{ext} such that its argument path contains a blocking path from EF: for every $x, y \in bl$ such that $x, y \in T$, $path_T(x, y) \not\subseteq argpath(C)$.*

Proof. Assume $path_T(x, y) \subseteq argpath(C)$. Then, there are two nodes $a_1, a_2 \in G$, with $args(a_1) = x$, and $args(a_2) =$ and a path $Pt \in paths_G(a_1, a_2)$ such that $Pt \in C$. But this contradicts with the fact that $connpr_G(x_1, x_2) = \emptyset$. Thus, the initial assumption was false. □

Lemma 5.17. *Let C be an elementary cycle in G_{ext} which contains a node y such that $(x, y) \in bl$ and $x, y \in T$. Then, $path_T(z, y) \in argpath(C)$, where $z = x \cdot i$ and $z < y$.*

Proof. Assume the opposite. As $y \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y) \in argpath(C)$. Let $z_2 \in path_T(succ_T(z), y)$ be such that $path_T(z_2, y) \in argpath(C)$ and $(prev_T(z_2), z_2) \notin argpath(C)$. Every node in $path_T(z, y)$, including z_2, has as incoming arcs the arc from its predecessor in EF and possibly blocking arcs. As $(prev_T(z_2), z_2) \notin argpath(C)$, z_2 has an incoming blocking arc: let (y_2, z_2) be such an incoming blocking arc, where $y_2 \in T$ and let $x_2 \in T$ be the corresponding blocking node: $(x_2, y_2) \in bl$. As (y_2, z_2) is a blocking arc, it means that $z_2 \in succ_T(x_2)$, or in other words $x_2 = prev_T(z_2)$. As $z_2 \in path_T(succ_T(z), y)$, it implies that $x_2 \in path_T(z, prev_T(y))$.

As $y_2 \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y_2) \in argpath(C)$. From lemma 3 we know that $path_T(x_2, y_2) \not\subseteq argpath(C)$. Thus, there is a node $z_3 \in path_T(succ_T(x_2), y_2)$ such that $path_T(z_3, y_2) \in argpath(C)$ and $(prev_T(z_3), z_3) \notin argpath(C)$. Like before in the case of z_2, z_3 has an incoming blocking arc (y_3, z_3) with $(x_3, y_3) \in bl$. In this case: $x_3 \in path_T(x_2, prev_T(y_2))$.

The process repeats itself ad infinitum: figure 9 describes it in the general case. One obtains a sequence of tuples (x_i, y_i, z_i) such that $(x_i, y_i) \in bl$, $x_{i+1} \in path_T\ (x_i, prev_T(y_i))$, $z_i = succ_T(x_i)$, $path_T(z_{i+1}, y_i) \in argpath(C)$, and $(y_i, z_i) \in argpath(C)$. If $y_i \neq y_j$, for every $i \neq j$, one has an infinite number of nodes in EF which is a contradiction with the fact that there is a finite amount of nodes in EF.

If however there exist $l < k$ such that $y_k = y_l$, then $x_l = x_{l+1} = \ldots x_k$, and $(y_k, z_k)\hat{}\ path_T(z_k, y_k - 1)\hat{}\ \ldots\ \hat{}\ (y_{l+1}, z_l)\hat{}\ path_T(z_{l+1}, y_l = y_k)$ is a cycle in G_{ext} strictly included in C. This contradicts with the fact that C is an elementary cycle (see Figure 10).

$$\begin{tikzcd}
 & & & x \arrow[dl] & \\
 & & x_2 \arrow[dl] \arrow[d] & & \\
 & z_2 \arrow[ddl] & x_3 \arrow[d] \arrow[dr] & & \\
 & & z_3 \arrow[d] & x_4 \arrow[dr] \arrow[r, dashed] & {} \\
y & & y_2 \arrow[uul] & & y_3 \arrow[ull]
\end{tikzcd}$$

Figure 9: Splitting blocking paths: infinite division

$$\begin{tikzcd}
x \arrow[dd] & & y_3 \arrow[dddl, thick] \\
 & z_4 \arrow[ur, thick] & \\
x_2, x_3, x_4 \arrow[ur] \arrow[dr] \arrow[dd] & & \\
 & z_3 \arrow[ddr, thick] & \\
z_2 \arrow[d] & & \\
y & & y_2, y_4 \arrow[ul, "" ', shift left=0] \arrow[ull] \arrow[uuuul, thick]
\end{tikzcd}$$

Figure 10: Splitting blocking paths: the case where $x_2 = x_3 = x_4$ and $y_2 = y_4$: z_4, y_3, z_3, y_2, z_4 form a cycle

$$\begin{tikzcd}
 & x \arrow[dl, dashed] \arrow[d] \\
t \arrow[r] & z \arrow[d] \\
 & y
\end{tikzcd}$$

Figure 11: If (t, z) is a blocking arc, (x, t) is a blocking pair, and the content of (t, z) is justified similarly to the content of (x, z), then $paths_G(t, y) = paths_G(x, y)$

Thus, in both cases we obtained a contradiction, and the initial assumption was false.

Lemma 5.18. *There are no elementary blocking cycles in G_{ext}.*

Proof.

Assume $C = (a_1, \ldots, a_n = a_1)$ is a blocking cycle in G_{ext} which contains a blocked node $y \in T$: $(x, y) \in bl$. Then, from lemma 5.14 $argpath(C)$ is a cycle in $EF^{'}$. Also, according to lemma 3 $argpath(C)$ does not contain $path_T(x, y)$, but according to lemma 3 it does contain $path_T(z, y)$, where $z = succ_T(x)$. As $z \in argpath(C)$, $argpath(C)$, and $(x, z) \not\subseteq argpath(C)$ there has to be a blocking arc of the type $(t, z) \in argpath(C)$. The situation is described in Figure 11.

Due to the construction of $argpath(C)$, there have to be unary predicates $p, q, r \in upreds(P)$ such that $(p(t), q(z)) \in A_{ext}$ and $(q(z), r(y)) \in conn_G$. But, as $prev_T(z) = x$, (x, t) is a blocking pair, and $(p(t), q(z)) \in A_{ext}$ implies $(p(x), q(z)) \in A$. Together with $(q(z), r(y)) \in conn_G$ it implies that $(p(x), q(z)) \in conn_G$, and thus $(p, q) \in connpr_G(x, y)$, which is in contradiction with $(x, y) \in bl$.

Thus, the initial assumption was false and G_{ext} does not contain any elementary blocking cycle.

Lemma 5.19. *Let C be an elementary caching cycle in G_{ext} for which $argpath(C)$ contains a cached node y such that $(x, y) \in ch$ and for every pair $(s, t) \in ch$, with $s, t \in T$: $right_T(s, x)$ or $t \notin C$. Then, $path_T(z \cdot i, y) \subseteq argpath(C)$, where $z = common_T(x, y)$ and $z < z \cdot i \leqslant y$.*

Proof.

Assume that $argpath(C)$ does not contain $path_T(z \cdot i, y)$. As $y \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y) \in argpath(C)$. Let $z_2 \in path_T(succ_T(z \cdot i), y)$ be such that $path_T(z_2, y) \in argpath(C)$ and $(prev_T(z_2), z_2) \notin argpath(C)$. Every node in $path_T(z, y)$, including z_2, has as incoming arcs the arc from its predecessor in EF and possibly blocking and/or caching arcs. $(prev_T(z_2), z_2) \notin argpath(C)$, so z_2 must have either an incoming caching arc or an incoming blocking arc which is part of $argpath(C)$.

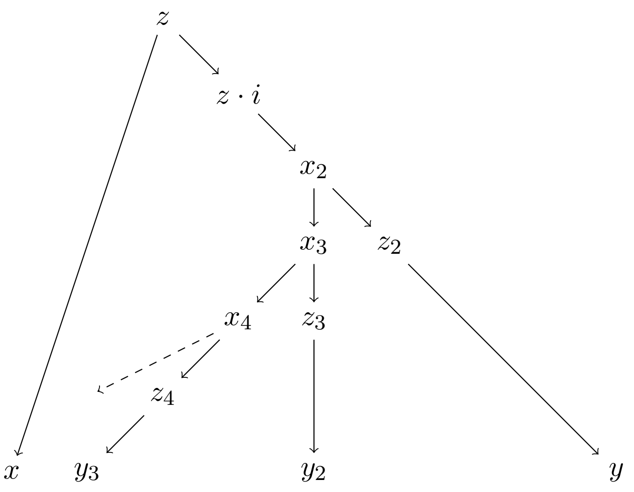


Figure 12: Splitting blocking paths in a potential caching cycle

- (i) Assume z_2 has an incoming caching arc (*). Then, there is a caching pair $(prev_T(z_2), t)$ and $prev_T(z_2)$ is a caching node. We have that $prev_T(z_2) \in path_T(z \cdot i, prev_T(y))$ and thus $z_2 \geqslant_T z \cdot i$. At the same time $x > z$ and due to the expansion and caching strategy $right_T(y, x)$. Thus: $x = z \cdot j \cdot s$, for some $s \in \langle \mathbb{N}^* \rangle$, and $j \in \mathbb{N}^*$, so it holds that $right_T(z_2, x)$. This in contradiction to the fact that x is the right-est caching node in $argpath(C)$. Thus (*) was false, and there are no incoming caching arcs to z_2 which are part of $argpath(C)$.
- (ii) Assume z_2 has an incoming blocking arc. Then, there is a blocking pair $(x_2 = prev_T(z_2), y_2)$ and x_2 is a blocking node with: $x_2 \in path_T(z \cdot i, prev_T(y))$. As there is no cycle which contains a blocking path, $path_T(x_2, y_2)$ is not part of the cycle. The argument follows similarly to the argument in the proof of lemma 3: a sequence of tuples (x_i, y_i, z_i) is constructed with similar properties as in the other lemma. This situation is described in Figure 12. The only difference to lemma 3, is that we always have to show that (x_i, y_i) cannot be a caching path, which is done similarly to item (i). As in the proof of lemma 3 we eventually reach a contradiction.

Thus, in both cases we reach a contradiction, and the original assumption was false: $path_T(z \cdot i, y) \subseteq argpath(C)$. $\Box$

Lemma 5.20. *Let C be an elementary caching cycle in G_{ext} such that its argument path contains $n+1$ distinct caching nodes, for $n \in \mathbb{N}$. Then, there is an elementary cycle in G_{ext} such that its argument path contains n distinct caching nodes.*

Proof. Let $Pt = argpath(C)$. Let $(x, y) \in ch$ such that: $y \in Pt$, and for every pair $(s, t) \in ch$, with $s, t \in T$: $right_T(s, x)$ or $t \notin C$. Then, according to lemma 5.19: $path_T(z \cdot i, y) \subseteq Pt$, where $z = common_T(x, y)$. In the following we show how to transform C in a cycle which does not contain y. (x, y) is a caching pair, so there must be a successor of x in T, $x \cdot j$, such that $(y, x_j) \subseteq Pt$.

There are two distinct cases:

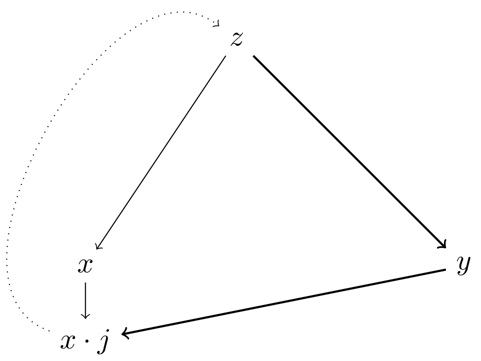


Figure 13: Reducing the number of cached nodes which appear in atoms in a cycle of G: $(x, y) \in ch$ and y is eliminated.

- (i) $path_T(z, y) \subseteq Pt$: let $Pt_1 = path_T(z, y)\hat{}(y, x \cdot j)$ and $Pt_2 = path_T(z, x \cdot j)$. Then, for every path $Pg_1 \in paths_{G_{ext}}(p(z), q(x \cdot j))$, for some $p, q \in upreds(P)$, such that $argpath(Pg_1) = Pt_1$, there is a path $Pg_2 \in paths_{G_{ext}}(p(z), q(x \cdot j))$, such that $argpath(Pg_2) = Pt_2$. This follows from the fact that $connpr_G(z, y) \subseteq connpr_G(z, x)$ and $connpr_G(y, x \cdot j) = connpr_G(x, x \cdot j)$ (from the caching condition and construction of G_{ext}).

  Thus, a path $Pg \in paths_{G_{ext}}$ with $argpath(Pg) = path_T(z, y)\hat{}(y, x \cdot j)\hat{}R$, for some $R \in paths_{G_{ext}}$, is a cycle iff there is another path $Pg' \in paths_{G_{ext}}$ with $argpath\ (Pg') = path_T(z, y)\hat{}(y, x \cdot j)\hat{}R$, which is is a cycle. $argpath(Pg')$ does not contain cached node y and does not introduce any new cached node, so, it decreases the number of cached nodes in the cycle. Figure 13 depicts this case: the thick lines are the part from $argpath(Pg)$ which is replaced with $path_T(z, x \cdot j)$.

- (ii) $path_T(z \cdot i, y) \subseteq Pt$, but $path_T(z, y) \not\subseteq Pt$: in this case $z \cdot i$ has an incoming blocking or caching arc $(t, z \cdot i)$, which translates in its predecessor z being a blocking or caching node and $(z, t) \in bl \cup ch$. In either of the cases, $(t, z \cdot i)$ is justified similarly to $(z, z \cdot i)$ and thus one can obtain an equivalent cycle by substituting t with z in C. The new cycle $C' = C_{t|y}$ fulfills the condition that $path_T(z, y) \subseteq argpath(C')$ and $path_T(z, y) \subseteq Pt$ and thus falls into case (i). Figure 14 depicts this case: the thick lines are the part from $argpath(C)$ which is replaced with $path_T(z, z \cdot i)$.

Lemma 5.21. *There are no caching cycles in G_{ext}.*

Proof. Assume C is a caching cycle in G_{ext} which contains n caching nodes. Then, by repeated application of lemma 5.20, one obtains a cycle with 0 caching nodes, thus a cycle which is either a blocking or local cycle. According to lemmas 5.18 and 5.15 there are no such cycles in G_{ext}, thus the initial assumption is false, and there are no caching cycles in G_{ext}. □

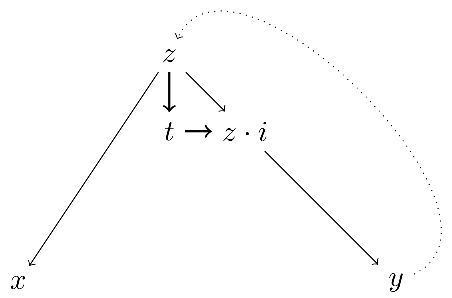


Figure 14: Reducing the number of cached nodes which appear in atoms in a cycle of G: reducing a cycle C in which $(z, z \cdot i) \not\subseteq argpath(C)$ to a cycle in which $(z, z \cdot i) \subseteq argpath(C)$.

5.6 Completeness

Proposition 5.22 (completeness)**.** *Let P be a FoLP and $p \in upreds(P)$. If p is satisfiable w.r.t. P, then there exists a complete clash-free completion structure for p w.r.t. P.*

Proof.

First, we introduce an operation which replaces the node y of a completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$, where $EF = (F, ES)$ and $G = (V, A)$, with a matchable unit completion structure $UC = \langle EF' = (F', ES'), \textsc{ct}', \textsc{st}', G'\rangle$ with root ε. The result of the operation is a new completion structure obtained by (i) deleting $T_c[y]$ from CS, where $y \in T_c$, and (ii) adding UC instead using the $expand$ operation introduced in section 4.2, and is denoted with $replace_{CS}(y, UC)$.

i) The removal of $T_c[y]$ transforms CS as follows:

- $ES = ES - \{(u, v) \mid u \in T_c[y]\}$;
- $\textsc{ct}$ and $\textsc{st}$ are undefined for $\{u \mid u \in T_c[y]\} \cup \{(u, v) \mid u \in T_c[y]\}$;
- $V = V - \{a \mid args_1(a) \in T_c[y]\}$, $A = A - \{(a, b) \mid args_1(a) \in T_c[y]\}$;
- $T_c = T_c - T_c[y]$.

ii) The addition of UC: $expand_{CS}(x, UC)$.

If p is satisfiable w.r.t. P then p is forest-satisfiable w.r.t. P (Proposition 3.4). We construct a clash-free complete completion structure for p w.r.t. P, by guiding the application of the match, blocking, caching, and redundancy rules with the help of a forest model of P which satisfies p. The proof is inspired by completeness proofs in Description Logics for tableau, for example in [I. Horrocks and Tobies, 1999], but requires additional mechanisms to eliminate redundant parts from Open Answer Sets.

Lemma 5.23. *Let (U, M) be a forest model for a FoLP P, with the extended forest $EF = \langle\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES\rangle$, and the labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$ as in definition 3.3. Then, for every node $x \in U$, there is a unit completion structure for P: $UC = \langle EF', \textsc{ct}, G\rangle$, with $EF' = (\{T_{\varepsilon'}\}, ES')$, and $G = (V, A)$, which satisfies the following:*

- *$y \in N_{EF'}$ iff $y_{\varepsilon'||x} \in N_{EF}$;*
- *$(\varepsilon', y) \in A_{EF'}$ iff $(x, y_{\varepsilon'||x}) \in A_{EF'}$;*
- *$\textsc{ct}(\varepsilon') = \mathcal{L}(x) \cup not\ (upreds(P) - \mathcal{L}(x))$;*
- *$\textsc{ct}(y) \subseteq \mathcal{L}(y_{\varepsilon'||x}) \cup not\ (upreds(P) - \mathcal{L}(y_{\varepsilon'||x}))$, for every $y \in N_{EF'}$;*
- *$\textsc{ct}(\varepsilon', y) = \mathcal{L}(x, y_{\varepsilon'||x}) \cup not\ (upreds(P) - \mathcal{L}(x, y_{\varepsilon'||x}))$, for every $y \in N_{EF'}$.*

Proof. Follows from the completeness of algorithm $\mathcal{A}_2$.

Now we proceed to the actual construction. Let U, M be the forest model which guides the expansion with $EF = \langle\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES\rangle$, where $p \in \mathcal{L}(\varepsilon)$ and let $CS = \langle EF', \textsc{ct}, \textsc{st}, G\rangle$ be an initial completion structure for checking satisfiability of p w.r.t. P with $EF' = \langle\{T'_{\varepsilon'}\} \cup \{T'_a \mid a \in cts(P)\}, ES'\rangle$, where $p \in \textsc{ct}(\varepsilon')$. We will expand CS in a depth-first fashion (the order of processing trees is not important, just that their contents are expanded depth-first; the expansions of different trees can also be interleaved). Always a node with status $unexp$ is selected for expansion.

Let π be a function which relates nodes from the extended forest in the completion structure in construction to nodes in the forest model: $\pi : N_{EF'} \to U$. We show that at any point during the construction the following property holds: ‡for every node $x \in N_{EF'}$ there is a node $\pi(x) \in N_{EF}$, such that $\textsc{ct}(x) \subseteq \mathcal{L}(\pi(x)) \cup not\ (upreds(P) - \mathcal{L}(\pi(x)))$. Intuitively, the positive content of a node/edge in the completion structure is contained in the label of the corresponding forest model node, and the negative content of a node/edge in the completion structure cannot occur in the label of the corresponding forest model node.

The property will be proved by induction and it is used at every step of the construction (for nodes for which it was already proved to hold): as such the induction step coincides with the construction step.

Base case: We set $\pi(\varepsilon') := \varepsilon$ and $\pi(a) := a$, for every $a \in cts(P)$. That the induction hypothesis is fulfilled follows from the way the initial completion structure for p w.r.t. P was defined.

Induction/Construction step: Let x be the node currently selected for expansion in EF': $\textsc{st}(x) := unexp$. Perform the following operations:

(i) Check whether the blocking or caching conditions are met:

- assume there is a node $y \in N_{EF'}$ such that (y, x) form a blocking pair. Then mark x as a blocked node and stop its expansion.
- assume there is a node $y \in N_{EF'}$ such that (y, x) form a caching pair. Then mark x as a cached node and stop its expansion.

Naturally, in both cases (‡) still holds, as we have not modified the content of nodes and we also did not add any new nodes. Note that when applying the blocking or caching rule we no longer use the guidance of (U, M): (U, M) might justify in a different way the atoms which have x and its successors as one of their arguments; we are interested in finding a finite representation of a model which satisfies p, not necessarily of the original model which we used for guidance (actually the soundness proof constructs a non-forest model from a clash-free complete completion structure).

(ii) If x is neither blocked nor cached, according to the induction hypothesis, there is a node $\pi(x) \in N_{EF}$ such that $\textsc{ct}(x) \subseteq \mathcal{L}(\pi(x)) \cup not\ (upreds(P) - \mathcal{L}(\pi(x)))$. Let UC be a unit completion structure with root ε' corresponding to node $\pi(x)$ as in Lemma 5.23. UC has the property that $\textsc{ct}(\varepsilon'') = \mathcal{L}(\pi(x)) \cup not\ (upreds(P) - \mathcal{L}(\pi(x)))$ and $\textsc{ct}(a) \subseteq \mathcal{L}(a) \cup not\ (upreds(P) - \mathcal{L}(a))$, for every $a \in cts(P)$; this, together with the induction hypothesis, implies that $x \in N_{EF'}$ is matchable with UC. Apply the *Match* rule for x and UC.

For every node y added to/updated from EF' by addition of UC: $y \in N_{EF'}$ and $(x, y) \in A_{EF'}$, we have that: $\textsc{ct}(y) \subseteq \mathcal{L}(y_{x||\pi(x)}) \cup not\ (upreds(P) - \mathcal{L}(y_{x||\pi(x)}))$. We set $\pi(y) = y_{x||\pi(x)}$, for every such node, and the induction hypothesis holds.

(ii) Check whether the redundancy rule condition is met: assume there is a node $y \in N_{EF'}$ such that (y, x) form a redundancy pair.

Note that unlike the models constructed by our algorithm, arbitrary forest models might contain 'redundant' nodes (or better said they translate to completion structures which contain such nodes). A redundancy pair (y, x) signals a redundant computation in the form of the tree in the extended forest from y to x The way to overcome this is to simply ignore the redundancy when constructing a completion structure. As the redundant part of the model is first incorporated in the completion structure, when encountering such a redundancy pair we modify the structure by cutting out the redundant part: y is replaced with x (technically with the completion structure at x): $replace_CS(y, CS_x)$.

As concerns the image of y under π in EF, it is changed to the previous image of x: $\pi(y) := \pi(x)$. The induction hypothesis still holds.

Given that the construction process described above terminates after a finite amount of time, its result is obviously a clash-free complete completion structure: the extended forest has been constructed by appending UCS-s to matchable nodes of the forest, no rule can be further applied, all redundant nodes are eliminated, and every node is expanded, blocked, or cached. Next we show by reductio ad absurdum that the construction can always be performed in a finite amount of steps.

We show that the number of operations related to constructing a path of the completion structure is finite. Assume the opposite, that the construction of a path in the structure does not terminate. First, one can only apply blocking or caching once on every path. Second, every completion structure has a finite number of nodes (from the Termination theorem). The only possibility for the construction to not terminate is by application of the redundancy rule an infinite number of times: note that also in this case, the path in construction should always have a finite number of nodes. Thus, in this case, there will be a repeated processing of chunks of the forest model which are found to be redundant.

In order to formalize this scenario, we first introduce the notion of *relaxed completion structure* which is a completion structure constructed in the usual way, except for the fact that it can contain redundancy pairs: the redundancy rule is not taken into account. Note that any completion structure is a relaxed completion structure, while the reciprocal statement is not true.

Lemma 5.24. *Let (x, y) and (y, z) be two redundancy pairs in a relaxed completion structure. Then (x, z) is still a redundancy pair.*

Proof. Follows directly from the definition of redundancy pair and transitivity of the subset-equal relationship. □

Formally, let $(x_i, y_i)_{i \geqslant 0}$ be the infinite sequence of redundancy pairs which are identified during the construction process on the same path of the completion structure. Note that these redundancy pairs do not coexist at any time during the construction process: each time a new pair is identified, previous redundancies have already been removed. Let also $CS^0 = \langle EF^0, ct^0, st^0, G^0 \rangle$ be a relaxed completion structure which is constructed similarly to the completion structure in discussion, CS, all steps being the same except that in the case of CS^0 the redundancy rule does not apply. Starting from CS^0, we define inductively a sequence of relaxed completion structures $(CS^i)_{i\geqslant 0}$, each one (except for CS^0) being obtained from the previous completion in the sequence by elimination of the redundant part indicated by the redundancy pair (x_i, y_i). Formally, $CS^i = replace_{CS^{i-1}}(x_{i-1}, CS^{i-1}[y_{i-1}])$. We also introduce the notation u^i to denote the new node in the relaxed completion structure CS^i corresponding to u in CS^0, also denoted as u^0 (if it was not deleted along the way). We have that for every $u^i \in N_{EF^i}$:

$$u^{i+1} = \begin{cases} u^i, \text{ if } u^i \leqslant x_i \\ u^i_{y_i\|x_i} \text{ if } u^i \geqslant y_i \\ undefined, \text{ otherwise} \end{cases}$$

It is clear from the previous identity that for every node $u \in N_{EF^i}$, there exists a node $v \in N_{EF^0}$, such that $u = v^i$ (as nodes are always removed from the original relaxed completion structure). Let $u_i, v_i \in N_{EF^0}$ be such that $x_i = u_i^i$ and $y_i = v_i^i$, for every $i \geqslant 0$. As each redundancy pair 'appears' later in the construction process we have that $v_{i+1} > v_i$, for every $i > 0$. As the path in CS^0 to which u_i and v_i belong is infinite and the result of removing every redundancy pair is a finite pair, an infinite part of the path is eventually removed. This infinite part is formed from chained pairs of nodes which correspond to redundancy pairs in some 'future' relaxed completion structure. The following lemma formalizes this observation.

Lemma 5.25. *There is a sequence $(k_i)_{i\geqslant 0}$ (possibly infinite) such that $v_{k_i} = u_{k_{i+1}}$ and an index $n \geqslant 0$ such that $path(v_{k_0}, u_{k_n})$ is a path of infinite length in CS^0.*

Proof. We start by constructing a sequence of pairs $(v_{l_i}, u_{l_i})_{l\geqslant 0}$ such that $v_{l_{i+1}} \geqslant u_{l_i}$, for every $i \geqslant 0$. For this we simply eliminate all pairs (v_i, u_i) from the original sequence for which there is an index j such that $v_j \leqslant v_i < u_i < u_j$. Note that this sequence might be finite. The number of nodes after applying all transformations corresponding to the redundancy pairs is: $\sum_{u_{l_i} < v_{l_{i+1}}} |path(u_{l_i}, v_{l_{i+1}})|$. This is a finite number, thus also $|\{i \mid u_{l_i} < v_{l_{i+1}}\}|$ is also finite. We have that $\sum_i |path(v_{l_i}, u_{l_i})|$ is infinite. Depending on the length of the sequence $(v_{l_i}, u_{l_i})_{l\geqslant 0}$ there are two possibilities:

- the sequence is finite: then there must be an index l_m such that $path(v_{l_m}, u_{l_m})$ is infinite. Let $k_0 = k_n = m$.
- the sequence is infinite: as, $|\{i \mid u_{l_i} < v_{l_{i+1}}\}|$, there must be an index l_m such that $u_{l_i} = v_{l_{i+1}}$, for every $i \geqslant m$. Let $k_0 = m$ and k_n some arbitrary number at infinite distance from k_0.

The following lemma will prove useful in concluding our argument regarding the impossibility of applying the redundancy rule an infinite number of times when constructing a path in a completion structure guided by a forest model.

Lemma 5.26. *For every $i \geqslant 0$, $rank(u_i) = rank(v_i)$.*

Proof. As (x_i, y_i) is a redundancy pair, it is clear that $rank(u_i^i) = rank(v_i^i)$. We show that $rank(u_i^j) = rank(v_i^j)$ implies $rank(u_i^{j-1}) = rank(v_i^{j-1})$, for every $0 < j \leqslant i$. Figure 15 depicts the possible positions of u_i^{j-1}, v_i^{j-1} relative to the positions of x_{j-1} and y_{j-1}. There are two different situations:

- a) $y_{j-1} \leqslant u_i^{j-1}$ (Figure 15 a)): again this case splits in two subcases:

$$\begin{tikzcd}
CS_{j-1}: \arrow[d] & CS_j: \arrow[d] \\
x_{j-1} \arrow[r, dashed] \arrow[d] & x_{j-1} \arrow[d] \\
y_{j-1} \arrow[ur, dashed] \arrow[d] & u_i^j = (u_i^{j-1})_{y_{j-1}||x_{j-1}} \arrow[d] \\
u_i^{j-1} \arrow[d] & v_i^j = (v_i^{j-1})_{y_{j-1}||x_{j-1}} \\
v_i^{j-1} &
\end{tikzcd}$$

a) $y_{j-1} \leqslant u_i^{j-1}$

$$\begin{tikzcd}
CS_{j-1}: \arrow[d] & CS_j: \arrow[d] \\
u_i^{j-1} \arrow[d] & u_i^j = u_i^{j-1} \arrow[d] \\
x_{j-1} \arrow[r, dashed] \arrow[d] & x_{j-1} \arrow[d] \\
y_{j-1} \arrow[ur, dashed] \arrow[d] & v_i^j = (v_i^{j-1})_{y_{j-1}||x_{j-1}} \\
v_i^{j-1} &
\end{tikzcd}$$

$b) u_i^{j-1} \leqslant x_{j-1}$

Figure 15: Backward preservation of equal rankings for (u_i^j, v_i^j) pairs.

- $rank_j(u_i^j) = rank_j(x_{j-1}) = k$: then, there exist $a, b \in upredsP$ such that $rank_j(a(x_{j-1})) = k$ and $(a, b) \in connpr_{G_j}(x_{j-1}, v_i^j)$; this, implies that $rank_{j-1}(a(x_{j-1})) = k$ and $(a, b) \in connpr_{G_{j-1}}(y_{j-1}, v_i^{j-1})$(see figure); from the fact that (x_{j-1}, y_{j-1}) is a redundancy pair, it results that: $rank_{j-1}(a(y_{j-1})) = k$ and together with $(a, b) \in connpr_{G_{j-1}}(y_{j-1}, v_i^{j-1})$, it results $rank_{j-1}(b(v_i^{j-1})) = k$, thus $rank_{j-1}(v_i^{j-1}) = k = rank_{j-1}(u_i^{j-1})$;
- $rank_j(u_i^j) > rank_j(x_{j-1})$: similar to the previous case;

- b) $u_i^{j-1} \leqslant x_{j-1} < y_{j-1} < v_i^{j-1}$ (Figure 15 b)): in this case, $rank_j(u_i^j) = rank_j(x_{j-1}) = k$. Then there exists $a, b, c \in upreds(P)$ such that $rank_j(a(u_i^j)) = k$, $(a, b) \in connpr_{G_j}(u_i^j, x_{j-1})$, and $(b, c) \in connpr_{G_j}(x_{j-1}, v_i^j)$.

  From the figure, one can see that $rank_{j-1}(x_{j-1}) = k$. As (x_{j-1}, y_{j-1}) is a redundancy pair, $rank_{j-1}(y_{j-1}) = k$. If $(a, b) \in connpr_{G_{j-1}}(u_i^{j-1}, x_{j-1})$ and $rank\ (a(u_i^{j-1}))$, then $(a, b) \in connpr_{G_{j-1}}(u_i^{j-1}, y_{j-1})$ (again from the redundancy of (x_{j-1}, y_{j-1})).

From lemma 5.25 and 5.26 it results that there is a sequence of nodes $(u_{k_i}, v_{k_i})_{0 \leqslant i \leqslant n}$ such that $u_{k_i} = v_{k_{i+1}}$, and $rank(u_{k_i}) = rank(v_{k_i}) = r$, for every $0 \leqslant i \leqslant n$ and the path $path(u_{k_0}, v_{k_n})$ is infinite. As $rank(u_{k_0}) = rank(v_{k_0}) = r$, this implies that there is a path of infinite length of rank r in G_0. As G_0 reflects the dependencies between the atoms in the forest model, this is equivalent to the existence of an atom in the forest model which is motivated by an infinite chain of atoms in the model. This contradicts with the fact that any atom in an open answer set is justified in a finite number of steps[Heymans et al., 2006, Theorem 2]. Thus, the construction of a complete clash-free completion structure from a forest model does terminate.

5.7 Complexity

Proposition 5.10 implies the following complexity result for our algorithm $\mathcal{A}_3$:

Proposition 5.27. *$\mathcal{A}_3$ runs in the worst case in non-deterministic exponential time.*

However, one can transform the algorithm to a deterministic procedure which can be executed in exponential time. The deterministic procedure which we will call $DET - \mathcal{A}_3$ consists in constructing an AND/OR extended forest with depth double in the size of the largest depth encountered when running the nondeterministic algorithm. At odd levels, there are OR nodes with unexpanded content (they contain just the constraints imposed by their predecessor or the predicate checked to be satisfiable in case of one root node and an empty set for the other root nodes), while at even levels, there are AND saturated 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 every OR node, each of its 'realizations' spawns a new copy of the graph G. We call such a structure an AND/OR completion structure.

Blocking and caching are applied by considering only pairs of AND nodes in the extended forest. For simplicity, we consider the stricter caching condition used in the proof of lemma 5.10.

A leaf of the AND/OR extended forest is labeled with *false* if it is unexpanded and it is not a blocked or cached node, with *true* if it is a blocked node, and it is labeled with the label of its corresponding caching node otherwise (if the leaf is cached node). A predicate p is satisfiable in such a structure if the root node of every tree in the structure evaluates to *true*. In this case the structure is called a *successful* AND/OR completion structure. The evaluation can be done straightforwardly as the evaluation of a caching node does not depend on the evaluation of its cached done due to the fact that, like before, the extended forest is constructed in a depth-first manner.

Proposition 5.28. *$DET - \mathcal{A}_3$ is sound, complete, and runs in the worst case in deterministic exponential time.*

Proof Sketch.

Soundness

From a successful AND/OR completion structure we construct a clash-free complete completion structure.

For every pair (S, r) for which there is at least a node x in the extended AND/OR forest with $\text{CT}(x) = S$ and $rank(x) = r$, let $x_{(S,r)}$ be the 'witness' AND node for (S, r), i.e. the node which is expanded and which will be a caching node in every caching pair of nodes with profile (S, r).

Assume that the root node of every tree in the successful deterministic structure evaluates to *true*. For every OR node, pick a successor which is true and add it to the completion structure in construction. For every AND node y, if it is blocked or expanded, simply add it to structure in construction. If y is cached and $x_{(\text{CT}(y),rank(y))}$ has not been added to the completion structure in construction, copy $x_{(\text{CT}(y),rank(y))}$ instead of y to the structure.

Completeness

From a clash-free complete completion structure we construct a successful AND/OR completion structure. At every OR node we always add as the first successor of the node the unit completion structure chosen when constructing the clash-free complete completion structure. It is easy to see that a deterministic structure constructed in such a way is successful.

Complexity

Using a similar argument as in lemma 5.10 one can show that the size of a successful deterministic structure is still exponential in the size of P: clearly the depth of the AND/OR extended forest is still exponential in the size of P (it is double the depth of the deepest complete completion structure constructed using the nondeterministic algorithm) and the caching argument still holds.

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 [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 and f-hybrid knowledge bases.

Proposition 5.29. *Satisfiability checking of a unary predicate p w.r.t. a FoLP P is* EXPTIME*-complete.*

Finally, this result translates in a similar result concerning f-hybrid knowledge bases. f-hybrid knowledge [Feier and Heymans, 2009] are a tight combination of FoLPs and the DL $\mathcal{SHOQ}$. As $\mathcal{SHOQ}$ is known to be EXPTIME-complete, it follows that f-hybrid knowledge bases are EXPTIME-hard. We also know that reasoning with f-hybrid knowledge bases can be reduced to reasoning with FoLPs. Thus, it can be deduced that they are EXPTIME-complete.

Proposition 5.30. *Satisfiability checking of a unary predicate p/concept C w.r.t. an f-hybrid knowledge base KB is* EXPTIME*-complete.*

6 Conclusions: Reasoning with FoLPs and Beyond

6.1 Discussion

We presented an optimal worst case algorithm for reasoning with Forest Logic Programs. The algorithm exploits the forest model property of the fragment and builds on techniques introduced in the previous year of the project, like pre-computing all possible building blocks of a model in the form of trees of depth 1. However due to the introduction of new termination techniques, the worst case complexity drops one exponential level compared to its previous variants: from double exponential time to exponential time.

Thus, while FoLPs can simulate reasoning with the DL $\mathcal{SHOQ}$, and allow for additional features like a minimal model based semantics and a cleaner syntax, the worst case reasoning complexity stays the same. As reasoning with the tight combination of FoLPs and $\mathcal{SHOQ}$ ontologies, f-hybrid knowledge bases, can be reduced to reasoning with FoLPs, the algorithm can be employed also for reasoning with f-hybrid knowledge bases. It also establishes that satisfiability checking w.r.t. f-hybrid knowledge bases is EXPTIME-complete.

While not mentioned explicitly here, $\mathcal{A}_2$ also identifies and eliminates so-called *redundant unit completion structures*: these are structures which are strictly less general than others, so they can always be replaced in a model with other more general structures. Assuming that the new algorithm $\mathcal{A}_3$ also employs this technique of discarding redundant unit completion structures, it addresses computational redundancy issues across three orthogonal axis: (i) *local redundancy*: eliminating redundant unit completion structures eliminates local redundancy, i.e. redundancy among the successors of a single node, (ii) *redundancy along a path*: achieved by means of the redundancy rule, and (iii) *redundancy across paths*: achieved by means of the caching rule.

6.2 Related Work

Datalog$^{\pm}$ [Calì et al., 2009b,a] is an extension of Datalog which can simulate some DLs from the DL-Lite family [Calvanese et al., 2007]. The extension consists in allowing a special type of rules with existentially quantified variables in the head, called tuple generating dependencies (TGDs). Note that our free rules are different from TGDs, as they allow for universally quantified variables which do not appear in the body of the rule to appear in the head.

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. Two such restrictions are: (1) every rule should have a guard, an atom which contains all variables in the rule body, giving rise to *guarded Datalog*$^{\pm}$, and (2) every rule should have a singleton body atom, giving rise to *linear Datalog*$^{\pm}$. The guardedness condition has been relaxed to *weakly-guardedness*, where the weak guard has to contain only the variables in the body that appear in so-called affected positions, positions where newly invented values can appear during reasoning [Calì et al., 2008]. Reasoning relies on a proof technique from database theory, the chase algorithm, which repairs databases according to the set of dependencies.

Some further generalizations to the guarded fragment of Datalog$^{\pm}$ are so-called *sticky sets* of TGDs [Calì et al., 2010a], *weakly-sticky* sets of TGDS, and *sticky-join* sets of TGDs [Calì et al., 2010b] which generalize both sticky sets and linear TGDs. All these fragments are defined by imposing restrictions on multiple occurrences of variables in rule bodies. The syntactical restrictions on rules bodies are orthogonal to the ones we imposed for achieving decidability on FoLPs: neither Datalog$^{\pm}$ rules are enforced to have a tree-shape like FoLPs, nor variables in FoLP rules have to fulfill the conditions required for the different sets of TGDs to belong to one of the previously mentioned decidable fragments of Datalog$^{\pm}$. TGDs do not contain negation. However, so-called stratified normal TGDs have been introduced, which are TGDs whose body atoms can appear in a negated form together with a semantics in terms of canonical models. FoLPs support full negation as failure (under the stable models semantics).

6.3 Future Work

We presented a worst-case optimal tableau algorithm for reasoning with FoLPs. From a practical point of view, FoLPs are interesting as they allow the simulation of reasoning with $\mathcal{SHOQ}$ ontologies, while at the same time having a nonmonotonic semantics. From a theoretical point of view, the stable model semantics combined with the open domains of interpretation gave rise to unusual challenges as concerns termination conditions for a tableau algorithm. The knowledge compilation technique in [Feier and Heymans, 2010] also defines and eliminates *redundant unit completion structures*, which are structures which are strictly less general than others. As such, they can always be replaced in a model with other more general structures. As the new algorithm can also reuse that optimization, it addresses computational redundancy issues across three orthogonal axis: (i) *local redundancy*: be eliminating the set of redundant unit completion structures (ii) *redundancy along a branch*: by means of the redundancy rule, and (iii) *redundancy across branches*: means of the caching rule.

Among rule-based formalisms, one which allows for unsafe rules is Datalog$^{\pm}$ [Calì et al., 2009b,a], an extension of Datalog which can simulate some DLs from the DL-Lite family [Calvanese et al., 2007]. The extension consists in allowing 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. Such restrictions are: (1) every rule should have a guard, an atom which contains all variables in the rule body, giving rise to *guarded Datalog*$^{\pm}$, and (2) every rule should have a singleton body atom, giving rise to *linear Datalog*$^{\pm}$. The guardedness condition has been relaxed to *weakly-guardedness*, where the weak guard has to contain only the variables in the body that appear in so-called affected positions, positions where newly invented values can appear during reasoning [Calì et al., 2008]. Reasoning relies on a proof technique from database theory, the chase algorithm, which repairs databases according to the set of dependencies. The guarded fragment of Datalog$^{\pm}$ has been generalized to so-called *sticky sets* of TGDs [Calì et al., 2010a], *weakly-sticky* sets of TGDS, and

$$\begin{tikzcd}
depth = d \arrow[dr, dashed] & & & & \\
& p(x) \arrow[dl] & r(x) \arrow[dr] & & \\
q(x_1) \arrow[urr] & & & s(x_2) & rank = d \arrow[l, dashed]
\end{tikzcd}$$

Figure 16: Computing the rank of a node in the presence of backward arcs in the tree

sticky-join sets of TGDs [Calì et al., 2010b]: the fragments are defined by imposing restrictions on multiple occurrences of variables in rule bodies. These restrictions are orthogonal to the ones we imposed for achieving decidability on FoLPs: neither Datalog$^{\pm}$ rules are enforced to have a tree-shape like FoLPs, nor variables in FoLP rules have to fulfill the conditions required for the different sets of TGDs to belong to one of the previously mentioned decidable fragments of Datalog$^{\pm}$.

A formalism related to FoLPs is $\mathbb{FDNC}$ [Šimkus and Eiter, 2007]. $\mathbb{FDNC}$ is an extension of ASP with function symbols which as FoLPs has the forest model property. $\mathbb{FDNC}$ rules are required to be safe unlike FoLP ones and as such they are amenable to a bottom-up reasoning technique. The complexity for standard reasoning tasks for $\mathbb{FDNC}$ is ExpTime-complete and worst-case optimal algorithms are provided.

Another interesting fragment of Open Answer Set Programming which has been proved to be decidable is *Conceptual Logic Programs under the Inverted World Assumption*. The fragment has the *tree model property* and can simulate the description logic $\mathcal{SHIQ}$. Conceptual Logic Programs (CoLPs) are FoLPs in which constants are disallowed. The Inverted World Assumption refers to the fact that the signature of the programs is such that for every binary f, there exists an inverse binary predicate f^i; semantically, the assumption refers to the following condition: for every Open Answer Set (U, M), and for every binary predicate f it holds that: $f^i(x, y) \in M$ iff $f(y, x) \in M$. IWA is equivalent to adding $f(X, Y) \leftarrow f^i(Y, X)$ and $f^i(Y, X) \leftarrow f(X, Y)$ to the original program and evaluating it under the usual semantics.

In [Heymans et al., 2006] satisfiability checking w.r.t. IWA has been reduced to checking emptiness of a two-way alternating tree automata. However, there is no practical algorithm for dealing with such programs. We plan to investigate how the algorithm $\mathcal{A}_3$ can be adapted for reasoning with CoLPs under IWA: the non-trivial part of such an adaptation consists in dealing with the IWA. A natural step in this direction is to generalize the notion of UCS to UCS under IWA: a UCS would be still an extended tree with depth 1, but the root can have an outgoing arc to its predecessor. Matching UCSs consists in this case in a double match (between 2 pairs of successor nodes). We conjecture that any arbitrary tree model of a CoLP under IWA can be reduced to an *exponential* size structure which can then be unraveled to a finite bounded size model by applying similar transformations to the ones used for reducing a FoLP model in the completeness proof. However, it is questionable whether the classical tableau approach still works. In the presence of backwards arcs in the forest (from nodes to their predecessors), the rank of a node/atom is no longer a function of predecessor nodes: it might be needed to traverse the whole structure in order to compute it. Figure 16 gives an example where the rank of a node depends on the content of one of its neighbors: both $x \cdot 1$ and $x \cdot 2$ are the successors of x in a tree, while the arcs depicted in the figure are arcs in the atom dependency graph of the constructed model. Before fully expanding $x \cdot 1$ there is no path in G from a predicate with argument x to a predicate with argument $x \cdot 2$, and thus the rank of $x \cdot 2$ is $d + 1$. However after fully expanding $x \cdot 1$, the rank of $x \cdot 2$ is d.

One possibility would be to generate a completion structure with size within the computed bound and check whether the completion structure is clash-free and complete. Formal proofs for reasoning with this fragment are subject of future work.

Further on, we plan to formally define the fragment of FoLPs under the IWA and to investigate reasoning with this fragment. Such a fragment would allow the simulation of the expressive DL $\mathcal{SHOIQ}$.

Acknowledgement

The author would like to thank Prof. Thomas Eiter for the helpful suggestions and discussions regarding this work.

References

F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. P.-S. (eds). The description logic handbook: Theory, implementation, and applications. In *Description Logic Handbook*. Cambridge University Press, 2003. ISBN 0-521-78176-0.

A. Calì, G. Gottlob, and M. Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In F. Baader, C. Lutz, and B. Motik, editors, *Description Logics'08*, volume 353. CEUR-WS.org, 2008.

A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog$\pm$: A Unified Approach to Ontologies and Integrity Constraints. volume 9, pages 14–30, 2009a.

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

A. Calì, G. Gottlob, and A. Pieris. Advanced Processing for Ontological Queries. *Proceedings of the VLDB Endowment*, 3(1):554–565, 2010a.

A. Calì, G. Gottlob, and A. Pieris. Query Answering under Non-Guarded Rules in Datalog$\pm$. In *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems (RR 2010)*, pages 1–17, 2010b.

D. Calvanese, G. de Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *JAR*, 39(3):385–429, 2007.

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.

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.

C. Feier and S. Heymans. An optimization for reasoning with forest logic programs. In *Proc. of Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*. CoRR, 2010. URL `http://stijnheymans.net/pubs/aspocp2010.pdf`.

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

C. Feier, H. Aït-Kaci, J. Angele, J. de Bruijn, H. Citeau, T. Eiter, A. E. Ghali, V. Kerhet, E. Kiss, R. Korf, T. Krekeler, T. Krennwallner, S. Heymans, A. M. (FUB), M. Rezk, and G. Xiao. D3.3 Complexity and optimization of combinations of rules and ontologies. Technical report, 2010.

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

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.

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.

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.

U. S. I. Horrocks and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proc. 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, volume LNAI 1705, pages 161–180. Springer Verlag, 1999.

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.

M. Šimkus and T. Eiter. $\mathbb{FDNC}$: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007)*, 2007.