Ludwig-Maximilians-Universität München
Department 'Institut für Informatik'
Lehr- und Forschungseinheit Programmier- und Modellierungssprachen

Master thesis

Approximating (Probabilistic) Logic Programs On Large Domains

Bao Loi Quach
B.Quach@campus.lmu.de
Matriculation number: 11368060

Submission Date: September 30, 2022
Mentor: Dr. Felix Weitkämper
Supervisor: Prof. Dr. Eyke Hüllermeier

Declaration Of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others.

Munich, September 30 2022

Bao Loi Quach

Abstract

Probabilistic logic programs are logic programs in which some of the clauses are annotated with probabilistic facts. The behaviour of relations in these clauses can get very complex for large domain sizes and in settings of uncertainty. In recent years the asymptotic behaviour of relational representations, in which queries are completely independent of the domain size have aroused interest. The motivating background of this thesis is the finding of Weitkämper [12]. He showed that every probabilistic logic program under the distribution semantics is asymptotically equivalent to an acyclic probabilistic logic program consisting only of determinate clauses over probabilistic facts. To transform a probabilistic logic program to a logic program with only determinate clauses several steps have to be done. These steps include rewriting the probabilistic logic program to a formula of least fixed-point logic and then applying asymptotic quantifier elimination on the formula. Weitkämper also showed that least fixed-point logic has asymptotic quantifier elimination. Quantifier-free first-order formulas can be rewritten to an acyclic determinate stratified DATALOG formula and a DATALOG formula is a synonym for a (probabilistic) logic program. This was proven in Theorem 9.1.1 by Ebbinghaus and Flum [6]. The goal of this thesis is to implement an algorithm which handles this transformation of a probabilistic logic program into an acyclic determinate logic program and show all the theoretical steps which are necessary for the implementation.

Contents

1 Introduction

1.1 Motivation

Statistical relational artificial intelligence is a field which specifies statistical models for relational data. One framework under this field is an extension to logic programming to incorporate probabilistic information. This includes probabilistic logic programming which brings together approaches from logic and probability to reason about and learn from relational domains in a setting of uncertainty. For probabilistic logic programs inference is computationally very expensive. This is a great hurdle since it is hard to predict or undesirable behaviour of the model when applied to domains of different sizes can occur.

The motivation for this thesis comes from the paper 'An asymptotic analysis of probabilistic programming, with implications for expressing projective families of distributions' of Weitkämper [12]. He showed that for probabilistic logic programming under the distribution semantics every probabilistic logic program is asymptotically equivalent to an acyclic determinate probabilistic logic program. In consequence of this result it is shown that the probabilities of queries expressed by a logic program converge as domain size increases.

The paper starts with introducing the framework of families of distributions and the notion of projectivity. A more detailed exposition of this topic and their background can be found in the work of Jaeger and Schulte [7] [8].

Definition 1. A *family of distributions* for a relational vocabulary $\mathcal{S}$ is a sequence $(Q^{(n)})_{n\in\mathbb{N}}$ of probability distributions on the sets Ω_n of all $\mathcal{S}$-structures with domain $\{1, ..., n\} \subseteq \mathbb{N}$.

Definition 2. A family of distributions is called *exchangeable* if every $Q^{(n)}$ is invariant under $\mathcal{S}$-isomorphism.

It is called *projective* if, in addition, for all $m < n \in \mathbb{N}$ and all $\omega \in \Omega_m$ the following holds:

$$Q^{(m)}(\{\omega\}) = Q^{(n)}\left(\{\omega' \in \Omega_n | \omega \text{ is the substructure of } \omega' \text{ with domain } \{1, \ldots, n\}\ \}\right)$$

Projectivity encapsulates a strong form of domain size independence. For instance, we have the query $R(x)$, where R is a relation symbol in $\mathcal{S}$. In an exchangeable family of distributions, the unconditional probability of $R(x)$ holding in a world is independent of the precise interpretation of x, and depends only on the domain size. If the family of distributions is projective, then the probability of $R(x)$ is independent even of the domain size. This implies that the computational complexity of quantifier-free queries is constant with domain size, since queries can always be evaluated in a domain consisting just of terms mentioned in the query itself.

Jaeger and Schulte [8] provided a complete characterisation of projective families of distributions in terms of exchangeable arrays. But it is not clear how this representation translates to the statistical relation formalisms currently in use, for instance probabilistic logic programming. It is shown later on that there are indeed projective families of distributions that are not expressible by a probabilistic logic program but that for projective family distributions induced by a probabilistic logic program, the independence property holds.

The rest of the motivating paper deals with the interplay between asymptotic behaviour of logical theories as they have been studied in finite model theory and the families of distributions that are induced by them. Therefore a notion of asymptotic equivalence of families of distributions is introduced. Abstract distributions acts like a bridge between

notions from finite model theory and probabilistic logic programming. This is built on the relational Bayesian network specifications of Cozman an Maua [5], which combine random and independent root predicates with non-root predicates that are defined by first-order formulas. Abstract distributions in the paper 'Asymptotic analysis of probabilistic logic programming' generalise from the idea of Cozman and Maua, by abstractions away from first-order logic to a general logical language.

Definition 3. Let $\mathcal{R}$ be a vocabulary. Then a *logical language* $L(\mathcal{R})$ consists of a collection of functions φ which take an $\mathcal{R}$-structure M and return a subset of M^n for some $n \in \mathbb{N}$ (called the *arity* of φ). In analogy to the formulas of first-order logic, we refer to those functions as $L(\mathcal{R})$*-formulas* and write $M \models \varphi(\vec{a})$ whenever $\vec{a} \in \varphi(M)$.

Definition 4. Let $\mathcal{S}$ be a relational vocabulary, $\mathcal{R} \subseteq \mathcal{S}$, and let $L(\mathcal{R})$ be a logical language over $\mathcal{R}$. Then an *abstract L-distribution over* $\mathcal{R}$ *(with vocabulary* $\mathcal{S}$*)* consists of the following data:

- For every $R \in \mathcal{R}$ a number $q_R \in \mathbb{Q} \cap [0, 1]$.
- For every $R \in \mathcal{S} \backslash \mathcal{R}$, an $L(\mathcal{R})$-formula ϕ_R of the same arity as R.

In the following it is assumed that all vocabularies are finite. The semantics of an abstract distribution is only defined relative to a domain D, which is also assumed to be finite. It is defined as the following:

Definition 5. Let $L(\mathcal{R})$ be a logical language over $\mathcal{R}$ and let D be a finite set. Let T be an abstract L-distribution over $\mathcal{R}$. Let Ω_D be the set of all $\mathcal{R}$-structures with domain D. Then the *probability distribution on* Ω_D *induced by* T*, written* $Q_T^{(D)}$*,* is defined as follows:

- For all $\omega \in \Omega_D$, if $\exists_{\vec{a} \in \vec{D}} \exists_{R \in \mathcal{S} \backslash \mathcal{R}} : R(\vec{a}) \nLeftrightarrow \phi_R(\vec{a})$, then $Q_T^{(D)}(\{\omega\}) := 0$
- Otherwise, $Q_T^{(D)}(\{\omega\}) := \prod_{R \in \mathcal{R}} (q_R^{|\{\vec{a} \in \vec{D} | R(\vec{a})\}|}) \times \prod_{R \in \mathcal{R}} (1 - q_R)^{|\{\vec{a} \in \vec{D} | \neg R(\vec{a})\}|}$

This means all the relations in $\mathcal{R}$ are independent with probability q_R and the relations in $\mathcal{S} \setminus \mathcal{R}$ are defined deterministically by the $L(\mathcal{R})$-formulas ϕ_R. Abstract first-order language distributions do not necessarily give rise to projective families. As seen in the following example:

Example 1. Let $\mathcal{R} = \{R, P\}$, $\mathcal{S} = \{R, P, S\}$, for a unary relation R a binary relation P and a unary relation S. Then an abstract distribution over (R) has numbers q_R and q_P which encode probabilities. Consider the FOL-distribution T with $\varphi_S = \exists_y (R(x) \wedge P(x, y))$. For any domain D, $Q_T^{(D)}$ is obtained by making an independent choice of $R(a)$ or $\neg R(a)$ for every $a \in D$, with a q_R probability of $R(a)$. Similarly, an independent choice of $P(a, b)$ or $\neg P(a, b)$ is made for every pair (a, b) from D^2, with a q_P probability of $P(a, b)$. Then, for any possible $\mathcal{R}$-structure, the interpretation of S is determined by $\forall_x S(x) \leftrightarrow \varphi_S(x)$. The resulting family of distributions is not projective, since the probability of $Q(a)$ increases with the size of the domain as more possible candidates b for $P(a, b)$ are added.

If the φ are all given by quantifier-free formulas then the induced families of distribution are indeed projective. Such abstract $L(\mathcal{R})$-distributions in which $L(\mathcal{R})$ is the class of quantifier-free first-order language formulas over R, are called quantifier-free distributions. This leads to following proposition:

Proposition 6. *Every abstract quantifier-free distribution induces a projective family of distributions.*

In the next part an overview of *asymptotic quantifier elimination* is given. This is a big part of the algorithm of this thesis and is also explained in more detail in 2.5 and 3.6. First the notion of asymptotic equivalence for families of distributions is introduced:

Definition 7. Two families of distributions $(Q^{(n)})$ and $(Q'^{(n)})$ are *asymptotically equivalent* if $\lim_{n\to\infty} \sup_{A\subseteq\Omega_n} |Q^{(n)}(A) - Q'^{(n)}(A)| = 0$

Note that in measure theoretic terms, the families of distributions of $Q^{(n)}$ and $Q'^{(n)}$ are asymptotically equivalent if and only if the limit of the total variation difference between them is 0. The notion can be extended to abstract distributions with them being asymptotically equivalent if they induce asymptotically equivalent families of distributions. This leads to the following setting for asymptotic quantifier elimination:

Definition 8. Let $L(\mathcal{R})$ be an extension of the class of quantifier-free $\mathcal{R}$-formulas. Then $L(\mathcal{R})$ *has asymptotic quantifier elimination* if every abstract $L(\mathcal{R})$ distribution is asymptotically equivalent to a quantifier-free distribution over $L(\mathcal{R})$.

First-order logic indeed has asymptotic quantifier elimination. The asymptotic theory of relational first-order logic can be summarised as follows by Ebbinghaus and Flum [6] in chapter 4:

Definition 9. Let $\mathcal{R}$ be a relational vocabulary. Then the first order theory RANDOM($\mathcal{R}$) is given by all axioms of the following form, called *extension axioms over* $\mathcal{R}$:

$$\forall_{v_1,\dots,v_r}\left(\bigwedge_{1\le i\le j\le r} v_i \neq v_j \to \exists_{v_{r+1}}\left(\bigwedge_{1\le i\le r} v_i \neq v_{r+1} \wedge \bigwedge_{\varphi\in\phi} \varphi \wedge \bigwedge_{\varphi\in\Delta_{r+1}\setminus\Phi} \neg\varphi\right)\right)$$

where $r \in \mathbb{N}$ and Φ is a subset of

$$\Delta_{r+1} := \{R(\overline{r}|R \in \mathcal{R}, \vec{x}\} \text{ a tuple from } \{v_1, ..., v_{r+1} \text{ containing } v_{r+1}\}$$

RANDOM($\mathcal{R}$) eliminates quantifiers, for instance for each formula $\varphi(\vec{x})$ there is a quantifier-free formula $\varphi'(\vec{x})$ such that RANDOM($\mathcal{R}$) $\vdash \forall_{\vec{x}}(\varphi(\vec{x}) \leftrightarrow \varphi'(\vec{x}))$. Sometimes it helps to characterise this quantifier-free formula more explicitly:

Proposition 10. *Let $\varphi(\vec{x})$ be a formula of first-order logic. Then:*

- *$\varphi'(\vec{x})$ as described earlier can be chosen such that only those relation symbols occur in φ' that occur in φ.*
- *If every atomic subformula of φ contains at least one free variable not in $\vec{x}$, and no relation symbol occurs with different variables in different literals, then either* RANDOM($\mathcal{R}$) $\vdash \forall_{\vec{x}}\varphi(\vec{x})$ *or* RANDOM($\mathcal{R}$) $\vdash \forall_{\vec{x}}\neg\varphi(\vec{x})$

RANDOM($\mathcal{R}$) is important since it has the role of the asymptotic limit of the class of all $\mathcal{R}$-structures. With this summary one can see that first-order logic has indeed asymptotic quantifier elimination.

Now an introduction to least fixed-point logic is given since it is an adequate representation for probabilistic logic programs. It is explained in more detail in the background section how least fixed-point logic is used in the theory of the algorithm of this thesis. Atomic second-order variables can occur as subformulas of least fixed-point formulas. The following representation is taken from Weitkämper [12].

Definition 11. Assume an infinite set of second-order variables, indicated customarily by upper-case letters from the end of each alphabet, each annotated with a natural number arity. Then an *atomic second-order formula* φ is either a (first-order) atomic formula, or an expression of the form $X(t_1, ..., t_n)$, where X is a second-order variable or arity n and $t_1, ..., t_n$ are constants or (first-order) variables.

Definition 12. A formula φ is called *positive in a variable* x if x is in the scope of an even number of negation symbols in φ. A *formula in least fixed-point logic* or *LFP formula* over a vocabulary $\mathcal{R}$ is defined inductively as follows:

1. Any atomic second-order formula is an LFP formula.
2. If φ is an LFP formula, then so is $\neg\varphi$.
3. If φ and ψ are LFP formulas, then so is $\varphi \vee \psi$
4. If φ is an LFP formula, then so is $\exists x\varphi$ for a first-order variable x.
5. If φ is an LFP formula, then so is $[\mathrm{LFP}_{\vec{x},X}\varphi]\vec{t}$, where φ is positive in the second-order variable X and the lengths of the string of first-order variables $\vec{x}$ and the string of terms $\vec{t}$ coincide with the arity of X.

An occurrence of a second-order variable X is *bound* if it is in the scope of an LFP quantifier $\mathrm{LFP}_{\vec{x},X}$ and *free* otherwise.

In the next definition one can see that each LFP formula φ can be associated with an operator:

Definition 13. Let $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$ be an LFP formula, with the length of $\vec{x}$ equal to the arity of X, and let ω be an $\mathcal{R}$-structure with domain D. Let $\vec{b}$ and $\vec{S}$ be an interpretation of $\vec{u}$ and $\vec{Y}$ respectively. Then we define the operator $F^{\varphi} : \mathfrak{P}(D^k) \to \mathfrak{P}(D^k)$ as follows:

$$F^{\varphi}(R) := \left\{ \vec{a} \in D^k | \omega \models \varphi(\vec{a}, \vec{b}, R, \vec{S}) \right\}.$$

Fact 14. *For every LFP formula $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$ and every $\mathcal{R}$-structure on a domain D and interpretation of variables as in Definition 1.1.14, there is a relation $R \subseteq D^k$ such that $R = F^{\varphi}(R)$ and that for all R' with $R' = F^{\varphi}(R')$ we have $R \subseteq R'$.*

Definition 15. This R from 14 is called the *least fixed-point* of $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$.

Now that all notions regarding least fixed-point logic were introduced it is time to define the semantics of least fixed-point logic:

Definition 16. By induction on the definition of an LFP formula, we define when an LFP formula $\varphi(\vec{X}, \vec{x})$ is said to *hold* in an $\mathcal{R}$-structure ω for a tuple $\vec{a}$ from the domain of ω and relations $\vec{A}$ of the correct arity:

The first-order connectives and quantifiers $\neg, \vee$ and $\exists$ as well as $\wedge$ and $\forall$ defined from them in the usual way are given the usual semantics.

An atomic second-order formula $X(\vec{x}, \vec{c})$ holds if and only if $(\vec{a}, \overrightarrow{c}_{\omega}) \in A$.

$[LFP_{\vec{x},X}\varphi]\vec{t}$ holds if and only if $\vec{a}$ is in the least fixed-point of $F^{\varphi(\vec{x},X)}$.

All the necessary introductory facts and definitions for probabilistic logic programming were presented. Now it is time to introduce probabilistic logic programming and provide the theoretical base of the algorithm of this thesis. A probabilistic logic program can be

considered as a stratified DATALOG program over probabilistic facts. This idea is taken from Riguzzi and Swift [11]. Keep in mind that in particular, probabilistic logic programs as used here do not involve function symbols, unstratified negation or higher-order constructs. The syntax and semantics of stratified DATALOG programs without probabilistic facts are explained in more detail and on how they are helpful for the implementation in the background 2.1 and methods 3 section.

Definition 17. A *probabilistic logic program* consists of probabilistic facts and deterministic rules, where the deterministic part is a stratified DATALOG program. We consider it in our framework of abstract distribution semantics as follows:

$\mathcal{R}$ is given by relation symbols R' for every probabilistic fact $p_R :: R(\vec{x})$, with $q_{R'} := p_R$. Their arity is just the arity of R.

$\mathcal{S}$ is given by the vocabulary of the probabilistic logic program and additionally the R' in $\mathcal{R}$.

Let Π be the stratified DATALOG program obtained by prefixing the program $\{R'(\vec{x}) \leftarrow R(\vec{x}) | R' \in \mathcal{R}\}$ to the deterministic rules of the probabilistic logic program.

Then ϕ_P for a $P \in \mathcal{S} \backslash \mathcal{R}$ is given by $(\Pi, P)\vec{x}$.

In the following fact, taken from [6], one can see that the semantics for probabilistic logic programming is related to the LFP distribution semantics introduced above:

Fact 18. *For every stratifiable DATALOG formula* $(\Pi, P)\vec{x}$ *as above, there exists an* LFP *formula* $\varphi(\vec{x})$ *over the extensional vocabulary* $\mathcal{R}$ *of* Π *such that for every* $\mathcal{R}$*-structure* ω *and every tuple* $\vec{a}$ *of elements of* ω *of the same length as* $\vec{x}$, $\omega \models \varphi(\vec{a})$ *if and only if* $\omega \models (\Pi, P)\vec{a}$.

For the implementation of this thesis it sufficient to consider formulas in the so-called bounded fixed-point logic whose expressiveness lies between first-order logic and least fixed-point logic [6]. Even though we also allowed second-order variables in the inductive definitions above, we assume from now on unless mentioned otherwise that LFP formulas do not have free second-order variables. Blass et al. [4] showed that LFP can be asymptotically reduced to first-order logic. With this information one can conclude that abstract LFP distributions and therefore probabilistic logic programs have asymptotic quantifier elimination. They also showed that RANDOM($\mathcal{R}$) not only eliminates classical quantifiers, but also least fixed-point quantifiers:

Fact 19. *Let* $\varphi(\vec{x})$ *be an* LFP *formula over* $\mathcal{R}$. *Then there is a finite subset* G *of* RANDOM($\mathcal{R}$) *and a quantifier-free formula* $\varphi'(\vec{x})$ *such that* $G \vdash \forall_{\vec{x}} \varphi(\vec{x}) \leftrightarrow \varphi'(\vec{x})$.

Putting this together, we can derive the following:

Theorem 20. *Least fixed-point logic has asymptotic quantifier elimination.*

Quantifier-free first-order formulas must be translated back to a stratifiable DATALOG program to achieve a characterisation within probabilistic logic programming. Indeed quantifier-free formulas can be mapped to a subset of stratified DATALOG that is well-known from logic programming:

Definition 21. A DATALOG program, DATALOG formula or probabilistic logic program is called *determinate* if every variable occurring in the body of a clause also occurs in the head of that clause.

Example 2. For example $R(x) : -P(x)$ or $Q(x, y) :\text{-}\ R(x)$ are determinate clauses in this sense. Indeterminate clauses include $R(x) :\text{-}\ P(y)$ or $R(x) :\text{-}\ Q(x, y)$.

This determinacy corresponds exactly to the fragment of probabilistic logic programs identified by Jaeger and Schulte [7]. Also Ebbinghaus and Flum [6] showed in their proof of Theorem 9.1.1:

Fact 22. *Every quantifier-free first order formula is equivalent to an acyclic determinate DATALOG formula.*

Then, we can conclude from Proposition 6:

Proposition 23. *Every determinate probabilistic logic program is projective.*

This is great since projectivity leads to domain-size independence which is good for approximating inference on large datasets. From Fact 22 we can conclude the following theorem:

Theorem 24. *Every probabilistic logic program is asymptotically equivalent to an acyclic determinate probabilistic logic program.*

1.2 Purpose of the work

Now that all the motivating and theoretical background for this thesis is presented we come to the purpose of this work. In Definition 17 we have seen that a probabilistic logic program consists of probabilistic facts and deterministic rules and in which the deterministic rules are a stratified DATALOG program. Also in Fact 18 we have seen that for every DATALOG formula in a DATALOG program there exists an LFP formula and from Theorem 20 we know that least fixed-point logic has asymptotic quantifier elimination. After applying quantifier elimination to LFP formulas we get quantifier-free first-order formulas which are equivalent to an acyclic determinate stratified DATALOG program. This information is taken from Fact 22.

Until this part all the steps for the transformation to a determinate probabilistic logic program considered only the stratified DATALOG program. But it is important to know for transforming a probabilistic logic program through asymptotic quantifier elimination to an equivalent acyclic determinate probabilistic logic program that there exist probabilistic facts. These probabilistic facts can then be added back to the determinate probabilistic logic program. The main focus of this work was to develop an algorithm which transforms formulas of a stratified DATALOG program to LFP formulas and then apply quantifier elimination on them to get the final result of an acyclic determinate probabilistic logic program. The implementation was realised with the programming language 'Prolog' and the dialect SWI-Prolog [3] since it has similar syntax to probabilistic logic programs.

1.3 Structure of the work

First we will introduce necessary background information for the implementation in more detail than in the introduction which focus was more on the motivating part. This includes further information of logic programs and probabilistic logic programming, least fixed-point logic, how to reach the fixed-point, quantifier elimination and how to transform quantifier-free first-order formulas to an acyclic determinate probabilistic logic program. After that we dig down deeper and show important parts of the actual implementation. Finally we conclude this thesis with a conclusion and show an improvement of the algorithm considering not only variables in a DATALOG program but can also constants.

2 Background

2.1 Logic programs

To understand what the algorithm does, some definitions and theoretical background are provided. The definitions and explanations for this subsection are from the book Finite Model Theory of Ebbinghaus and Flum [6] and the motivating paper of this work [12].

Definition 25. A logic program is a finite set Π of clauses of the form $\gamma \leftarrow \gamma_1, ..., \gamma_l$, where $l \geq 0$ and where $\gamma, \gamma_1, ..., \gamma_l$ are atomic or negated atomic first-order formulas.

Definition 26. Clauses like $\gamma \leftarrow \gamma_1, ..., \gamma_l$, where $l \geq 0$ and where $\gamma, \gamma_1, ..., \gamma_l$ are defined by:

- Head of the clause γ.
- Body of the clause the sequence $\gamma_1, ..., \gamma_l$.

Definition 27. In a logic program all relational symbols τ occurring in the head of some clause are called intentional. They are denoted by $(\tau,\Pi)_{\text{int}}$

Definition 28. Extensional relation symbols are all the other relation symbols occurring in a clause of the set Π. This means $(\tau,\Pi)_{\text{ext}} = \tau \setminus (\tau,\Pi)_{\text{int}}$.

Definition 29. A DATALOG program is a general logic program, in which no intentional symbol occurs negated in the body of any clause.

Example 3. An example for a DATALOG program looks like this in the final implementation:

```
Datalog = [t-[x,y]-[e-[x,y]],t-[x,z]-[t-[x,y], e-[y,z]]].
```

The DATALOG program is interpreted in this way: Whenever [e-[x,y] then t-[x,y] and whenever [t-[x,y] and e-[y,z]] then t-[x,z].

Definition 30. A DATALOG formula has the form $(\Pi, P\bar{t})$ where P is an intentional symbol of Π, say of arity r, and $\bar{t} = t_1, ..., t_r$ are terms. It is a formula of vocabulary τ_{ext}, its free variables being those occurring in some t_i.

Definition 31. A stratified DATALOG program (by short: S-DATALOG program) Σ consists of a sequence $\Pi_0, .., \Pi_n$ of DATALOG programs of vocabularies $\tau_0, ..., \tau_n$, where $(\tau_{m+1}, P_{m+1})_{\text{ext}} = \tau_m$ for $m < n$.

Definition 32. An S-DATALOG formula $(\Sigma, P\bar{t})$ exists if P is an intentional symbol of one of the constituents Π_i.

Example 4. For example, $\Sigma := \Pi_0, \Pi_1$ with $\Pi_0 : \; Px \leftarrow Exy$ and $Lx \leftarrow \neg Px$ is an S-DATALOG program. In any tree, the S-DATALOG formula $(\Sigma, L)x$ defines the set of leaves.

Definition 33. If Π is a logic program, a *stratification* of Π is a partition of Π into $\Pi_0, ..., \Pi_n$ such that $\Pi_0, ..., \Pi_n$ is an S-DATALOG program.

Definition 34. The breadth of an S-DATALOG program $\Pi_0, \Pi_1, ..., \Pi_k$ is by definition, given by $max\_i \leq k \; breadth(\Pi_i)$.

According to Theorem 9.1.1 of the Finite Model Theory book of Ebbinghaus and Flum the logic of S-DATALOG is equivalent to bounded fixed-point logic, by short FO(BFP), a subset of least fixed-point logic. It is shown that for the direction $S\text{-}DATALOG \leq FO(BFP)$ that all clauses with form $Q\overline{x}_{\mathrm{Q}} \leftarrow \gamma_1, ..., \gamma_l$, where $\overline{x}_Q$ is a sequence of distinct variables can be transformed into a formula of form $\varphi_Q(\overline{x}_Q) := \bigvee\{\exists\overline{y}(\gamma_1 \wedge ... \wedge \gamma_l) \mid Q\overline{x}_{\mathrm{Q}} \leftarrow \gamma_1, ..., \gamma_l\}$, where $\overline{y}$ contains all the variables in the clause that are not in $\overline{x}_Q$. By definiton of the semantics of DATALOG, the formula $(\Pi, P)\overline{t}$ is equivalent to $[S - LFP_{\overline{x}_{P^1},P^1,...,\overline{x}_{P^k},P^k\varphi_{P^1},...,\varphi_{P^k}}]\overline{t}$.

Definition 35. An acyclic logic program is one in which no intentional relation symbol occurs in the body of any clause. (At first glance this is a stronger condition than the usual definition of acyclicity, but by successively unfolding head atoms used in the body of a clause every acyclic logic program in the usual sense can easily be brought to this form.)

2.2 Probabilistic Logic programs

Definition 36. Probabilistic logic programs are logic programs in which some of the facts or rules are annotated with probabilities. [2]

Example 5. An example of a probabilistic logic program:

```
1 0.3::blue(X) :- node(X).
2 0.2::connected(X,Y) :- node(X), node(Y).
```

In the implementation of this work probabilities are not considered. This means we can apply the algorithm on the probabilistic logic program without having any annotated probabilities. If the algorithm is applied to this probabilistic logic program then the result will be that for large domains every node is connected via transitive closure. This knowledge can be used to compute for Example 5 that the probability for two nodes to be connected and both are blue is nine percent.

2.3 Least Fixed-Point Logic

The following definitions are taken from the work of Kreutzer [9]. This works gives a good overview about (least) fixed-point logics which are used to model recursion in the Prolog algorithm of this thesis.

Definition 37. Fixed-point logics are logics with an explicit operator for forming fixed points of definable mappings. They are well suited for modelling recursion in logical languages.

Least fixed-point logic was already introduced in the introduction at Definition 12 with all its properties.

Definition 38. Let A be a set and $F : Pow(A) \rightarrow Pow(A)$ be a function. F is called monotone if for all $X \subseteq Y \subseteq A, F(X) \subseteq F(Y)$. A fixed point P of F is any set $P \subseteq A$ such that $F(P) = P$. A least fixed-point of F is a fixed point that is contained in any other fixed point of F.

As mentioned above in 2.1 a DATALOG formula can be written as an S-LFP formula $[S - LFP_{\overline{x}_{P^1},P^1,...,\overline{x}_{P^k},P^k\varphi_{P^1},...,\varphi_{P^k}}]\overline{t}$. Kreutzer showed in his work in Lemma 3.19 on

page 41 that simultaneous fixed points do not increase the expressive power of LFP. This means formulas in S-LFP are equivalent to formulas in LFP. Lemma 3.18 on page 39 it is stated that: For any system S of fomulae in LFP, positive in their free fixed-point variables, $[\mathbf{lfp}R_i : S](\bar{t})$ is equivalent to a formula in LFP (without simultaneous inductions).

Proof. Let

$$S := \begin{cases} R_1\overline{x}_1 & \leftarrow & \varphi_1(R_1, ..., R_{k,\overline{x}_1}) \\ & & . \\ & & . \\ & & . \\ R_{k-1}\overline{x}_{k-1} & \leftarrow & \varphi_{k-1}(R_1, ..., R_{k,\overline{x}_{k-1}}) \\ R_k\overline{x}_k & \leftarrow & \varphi_k(R_1, ..., R_{k,\overline{x}_k}) \end{cases}$$

be a system of formulas in LFP. Then $[\mathbf{lfp}R_1 : S]$ is equivalent to the formula $[\mathbf{lfp}R_1 : T]$, where

$$T := \begin{cases} R_1\overline{x}_1 & \leftarrow & \varphi_1(R_1, ..., R_{k-1}, R_k\overline{u}/[\mathbf{lfp}_{R_k,\overline{x}_k}\varphi_k](\overline{u}), \overline{x}_1) \\ & & . \\ & & . \\ & & . \\ R_{k-1}\overline{x}_{k-1} & \leftarrow & \varphi_{k-1}(R_1, ..., R_{k-1}, R_k\overline{u}/[\mathbf{lfp}_{R_k,\overline{x}_k}\varphi_k](\overline{u}), \overline{x}_k) \end{cases}$$

The new system T is obtained from S by removing the rule R_k and substituting in the formulas φ_1 to φ_{k-1} every occurrence of an atom $R_{\overline{u}}$ by the least fixed-point of φ_k.

This procedure of substituting in the formulas φ_1 to φ_{k-1} is used in the implementation of this thesis. Substituting in the formulas comes in handy to get formulas of LFP without losing any information. The order of substituting in the formulas is computed in the algorithm and is described later on. These steps work fine for the case with only positive clauses in the given logic program. For clauses which contain negated predicates the order for substituting in the formulas and computing the formulas is depending on a strata.

2.4 Iterative Fixed Point Logic

The work of Blass et. al extends LFP to a logic containing the more powerful iterative-fixed-point operator [4]. Any structure with universe $\{1, 2, ..., n\}$ of sentence φ of this extended logic approaches 0 or 1, this means false or true, as n tends to infinity. In their work the LFP formation rule is described as the following:

Definition 39. Let φ be a formula with only positive occurrences of the l-ary predicate symbol $\dot{P}$, let $v_1, .., v_l$ be distinct variables, and let $u_1, ..., u_l$ be variables. Then

$$(u_1, ..., u_l) \in (\mathbf{LFP}, \dot{P}, v_1, ..., v_l)\varphi$$

$u_1, ..., u_l$ are the free variables of the formula φ and $v_1, ..., v_l$ are the bound variables of this formula. What a LFP formula is was already defined in Definition 12 but here another notation of it was given to emphasise the syntactic similarity to the IFP formula which is introduced in the following:

Definition 40. Let φ be a formula, P an l-ary predicate symbol, $v_1, ..., v_l$ distinct variables, and $u_1, ..., u_l$ variables. Then

$$(u_1, ..., u_l) \in (\mathbf{IFP}, \dot{P}, v_1, ..., v_l)\varphi$$

is also a formula.

The syntactic properties of this new formula are the same as for **LFP**. If the predicate P occurs only positively in φ, then IFP and LFP agree. Therefore the logic of First-Order (FO) + IFP extends the logic FO + LFP by adding the iterative fixed point formation rule to the formation rules of first-order logic. The question arises as to how the individual steps of the iteration leading to the iterative fixed-point can best be shown. The authors therefore presented a notation for the iteration steps:

Definition 41. Iteration Stage Formation Rule. For φ, $\dot{P}$, $\overline{v}$, $\overline{u}$ as in the preceding two formation rule and for α an ordinal number

$$(u_1, ..., u_l) \in (\alpha, \dot{P}, v_1, ..., v_l)\varphi$$

is a formula.

This formation rule has the same syntactic properties as IFP except in place of the 'iterative fixed-point' 'the αth stage of P_α' is notated. The iteration stage formation rule adds no expressive power to first-order logic unless α is infinite. For this work we only look at α with finite value. For the first iteration step $\overline{u} \in (0, \dot{P}, v)\varphi$ is always false and $\overline{u} \in (k+1, \dot{P}, v)\varphi$ is equivalent to the result of first substituting $\overline{u}$ for $\overline{v}$ in φ. For this step renaming the bound variables of the formula can be necessary. After that every subformula of the form $\dot{P}(\overline{w})$ (for arbitrary $(\overline{w})$ is replaced with $\overline{w} \in (k, \dot{P}, v)\varphi$. Finally the disjunction of the result with $\overline{u} \in (k, \dot{P}, v)\varphi$ is formed. An example of how this formation rule works is shown in the following.

Example 6. Given the formula $\varphi := e(x, y) \vee \exists a(t(x, a) \wedge e(a, y))$.
The result of $x, y \in (0, \dot{t}, a)\varphi$ is false, 0 is the iteration step number. In the next step $x, y \in (1, \dot{t}, a)\varphi$ results into $e(x, y) \vee \exists a(false \wedge e(a, y)) \vee false$. With basic knowledge of first-order logic this is equivalent to $e(x, y)$. $x, y \in (2, \dot{t}, a)\varphi$ results into $(e(x, y) \vee \exists a(e(x, a) \wedge e(a, y))) \vee e(x, y)$. For the next step renaming bound variables has to be done. Therefore $x, y \in (3, \dot{t}, a)\varphi$ leads to $(e(x, y) \vee \exists a(((e(x, a)) \vee \exists a_3(e(x, a_3) \wedge e(a_3, a))) \vee e(x, a)) \wedge e(a, y) \vee (e(x, y) \vee \exists a(e(x, a) \wedge e(a, y))) \vee e(x, y)$.

For finite α any first-order formula with the iteration stage formation rule (FO + IS) can be reduced to a first-order formula with the help of recursive application. Now that the iteration stage formation rule was explained the question arises on how to calculate the total number of iterations α to reach the fixed-point. As k increases the sequence of predicates defined by $\overline{u} \in (k, \dot{P}, \overline{v})\psi$ can not strictly increase asymptotically more than $p(l) = 2^{m \cdot l^r}$, where l is the number of free variables of ψ and m and r are the number of predicate symbols and the maximum arity of a predicate symbol of ψ. From the preceding it follows $\overline{u} \in (IFP, \dot{P}, \overline{v})\psi$.

IFP can be replaced by $p(l)$ or any larger number. Doing this any FO + IFP formula can be transformed into a formula of FO + IS in which the iteration stage formation rule is applied only with $\alpha = p(l)$. As mentioned before FO + IS formulas can then be reduced to first-order formulas.

2.5 Quantifier-free Formulas

Since determinate logic programs correspond quantifier-free formulas, we need to find a way to eliminate quantifiers. In the next part the process of quantifier elimination, in

particular asymptotic quantifier elimination, will be presented. All information and steps about this process is taken from the bachelor thesis of Marian Lingsch Rosenfeld [10].

Quantifier elimination is a process in which a formula φ with quantifiers is rewritten to a formula ψ without any quantifiers. One version of this process is called asymptotic quantifier elimination with respect to families of distribution of measures (P_n) over the set of finite models with universe size $n \in \mathbb{N}$ and a formula φ. Here a quantifier-free formula ψ is constructed such that $\lim_{n\to\infty} P_n(\varphi \leftrightarrow \psi) = 1$. From asymptotic quantifier elimination the 0-1 law for finite models follows, which states that for any closed formula φ in first-order logic without constants nor function function symbols $\lim_{n\to\infty} P_n(\varphi) \in \{0, 1\}$.

2.5.1 Exclusive Quantifiers

In his work Lingsch Rosenfeld introduced exclusive quantifiers and proved that closed formulas containing only exclusive quantifiers obey the 0-1 law.

Definition 42. An exclusive existential (universal) quantifier $\exists x/x_1, ..., x_n (\forall x/x_1, ..., x_n)$ read as there exists an (for all) x other than $x_1, ..., x_n$ is defined by:

- $\exists x/x_1, ..., x_l = \exists x(x \neq /x_1 \wedge ... \wedge x \neq x_l \wedge \varphi)$.
- $\forall x/x_1, ..., x_l = \forall x(x \neq /x_1 \wedge ... \wedge x \neq x_l \wedge \varphi)$.

Definition 43. A Γ-formula φ is a formula which contains no conventional quantifiers. This means φ can still contain exclusive quantifiers. The excluded variables must contain all free variables of the formula the quantifier is being applied to.

Definition 44. A free elementary part in formula φ is an atomic formula in which all the variables are free.

Definition 45. In the formula $\forall x(p(x, z) \wedge q(z))$ $q(z)$ is a free elementary part and $p(x, z)$ is a non-free elementary part.

Lemma 46. *For every formula ϕ there exists a Γ-formula ψ such that $\phi \equiv \psi$.*

Example 7. Applying Lemma 46 results in:

$$\begin{aligned}
&\forall x \exists y((p(x, y, z) \wedge q(x)) \vee p(x, x, z)) \\
&\equiv \forall x(((\exists y/z, x(p(x, y, z) \wedge q(x))) \vee (p(x, z, z) \wedge q(x)) \vee (p(x, x, z) \wedge q(x))) \vee p(x, x, z)) \\
&\equiv (\forall x/z((\exists y/z, x(p(x, y, z) \wedge q(x))) \vee (p(x, z, z) \wedge q(x)) \vee (p(x, x, z) \wedge q(x))) \vee p(x, x, z)) \wedge \\
&\quad (((\exists y/z, x(p(z, y, z) \wedge q(z))) \vee (p(z, z, z) \wedge q(z)) \vee (p(z, z, z) \wedge q(z))) \vee p(z, z, z)).
\end{aligned}$$

2.5.2 Disjunctive Normal Form

Definition 47. A formula φ is in disjunctive normal form (dnf) if $\varphi = \bigvee_{i\in I} \bigwedge_{j\in J} \varphi_{ij}$, where φ_{ij} is a literal, for some finite index sets I, J_i.

To visualise this a small example of a formula in dnf and a non-dfn formula is given below:

Example 8. $(p(x) \wedge q(x)) \vee \neg q(x)$ is in dnf, while $\neg(p(x) \wedge q(x))$ is not

Lingsch Rosenfeld then proceeds by proofing the following:

Lemma 48. *For every formula ϕ without quantifiers there exists a formula ψ without quantifiers in disjunctive normal form such that $\phi \equiv \psi$.*

Without quantifiers means in this case without any classical quantifiers. Formulas with exclusive quantifiers fulfill Theorem 48. Now the question arises how a possible algorithm for building the disjunctive normal form of a formula can look like. Lingsch Rosenfeld provided a pseudo-code for such an algorithm:

Listing 1: Pseudo-code for building the disjunctive normal form of a formula

```
def build_negation_normal_form(phi):
    if phi == not (p or q):
        return (not p) and (not q)
    elif phi == not (p and q ) :
        return (not p) or (not q)
    elif phi == not not p :
        return p
    else:
        return phi

def build_dnf_rec(phi) :
    if phi == p or q :
        return build_dnf_rec (p) or build_dnf_rec(q)
    elif phi == p and (q or r) :
        s = build_dnf_rec(p)
        return (s and build_dnf_rec(q)) or (s and build_dnf_rec(r))
    elif phi == (q or r) and p :
        s = build_dnf_rec(p)
        return (s and build_dnf_rec(q)) or (s and build_dnf_rec(r))
    else:
        return phi

def build_dnf(phi):
    psi = build_negation_normal_form(phi)
    new psi = build_dnf_rec(psi)
    while new psi != psi :
        psi = new_psi
        new psi = build_dnf_rec(psi)
    return psi
```

This pseudo-code was used as a base in the implementation of this thesis for building formulas in disjunctive normal form. Later on in section 3.5 it is shown what needs to be added to realise this pseudo-code with Prolog.

2.5.3 Replace non-free elementary parts with either true or false

To apply quantifier elimination on first-order formulas they have to be separated into free elementary parts and non-free elementary parts.

Lemma 49. *Every Γ-formula ϕ can be represented as $\bigvee_{i\in I} \phi_i \wedge \psi_i$ for some index set I, where ϕ_i contains only free elementary parts and ψ_i contains non-free elementary parts.*

Lingsch Rosenfeld proved this in his work and pointed out that there are only finitely many quantifiers in ϕ and the free elementary parts are moved up one quantifier each time

the procedure of separating the parts is applied. This means that at some point all free elementary parts will be removed from the range of all quantifiers.

Definition 50. Let ϕ be a formula and $P_n : \mathcal{M} \to (0,1)$ be a probability measure over the set of all models with n elements, arising from a list of probabilistic facts. Then ϕ is called 0-admissible if for every $k \in \mathbb{N}$

$$\lim_{n\to\infty} n^k P_n(\phi) = 0.$$

ϕ is called 1-admissible if for every $k \in \mathbb{N}$

$$\lim_{n\to\infty} n^k (1 - P_n(\phi)) = 0.$$

If ϕ is 0-admissible or 1-admissible it is called admissible.

He then continues by proofing following Lemma:

Lemma 51. *Let $\phi = \Pi x/y_1, ..., y_l \psi$ where ψ does not contain quantifiers, free elementary parts, constant symbols nor function symbols. Then it is the case that ϕ is admissible.*

Now the question is what will happen to the non-free elementary parts? Lingsch Rosenfeld showed in his Theorem 3.1.20 what asymptotically happens:

Theorem 52. *Let $P_n : \mathcal{M} \to (0,1)$ be a probability measure over the set of all models with n elements. Then every Γ-formula ϕ with non-free elementary parts, constant symbols nor function symbols is admissible. This implies that $\lim_{n\to\infty} P_n(\phi) \in \{0,1\}$.*

How this theorem can be applied is shown in the example below:

Example 9. Let P_n be as in the preceding and p be a unary predicate symbol with $P_n(p(x)) = s \in (0,1)$ then:

$$\lim_{n\to\infty} P_n(\forall x p(x)) = \lim_{n\to\infty} s^n = 0.$$

$$\lim_{n\to\infty} P_n(\exists x p(x)) = \lim_{n\to\infty} 1 - (1-s)^n = 1.$$

0 means in this case 'false' and 1 'true'. For formulas with (exclusive) quantifiers we can asymptotically replace non-free elementary parts with 'false' if they are surrounded by the $\forall$ quantifier and with 'true' if they are surrounded by the $\exists$ quantifier. Let's look at an example on how the replacement can look like:

Example 10. Given the formula: $e(x,y) \vee \exists z/x, y(e(x,z) \wedge e(z,y)) \vee (e(x,x) \wedge e(x,y) \vee (e(x,y) \wedge e(y,y))$. The non-free elementary parts $(e(x,z) \wedge e(z,y))$ can be replaced with 1 (true) according to Theorem 52. This results in: $e(x,y) \vee \exists z/x, y(true \wedge true) \vee (e(x,x) \wedge e(x,y) \vee (e(x,y) \wedge e(y,y))$ and can then be reduced to $true$.

After replacing the non-free elementary part we can reduce the formulas according to common rules of first-order logic. This reduction leads in the end to quantifier-free formulas which we now can use to build the determinate logic program. This is the main goal of the implementation of this thesis.

2.6 Building Determinate Logic Programs

As mentioned above we need to transform the formulas back into a logic program which now is determinate. Instead of going through the whole iteration stage formation rule until the fixed-point is reached we apply the quantifier elimination process to reduce computation time since the number of iterations α can get very high.

The iteration stage formation rule process is applied to a formula φ: Recall the Definition 26 of a clause and with the intentional symbol P in its head. It has the form: $P\overline{x}_{\mathcal{P}} \leftarrow \gamma_1, ..., \gamma_l$. $\overline{x}$ are the free variables of the predicate symbol P. The formula φ that we get after transforming it to a formula of least fixed-point logic, applying the iteration stage formation rule and have done quantifier elimination on it has the form: $\varphi := \bigvee_{i=1,..,l}^{l}(\gamma_{1i}, ..., \gamma_{li})$.

This formula is split at the disjunctions and then can be rewritten to a logic program of form: $\Pi := \{P(\overline{x}) \leftarrow \gamma_{1i}, ..., \gamma_{li}\}_{i=1,...,l}^{l}$. The base of transforming a formula back to a logic program is taken from Lemma 9.1.2 of Ebbinghaus and Flum [6].

3 Methods

In this part the actual algorithm for approximating (probabilistic) logic programs is explained. The implementation is implemented the programming language 'Prolog'. Prolog was chosen since its syntax and semantics are similar to (probabilistic) logic programming. Also for the algorithm the Prolog dialect SWI-Prolog [3] was used. To start the algorithm on a local machine it is necessary to download SWI-Prolog first.

3.1 DATALOG program and transformation into S-LFP formulas

The algorithm starts with a logic program for which the presentation needs to be simplified. For any intentional symbol P of logic program Π of arity r there should be distinct variables $\overline{x}_P = x_{P,1}...x_{P,r}$ such that any clause in Π with head predicate P has the form $P\overline{x}_P \leftarrow \gamma_1, ..., \gamma_l$. If that is not the case suitable changes must be carried out.

Example 11. The algorithm rewrites OldClause to NewClause:

```
1 OldClause = [r-[x,y,x]-[p-[x], s-[x]]].
2 NewClause = [r-[x, y, x2]-[p-[x], s-[x], same-[x, x2]]].
```

Example 12. The algorithm rewrites OldClause to NewClause and makes suitable changes to free variables in the body of clauses:

```
1 OldClause = [t-[x,y]-[e-[x,y]],t-[x,z]-[t-[x,y], e-[y,z]]].
2 NewClause = [t-[x, y]-[e-[x, y]], t-[x, y]-[t-[x, at], e-[at, y]]].
```

In the next step the changed logic program formulas are transformed into simultaneous fixed-point logic formulas. For every intentional P the formula $\varphi_P(\overline{x}_P) := \bigvee\{\exists\overline{y}(\gamma_1, ..., \gamma_l) \mid P\overline{x}_P \leftarrow \gamma_1, ..., \gamma_l \ in \ \Pi\}$, where $\overline{y}$ contains the variables $\gamma_1, ..., \gamma_l$ that are distinct variables in $\overline{x}_P$ is constructed. This is the case if the intentional symbols $P_1, ..., P_k$ in Π do not occur negated in the body of any clause. For further steps the S-LFP formulas are saved into a list.

The algorithm starts with the compute_lp_form function.

Listing 2: The starting point of the algorithm

```
compute_lp_form(Program, Pred, SubVars,SortedLogProg) :-
    contains_term(not(_), Program),
    compute_for_negation(Program, Pred,SubVars,Freevars, Lfp),
    compute_alpha(Lfp,AllLfps, _,AlphaConstr),
    replace_lfps(Lfp, AlphaConstr, _),
    get_iter_numbers(AlphaConstr, Iterations),
    reverse(AllLfps,RevAllLfps),
    reverse(Iterations, RevIterations),
    resolve_isfr(RevAllLfps,RevIterations,RevAllLfps,ReducedInnerForm),
    flatten1(ReducedInnerForm, FlatReducedInnerForm),
    convert_to_lp(Pred,Freevars, FlatReducedInnerForm, LogProg),
    sort(LogProg, SortedLogProg),!.

compute_lp_form(Program, Pred,SubVars,SortedLogProg) :-
    \+contains_term(not(_), Program),
    compute_for_pos(Program, Pred, SubVars,Freevars,Lfp),
    compute_alpha(Lfp, AllLfps, _,AlphaConstr),
    replace_lfps(Lfp, AlphaConstr, _),
    get_iter_numbers(AlphaConstr, Iterations),
    reverse(AllLfps,RevAllLfps),
    reverse(Iterations, RevIterations),
    resolve_isfr(RevAllLfps,RevIterations,RevAllLfps,ReducedInnerForm),
    flatten1(ReducedInnerForm, FlatReducedInnerForm),
    convert_to_lp(Pred,Freevars, FlatReducedInnerForm, LogProg),
    sort(LogProg, SortedLogProg),!.
```

First of all the algorithm checks if the logic program contains any negated predicates because for clauses with negated parts some additional steps are needed. But for both positive and negated clauses the computation of the S-LFP formulas work the same.

Example 13. An example call for the compute_lp_form function can look like this:

```
/*compute_lp_form(Datalog, DesiredPredicate, SubstituteVars, LP).*/
compute_lp_form([t-[x,y]-[e-[x,y]],t-[x,z]-[t-[x,y], e-[y,z]]], t, [a,b,c], LP).
```

This call will give a determinate logic program for predicate t as result.
The steps of making necessary changes to the clauses and transforming for each intentional symbol of the logic program all happens in the following function:

Listing 3: Structure of datalog_to_formula

```
 datalog_to_formula(Datalog, Subvars, slfp(Preds,ReducedPhis,FreeVars)) :-
  Datalog = Pi-P,
  intentionals(Pi, Ipreds),
```

```
4   do_substitution(Pi, Ipreds, Subvars, NewPi),
5   ord_subtract(Ipreds,[P],Ipreds_aux),
6   Preds = [P|Ipreds_aux],
7   getFreeVarsInRightOrder(Preds,NewPi, FreeVars),
8   maplist(program_to_formula(NewPi),Preds,Phis),
9   maplist(reduced_form, Phis, ReducedPhis).
```

First the intentional symbols are saved into 'Ipreds'. After that the do_substitution function is called for doing the necessary changes to the clauses as explained in Example 11 and Example 12. For duplicate variables '[x,x]' in the head of a clause the relation 'same-[x,x2]' is used. Therefore it is blocked for any other use. For each intentional symbol the suitable changes are applied.

Clauses with the same intentional symbol get the same head variables and 'SubVars' is used to replace former head variables which had occurred in the body of the clause as seen in Example 12. 'SubVars' can be any list containing atoms that are different from the variables of the DATALOG program. This list must be created by the user and then put into the 'compute_lp_form' as a parameter.

It is also checked if there are duplicate variables in the head of the clause. If yes, suitable changes are made as seen in Example 11, and the fixed predicate 'same' is added for the duplicate variables. This also works for multiple duplicates. In the end, all changed clauses are stored together with the unchanged clauses. The resulting DATALOG program is now prepared for the transformation into an S-LFP formula.

After executing line 4 of Listing 3 the DATALOG of Example 12 is changed to the form of 'NewClause'. The function 'program_to_formula' is then used to transform the DATALOG program into an S-LFP formula. 'Preds' are all the intentional symbols ordered with the 'P' being the first one. The 'P' is given at the beginning with the compute_lp_form function.

Depending on the position of the predicates in 'Preds' the free variables for each intention symbol are stored in 'FreeVars'. Later on, the free variables of 'P' will be the head variables of the resulting logic program. To understand how the transformation from logic program to formula looks like, the structure of 'program_to_formula' and its helpers is shown:

Listing 4: Structure of program_to_formula and its helpers

```
1   program_to_formula(Program,Q,Phi) :-
2       program_to_formula1(Q,Program,Phi).
3
4   program_to_formula1(_,[],false).
5   program_to_formula1(Q,[Head-Body|Clauses],F or Fs) :-
6       Head = Q-_,
7       clause_to_formula(Head-Body,F),
8       program_to_formula1(Q,Clauses, Fs).
9   program_to_formula1(Q,[Head-_|Clauses],F) :-
10      Head \= Q-_,
11      program_to_formula1(Q,Clauses, F).
12
13  clause_to_formula(Head-Body,Psi) :-
14      include(atom, Body, AtomsInBody),
15      AtomsInBody == [],
```

```
16      variables_free_in_body(Head-Body,Freevars),
17      clause_to_formula_aux(Head-Body,Freevars,Psi).
18
19  clause_to_formula(Head-Body,Psi) :-
20      include(atom, Body, AtomsInBody),
21      AtomsInBody \= [],
22      clause_to_formula_aux(Head-Body,[],Psi).
23
24
25  clause_to_formula_aux(_-Body,Freevars,exists(Freevars,Psi)) :-
26      conjoin_body(Body,Psi).
27
28  conjoin_body([],[]).
29  conjoin_body(Parts,Conj and true) :-
30      Parts \= [],
31      maplist(term_to_atom,Parts, AtomicTerm),
32      atomic_list_concat(AtomicTerm, ' and ', AtomicConjunction),
33      term_to_atom(Conj, AtomicConjunction).
```

First the bodies of the relevant clauses are taken and transformed into conjunctions adding 'and true' at the end with the help of 'conjoin_body'. For bodies which contain variables that differ from the free variables of the relevant relation, an existential quantifier is added to the conjunction containing the different variables as bound variables. This can be seen in line 25 of Listing 4.

Bodies which do not have any other variables than the free ones contain an empty list of bound variables. The existential quantifier with empty bound variables is later on discarded. To each existential quantifier formula an 'or false' is added. In the end all these transformed bodies are written together as a disjunction with the content of one part of the disjunction being a conjunction. For Example 12 this leads to this result:

Example 14. Example of how the algorithm transforms a DATALOG program into a formula which is not reduced yet

```
program_to_formula([t-[x, y]-[e-[x, y]], t-[x, y]-[t-[x, at], e-[at, y]]],t,Formula),
Formula = exists([],e-[x, y] and true) or (exists([at],t-[x, at] and e-[at, y] and true)
          or false).
```

Because the formula can contain empty bound variables and unnecessary 'true' and 'false' the S-LFP formulas are reduced. Parts which are already reduced are unchanged after applying the function 'reduced_form'.

Example 15. Example of how the algorithm transforms a DATALOG program into an S-LFP formula

```
DatalogForm = [t-[x,y]-[e-[x,y]],t-[x,z]-[t-[x,y], e-[y,z]]]-t-[x,y],
S-LFP = exists([],e-[x, y] and true) or (exists([at],t-[x, at] and e-[at, y] and true)
        or false),
Reduced_S-LFP = e-[x, y] or exists([at],t-[x, at] and e-[at, y]).
```

Example 16. The result of datalog_to_formula of a formula with negation

```
DatalogForm = [p-[x]-[not(r-[x, y]), q-[x]], p-[x]-[r-[x, y], not(q-[x])],
               r-[x, y]-[q-[x]]],
Result = slfp([p,r],
              [exists([ap],not(r-[x, ap]) and q-[x]) or exists([ap],r-[x, ap]
              and not(q-[x])),q-[x]],
              [[x], [x, y]]).
```

Here for each of the intentional symbols ‘p’ and ‘r’ the S-LFP formula is computed and the free variables are stored. They all have the same position in the list surrounding them.

3.2 Useful helper functions

As mentioned before the ‘reduced_form’ function is used to reduce a formula. This includes empty bound variables and redundant parts in either a conjunction or disjunction. This helper function is used several times in the algorithm to avoid large formulas which also reduces the run time of the algorithm.

Example 17. Example with empty bound variables

```
reduced_form(exists([], e-[x,y]), RedForm1),
RedForm1 = e-[x,y],
reduced_form(forall([], e-[x,y]), RedForm2),
RedForm2 = e-[x,y].
```

Example 18. Example with ‘true’ and ‘false’

```
reduced_form(exists([at], true or e-[x,y]), RedForm1),
RedForm1 = true,
reduced_form(exists([at], true and e-[x,y]), RedForm2),
RedForm2 = exists([at], e-[x,y],
reduced_form(exists([at], false or e-[x,y]), RedForm3),
RedForm3 = exists([at], e-[x,y],
reduced_form(exists([at], false and e-[x,y]), RedForm4),
RedForm4 = false.
```

For Example 18 the result is the same using the ‘forall’ quantifier instead of the ‘exists’ quantifier. Also ‘reduced_form’ discards redundant parts in a formula.

Example 19. Example with redundant parts

```
reduced_form((e-[x,y] and e-[x,y]) or t-[x,y], RedForm1),
RedForm1 = (e-[x, y] or t-[x, y]),
reduced_form(e-[x,y] or e-[x,y] or t-[x,y], RedForm2),
RedForm2 = (e-[x, y] or t-[x, y]).
```

The bracketing of the first formula in of Example 19 must be considered or else reducing the formula does not work since the operators 'and' and 'or' were implemented left-associative. But in order to avoid large disjunctions later on the 'notsath' function was implemented which is explained in Listing 27. Taking a look at the structure of 'reduced_form':

Listing 5: Structure of reduced_form

```
1  reduced_form(Phi or false, Phi_r) :-
2      reduced_form(Phi, Phi_r),!.
3  reduced_form(false or Phi, Phi_r) :-
4      reduced_form(Phi, Phi_r),!.
5  reduced_form(_ or true, true) :- !.
6  reduced_form(true or _, true) :- !.
7  reduced_form(_ and false, false) :- !.
8  reduced_form(false and _, false) :- !.
9
10 reduced_form(Phi,Psi) :-
11     reduces_to(Phi,Psi),
12     \+reducible(Psi), !.
```

Simple cases with either 'true' or 'false' are covered from lines 5 - 8 in Listing 5. If 'false' is part of a disjunction but the rest of the disjunction is something else the outcome of 'reduced_form' is dependent on the other part as seen in lines 1 - 4.

For all other cases the 'reduced_form' call from lines 10 - 12 is executed. 'reduces_to' is responsible for bringing a formula to the reduced form and 'reducible' is used to continue with the reduction process until the formula cannot be reduced further.

Another helper function which is frequently used throughout the entire algorithm is 'copy_list'. Oftentimes variables have to be substituted, and to replicate elements of the same value in a list 'copy_list' is used:

Listing 6: Structure of copy_list

```
1  copy_list(L, N, Copies) :-
2      length(Lists, N),
3      maplist(=(L), Lists),
4      append(Lists, Copies).
```

Here 'L' is the element which should be replicated, 'N' is the number of times the element should be replicated and 'Copies' is then the resulting list.

3.3 Building the LFP formula

3.3.1 Stratification for clauses with negated parts

Logic programs with negated parts have to be stratified. During this process each predicate, intentional and extensional, is labeled with a number. This is called a stratum.

The stratum gives the order in which the LFP formula of this predicate should be computed. In the end each of the stratum are stored in a list called strata. With the help of the Prolog library 'Constraint Logic Programming over Finite Domain' (clpfd) the

stratification is implemented [1]. This library supports integer linear programming and provides these arithmetic constraints:

- Expr1 #= Expr2 Expr1 equals Expr2
- Expr1 #\= Expr2 Expr1 is not equal to Expr2
- Expr1 #>= Expr2 Expr1 is greater than or equal to Expr2
- Expr1 #=< Expr2 Expr1 is less than or equal to Expr2
- Expr1 #> Expr2 Expr1 is greater than Expr2
- Expr1 #< Expr2 Expr1 is less than Expr2

The following code snippet shows the usage of the '#<' operator since it is desired that if negated predicates in the body of a clause also are a head-predicate of a clause the LFP formula of this clause should be computed first.

Listing 7: Implementation of the stratification

```
:- use_module(library(clpfd)).
stratify(Program,Predicates,Strata) :-
    length(Strata,N),
    Strata ins 0..N,
    stratification(Program,Predicates,Strata),
    once(labeling([],Strata)).

stratification([],_,_).
stratification([Clause|Program],Predicates,Strata) :-
    safe_clause(Clause,Predicates,Strata),
    stratification(Program,Predicates,Strata).

safe_clause(_-[],_,_).
safe_clause(Head-[Body1|Bodyrest],Predicates,Strata) :-
    Body1 = not(Atom),
    Head = Headpred-_,
    Atom = Bodypred-_,
%%    select(Headpred,Predicates,H,Strata),
    nth1(N_H,Predicates,Headpred),
    nth1(N_H,Strata,H),
%%    select(Bodypred,Predicates,B,Strata),
    nth1(N_B,Predicates,Bodypred),
    nth1(N_B,Strata,B),
    B #< H,
    safe_clause(Head-Bodyrest,Predicates,Strata).
safe_clause(Head-[Body1|Bodyrest],Predicates,Strata) :-
    Body1 \= not(_),
    Head  = Headpred-_,
    Body1 = Bodypred-_,
%%    select(Headpred,Predicates,H,Strata),
    nth1(N_H,Predicates,Headpred),
    nth1(N_H,Strata,H),
```

```
%%    select(Bodypred,Predicates,B,Strata),
    nth1(N_B,Predicates,Bodypred),
    nth1(N_B,Strata,B),
    B #=< H,
    safe_clause(Head-Bodyrest,Predicates,Strata).
```

The function 'stratify' starts the stratification of the logic program with the recursive function 'stratification'. It checks for each clause in the DATALOG if the relations in the body are negated or not. If there is a negated part in the body of the clause then the body-predicate will be labeled with a number lower than the head-predicate of the clause.

If the body part is not negated, the body predicate has to be labeled as equal order or lower. The ordering is then saved to a list to get the whole 'Strata' in the end. This process is done recursively for each clause in the DATALOG program with the help of the 'safe_clause' function. To visualise the result in a readable form, the 'labeling' function has to be applied to the strata. By assigning each stratum a value the next step of substituting in necessary LFP formulas can be correctly performed.

Example 20. Example call of the stratify function.

```
Example = [p-[x]-[not(r-[x, y]), q-[x]], p-[x]-[r-[x, y], not(q-[x])], r-[x, y]-[q-[x]]],
once(stratify(Example, [p,q,r], Strata)),
Strata = [1,0,0].
```

In Example 20 it is seen that the computation of the LFP formulas of predicates q and r has to be done before the computation of the LFP formulas of predicate p, since in its body a relation with negated intentional predicates occurs.

3.3.2 Building LFP formula

After building the S-LFP formula and getting the right computation order for clauses with negation by computing a stratification we construct for each intentional symbol an LFP formula from the S-LFP formula. First of all we add to each LFP formula the LFP-constructor if necessary (It is necessary if the head predicate of a clause also appears in the body of the clause). The starting point of adding or not adding the constructor is the 'compute_rec_lfp' function:

Listing 8: Structure of compute_rec_lfp

```
/* Adds recursively to all formulas the lfp-constructor if necessary */
compute_rec_lfp(_,[],_,[]).

compute_rec_lfp(Program, [FirstPred|RestPred],slfp(Preds,Slfps,FreeVars),
                AllLfps) :-
    intentionals(Program, Int),
    \+member(FirstPred, Int),
    compute_rec_lfp(Program,RestPred,slfp(Preds,Slfps,FreeVars),AllLfps).

compute_rec_lfp(Program, [FirstPred|RestPred],slfp(Preds,Slfps,FreeVars),AllLfps) :-
    intentionals(Program, Int),
    member(FirstPred, Int),
```

```
    compute_lfp(FirstPred, slfp(Preds,Slfps,FreeVars),FirstLfp, _),
    append([FirstLfp], Out, AllLfps),
    compute_rec_lfp(Program, RestPred,slfp(Preds,Slfps,FreeVars), Out).
```

Since we get a stratification for clauses with negated parts and there may exist a stratum for predicates which are not a head predicate and therefore do not have an S-LFP formula we need to check with '\+member(FirstPred,Int)' if the current predicate is part of the intentional symbols of the 'Program'. If the current predicate is not an intentional one it is skipped. But if the current predicate is an intentional symbol we apply the 'compute_lfp' function on it:

Listing 9: Structure of compute_lfp and build_lfp_constructor

```
/* Picks the right S-LFP formula for the build_lfp_constructor function*/
compute_lfp(Pred,slfp(Preds, Slfp,FreeVars),Lfp,RelvFreeVars) :-
 nth1(Index, Preds, Pred),
 nth1(Index, FreeVars, RelvFreeVars),
 nth1(Index, Slfp, RelvForm),
 build_lfp_constructor(Pred, RelvForm, Lfp).

 /* Adds the lfp-constructor to formulas if necessary */
build_lfp_constructor(Pred, exists(Boundvars, Phi), lfp(Pred, exists(Boundvars, Phi))) :-
        contains_term(Pred, Phi).
build_lfp_constructor(Pred, forall(Boundvars, Phi), lfp(Pred, forall(Boundvars, Phi))) :-
        contains_term(Pred, Phi).
build_lfp_constructor(Pred, Phi, lfp(Pred, Phi)) :-
        contains_term(Pred, Phi),
        Phi \= exists(_,_),
        Phi \= forall(_,_)
build_lfp_constructor(Pred, Phi, Phi) :-
        \+contains_term(Pred, Phi).
```

Because the 'datalog_to_formula' function stores as a result the predicates and the corresponding S-LFP formula on the same index we can easily get the S-LFP formula for the current predicate with the 'nth1' function.

Now for each S-LFP formula the 'build_lfp_constructor' function is called. To check if the current predicate is also part of its corresponding S-LFP formula we use the 'contains_terms' function which is part of the Prolog library 'occurs'.

This library is helpful for finding and counting sub-terms in a term and ' contains_term($Sub, Term$)' succeeds if Sub is contained in $Term$. If it is contained the LFP constructor is added. Keep in mind that for formulas which do not contain the head predicate itself the S-LFP formula can be copied unchanged.

Example 21. Here is an example on how the result of compute_rec_lfp can look like:

```
Program = [p-[x,x]-[r-[x],s-[y]],p-[y,y]-[p-[y,x], s-[y]], r-[x]-[m-[x]]],

SLFP = slfp([p, r],[exists([y],r-[x] and s-[y] and same-[x, x2]) or
                    exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]),
                    m-[x]],[[x, x2], [x]]),
```

```
6
7   compute_rec_lfp(Program, SLFP, LFP),
8
9   LFP = [m-[x], lfp(p,exists([y],r-[x] and s-[y] and same-[x, x2]) or
10                    exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]))].
```

In Example 21 the formula for intentional symbol 'r' stays the same for S-LFP and LFP but for 'p' the LFP constructor is added which has the form 'lfp(predicate, formula)'. But these are not the final LFP formulas. In Example 21 the formula of 'p' contains a relation with 'r' which formula is 'm-[x]'. This means that in the LFP formula of 'p' the relation with 'r' has to be replaced with 'm-[x]'. The starting point of such a replacement is the 'do_replacement1' function which uses the 'do_replacement' function:

Listing 10: do_replacement function for substitution

```
1   do_replacement1([LastPred],Preds, Lfps, FinalLfps) :-
2       do_replacement(LastPred,Preds,Lfps,FinalLfp),
3       nth1(Index, Preds,LastPred),
4       nth1(Index,Lfps, LastLfp),
5       select(LastLfp,Lfps,FinalLfp, FinalLfps).
6
7   do_replacement1([FirstPred|RestPred],Preds, Lfps, EndResultLfps) :-
8       nth1(Index, Preds,FirstPred),
9       nth1(Index,Lfps, CurrLfp),
10      PlusOne is Index + 1,
11      nth1(PlusOne,Lfps, NextLfp),
12      findall(Value, sub_term(Value, [NextLfp]), SuLi),
13      include(getLiteral, SuLi, Literals),
14      include(checkPredLiteral(FirstPred), Literals, RelvLiterals),
15      compute_replacer(RelvLiterals, CurrLfp, Replacers),
16      maplist(variables_in_literal, RelvLiterals, RelvVars),
17      maplist(reduced_form,Replacers, ReducedReplacers),
18      maplist(freevars, ReducedReplacers, ReplVars),
19      maplist(reduced_form, Replacers, RedReplacers),
20      flatten(ReplVars,FlatReplVars),
21      flatten(RelvVars,FlatRelvVars),
22      sort(FlatReplVars, SortedFlatReplVars),
23      sort(FlatRelvVars, SortedRelvVars),
24      length(SortedFlatReplVars, LenReplVars),
25      length(SortedRelvVars, LenRelvVars),
26      LenReplVars == LenRelvVars,
27      maplist(replace_rec, RedReplacers, ReplVars, RelvVars, NewReplacers),
28      replace_rec(NextLfp, RelvLiterals, NewReplacers, ReplForm),
29      select(NextLfp, Lfps, ReplForm, NewLfps),
30      do_replacement1(RestPred,Preds,NewLfps, EndResultLfps).
31
32  do_replacement1([FirstPred|RestPred],Preds, Lfps, EndResultLfps) :-
33      nth1(Index, Preds,FirstPred),
34      nth1(Index,Lfps, CurrLfp),
35      PlusOne is Index + 1,
```

```
36      nth1(PlusOne,Lfps, NextLfp),
37      findall(Value, sub_term(Value, [NextLfp]), SuLi),
38      include(getLiteral, SuLi, Literals),
39      include(checkPredLiteral(FirstPred), Literals, RelvLiterals),
40      compute_replacer(RelvLiterals, CurrLfp, Replacers),
41      maplist(variables_in_literal, RelvLiterals, RelvVars),
42      maplist(reduced_form, Replacers, RedReplacers),
43      maplist(freevars, RedReplacers, ReplVars),
44      flatten(ReplVars,FlatReplVars),
45      flatten(RelvVars,FlatRelvVars),
46      sort(FlatReplVars, SortedFlatReplVars),
47      sort(FlatRelvVars, SortedRelvVars),
48      length(SortedFlatReplVars, LenReplVars),
49      length(SortedRelvVars, LenRelvVars),
50      LenReplVars \= LenRelvVars,
51      replace_rec(NextLfp, [], [], ReplForm),
52      select(NextLfp, Lfps, ReplForm, NewLfps),
53      do_replacement1(RestPred,Preds,NewLfps, EndResultLfps).
54
55
56  do_replacement(Pred,AscOrdPreds, Lfps, FinalLfp) :-
57      nth1(Index,AscOrdPreds,Pred),
58      nth1(Index,Lfps, RelvLfp),
59      findall(Value, sub_term(Value, RelvLfp), SuLi),
60      include(getLiteral, SuLi, Literals),
61      sort(Literals, SortedLiterals),
62      subtract(AscOrdPreds, [Pred], RemPreds),
63      look_for_repl_literals(RemPreds, AscOrdPreds,Lfps, SortedLiterals, RelvLfp, FinalLfp).
```

Example 22. An example call of the this function can look like this:

```
1   do_replacement1([r, p],[r, p],[m-[x], lfp(p,exists([y],r-[x] and s-[y] and same-[x, x2])
2                   or exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]))],
3                   FinalLfp),
4
5   FinalLfp = [m-[x], lfp(p,exists([y],m-[x] and s-[y] and same-[x, x2])
6               or exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]))].
```

Note that the first two elements are the intentional symbols. The first one is used for the current predicate of the recursion and the second one for indexing. The order of these two elements is given by reversing the list 'Preds' from the 'datalog_to_formula' function. This has to be done to ensure that the replacement process ends with the desired formula which becomes the main result later on. Now coming back to the process of the replacement. First the index of the current predicate of the predicate list is obtained. With this index, the corresponding formula is taken as a possible candidate for replacement in the next formula.

To get the next LFP formula, the index is increased by one and with 'nth1' the next LFP formula is taken from the list. For the next LFP formula, all sub-terms are found with the 'sub_terms' function and the 'findall' function. 'sub_terms' is also part of the

'occurs' library. To find sub-terms which have the form predicate-variables and have as predicate the current predicate of the recursion, the 'getLiteral' and 'checkPredLiteral' function are used in combination with 'include'. 'include' is part of the Prolog library 'apply' which is mainly used to apply functions on a list. 'include(:Goal, +List1, ?List2)' is a filter function for which Goal succeeds and List2 contains those element Xi of List1 for which 'call(Goal,Xi)' succeeds.

For better functionality we put the LFP formula of the current predicate into a list if it is a candidate for replacement in the next LFP formula. If it is not a candidate we continue working with an empty list. This is done with the 'compute_replacer' function:

Listing 11: Structure of compute_replacer

```
1  /* Helper predicates for computing replacers*/
2  compute_replacer([], _, []).
3
4  compute_replacer([FirstSub|RestSub], Repl, ReplList) :-
5      \+sub_term(not(_), FirstSub),
6      append([Repl], Out, ReplList),
7      compute_replacer(RestSub, Repl, Out).
8
9  compute_replacer([FirstSub|RestSub], Repl, ReplList) :-
10     sub_term(not(_), FirstSub),
11     append([not(Repl)], Out, ReplList),
12     compute_replacer(RestSub, Repl, Out).
```

Whether there exists a replacer depends on the result of 'checkPredLiteral'. If for the next LFP formula there needs to be no replacement, then 'checkPredLiteral' and 'compute_replacer' have as result an empty list. Since negated relations can occur in a formula, it has to be checked separately if their LFP formula is a candidate for substitution.

After getting all substitution candidates for the next LFP formula, we have to extract the variables of the relevant literals which have to be replaced, and also get free variables of the substitution candidate. This has to be done because sometimes the variables of the substitute are different from variables of the relevant literals. The variables of the substitute have then to be replaced by the variables of the relevant literal. In lines 16 - 27 and lines 41 - 50 of Listing 10 this procedure is implemented.

To extract the variables the 'maplist/3' function is very helpful to apply the functions 'variables_in_literal' and 'freevars' to all elements of the lists containing relevant literals and the substitute. 'maplist(:Goal, ?List1, ?List2)' is also part of the library 'apply' and is true if Goal is successfully applied on all matching elements of the list.

After that, both parts of variables are flattened and sorted. This has to be done for the actual replacement function 'replace_rec' which does not work on nested lists or with duplicate variables. Also the variables of the relevant literal and the substitute have to be of the same number. If they are not, the 'do_replacement' function does not replace relations in the next LFP formula and continues the recursion with the next predicate.

Checking the number of variables works by using then function 'length' on the list containing variables and then comparing both lengths. To understand how the 'replace_rec' function works we will take a closer look at it:

Listing 12: replace_rec and replace function for replacement in formulas

```
1  replace_rec(NewForm, [],_, NewForm).
2  replace_rec(Formula, [FirstInner|RestInner], [FirstReplInner|RestReplInner],
3              ResNewForm) :-
4      replace(FirstInner, FirstReplInner, Formula, NewForm),
5      replace_rec(NewForm, RestInner, RestReplInner, ResNewForm).
6
7
8  replace(Subterm0, Subterm, Term0, Term) :-
9          (   Term0 == Subterm0 -> Term = Subterm
10         ;   var(Term0) -> Term = Term0
11         ;   Term0 =.. [F|Args0],
12             maplist(replace(Subterm0,Subterm), Args0, Args),
13             Term =.. [F|Args]
14         ).
```

'replace_rec' makes use of the 'replace' function. This function takes 'Subterm0' of 'Term0' and substitutes 'Subterm0' with 'Subterm'. The result of this is stored in the variable 'Term'. In line 9 of Listing 12 we first check if 'Term0' is just 'Subterm0'. Then the resulting 'Term' is 'Term0'. Then it is checked if 'Term0' is just a variable and not a full term.

If yes, no replacement has to be done and 'Term' equals 'Term0'. In Prolog, a term can be separated with the built-in predicate '=../2'. '?Term =.. ?List' transforms a Term into a list whose head is the functor of Term, and whose remaining arguments are the arguments of the term.

With the help of line 12 of Listing 12 and the backtracking behaviour of Prolog, on line 13 of Listing 12, 'Term' is successfully constructed. 'replace_rec' helps if we want to substitute more than one part of a formula. Recursion then replaces then each element of the list '[FirstInner|RestInner]' with elements of the list '[FistReplInner|RestReplInner]'. After each substitution the 'NewForm' is taken for the next step in the recursion.

Example 23. Here is an example of how replace_rec is used:

```
1  replace_rec(lfp(p,exists([y],r-[x] and s-[y] and t-[x] and same-[x, x2]) or
2                  exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2])),
3                  [r-[x]],[m-[x]],
4                  NewFormula),
5  NewFormula = lfp(p,exists([y],m-[x] and s-[y] and t-[x] and same-[x, x2]) or
6                   exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2])).
```

After substituting the current LFP formula into the next LFP formula we need to store this result for the next step of the recursion. For replacing elements in a list we can use the 'select/4' function of the library 'lists'. This library provides basic predicates for list manipulation and is commonly used in the Prolog community.

'select(?X, ?XList, ?Y, ?YList)' is true if XList is unifiable with YList apart from a single element at the same position that is unified with X in XList and with Y in YList. A typical use for this predicate is to replace an element, as shown in the example below:

Example 24. Replacing an element in a list:

```
select(b, [a,b,c,b], 2, X).
X = [a, 2, c, b] ;
X = [a, b, c, 2] ;
```

For the 'do_replacement1' function we just replace the the old LFP formula with the new one in the list of all LFP formulas. This new list is then used for the next step of the recursion. To make sure that in the end the desired LFP formula does not need any substitutions anymore we apply in the last step of 'do_replacement1' the 'do_replacement' function which uses the 'look_for_repl_literals' predicate, as seen in Listing 10 line 63.

Basically it does the same as 'do_replacement1' from Listing 10 lines 14 - 28. It is just done to ensure that the last LFP formula which is the desired one really has no more relations that need to be substituted. This last LFP formula is then taken to compute the iteration steps to reach the fixed-point. The other formulas are not relevant anymore for the result of the algorithm.

3.4 Compute iteration steps for Iteration Stage Formation Rule

In this part it is shown how the theoretical steps of section 2.4 are used in the implementation of this thesis. As mentioned in this section, there exists a number α which contains the number of iterations to reach the fixed-point of an IFP formula. We first start by computing this α and then replace the LFP constructor of the LFP formula with a constructor of the form 'alpha_α'. To compute the number α and the alpha constructor, the 'compute_alpha' function is used:

Listing 13: compute_alpha function and its helpers

```
compute_alpha(Formula, AllLfps, Boundvars, AtomAlphas1) :-
    getAllLfpFormulasOfAFormula(Formula, AllLfps),
    getBoundVarsOfFormula(Formula, Boundvars),
    countPredsFreeVarsArity(AllLfps, Boundvars, Alphas),
    maplist(atom_number, AtomAlphas, Alphas),
    maplist(atom_concat(alpha_),AtomAlphas,AtomAlphas1).

countPredsFreeVarsArity(AllLfps,Boundvars,Alphas) :-
    maplist(countAllPreds, AllLfps, LenPreds),
    maplist(getHighestArityOfPreds, AllLfps, MaxArity),
    maplist(freeVarsOfLfp, AllLfps, MaybeFreevars),
    real_freevars(MaybeFreevars, Boundvars, Freevars),
    maplist(length, Freevars, LenFreevars),
    maplist(pow,LenFreevars,MaxArity,Result),
    maplist(mult,LenPreds,Result, Exp),
    length(Exp, LenForCompAlpha),
    copy_list([2], LenForCompAlpha, Base),
    maplist(pow,Base,Exp,Alphas).

countAllPreds(lfp(_, Formula), LenPreds) :-
    findall(Value, sub_term(Value, Formula), SubTermList),
        include(getLiteral, SubTermList, Literals),
```

```
23      sort(Literals, SortedLiterals),
24      extract_inner_literals(lfp([_], Formula), InnerLiterals),
25      subtract(SortedLiterals, InnerLiterals, RelvLiterals),
26      maplist(pred_in_literal, RelvLiterals, Preds),
27      sort(Preds, SortedPreds),
28      length(SortedPreds, LenPreds).
29
30  getHighestArityOfPreds(lfp(_, Formula), Max) :-
31      findall(Value, sub_term(Value, Formula), SubTermList),
32          include(getLiteral, SubTermList, Literals),
33      sort(Literals, SortedLiterals),
34      extract_inner_literals(lfp([_], Formula), InnerLiterals),
35      subtract(SortedLiterals, InnerLiterals, RelvLiterals),
36      maplist(variables_in_literal, RelvLiterals, Vars),
37      maplist(length, Vars, Lens),
38      max_list(Lens, Max).
39
40
41  freeVarsOfLfp(lfp(_, Formula), Frees) :-
42      getFreeVarsOfFormula(Formula, Frees).
```

Remember that α is defined as $2^{m \cdot l^r}$, where m is the number of predicate symbols in the formula, r the maximum arity of a predicate symbol in the formula and l the number of free variables of the formula.

To get m, we first start by extracting for nested LFP formulas all the nested parts and store them in a list with the function 'getAllLfpFormulasOfAFormula'. Getting all the nested parts with an LFP constructor makes it easier to count all the predicate symbols which occur in the formula.

After gathering all the parts which contain an LFP constructor, the 'countAllPreds' function of line 20 of Listing 13 is called and gives as result the length of the list which contains all predicates of the given LFP formula. Just unique predicates are counted, without any duplicates. From lines 20 - 28 of Listing 13 one can see the structure of the 'countAllPreds' function. It starts by collecting all literals first and then sorts them to avoid duplicates.

Then for all nested LFP formulas the occurring literals are also collected with the 'extract_inner_literal' function in line 24 of Listing 13. To avoid duplicates with the previous collected and sorted literals the 'subtract(+Set, +Delete, -Result)' predicate is used. It deletes all elements in Delete from Set. After getting all literals all predicates from the literals are extracted and stored into the list 'Preds' with the function 'pred_in_literal'.

These are also sorted to avoid duplicates since literals with the same predicate but different variables can occur. For instance 'r-[x]' and 'r-[y]'. Of this sorted predicate list the length is taken, and that is equal to m, the number of all predicate symbols of the LFP formula. To get r, the maximum arity of a predicate symbol, the 'getHighestArityOfPreds' function is used. Until line 35 of Listing 13, it works the same as 'countAllPreds', but instead of extracting the predicate symbols, the variables of each literal are extracted. With 'maplist', the 'length' function is then applied to all lists of variables.

This leads to a list of numbers and 'max_list' gives as result the largest number in the list. This value is equal to r. The last desired value is l, the number of free variables of the LFP formula. Since nested LFP formulas can occur, the computation of free variables

should just be for formulas which are surrounded by a LFP constructor. 'freeVarsOfLfp' handles that and 'getFreeVarsOfFormula' gets the free variables of further nested LFP formulas in the formula. For a simple LFP formula, this is achieved by the function 'freevars'. This function gets all variables which are not bound by quantifiers.

But to ensure that no bound variables extracted from nested LFP formulas, we get all bound variables with 'getBoundVarsOfFormula' and use 'subtract' in 'real_freevars' to delete them from the set of 'MaybeFreevars' as seen in line 15 of Listing 13. Of the list of free variables the length is taken to get l.

Now that all relevant values are computed, the computation of α is no problem. From lines 14 - 18 of Listing 13, the computation of α is executed and the value is stored in the variable 'Alphas'. For each nested LFP formula, a new alpha constructor is needed.

In line 5, this numerical value is transformed into an atom, so that it is connectable with the constructor. All the new constructors are then stored in a list. This alpha constructor acts like a constructor for the IFP formula. As mentioned earlier, IFP and LFP are syntactically the same. To replace the LFP constructor with the alpha constructor, the 'replace_lfps' function is used:

Listing 14: Functions that are used for replacing the LFP constructor

```
1  /* Replace former lfp-constructor with a new constructor */
2  replace_lfps(Formula,[],Formula).
3
4  replace_lfps(Formula, [First|Rest], NewFormula) :-
5      term_to_atom(Formula, AtomFormula),
6      nth1(Index,[First|Rest], First),
7      replace_nth_word(AtomFormula,Index,lfp, First, InterResult),
8      term_to_atom(Term,InterResult),
9      replace_lfps(Term, Rest, NewFormula),!.
10
11 /* Auxilary predicate to replace the nthOccurence of ToReplace in Word with ReplaceWith */
12 replace_nth_word(Word,NthOcurrence,ToReplace,ReplaceWith,Result) :-
13     call_nth(sub_atom(Word,Before,_Len,After,ToReplace),NthOcurrence),
14     sub_atom(Word,0,Before,_,Left), % get left part
15     sub_atom(Word,_,After,0,Right), % get right part
16     atomic_list_concat([Left,ReplaceWith,Right],Result).
```

'replace_lfps(Lfp,Alpha_Constr,AlphaForm)' transforms the LFP formula from term to atom. This is done because 'replace_nth_word' just works on atoms and strings. 'replace_nth_word(AtomFormula,Index,lfp, First, InterResult)' replaces the occurrence of 'lfp' at 'Index' with 'First'.

The result is then stored in the variable 'InterResult'. Since 'replace_lfps' is a recursive function, we can replace every occurrence of the LFP constructor with the corresponding alpha constructor. The following is an example to visualise how the LFP formula is transformed to a formula with an alpha constructor:

Example 25. Example of how a LFP formula is transformed into an IFP formula:

Listing 15: Example 'replace_lfps'

```
1  replace_lfps(lfp(t,e-[x, y] or lfp(w, k-[x,y]) or exists([at],t-[x, at] and e-[at, y])),
2               [alpha_256, alpha_5],Result),
```

```
3  Result = alpha_256(t,e-[x, y] or alpha_5(w,k-[x, y])
4             or exists([at],t-[x, at] and e-[at, y])).
```

Instead of using the previous replacement function 'replace_rec', 'replace_lfps' with 'replace_nth_word' was chosen because 'replace_rec' would have replaced all occurrences of 'lfp' in the first recursion step already. 'replace_nth_word' makes it easy to replace values at certain positions.

After rewriting the LFP formula to IFP with the alpha constructor, the iteration stage formation rule is executed. First the iteration number for all LFP formulas has to be extracted from the constructor. This is done by the 'get_iter_numbers' function:

Listing 16: Extracting the number of iterations with 'get_iter_numbers'

```
1  get_iter_numbers([],[]).
2
3  get_iter_numbers([FirstAlpha|RestAlpha],Iterations) :-
4      split_string(FirstAlpha, "_", "", Split),
5      subtract(Split, ["alpha"], Result),
6      flatten1(Result, IterString),
7      number_string(IterNumber, IterString),
8      append([IterNumber], Out, Iterations),
9      get_iter_numbers(RestAlpha, Out).
```

If nested IFP formulas exist, all constructors are stored in a list and split at underscore (_). To get the numeric value of the iterations the function 'number_string' is used. All the available iterations are then stored to a list and as a result the list 'Iterations' is a list of numeric values.

In the previous part, where the function 'compute_alpha' was explained, we got all LFP formulas in a formula (including the nested ones) in a list. This list comes in handy, for the next step the execution of the iteration stage formation rule.

3.5 Implementation Of The Iteration Stage Formation Rule

The implementation of this part is based on section 2.4 and an example of the executing the iteration stage formation rule is seen in Example 6. If the desired formula is a nested IFP formula, we need to reverse the list first for further processing, since we start by resolving the inner most IFP formula first until we traversed the whole formula.

The idea is to replace all the inner IFP formulas by starting with the inner most one. To get the right order of execution the list with the iterations has to be also reversed. These preparation steps for 'resolve_isfr' are shown in line 9 and 22 of Listing 2. The structure of the function is shown below:

Listing 17: Structure of resolve_isfr

```
1  resolve_isfr([Formula],[FirstIter],[Formula],[ReducedForm]) :-
2      Formula = lfp(Pred,Form),
3      isfr1(FirstIter,0,Form,[Pred],false,ReducedForm).
4
5  resolve_isfr([Formula], _,_,[Formula]) :-
```

```
6       Formula \= lfp(_,_).
7
8   resolve_isfr([FirstForm|_], [FirstIter|RestIter],Formulas,Res) :-
9       get_inner_pred(FirstForm,Pred),
10      get_form(FirstForm,Form),
11      isfr1(FirstIter,0,Form,[Pred],false,ReducedForm),
12      length(Formulas, LenFormulas),
13      copy_list([ReducedForm], LenFormulas, EnoughReducedForm),
14      copy_list([FirstForm], LenFormulas, EnoughFirstForm),
15      maplist(replace,EnoughFirstForm,EnoughReducedForm,Formulas, NewFormulas),
16      length(NewFormulas, LenNewFormulas),
17      LenNewFormulas > 1,
18      removehead(NewFormulas, WHReducedNewFormulas),
19      resolve_isfr(WHReducedNewFormulas, RestIter, WHReducedNewFormulas,Res).
20
21  resolve_isfr([FirstForm|_], [FirstIter|RestIter],Formulas,Res) :-
22      get_inner_pred(FirstForm,Pred),
23      get_form(FirstForm,Form),
24      isfr1(FirstIter,0,Form,[Pred],false,ReducedForm),
25      length(Formulas, LenFormulas),
26      copy_list([ReducedForm], LenFormulas, EnoughReducedForm),
27      copy_list([FirstForm], LenFormulas, EnoughFirstForm),
28      maplist(replace,EnoughFirstForm,EnoughReducedForm,Formulas, NewFormulas),
29      length(NewFormulas, LenNewFormulas),
30      LenNewFormulas == 1,
31      resolve_isfr(NewFormulas, RestIter, NewFormulas,Res).
```

From lines 5 - 6 the formula stays the same if there is no fixed-point in the formula and from lines 1 - 3 in Listing 17 the iteration stage formation rule is applied to the whole formula and the result is stored in 'ReducedForm'.

The actual implementation of the iteration stage formation rule is done in the function 'isfr1'. Since IFP and LFP are syntactically similar, it is enough working on the previously obtained LFP formula. The 'isfr1' function will be presented in more detail later on in this thesis. Lines 8 - 19 of Listing 17 are executed if the IFP formula is a nested one.

Line 11 starts by applying the iteration stage formation rule on the IFP formula whose turn it is now and since the given 'AllLfps' list contains a list containing the whole formula and the nested ones we can apply the 'replace' function to all parts of this list of formulas. Below is an example to visualise this procedure:

Example 26. An example on how nested formulas are replaced in the outer formulas

```
1   ToBeReplaced = [lfp(r,m-[x] and r-[x]), lfp(r,m-[x] and r-[x]), lfp(r,m-[x] and r-[x])],
2
3   Replacer = [false,false,false],
4
5   AllLfps = [lfp(r,m-[x] and r-[x]), lfp(r,m-[x] and lfp(r,m-[x] and r-[x])),
6              lfp(p,exists([y],lfp(r,m-[x] and lfp(r,m-[x] and r-[x])) and s-[y] and
7              same-[x, x2]) or exists([y],p-[x, x] and s-[x] and same-[y, x]
8              and same-[y, x2]))],
9
```

```
10  maplist(replace, ToBeReplaced, Replacer, AllLfps, ReplacedLfps),
11
12  ReplacedLfps = [false, lfp(r,m-[x] and false),
13                  lfp(p,exists([y],lfp(r,m-[x] and false) and s-[y] and same-[x, x2]) or
14                  exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]))].
```

'ToBeReplaced' and 'Replacer' have to be of the same length as 'AllLfps' in order to work on each formula with the function 'replace' and the value of the 'Replacer' is the result of applying 'isfr1' on 'lfp(r,m-[x] and r-[x])' as seen in Listing 17 in line 11. This replacement procedure is done until only the desired LFP formula is left and no replacement can be done anymore. For Example 26 the result in the end looks like this:

Listing 18: Last result of Example 26

```
1  LastLfp = [lfp(p,exists([y],false and s-[y] and same-[x, x2])
2            or exists([y],p-[x, x] and s-[x] and same-[y, x] and same-[y, x2]))],
3  LastIteration = [Iteration],
4  resolve_isfr(LastLfp, LastIteration, LastLfp, Result),
5  Result = [false].
```

The result is '[false]' since the resolve_isfr function from lines 1 - 3 in Listing 17 is called and the iteration stage formation rule has as result 'false' for the last LFP formula. Also after each recursion step for the 'NewFormulas' which are the replaced ones the head is removed to start the next recursion step with the next possible larger nested LFP formula.

This is done to get in the end a LFP formula in which all nested LFP formulas are resolved through substitution. Now that the handling of nested LFP formulas was explained the question arises on how the essential part the implementation of the iteration stage formation rule looks like. In Example 6 it was shown how the computation for the iteration stage formation rule looks like for three steps if done by hand.

In the following the structure of the function 'isfr1' is shown and how the theoretical steps are realised with Prolog. As already mentioned in 2.4 the iteration is done from 0 - α. The next result of the next iteration is always achieved by substituting in the free variables into the bound variables.

For this step renaming bound variables can be necessary. Then for each step the previous result has to be substituted into the subformulas which are part of the fixed-point. In the end a disjunction is formed which has the form: NewResult or PrevResult. How all the steps are realised is shown below:

Listing 19: Strucutre of isfr1

```
1  isfr1(0,_,_,_,Term,ReducedTerm) :-
2      reduced_form(Term, ReducedTerm).
3
4  isfr1(Iter, Counter, Term, Pred, PrevResult, ReducedQuantTerm) :-
5      Iter > 0,
6      Counter >= 0,
7      NewCounter is Counter + 1,
8      reduced_form(PrevResult, ReducedPrevResult),
9      getBoundVarsOfFormula(Term, Boundvars),
```

```
    length(Boundvars, LenBoundvars),
    copy_list([NewCounter], LenBoundvars, EnoughIterIndices),
    maplist(atom_concat, Boundvars, EnoughIterIndices, NewBoundvars),
    freevars(ReducedPrevResult, FreevarsPrevResult),
    flatten1(Pred,FlatPred),
    find_relv_literal(Term, FlatPred, Literal),
    Literal \= [],
    variables_in_literal(Literal, LiteralVars),
    replace_vars(Boundvars, NewBoundvars, ReducedPrevResult,NewPrevResult),
    replace_rec(NewPrevResult,FreevarsPrevResult,LiteralVars,NewPrevResult1),
    term_to_atom1(Term, AtomFormula),
    term_to_atom1(Literal, AtomLiteral),
    term_to_atom1(NewPrevResult1, AtomNewPrevResult),
    replace_nth_word(AtomFormula,1,AtomLiteral, AtomNewPrevResult, InterResult),
    term_to_atom(ReplacedTerm, InterResult),
    reduced_form(ReplacedTerm, ReducedTerm),
    reduced_quant_elim(ReducedTerm, ReducedQuantTerm),
    transform_term(ReducedPrevResult, ReducedQuantTerm, TransfTerm),
    terminate_isfr(TransfTerm),!.

isfr1(Iter, Counter, Term, Pred, PrevResult, NewResult) :-
    Iter > 0,
    Counter >= 0,
    NewCounter is Counter + 1,
    NewIter is Iter - 1,
    reduced_form(PrevResult, ReducedPrevResult),
    getBoundVarsOfFormula(Term, Boundvars),
    length(Boundvars, LenBoundvars),
    copy_list([NewCounter], LenBoundvars, EnoughIterIndices),
    maplist(atom_concat, Boundvars, EnoughIterIndices, NewBoundvars),
    freevars(ReducedPrevResult, FreevarsPrevResult),
    flatten1(Pred,FlatPred),
    find_relv_literal(Term, FlatPred, Literal),
    Literal \= [],
    variables_in_literal(Literal, LiteralVars),
    replace_vars(Boundvars, NewBoundvars, ReducedPrevResult,NewPrevResult),
    replace_rec(NewPrevResult,FreevarsPrevResult,LiteralVars,NewPrevResult1),
    term_to_atom1(Term, AtomFormula),
    term_to_atom1(Literal, AtomLiteral),
    term_to_atom1(NewPrevResult1, AtomNewPrevResult),
    replace_nth_word(AtomFormula,1,AtomLiteral, AtomNewPrevResult, InterResult),
    term_to_atom(ReplacedTerm, InterResult),
    reduced_form(ReplacedTerm, ReducedTerm),
    reduced_quant_elim(ReducedTerm, ReducedQuantTerm),
    transform_term(ReducedPrevResult,ReducedQuantTerm,  TransfTerm),
    \+terminate_isfr(TransfTerm),
    isfr1(NewIter, NewCounter, Term, Pred, ReducedQuantTerm or ReducedPrevResult,
          NewResult),!.

isfr1(Iter, Counter, Term, Pred, _, ReducedQuantTerm) :-
```

```
60      Iter > 0,
61      Counter >= 0,
62      reduced_form(Term, ReducedTerm),
63      reduced_quant_elim(ReducedTerm, ReducedQuantTerm),
64      flatten1(Pred, FlatPred),
65      find_relv_literal(Term, FlatPred, Literal),
66      Literal == [].
```

To explain how it works, an example is given. It is actually the same formula from Example 6 but now with the syntax of the Prolog algorithm and detailed explanation on how the steps are implemented. For formulas which do not have fixed-point functor or reached the last step of the iteration stage formation rule, the 'isfr1' function from lines 11 - 12 in Listing 19 is applied to the formula.

The formula is just reduced according to rules of first-order logic and the result is stored in 'ReducedTerm'. Here is an explanation of the parameters of the call 'isfr1(Iter, Counter, Term, Pred, PrevResult, ReducedQuantTerm)': Iter is the number α which is the number of the iterations that have to be done until the fixed-point is reached, Counter is used to track in which iteration step the recursion is currently, Term is the formula which is inside the LFP/IFP constructor, Pred is the predicate symbol which has the fixed-point and ReducedQuantTerm is the actual formula without any quantifiers reached at the fixed-point.

Actually, to reduce computation time, after each iteration step quantifier elimination is applied to the formula with the function 'reduced_quant_elim'. It is explained later on how this is implemented. We will now concentrate on the iteration stage formation rule first.

Example 27. The formula 'GivenFormula' is given and the iteration stage formation rule is applied to it

```
1  GivenFormula = lfp(t,e-[x, y] or exists([at],t-[x, at] and e-[at, y])),
2  Iterations = 256,
3  isfr1(Iterations,0,e-[x, y] or exists([at],t-[x, at] and e-[at, y]), [t], false, Result),
4  Result = true.
```

Since the current iteration value is larger than 0 the computation starts at line 14 of Listing 19. The difference between the implementation of 'isfr1' from lines 14 - 38 and lines 40 - 67 is that in line 38 the function 'terminate_isfr' is used also to reduce computation time.

As mentioned before, the result of each iteration is a disjunction, but this disjunction can grow very large and may contain duplicates. 'terminate_isfr' is then used to detect those duplicates, and if the formula does not semantically change, the iteration stage formation rule is terminated. It is explained later on how that function is implemented. Looking at Example 27, the execution of 'isfr1' starts at line 40 of Listing 19. 'false' is here chosen as the first previous result because this is for all formulas the result in the beginning of the iteration stage formation rule, as mentioned in 2.4.

For each iteration step, the previous result is reduced to avoid unnecessarily large formulas for substituting into. To extract the bound variables and free variables that are also relevant for the substitution process, the functions 'getBoundVarsOfFormula' and 'freevars' are used. To find the subformula which has to be replaced, the function 'find_relv_literal' is used. The results of these preparation steps for the first iteration are as follows:

Listing 20: Bound variables, Free variables of the previous result and finding subformulas

```
1 getBoundVarsOfFormula(e-[x, y] or exists([at],t-[x, at] and e-[at, y]),BoundVars),
2 freevars(false, FreeVars),
3 find_relv_literal(e-[x, y] or exists([at],t-[x, at] and e-[at, y]),Subformulas),
4
5 BoundVars = [at],
6 FreeVars = [],
7 Subformulas = t-[x,at].
```

Since the previous result is just 'false', there are non-free variables. It could be possible that the previous result contains the bound vars of the current formula. 'replace_vars' renames those variables with the help of 'Counter'. For each iteration, the name of the bound variable is concatenated with the 'Counter' number. But since the previous result is still false, nothing changes:

Listing 21: Replacing bound variables

```
1 replace_vars([at],[at1],false,PrevResReplBoundVars),
2 PrevResReplBoundVars = false,
3
4 replace_rec(false,[],[x, at],NewPrevResult),
5 NewPrevResult = false.
```

Then with 'replace_rec' the changed variables of the previous result have to be substituted for the variables of the subformula. This is done to avoid duplicate variables. But since in Listing 21 the previous result, is 'false', it stays 'false' because it has no variables. Now that all preparation steps are done the previous result 'false' can be substituted for the subformula 't-[x,at]' in the formula 'e-[x, y] or exists([at],t-[x, at] and e-[at, y])'. This is done from lines 57 - 61 in Listing 19, and the result is:

Listing 22: Substitution of the subformula with the previous result

```
1 term_to_atom1(e-[x, y] or exists([at],t-[x, at] and e-[at, y]), AtomFormula),
2 replace_nth_word(AtomFormula,false,AtomResult),
3 term_to_atom(Result, AtomResult),
4 Result = e-[x, y] or exists([at],false and e-[at, y]),
5
6 reduced_form(Result, ReducedResult),
7 ReducedResult = e-[x,y].
```

For the substitution the, 'replace_nth_word' function is used. This function was already used as shown in Listing 15. The formula has to be transformed into an atom first for this process. 'term_to_atom1' is used for that since Prolog does not add any spacing for 'true' or 'false'. If the formula is not 'true' or 'false' the regular 'term_to_atom' function is called:

Listing 23: Structure of term_to_atom1

```
1  term_to_atom1(false, 'false ').
2  term_to_atom1(true, 'true ').
3  term_to_atom1(Term, Res) :-
4      Term \= false,
5      Term \= true,
6      term_to_atom(Term, Res).
```

To reduce large formulas, the 'reduced_form' function is applied as seen in line 6 of Listing 22. To check if 'isfr1' can already be terminated, the 'terminate_isfr' function is used. This is done to avoid large disjunctions. For this step the result has to be transformed into a formula of the form 'result and not(previous result)', which the 'transform_term' function does:

Listing 24: Structure of transform_term

```
1  /* A = A or B <=> B => A <=> not(B) or A <=> B and not(A) */
2  transform_term(A, B, B and FlatNotConj) :-
3      split_term_at_or(A,Split),
4      flatten(Split, FlatSplit),
5      maplist(split_term, FlatSplit, SplitConjs),
6      maplist(flatten,SplitConjs, FlatSplitConjs),
7      maplist(addnot,FlatSplitConjs, NotList),
8      maplist(list_to_conj,NotList,NotConj),
9      flatten(NotConj, FlatList),
10     list_to_disj(FlatList,DisjList),
11     flatten1(DisjList, FlatNotConj),!.
```

In the comment of line 1 it is already seen how the transformation process to the desired form looks like. An example call of this function looks like this:

Example 28. Example call of transform_term

```
1  transform_term(transform_term((a-[x] and b-[x] or e-[x]), c-[x] or d-[x],Result),
2  Result = (c-[x] or d-[x]and(not(a-[x]) and not(b-[x])) or not(e-[x])).
```

This transformed function is then the only parameter for the 'terminate_isfr' function:

Listing 25: Structure of terminate_isfr

```
1  terminate_isfr(TransfTerm) :-
2      while_dnf(TransfTerm,Dnf),
3      split_term_at_or(Dnf,SplitTerm),
4      flatten(SplitTerm, FlatSplitTerm),
5      maplist(split_term, FlatSplitTerm, SplitConjs),
6      maplist(flatten, SplitConjs, FlatSplitConjs),
7      notsat(FlatSplitConjs),!.
```

This transformed formula is then transformed into a formula in disjunctive normal form. As a base for the Prolog implementation, the pseudo-code provided by Lingsch Rosenfeld

was used, as mentioned in 2.5.2. Since Prolog does not provide any loops, the while-loop from lines 26 - 28 of Listing 1 has to be implemented with endrecursion. Therefore we implemented also the 'while_dnf' function, in addition to the already provided pseudo-code for building the negation normal form and building the dnf formula with this form. The structure of the additional endrecursive 'while_dnf' function looks like this:

Listing 26: Structure of while_dnf

```
1  while_dnf(Phi, NewPsi_r) :-
2      build_negation_normal_form(Phi,Psi),
3      build_dnf_rec(Psi, NewPsi),
4      Psi \= NewPsi,
5      while_dnf(NewPsi, NewPsi_r).
6
7  while_dnf(Phi, NewPsi) :-
8      build_negation_normal_form(Phi,Psi),
9      build_dnf_rec(Psi, NewPsi),
10     Psi == NewPsi.
```

As seen in line 10 of Listing 26, the function for building the negation normal form and building from that the disjunctive normal form has to be called until they both have the same result. An example on how the disjunctive normal form of a formula is constructed with 'while_dnf' is given below:

Example 29. Example call of while_dnf

```
1  while_dnf((p-[x] or s-[x]) and (not(p-[x]) or m-[x]),Dnf),
2  Dnf = (p-[x] or s-[x] and not(p-[x]) or m-[x],(p-[x] and
3         not(p-[x]))or(s-[x] and not(p-[x]))or((p-[x] and m-[x])or(s-[x] and m-[x])).
```

With this disjunctive normal form, it is easier to check if there are conjunctions which contain for example 'p-[x]' and 'not(p-[x])'. Forms like this can easily happen and can not be reduced by 'reduced_form' if they are at first a part of a disjunction since 'reduced_form' works on disjunctions just from the left side of the formula.

To check if a relation and its negated form occur together in a formula in a form that they can be written together into a conjunction the 'notsath' function is used. But it works on lists. So some preparation steps need to be done before applying this function. These steps are executed from lines 4 - 7 of Listing 25. The formula is first split at its disjunctions.

Example 30. Example call of split_term_at_or

```
1  split_term_at_or((p-[x] and not(p-[x]))or(s-[x] and not(p-[x]))or((p-[x] and
2                   m-[x])or(s-[x] and m-[x])),SplitTerm),
3  SplitTerm = [[[p-[x] and not(p-[x])], [s-[x] and not(p-[x])]], [[p-[x] and m-[x]],
4               [s-[x] and m-[x]]]],
5  flatten(SplitTerm, FlatSplitTerm),
6  FlatSplitTerm = [p-[x] and not(p-[x]), s-[x] and not(p-[x]), p-[x] and m-[x],
7                   s-[x] and m-[x]].
```

This function transforms a formula into a list which is split at all disjunctions, so all elements in the list are conjunctions or a single relation then. 'SplitTerm' has to be flattened with the function 'flatten' to erase the unnecessary nesting of the list. In the end, the result is a list in the form of 'FlatSplitTerm'.

Transforming a formula with the operator '=..' would not work in this case to reach the desired form of list. It would not consider all 'or' operators and not split at all disjunctions. The formula is then further split at its conjunctions by the function 'split_term'.

Example 31. Example call of split_term

```
maplist(split_term([p-[x] and not(p-[x]), s-[x] and not(p-[x]), p-[x] and m-[x],
                    s-[x] and m-[x]], SplitConj),
SplitConjs = [[[p-[x]], [not(p-[x])]], [[s-[x]], [not(p-[x])]], [[p-[x]], [m-[x]]],
              [[s-[x]], [m-[x]]]],
maplist(flatten, SplitConjs, FlatSplitConjs),
FlatSplitConjs = [[p-[x], not(p-[x])], [s-[x], not(p-[x])], [p-[x], m-[x]],
                  [s-[x], m-[x]]].
```

'split_term' is here used in combination with 'maplist' to store the split result of each conjunction in its own list to get a list of lists. This is necessary because 'notsath' checks if a relation occurs together in a conjunction with its negated form. So each conjunction will be looked at separately.

If that is the case further iteration steps of the iteration stage formation rule will not lead to a new formula since the old formula actually implies the result of the next iteration stage formation rule step. The structure of the 'notsath' function looks like this:

Listing 27: Structure of notsath

```
/* Helper predicates for computing replacers*/
notsath(List) :- member(X,List), member(not(X), List).
notsat(List) :- member(InnerList,List) -> notsath(InnerList).
```

To check if a relation and its negation occur in the same list it uses the 'member(X,List)' function which just checks if an element 'X' is part of the list 'List'. As mentioned before if 'notsath' results in 'true' the function 'isfr1' is terminated and the result of 'reduced_quant_term' is the resulting formula. How this result is achieved with 'reduced_quant_term' will be covered in the next subsection.

3.6 Quantifier-free formulas

From Fact 22 of the introduction of this thesis it is known that every quantifier-free first-order formula is equivalent to an acyclic determinate DATALOG program. Since after transforming a DATALOG program into an S-LFP formula (which was explained in section 3.1) the formula contains quantifiers, they must be (asymptotically) eliminated to transform a formula back into a (probabilistic) logic program.

The theoretical background was covered in 2.5 and is based on the work of Lingsch Rosenfeld [10]. In Listing 19 in line 36, 63 and 73 the starting point of the quantifier elimination process in the algorithm is shown. As mentioned before it is done after each iteration of the iteration stage formation rule to reduce computation time and unnecessarily large formulas. Here is an example of how the result of 'reduced_quant_elim' can look like:

Example 32. Example call of reduced_quant_elim

```
reduced_quant_elim(e-[x, y] or exists([at],e-[x, at] and e-[at, y]), Result),
Result = true.
```

To understand why the 'Result' is 'true' and what steps are necessary to achieve it the structure of 'reduced_quant_elim' and its steps are shown below:

Listing 28: Structure of reduced_quant_elim

```
reduced_quant_elim(Term, ReducedTerm) :-
    first_step_qe(Term, FirstStepTerm),
    build_dnf(FirstStepTerm, DnfTerm),
    split_up_free_elems(DnfTerm, SplitFepTerm),
    map_non_fep(SplitFepTerm, MappedTerm),
    reduced_form(MappedTerm, ReducedTerm),!.
```

'Term' is the formula we get after a step of the iteration stage formation rule and 'ReducedTerm' is the formula without any quantifiers. The first step is covered by the function 'first_step_qe'. It rewrites the conventional quantifiers of a formula as exclusive quantifiers and rewrites the formula according to Lemma 46. The exclusive quantifiers were introduced in Definition 42 and an example on applying Lemma 46 is shown in Example 7. First let us look at the structure of 'first_step_qe':

Listing 29: Structure of first_step_qe

```
first_step_qe(Term, NewFormula) :-
    split_term(Term, SplittedTerm),
    flatten(SplittedTerm, FlattenedTerm),
    include(is_quantifier_formula,FlattenedTerm, AllQuantFormulas),
    helper_add_excl_quantifier(AllQuantFormulas, Res),
    maplist(helper_resolve_inner, Res, ResolvQuantForms),
    replace_rec(Term, AllQuantFormulas, ResolvQuantForms, NewFormula).
```

At first the formula is split at all outer operators and then filtered for parts of the formula which contain a quantifier. This is done by 'split_term' and 'is_quantifier_formula' as seen in line 2 and 4 of Listing 29. Here is an example of what is happening after these two functions are applied to a formula:

Example 33. Example of what happens after lines 2 - 4 of Listing 29

```
split_term(e-[x, y] and forall([at],e-[x, at]) or exists([at],e-[x, at] and e-[at, y]),
           SplitTerm),
SplitTerm = [e-[x, y]], [[forall([at],e-[x, at])], [exists([at],e-[x, at] and e-[at, y])]]],
flatten(SplitTerm, FlatSplitTerm),
FlatSplitTerm = [e-[x, y], forall([at],e-[x, at]), exists([at],e-[x, at] and e-[at, y])],
include(is_quantifier_formula,FlatSplitTerm, AllQuantFormulas),
AllQuantFormulas = [forall([at],e-[x, at]), exists([at],e-[x, at] and e-[at, y])].
```

In the end 'AllQuantFormulas' contains all the parts of the formula which are dependent on a quantifier. All these parts can then be rewritten to parts with exclusive quantifiers and that is done with the help of the 'helper_add_excl_quantifier' function. The structure of this function is the following:

Listing 30: Structure of helper_add_excl_quantifier

```
1  helper_add_excl_quantifier([],[]).
2  helper_add_excl_quantifier([FirstQuantForm|RestQuantForm], HelperExclQuantForm) :-
3      getInnerFormula(FirstQuantForm, InnerForm),
4      extract_bound_vars(FirstQuantForm, Boundvars),
5      freevars(FirstQuantForm, Freevars),
6      helper_qe_repl_quant(InnerForm, Boundvars, Freevars, NewInnerForms),
7      helper_dis_or_con(FirstQuantForm, InnerForm, Boundvars, Freevars, NewInnerForms,
8                        NewQuantForm),
9      append([NewQuantForm], Out, HelperExclQuantForm),
10     helper_add_excl_quantifier(RestQuantForm, Out).
```

If the result of 'AllQuantFormulas' of Example 33 is taken and the 'helper_add_excl_quantifier' function is applied to it, this function produces the following output:

Example 34. Example of calling helper_add_excl_quantifier

```
1  helper_add_excl_quantifier([forall([at],e-[x, at]), exists([at],e-[x, at] and e-[at, y])],
2                             Result),
3  Result = [forall([at],[x],e-[x, at] and e-[x, x]), exists([at],[x, y],(e-[x, at] and
4            e-[at, y])or(e-[x, x] and e-[x, y])or(e-[x, y] and e-[y, y]))].
```

In this algorithm the formulas with exclusive quantifiers have the form 'quantifier(bound variables, exclusive variables, inner formula)'. This form is achieved by first extracting the inner formula of a formula with a conventional quantifier and then the bound variables and free variables of this inner formula are extracted. This is all done with the help of the functions of lines 3 - 5 of Listing 30. After this the function 'helper_qe_repl_quant' is called and produces the following output:

Example 35. Example of calling helper_qe_repl_quant

```
1  helper_qe_repl_quant(e-[x, at],[at],[x],ExclTerm1),
2  ExclTerm1 = [e-[x,x]],
3  helper_qe_repl_quant(e-[x, at] and e-[at, y],[at],[x, y],ExclTerm2),
4  ExclTerm2 = [e-[x, x] and e-[x, y], e-[x, y] and e-[y, y]].
```

The terms of 'ExclTerm1' and 'ExclTerm2' are all achieved by substituting free variables of a formula for the bound variables. They have to be added back to the original quantifier formula to get to the form of a exclusive quantifier form.

According to Lemma 3.1.9 of Lingsch Rosenfeld [10] 'ExclTerm1' has to be added as conjunction to the original formula since it is dependent on the 'forall'-quantifier and the terms of 'ExclTerm2' have to be added as a disjunction to the original formula because

it is dependent on the 'exists'-quantifier. To build this conjunction or disjunction, the 'helper_dis_or_con' function is used:

Listing 31: Structure of helper_dis_or_con

```
1  helper_dis_or_con(QuantForm, Form,Boundvars, Freevars, [],
2                    exists(Boundvars, Freevars,Form)) :-
3      QuantForm = exists(_,_).
4  helper_dis_or_con(QuantForm, Form,Boundvars, Freevars, [],
5                    forall(Boundvars, Freevars,Form)) :-
6      QuantForm = forall(_,_).
7  helper_dis_or_con(QuantForm, InnerForm, Boundvars,Freevars, [FirstForm|RestForm],
8                    DisOrCon) :-
9      QuantForm = exists(_,_),
10     helper_dis_or_con(QuantForm, InnerForm or FirstForm,Boundvars,Freevars, RestForm,
11                       DisOrCon).
12 helper_dis_or_con(QuantForm, InnerForm, Boundvars,Freevars,[FirstForm|RestForm],
13                   DisOrCon) :-
14     QuantForm = forall(_,_),
15     helper_dis_or_con(QuantForm, InnerForm and FirstForm, Boundvars,Freevars,RestForm,
16                       DisOrCon).
```

Depending on the quantifier of the formula, the operator 'or' or 'and' is added for each part with a term of '[FirstForm|RestForm]'. The parameter '[FirstForm|RestForm]' contains all the terms which are the output of 'helper_qe_repl_quant'. After the recursive call of this function is done, it terminates and the new formula is written to the exclusive quantifier constructor. Then all transformed formulas are added to a list which corresponds to the output of Example 35.

This works only for non-nested quantified formulas and to resolve inner nested quantified formulas the 'helper_resolve_inner' function is applied as seen in line 6 of Listing 29. This has to be done since the nested quantifier formulas have to be considered when substituting the free variables for the bound ones.

Example 36. Example of transforming nested quantifier formulas into nested exclusive quantifier formulas

```
1  helper_add_excl_quantifier([exists([at],forall([bt],e-[x, bt]) and e-[at, y])], First),
2  First = [exists([at],[x, y],(forall([bt],e-[x, bt]) and e-[at, y])or
3           (forall([bt],e-[x, bt]) and e-[x, y])or
4           (forall([bt],e-[x, bt]) and e-[y, y]))]),
5
6  maplist(helper_resolve_inner, First, Second),
7  Second = [exists([at],[x, y],(forall([bt],[x],e-[x, bt] and e-[x, x]) and e-[at, y])or
8           (forall([bt],[x],e-[x, bt] and e-[x, x]) and e-[x, y])or
9           (forall([bt],[x],e-[x, bt] and e-[x, x]) and e-[y, y]))].
```

'helper_inner_resolve' actually takes care of all the inner formulas which do not have the exclusive quantifier form yet. It applies the 'helper_add_excl_quantifier' function to all these parts as seen in line 7 of the following Listing:

Listing 32: Structure of helper_inner_resolve

```
helper_resolve_inner(Formula, Formula) :-
    getAllQuantifierFormulas(Formula, Inners),
    Inners = [].
helper_resolve_inner(Formula, NewFormula1) :-
    getAllQuantifierFormulas(Formula, Inners),
    Inners \= [],
    helper_add_excl_quantifier(Inners, ResolvedInners),
    replace_rec(Formula, Inners, ResolvedInners, NewFormula),
    helper_resolve_inner(NewFormula, NewFormula1).
```

The function 'first_step_qe' ends with taking the original formula and replacing all the parts with a quantifier with the new generated exclusive quantifier parts. Here is an example on how the result of 'first_step_qe' can look like then:

Example 37. Example call of first_step_qe

```
first_step_qe(e-[x, y] or forall([bt],e-[x, bt]) and exists([at],e-[x, at] and e-[at, y]),
              Result),
Result = (e-[x, y] or forall([bt],[x],e-[x, bt] and e-[x, x]) and
         exists([at],[x, y],(e-[x, at] and e-[at, y])or
         (e-[x, x] and e-[x, y])or(e-[x, y] and e-[y, y])).
```

In Example 37 it can be seen that parts without any quantifiers are unchanged and that parts that are dependent on 'forall' are connected with a conjunction. Parts that are dependent on 'exists' are connected with disjunctions.

Now that the original formula is rewritten to a formula with exclusive quantifiers, the next step consists of converting the formula into disjunctive normal form. This step can be seen in line 3 of Listing 28 and has to be done to make it easier to apply Lemma 3.1.16 of Lingsch Rosenfeld [10], which states that every formula according to Definition 43 can be separated into free elementary parts and non-free elementary parts.

The formula then has this form: $\bigvee_{i \in I} \phi_i \wedge \psi_i$, where ϕ_i contains only free elementary parts and ψ_i contains non-free elementary parts. To split these two parts the 'split_up_free_elems' part is used:

Listing 33: Structure of split_up_free_elems

```
split_up_free_elems(Term, FinalDisjTerm) :-
    split_term_at_or(Term, SplittedTerm),
    flatten(SplittedTerm, Flattened),
    split_up_free_elems_term(Flattened, Res),
    list_to_disj(Res, FinalDisj),
    flatten1(FinalDisj, FinalDisjTerm).
```

The 'split_up_free_elems_term' function does the main work, and the formula needs to be prepared by splitting it at the 'or's again, storing the split parts in a list and then flatten the result. An example call of 'split_up_free_elems_term' looks like the

following:

Example 38. Example call of split_up_free_elems_term

```
split_up_free_elems_term([e-[x, y], forall([bt],[x],e-[x, bt] and e-[x, x] and t-[bt, x]),
                          exists([at],[y, y],(e-[at, y] and t-[y, y] and s-[at, y])or
                          (e-[y, y] and t-[y, y]))],Result),
Result = [e-[x, y], forall([bt],[x],e-[x, bt] and t-[bt, x] and e-[x, x]),
          exists([at],[y, y],(e-[at, y] and t-[y, y] and s-[at, y])or
          (e-[y, y] and t-[y, y]))].
```

Most of the times 'build_dnf' already puts the formula into the right shape that the inner quantified formulas are a conjunction of free elementary parts and non-free elementary parts. In Example 38 it can be seen that the conjunctions are transformed so that non-free elementary parts are standing together and the free elementary parts are standing together. After this step non-free elementary parts of the formula can be replaced by either false or true. This depends on what quantifier they are dependent on. In Theorem 52 this procedure is described and Example 9 shows how the formula looks like after this replacement. Keep in mind that this replacement is just asymptotically true. To realise this replacement procedure the function 'map_non_fep' function is used:

Listing 34: Structure of map_non_fep

```
1  map_non_fep(Term, Result) :-
2      split_term_at_or(Term, SplittedTerm),
3      flatten(SplittedTerm, Flattened),
4      getAllExclQuantifierFormulas(Flattened, ExclQuantFormula),
5      maplist(getAllExclQuantifierInnerPart, ExclQuantFormula, InnerParts),
6      maplist(split_term_at_or, InnerParts, SplitOrs),
7      flatten(SplitOrs, FlatSplitOrs),
8      maplist(split_term, FlatSplitOrs, SplitConjs),
9      maplist(flatten, SplitConjs, Flatten),
10     maplist(exclude(is_excl_quant_form), Flatten, Relv),
11     maplist(helper_inner_part1, ExclQuantFormula, ConjLiterals),
12     maplist(helper_map_non_fep, ConjLiterals, New),
13     flatten(New, FlatNew),
14     flatten(Relv, FlatRelv),
15     term_to_atom1(Term, AtomTerm),
16     maplist(term_to_atom1, FlatRelv, RelvAtoms),
17     maplist(term_to_atom1, FlatNew, FlatNewAtoms),
18     replace_nth_word_rec(AtomTerm, 1, RelvAtoms, FlatNewAtoms, NewTerm),
19     term_to_atom(Result, NewTerm).
```

Until line 11 of Listing 34 the formula is prepared for this replacement procedure. The formula is split into a list at all its disjunctions with nested lists that contain the parts which were part of a conjunction. Here is an example on how a formula can look like after the execution of line 10:

Example 39. Given the formula

```
Form = e-[x, y] or exists([at],[x, y],(e-[x, at] and e-[at, y])or
      (e-[x, x] and e-[x, y])or(e-[x, y] and e-[y, y]))
```

Applying the functions of Listing 34 until line then results in:

```
Relv = [e-[x, at], e-[at, y]], [e-[x, x], e-[x, y]], [e-[x, y], e-[y, y]]],
ConjLiterals = [exists([at],[x, y],[[e-[x, at], e-[at, y]], [e-[x, x], e-[x, y]],
               [e-[x, y], e-[y, y]]])]
```

'Relv' will be flattened in line 14 of Listing 34 and contains all literals that are part of the inner exclusive quantifier formula. Then with the help of 'helper_map_non_fep' for each literal it is checked if it is a non-free elementary part. If that is the case, this literal part can be replaced by either 'true' or 'false'. Therefore 'helper_map_non_fep' has as result a list that contains for each literal of 'Relv' the substitution value. Looking at 'Relv' the non-free elementary part are e-[x,at] and e-[at,y] since the bound variable 'at' is part of them. The remaining literals of 'Relv' are free elementary parts. For 'Relv' the function would then produce the following output:

```
helper_map_non_fep(exists([at],[x, y],[[e-[x, at], e-[at, y]], [e-[x, x], e-[x, y]],
                  [e-[x, y], e-[y, y]]]),Result),
Result = [true, true, e-[x, x], e-[x, y], e-[x, y], e-[y, y]].
```

To understand how 'Result' is achieved, we look at the structure of this function:

Listing 35: Structure of helper_map_non_fep

```
helper_map_non_fep(exists(Boundvars, _, Literals),FlatInnerPart) :-
    maplist(list_to_conj,Literals, InnerPart),
    length(InnerPart, LenInnerPart),
    copy_list([Boundvars], LenInnerPart, EnoughBoundvars),
    maplist(is_free_elem_part, EnoughBoundvars, Literals, FreeElems),
    maplist(flatten, FreeElems, FlatFreeElems),
    maplist(subtract, Literals, FlatFreeElems, NonFreeElems),
    filter_negated_parts(NonFreeElems, NegatedNonFreeElems),
    maplist(subtract, NonFreeElems, NegatedNonFreeElems, NormalNonFreeElems),
    maplist(length, NegatedNonFreeElems, LenNegatedNonFreeElems),
    maplist(length, NormalNonFreeElems, LenNormalNonFreeElems),
    maplist(copy_list([true]), LenNormalNonFreeElems, EnoughTrue),
    maplist(copy_list([false]), LenNegatedNonFreeElems, EnoughFalse),
    maplist(replace_rec, InnerPart, NormalNonFreeElems, EnoughTrue, NewInnerPart),
    maplist(replace_rec, NewInnerPart, NegatedNonFreeElems, EnoughFalse,
            NewInnerPartNegatedMapped),
    flatten(NewInnerPartNegatedMapped,FlatNewInnerPart),
    maplist(split_term, FlatNewInnerPart, SplitInnerPart),
    flatten(SplitInnerPart, FlatInnerPart).

helper_map_non_fep(forall(Boundvars, _, Literals),FlatInnerPart) :-
```

```
22      maplist(list_to_conj,Literals, InnerPart),
23      length(InnerPart, LenInnerPart),
24      copy_list([Boundvars], LenInnerPart, EnoughBoundvars),
25      maplist(is_free_elem_part, EnoughBoundvars, Literals, FreeElems),
26      maplist(flatten, FreeElems, FlatFreeElems),
27      maplist(subtract, Literals, FlatFreeElems, NonFreeElems),
28      filter_negated_parts(NonFreeElems, NegatedNonFreeElems),
29      maplist(subtract, NonFreeElems, NegatedNonFreeElems, NormalNonFreeElems),
30      maplist(length, NegatedNonFreeElems, LenNegatedNonFreeElems),
31      maplist(copy_list([true]), LenNegatedNonFreeElems, EnoughTrue),
32      maplist(length, NormalNonFreeElems, LenNormalNonFreeElems),
33      maplist(copy_list([false]), LenNormalNonFreeElems, EnoughFalse),
34      maplist(replace_rec, InnerPart, NormalNonFreeElems, EnoughFalse, NewInnerPart),
35      maplist(replace_rec, NewInnerPart, NegatedNonFreeElems, EnoughTrue,
36              NewInnerPartNegatedMapped),
37      flatten(NewInnerPartNegatedMapped,FlatNewInnerPart),
38      maplist(split_term, FlatNewInnerPart, SplitInnerPart),
39      flatten(SplitInnerPart, FlatInnerPart).
```

'helper_map_non_fep' takes the part '[[e-[x, at], e-[at, y]], [e-[x, x], e-[x, y]],[e-[x, y], e-[y, y]]]' and filters it for free elementary parts as seen in line 5 and 32 of Listing 35. This is done by generating a list of lists with the inner lists containing the bound variables and then excluding all elements which contain the bound variables. After filtering the example list the result is:

```
1   FreeElems = [[]], [[e-[x, x], e-[x, y]]], [[e-[x, y], e-[y, y]]]]
```

'FreeElems' is then flattened, and to get the parts which are non-free elementary parts the built-in function 'subtract' is used. This leads to:

```
1   NoFreeElems = [[e-[x, at], e-[at, y]], [], []]
```

Negated literals can also occur in the formula, and cannot be replaced with 'true' if they are dependent on the 'exists'-quantifier. Therefore it is checked if negated parts exist in line 8 and 28 of Listing 35 and the negated non-free elementary parts and the positive free elementary parts are separately stored in 'NegatedNonFreeElems' and 'NormalNonFreeElems'. Then from lines 12 - 13 and 31 - 33 the necessary amount of 'true' and 'false' for the substitution are computed.

Example 40. Example on what happens if a inner list contains a negated value

```
1   helper_map_non_fep(exists([at],[x, y],[[e-[x, at], not(e-[at, y])],[e-[x, x], e-[x, y]],
2                      [e-[x, y], e-[y, y]]]),Result1),
3   Result1 = [true, false, e-[x, x], e-[x, y], e-[x, y], e-[y, y]],
4
5   helper_map_non_fep(forall([bt],[x, y],[[e-[x, bt], not(e-[bt, y])]]),Result2),
6   Result = [false,true].
```

In Example 40 it can be seen that the negated values are being resolved to either 'true' or 'false' depending on the quantifier they are dependent on. Then in lines 14 - 15 and 34 - 35 the computed substitution values for the non-free elementary parts are replaced into the inner formula of the exclusive quantifier formula.

After this procedure the substituted conjunctions are then transformed into a list again. This list contains all the values the literals of the exclusive quantifiers have after the substitution process. Since later on they all will be replaced according to the order of the resulting list.

Now that we have the substitution values after executing line 12 of Listing 34 the substitution process can start. This is done from lines 15 - 18 of Listing 34. Before that the list of all literals and the list of substitution literals are flattened. Here is an example of how the process form lines 15 - 19 for a formula looks like:

Example 41. Given the formula

```
1  Formula = e-[x, y] or exists([at],[x, y],(e-[x, at] and not(e-[at, y]))or
2                                           (e-[x, x] and e-[x, y])or
3                                           (e-[x, y] and e-[y, y])).
```

'map_non_fep' will produce the following output from lines 13 - 19:

```
1  flatten([[true, false, e-[x, x], e-[x, y], e-[x, y], e-[y, y]]], FlatSubs),
2  FlatSubs = [true, false, e-[x, x], e-[x, y], e-[x, y], e-[y, y]],
3
4  flatten([[e-[x, at], not(e-[at, y])], [e-[x, x], e-[x, y]], [e-[x, y], e-[y, y]]],
5          FlatLiterals),
6  FlatLiterals = [e-[x, at], not(e-[at, y])], [e-[x, x], e-[x, y]], [e-[x, y], e-[y, y]],
7
8  term_to_atom(Formula, AtomTerm),
9  AtomTerm = 'or(e-[x,y],exists([at],[x,y],or(or(and(e-[x,at],not(e-[at,y])),
10             and(e-[x,x],e-[x,y])),
11             and(e-[x,y],e-[y,y])))))',
12
13 maplist(FlatLiterals, AtomLiterals),
14 AtomLiterals = ['e-[x,at]', 'not(e-[at,y])', 'e-[x,x]', 'e-[x,y]', 'e-[x,y]', 'e-[y,y]'],
15
16 maplist(FlatSubs, AtomSubs),
17 AtomSubs = [true, false, 'e-[x,x]', 'e-[x,y]', 'e-[x,y]', 'e-[y,y]'],
18
19 replace_nth_word_rec(AtomTerm, 1, AtomLiterals, AtomSubs, AtomFormula),
20 term_to_atom(AtomFormula, NewFormula),
21 NewFormula = exists([at],[x, y],(true and false)or(e-[x, x] and e-[x, y])or
22                                 (e-[x, y] and e-[y, y])).
```

To be able to use 'replace_nth_word_rec' for the substitution process the relevant parameters have to be transformed from datatype 'term' to 'atom' first. This can be done with the built-in function 'term_to_atom'. Then 'replace_nth_word_rec' substitutes like 'replace_nth_word' but works listwise. In the end all the literals have been replaced, and the formula can be transformed to datatype 'term' again, leading for example to a value like 'NewFormula'. The result of 'map_non_fep' is then reduced with 'reduced_form'

according to line 6 of Listing 28. In the following there is an example of how the reduced formulas look like:

Example 42. Example of reducing exclusive quantifier formulas

```
1 reduced_form(e-[x, y] or exists([at],[x, y],(true and false)or
2             (e-[x, x] and e-[x, y])or(e-[x, y] and e-[y, y])), Result1),
3 Result1 = (e-[x, y]or((e-[x, x] and e-[x, y])or(e-[x, y] and e-[y, y]))),
4
5 reduced_form(e-[x, y] or forall([at],[x, y],(false or true)or
6             (e-[x, x] and e-[x, y])or(e-[x, y] and e-[y,y])),Result2),
7 Result2 = true.
```

For 'Result1', 'exists(_,_, true and false)' will lead to 'false' and 'false' in a disjunction can be discarded. For 'Result2', 'forall(_,_, false or true)' will lead to 'true' and a disjunction regardless of how many parts are part of it, will lead to 'true' if true is one part of the disjunction. These reduced formulas are now quantifier-free and can be transformed into a determinate logic program in the next step. How this conversion is done is explained in the next subsection.

3.7 Conversion into determinate logic program

For the conversion into a determinate logic program the function 'convert_to_lp' was implemented:

Listing 36: Structure of convert_to_lp

```
1  convert_to_lp(Pred,Freevars,true,[Clause]) :-
2      Clause = Pred-Freevars-[].
3  convert_to_lp(_,_,false,[]).
4  convert_to_lp(Pred,Freevars,Formula, LP) :-
5      Formula \= true,
6      Formula \= false,
7      Head = Pred-Freevars,
8      split_term_at_or(Formula, SplittedTerm),
9      flatten(SplittedTerm, FlatSplittedTerm),
10     maplist(split_term, FlatSplittedTerm, SplitConjs),
11     length(SplitConjs, LenConjs),
12     copy_list([Head], LenConjs, EnoughHeads),
13     maplist(flatten, SplitConjs,FlatSplitConjs),
14     maplist(put_together, EnoughHeads, FlatSplitConjs, LP).
```

Oftentimes the asymptotic result of a formula will either lead to 'true' or 'false'. If the formula is 'true', then a clause is built with the head being the predicate symbol of the desired original formula, the variables of that head predicate are the free variables of the original formula, and the body is an empty list since this formula will asymptotically become 'true' on large domains.

Example 43. Given the logic program

```
Prog = [t-[x,y]-[e-[x,y]],t-[x,z]-[t-[x,y], e-[y,z]]],
```

the desired LFP formula for predicate symbol 't' is:

```
Lfp = lfp(t,e-[x, y] or exists([at],t-[x, at] and e-[at, y]))
```

This formula has the free variables '[x,y]'. Applying the iteration stage formation rule and quantifier elimination after each step will lead to this result:

```
RedForm = true
```

Calling 'convert_to_lp' will then lead to the following logic program:

```
convert_to_lp(t,[x,y],true,LogProg),
LogProg = [t-[x,y-[]].
```

Example 44. Given the logic program

```
Prog = [p-[x]-[r-[x],q-[x]], p-[x]-[r-[x],not(q-[x])]]
```

the desired LFP formula for predicate symbol 'p' is:

```
Lfp = not(q-[x]) and q-[x]
```

This formula has the free variables '[x]'. Since there is no fixed-point functor the result of this formula is 'false'. This seems sensible since 'not(q-[x]) and q-[x]' would always lead to 'false.

```
RedForm = false
```

Calling 'convert_to_lp' will then lead to the following logic program:

```
convert_to_lp(t,[x,y],true,LogProg),
LogProg = [].
```

This means this logic program will asymptotically always be 'false'.

Example 45. Given the logic program

```
Prog = [smokes-[x]-[stress-[x]], smokes-[x]-[friend-[x,y],influences-[y,x],smokes-[y]],
       influences-[x,y]-[person-[x],person-[y]],stress-[x]-[person-[x]],
       asthma-[x]-[smokes-[x]]]
```

the desired LFP formula for predicate symbol ‘smokes’ is then:

```
Lfp = lfp(smokes,person-[x] or exists([asmokes],friend-[x, asmokes]and
        (person-[asmokes] and person-[x]) and smokes-[asmokes]))
```

This formula has the free variables ‘[x]’. Applying the iteration stage formation rule and quantifier elimination after each step will lead to this result:

```
RedForm = person-[x]or(person-[x]or(friend-[x, x] and person-[x] and person-[x]))
```

Calling ‘convert_to_lp’ will then lead to the following logic program:

```
convert_to_lp(smokes,[x],RedForm,LogProg),
sort(LogProg,SortedLogProg),
LogProg = [smokes-[x]-[person-[x]], smokes-[x]-[person-[x]], smokes-[x]-[friend-[x, x],
           person-[x], person-[x]]],
SortedLogProg = [smokes-[x]-[friend-[x, x], person-[x], person-[x]],
                 smokes-[x]-[person-[x]]].
```

In the function ‘compute_lp_form’ the result of ‘convert_to_lp’ is sorted to avoid duplicate clauses. It can be seen that the result of ‘LogProg’ has the duplicate clauses ‘smokes-[x]-[person-[x]], smokes-[x]-[person-[x]]’.

From lines 8 - 14 of Listing 36 it is shown how a formula that is not ‘true’ or false is converted into a logic program. The formula ‘RedForm’ has the form as described in 2.6. There the formula can be split at its disjunctions and then rewritten to a logic program with each clause being a disjunction and the body of the clause contains the inner conjunctions of a disjunction.

As head for this clauses the desired predicate is used and as its variables the free variables that were computed earlier. From lines 8 - 10 the formula is split at its disjunctions, the inner conjunctions are also split, and then stored as a nested list like in other functions before.

Then enough heads for each clause are computed and attached to the body of the clause, making the outer list a logic program. The resulting logic program is the final output of the algorithm of this thesis.

4 Conclusion and future work

Based on the findings of the work of Weitkämper [12] whose main result is that every probabilistic logic program is asymptotically equivalent to an acylic determinate probabilistic logic program, an algorithm to achieve this main result was implemented in the Prolog dialect SWI-Prolog [3].

This algorithm can be used to evaluate the asymptotic probabilities of quantifier-free queries with respect to a probabilistic logic program. This is especially interesting to obtain an asymptotic value on how queries behave in large domains. Also, the algorithm works independently of the probability with which the clauses are annotated in a probabilistic logic program. This is possible because the probabilistic logic program can be separated into probabilistic facts and an ordinary logic program. To transform a probabilistic logic program into a quantifier-free formula only the logic program is necessary, but we have to keep in mind that the probabilistic facts exist.

After transforming the quantifier-free formula into a determinate logic program, the probabilistic facts can be reattached to the clauses for any further processing or computation. The algorithm can be broken down into the following steps: Building the S-LFP formula of the desired clauses, transforming the S-LFP formula into LFP formula, applying the iteration stage formation rule on the formula, eliminating quantifiers in the formula and then converting it into a determinate logic program.

While the algorithm works on simple logic programs and with negated relations, an implementation how to handle constants is missing and would be part of further work improving the implementation. In the following a short outlook is given on how an implementation considering constants can look like.

4.1 Future Work

Ebbinghaus and Flum [6] describe how constants can be handled when transforming them in a DATALOG program. As for other clauses with the same head predicate also clauses which contain constants should have the same head variables as regular clauses. For the body of such clauses it should be indicated that the head variable has a constant value.

Example 46. Given the DATALOG program

```
Datalog = [p-[const(min)]-[], p-[x]-[q-[x],s-[x,y]]]
```

The clause ‘p-[const(min)]-[]’ is then rewritten to ‘p-[x]-[const(x,min)]’ since there exists in ‘Datalog’ another clause with same head predicate and the head variables are ‘[x]’. The construct ‘const(x,min)’ indicates that the value min is a constant and that the variable ‘x’ is equal to constant value ‘min’. The changed DATALOG program looks like this then:

```
ChangedDatalog = p-[x]-[const(x,min)], p-[x]-[q-[x], s-[x, y]]]
```

Also a rough implementation of how substituting clauses with constants is provided:

Listing 37: Pseudo-code implementation to transform clauses with constants

```
subst_const(_,[],[],[]).
subst_const(Clause, [FirstPred|Rest],FinalConstClauses,FinalNewConcats) :-
```

```
3     filterClauseWithPred(Clause, FirstPred, AllRelvClauses),
4     include(is_clause_with_const, AllRelvClauses, ConstClauses),
5     getAllHeads(ConstClauses, ConstHeads),
6     exclude(is_clause_with_const, AllRelvClauses, NormalClauses),
7     getReplacer(Replacer, NormalClauses),
8     variables_in_literal(Replacer, ReplacerVars),
9     maplist(const_in_literal, ConstHeads, Constants),
10    flatten(Constants, FlattenedConstants),
11    maplist(is_constant, FlattenedConstants, Cons),
12    maplist(build_constant, ReplacerVars, Cons, Concats),
13    maplist(put_together, [Replacer], [Concats], FinalConcats),
14    append(FinalConcats, Output, FinalNewConcats),
15    append(ConstClauses, Out, FinalConstClauses),
16    subst_const(Clause, Rest, Out, Output).
```

It should work similarly as the 'do_substitution' function, but transforming the constant value takes place in lines 14 - 16 of Listing 37. The further process of transforming the DATALOG program with clauses containing constants is yet to be implemented.

5 References

References

[1] A.9 library(clpfd): Clp(fd): Constraint logic programming over finite domains. https://www.swi-prolog.org/pldoc/man?section=clpfd. Accessed: 2022-08-25.

[2] Introduction. - ProbLog: Probabilistic Programming. https://dtai.cs.kuleuven.be/problog/. Accessed: 2022-07-31.

[3] Robust, mature, free. Prolog for the real world. https://www.swi-prolog.org/. Accessed: 2022-09-16.

[4] A. Blass, Y. Gurevich, and D. Kozen. A Zero-One Law for Logic with a Fixed-Point Operator. *Information and Control*, 67(1-3):70–90, 1985.

[5] F. G. Cozman and D. D. Mauá. The finite model theory of Bayesian network specifications: Descriptive complexity and zero/one laws. *Int. J. Approx. Reason.*, 110:107–126, 2019. doi:10.1016/j.ijar.2019.04.003.

[6] H.-D. Ebbinghaus and J. Flum. *Finite model theory*, pages 165–169, 239–241. Springer Science & Business Media, 1999.

[7] M. Jaeger and O. Schulte. Inference, learning, and population size: Projectivity for SRL models. *arXiv preprint arXiv:1807.00564*, 2018.

[8] M. Jaeger and O. Schulte. A complete characterization of projectivity for statistical relational models. *arXiv preprint arXiv:2004.10984*, 2020.

[9] S. Kreutzer. *Pure and applied fixed-point logics.* Dissertation, RWTH Aachen, 2002.

[10] M. Lingsch Rosenfeld. Asymptotic quantifier elimination. Bachelorarbeit, LMU Munich, 2022.

[11] F. Riguzzi and T. Swift. A survey of probabilistic logic programming. *Declarative Logic Programming: Theory, Systems, and Applications*, pages 185–228, 2018.

[12] F. Weitkämper. An asymptotic analysis of probabilistic logic programming with implications for expressing projective families of distributions. *Theory and Practice of Logic Programming*, 2021, 2102.08777. URL https://arxiv.org/abs/2102.08777.