Proceedings of the
29nd Workshop on (Constraint) Logic Programming
(WLP 2015)

Dresden, Germany, September 22, 2015
Preface

This volume contains the papers presented at WLP 2015: 29th Workshop on (Constraint) Logic Programming held on September 22, 2015 in Dresden as part of the 38th edition of the German Conference on Artificial Intelligence (KI 2015).

The Workshops on (Constraint) Logic Programming are the annual meeting of the German Society of Logic Programming (GLP) and bring together researchers interested in logic programming, constraint programming, answer set programming, and related areas like databases and artificial intelligence (not only from Germany).

The WLP workshop series started 1988 in Berlin (in the first three years there were two workshops per year). Previous workshops have been held in Germany, Austria, Switzerland and Egypt.

The workshops provide a forum for exchanging ideas on declarative logic programming, nonmonotonic reasoning and knowledge representation, and facilitate interactions between research in theoretical foundations and in the design and implementation of logic-based programming systems.

For WLP 2015, the committee decided to accept 7 papers. The program also includes an invited talk by Steffen Hölldobler on Human Reasoning, Logic Programs, and Connectionist Systems.

We thank the programme committee members, authors of papers, and the KI 2015 workshop organizers. It was a pleasure to work together.

August 21, 2015
Leipzig, Dresden

Sibylle Schwarz
Steffen Hölldobler
## Program Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slim Abdennadher</td>
<td>German University in Cairo</td>
</tr>
<tr>
<td>Stefan Brass</td>
<td>University of Halle-Wittenberg</td>
</tr>
<tr>
<td>Gerhard Brewka</td>
<td>Leipzig University</td>
</tr>
<tr>
<td>François Bry</td>
<td>Ludwig-Maximilian University of Munich</td>
</tr>
<tr>
<td>Michael Hanus</td>
<td>CAU Kiel</td>
</tr>
<tr>
<td>Steffen Hölldobler</td>
<td>Technische Universität Dresden</td>
</tr>
<tr>
<td>Petra Hofstedt</td>
<td>Brandenburg University of Technology Cottbus</td>
</tr>
<tr>
<td>Torsten Schaub</td>
<td>University of Potsdam</td>
</tr>
<tr>
<td>Sibylle Schwarz</td>
<td>Hochschule für Technik, Wirtschaft und Kultur Leipzig</td>
</tr>
<tr>
<td>Dietmar Seipel</td>
<td>Univ. Wuerzburg, Dept. of Computer Science</td>
</tr>
<tr>
<td>Hans Tompits</td>
<td>Vienna University of Technology</td>
</tr>
<tr>
<td>Janis Voigtländer</td>
<td>University of Bonn</td>
</tr>
</tbody>
</table>
Table of Contents

Human Reasoning, Logic Programs and Connectionist Systems ............ 1
  Steffen Hölldobler

Planning Problems and Fixpoint Semantics ......................... 2
  Asmaa Afeefi

Default Rules in Functional Logic Programs ....................... 17
  Sergio Antoy and Michael Hanus

Bottom-Up Evaluation of Datalog: Preliminary Report ............ 21
  Stefan Brass and Heike Stephan

A practical view on renaming ................................... 36
  Marija Kulas

Refutationally Complete Hierarchic Theorem Proving with Definitions .... 51
  Adrian Rebola Pardo

Knowledge Engineering for Hybrid Deductive Databases ............ 66
  Dietmar Seipel

Learning Instance-Level Constraints in Folksonomies for Semi-supervised Clustering using CHR .................................. 79
  Maged Shalaby, Slim Abdennadher, Nada Sharaf and Ghada Fakhry
Abstract The suppression task, the selection task, the belief bias effect, spatial reasoning and reasoning about conditionals are just some examples of human reasoning tasks which have received a lot of attention in the field of cognitive science and which cannot be adequately modeled using classical two-valued logic. I will present an approach using logic programs, weak completion, three-valued Lukasiewicz logic, abduction and revision to model these tasks. In this setting, logic programs admit a least model and reasoning is performed with respect to these least models. For a given program, the least model can be computed as the least fixed point of an appropriate semantic operator and, by adapting the Core-method, can be computed by a recurrent connectionist network with a feed-forward core.
Planning Problems and Fixpoint Semantics

Asmaa Afeefi

International Center for Computational Logic
Technische Universität Dresden, 01062 Dresden, Germany

Abstract

We investigate the existence of a least fixpoint of planning problems in the context of fluent calculus. For that matter, we represent the planning problems as equational logic programs which is restricted to acyclic programs to obtain a least fixpoint of $T_P$. The acyclicity of the program is guaranteed by the existence of a level mapping. To that purpose, we extend the definition of level mapping to satisfy the equivalence class of the equational logic programs. Furthermore, we apply the extended level mapping on the instances of conjunctive planning problems by [1] and compute a least fixpoint.

1 Introduction

The human reasoning field is one wide discipline science which is increasingly growing. Reasoning about the world is based on the idea that human understanding and reasoning work via mental simulation [8]. Somehow, we imagine the situation playing out in front of us and by picturing how the world evolves. Then, we can simulate this evolving into a sequence of actions that transforms one state into another to reach the desired situation. For example, considering the case of rescuing an ill man from his apartment is built of an initial situation when the ill man is in his apartment. The sequence of actions starts with helping him to an ambulance, driving him to the hospital, and the final situation when the ill man is in the hospital.

This problem and its resolution method show a main stream problem of Artificial intelligence (AI) planning. Planning problems are an important branch of AI, which can only be solved by adding (some form of) frame axioms. These frame axioms may lead to a vastly increased search space. However, in an equational logic programming language, where equational theories are included into the logic framework, same planning problems can be solved without the frame axioms [11]. Knowing that: (1) the main semantic properties of logic programs hold also for equational logic programs [14], (2) the semantics of an equational logic program $\mathcal{P}$ can be given as the
least Herbrand model for $\mathcal{P}$, and (3) the least Herbrand model of $\mathcal{P}$ can also be presented by the least fixpoint of a function $T_\mathcal{P}$. Our goal is to investigate if there exists a least fixpoint for general programs with equational theories.

Many related works have investigated in this area. For instance, Hölldobler in [2] has investigated first order theories which are restricted to Horn clauses and include the equality relation. He proved that $T_\mathcal{P}$ operator is continuous for the equational logic programs. Consequently, there exists a least fixpoint for $T_\mathcal{P}$. He also stated that the least Herbrand model and the least fixpoint of an equational logic program are equivalent. He did not consider an equational logic program with negative literal occur in a body of a clause from $\mathcal{P}$.

Micheal Thielscher in his PhD thesis [12] has shown that general programs with equational theory has fixpoints, and we investigate the existence of a least fixpoint. Apt and Bezem have shown in [9] for every acyclic program that the immediate consequence operator has a unique fixpoint. They did not extend an cyclic program to include equational theory.

Tackling the problem of existence of least fixpoint for general programs and going beyond exist works, we conduct an hybrid approach that combines previous related work for equational logic program restriction to acyclic program. Our contribution is threefold: (i) We consider a general logic program with equational theories which is restricted to an acyclic program. The acyclicity of the program is guaranteed by the existence of a level mapping, Thus, we extend the definition of level mapping to satisfy the equivalence class of the equational logic programs, and (ii) We solve the instances of conjunctive planning problems in the context of fluent calculus by [1] with fixpoint semantics.

We assume the reader to be familiar with propositional and first-order logic as well as with logic programming [13].

2 Preliminaries

2.1 Semantics of Logic Programs

This sections discusses interpretations and models, concentrating on the class of Herbrand interpretations.

**Definition 2.1** (Herbrand Universe). The *Herbrand universe* $\mathcal{U}_L$ for the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ is the set of all ground terms which can be formed from constants and function symbols appearing in the language.
**Definition 2.2** (Herbrand Base). The *Herbrand base* $\mathcal{B}_L$ for the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ is the set of all ground atoms which can be formed by using predicate symbols from the language with ground terms from $\mathcal{U}_L$ as arguments. By $\mathcal{B}_P$ we denote the Herbrand base underlying the program $\mathcal{P}$.

**Definition 2.3** (Interpretation). An *interpretation* $I$ is a mapping from the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ to the set of truth values $\{\text{true}, \text{false}\}$. A *Herbrand* interpretation consists of non-empty domain where the domain is a set of ground terms. (In the case that the language has no constants, we add some constant, e.g. a, to form ground terms).

We define an interpretation $I$ as a subset of $\mathcal{B}_L$ with the understanding that all atoms which are in the set are considered *true*, while all the others which are not, are considered *false*.

**Definition 2.4** (Ground Instance). A *ground instance* of a program $\mathcal{P}$ with respect to Herbrand Base will be denoted by $g(\mathcal{P})$. In general, $g(\mathcal{P})$ is infinite.

**Definition 2.5** (Model). Let $I$ be an interpretation of the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$, $\mathcal{F}$ be a set of formulae, and $F$ a formula. $I$ is a *model* of a formula $F$, denoted by $I \models F$, if $I(F) = \text{true}$. $\models F$ if and only if $I \models F$ for all interpretations $I$.

$F$ is called a *logical consequence* of $\mathcal{F}$, denoted by $\mathcal{F} \models F$, if every model $I$ of $\mathcal{F}$ is also a model of $F$.

**Definition 2.6** (Ordering Among Interpretations). Let $\mathcal{I}$ be a collection of interpretations. An interpretation $I$ in $\mathcal{I}$ is called *minimal* in $\mathcal{I}$ if and only if there is no interpretation $J$ in $\mathcal{I}$ such that $J \subset I$. An interpretation $I$ is called *least* in $\mathcal{I}$ if and only if $I \subseteq J$ for any interpretation $J$ in $\mathcal{I}$. A model $M$ of a program $\mathcal{P}$ is called minimal (resp. least) if it is minimal (resp. least) among all models of $\mathcal{P}$.

### 2.2 Fixpoint Semantics and Logic Programs

In this project we study fixpoint semantics for logic programs. A logic program consists of a set of clauses.

**Definition 2.7** (Program). Let $A$ be an atom, and $B_i$, $1 \leq i \leq n$, a literal. A *program* is a set of clauses of the form

$$A \leftarrow B_1 \land \ldots \land B_n \quad n \geq 0$$
where $A$ is called \textit{head} and $B_1 \land \ldots \land B_n$ is called \textit{body} of the clause. A \textit{logic program} is a finite set of clauses. A \textit{definite clause} is a clause where every literal in the body of the clause is positive. A \textit{definite or (positive) logic program} consists of definite clauses.

\textbf{Definition 2.8} (Complete Lattice). A partially ordered set $U$ is a \textit{complete lattice} with a reflexive, antisymmetric, and transitive order relation $\leq$, if a least upper bound, $\text{lub}(X)$, and a greatest lower bound, $\text{glb}(X)$ exist for every subset $X$ of $U$.

\textbf{Definition 2.9} (Monotonic Function). A function $T$ over a complete lattice $U$ is \textit{monotonic} if $T(x) \leq T(y)$, whenever $x \leq y$.

\textbf{Definition 2.10} (Directed Set). We say $X \subseteq U$ is \textit{directed} set if every finite subset of $X$ has an upper bound of $X$.

\textbf{Definition 2.11} (Continuous Function). A function $T$ is continuous if

$$T(\text{lub}(X)) = \text{lub}(\{T(Y) \mid Y \in X\}),$$

where $X$ is a directed set.

We will also need the concept of ordinal powers of $T$. The ordinals are what we use to count with. The first ordinal 0 is defined to be $\emptyset$. Then we define $1 = \{\emptyset\} = \{0\}, \ 2 = \{\emptyset,\{\emptyset\}\} = \{0,1\}$, and so on. The first infinite ordinals is $\omega = \{0,1,2,...\}$, the set of all non-negative integers.

Every definite logic program $P$ has an immediate consequence operator $T_P$ associated with it. $T_P$ maps a Herbrand interpretation of $P$ to another Herbrand interpretation of $P$.

In particular, a continuous operator $T_P$ has a least fixpoint. The least fixpoint ($\text{lfp}$) can be computed by iterating $T_P$ starting with the empty interpretation.

$$
\begin{align*}
T_P \uparrow 0 &= \emptyset \\
T_P \uparrow 1 &= T_P(T_P \uparrow 0) \quad (= T_P(\emptyset)) \\
\vdots & \\
T_P \uparrow n + 1 &= T_P(T_P \uparrow n) \\
\vdots & \\
T_P \uparrow \omega &= \bigcup_{n \geq 0} T_P \uparrow n
\end{align*}
$$

The least fixpoint

$$\text{lfp}(T_P) = T_P \uparrow \omega$$
The purpose for finding a least fixpoint for a definite logic program is to construct the least model for the program in a bottom-up manner. This is done by $T_P$ as we mentioned before, and showing that the least fixpoint of $T_P$ is equal to the least model of the program.

### 2.3 Acyclic Programs

We are not only interested in definite (positive) logic programs, but also in programs which contain a clause with negative atom in its body. Therefore, not all of them have a least fixpoint. We consider the restriction to acyclic programs. Apt and Bezem have shown that for every acyclic program $P$ its immediate consequence operator $T_P$ has a unique fixpoint [9].

Let us start with the strict notion of an acyclic program. The acyclicity of a program is guaranteed by the existence of a level mapping.

**Definition 2.12 (Level Mapping).** Let $P$ be a program. A level mapping for $P$ is a function $l: B_P \rightarrow \mathbb{N}$ of variable-free atoms to natural numbers, e.g. if $l(A) = n$, we will say the level of $A$ is $n$.

**Definition 2.13 (Acyclic Program).** Let $P$ be a program and $l$ a level mapping for $P$. We call $P$ acyclic with respect to $l$ if for every $A \leftarrow L_1 \land \ldots \land L_n$ $(n \geq 0) \in g(P)$, $l(A) > l(L_i)$ for every $1 \leq i \leq n$. Moreover, $P$ is called acyclic if $P$ is acyclic with respect to some level mapping for $P$.

### 2.4 Fluent Calculus

The general fluent calculus is a first-order logic language with axioms of equality and a particular equational system. The fluent calculus is used for planning to solve the frame problem. We start with the definition of fluents.

**Definition 2.14 (Fluent).** Fluents are the non-variable terms not containing the symbol $\circ$ and 1. Simple fluents are fluents containing only constants. The ground fluents are fluents not containing any variables.

A binary function symbol $\circ$ is designed to represent terms in fluent calculus which have the property of associativity, commutativity, and admitting a unit element (constant) 1.

The set of fluent terms is the smallest set satisfying the following conditions:

- 1 is a fluent term,
- each fluent is a fluent term, and
• if \( s \) and \( t \) are fluent terms, then so is \( s \circ t \)

The binary function symbol \( \circ \) enjoys some typical properties expressed by
the following equational system \( \mathcal{E}_{AC1} \):

\[
\begin{align*}
X \circ Y & \approx Y \circ X \quad \text{(commutativity)} \\
X \circ (Y \circ Z) & \approx (X \circ Y) \circ Z \quad \text{(associativity)} \\
X \circ 1 & \approx X \quad \text{(unit element)}
\end{align*}
\]

2.5 Planning Problems

Conjunctive planning problems describe an initial situation, a goal situation,
a set of actions, and a question of whether there exists a series of actions
which is called a plan transforming the initial situation into the goal situation.

For conjunctive planning problems, we have multisets of fluents which are:

• \( \mathcal{I} \) is called initial state,

• \( \mathcal{G} \) is called final state, and

• An action \( A : \mathcal{C} \Rightarrow \mathcal{E} \) has the name \( A \), the conditions \( \mathcal{C} \), and the effects \( \mathcal{E} \).

Multisets containing simple fluents occurring in conjunctive planning problems (cpp) are called states. A state represents what simple fluents hold at a time.

2.6 Equational Logic Programming

Planning problems can only be solved by adding frame axioms which consequen-
tially increase the search space. However, in an equational logic programming
language, the same planning problems can be solved without frame axioms, which means that the equational theories are included into logic framework (see e.g. [11]).

An equational logic program is a set of program clauses as usual but the
infix predicate (relation) symbol \( \approx \) may be used in the body of these clauses
and will be interpreted as equality.

Let us start with the definition of equations.
Definition 2.15 (Equation). An equation is an expression of the form \( s \approx t \), where \( s \) and \( t \) are terms. An equational system \( \mathcal{E} \) is a set of universally closed equations.

The binary relation symbol \( \approx \), called equality, has some properties expressed by following universally closed axioms of equality \( \mathcal{E}_{\approx} \):

\[
\begin{align*}
X \approx X & \quad \text{(reflexivity)} \\
X \approx Y & \Rightarrow Y \approx X & \quad \text{(Symmetry)} \\
X \approx Y \land Y \approx Z & \Rightarrow X \approx Z & \quad \text{(transitivity)} \\
\bigwedge_{i=1}^{n} X_i \approx Y_i \Rightarrow f(X_1, ..., X_n) \approx f(Y_1, ..., Y_n) & \quad \text{(f-substitutivity)} \\
\bigwedge_{i=1}^{n} X_i \approx Y_i \land r(X_1, ..., X_n) \approx r(Y_1, ..., Y_n) & \quad \text{(r-substitutivity)}
\end{align*}
\]

Definition 2.16 (Congruence relation). Let \( s, t \) be terms, and \( \forall \) denote the universal closure.

\[
s \approx_{\mathcal{E}} t \iff \mathcal{E} \cup \mathcal{E}_{\approx} \models \forall s \approx t
\]

\( \approx_{\mathcal{E}} \) is the least congruence relation on terms generated by \( \mathcal{E} \). Let \([t]\) denote the congruence class of \( t \) generated by \( \approx_{\mathcal{E}} \). Obviously, \( s \approx_{\mathcal{E}} t \) iff \([s] = [t]\).

The abbreviation for \( p([t_1], ..., [t_n]) \) is \([p(t_1, ..., t_n)]\), for \( n \geq 0 \). We consider \([p(t_1, ..., t_n)] = [q(s_1, ..., s_m)] \) iff \( p = q \), \( n = m \), and \([t_i] = [s_i] \), \( 1 \leq i \leq n \).

In our project, we deal with equational logic programs and consider the restriction to acyclic programs.

3 Fixpoint Construction of Fluent Calculus

We are interested in computing a least fixpoint for conjunctive planning problems and extensions. As we mentioned before, we consider here equational logic programs with fluent calculus and consider the restrictions to acyclic program. We will define a semantic operator and then find the least fixpoint for simple instances by [1].

Let \( I \) be a Herbrand interpretation. \( \mathcal{E} \) is an equational system specifying the \( AC1 \) property for \( \circ \) and \( 1 \) and containing the actions of equality and admit a least congruence relation \( \approx_{\mathcal{E}} \) on the set of terms.

The semantic operator is defined as follows:

\[
T_{FC;\mathcal{P}}(I) = \{ [A'] \mid \text{there exist } A \leftarrow Body \in \mathcal{P} \land [A] = [A'] \land I(Body) = \top \}
\]

Now, we aim to find the least fixpoint of the fluent calculus instances. So the question is whether there exists a least fixpoint for each planning problem. We will discuss this issue in the next subsections.
3.1 Conjunctive planning problems (CPP)

A simple fluent calculus can be represented and solved by cpp. Actions in cpp are represented by a ternary predicate symbol `action/3` and are of the form

\[ \text{action}(C^{-I}, A, E^{-I}) \]

where \( C \) is called conditions, \( A \) encodes the name of action, and \( E \) is called effects.

An applicable action is represented with another ternary predicate symbol `applicable/3`.

\[ \text{applicable}(C \circ Z, A \circ Z) \leftarrow \text{action}(C, A, E) \]

Or, in words, an action \( A \) is applicable in state \( C \circ Z \) transforming it into state \( E \circ Z \) if there is an action named \( A \) with conditions \( C \) and effects \( E \).

With the help of ternary predicate `causes/3`, we can express that the current state is transformed into another one by executing the plan \([A|P]\) which is a sequence of actions.

\[ \text{causes}(I, [A|P], G) \leftarrow \text{applicable}(I, A, S) \land \text{causes}(S, P, G) \]

Or, in words, the execution of the plan \([A|P]\) transforms state \( I \) into state \( G \) if action \( A \) is applicable in state \( I \) and its application yields state \( S \) and there is a plan \( P \) which transforms state \( S \) into \( G \).

The future state \( X \) is the same state as the current state when there is no plan, such that

\[ \text{causes}(X, [], X) \]

We consider acyclic programs to solve planning problems. Apt and Bezem have investigated acyclic programs. They have shown in [9] that every acyclic program has a unique fixpoint. Furthermore, they also have concluded from [10] that if \( P \) is a definite (positive) logic program then it has a least Herbrand model of \( P \). Extending the program \( P \) with equational theory and taking into account the advantage of the acyclicity we will obtain a program whose \( T_P \) has a least fixpoint. Now, we will start with a positive program and show that its \( T_P \) has a least fixpoint.

We extend the definition of level mapping to satisfy the equivalence classes of the equational logic programs as follows:

- If \( l(p(t)) = n \) and \( t \approx_E s \) then \( l(p(s)) = n \).
- If \( l(p(t_1, ... , t_m)) = n \) and \( t_i \approx_E s_i \), \( n \in \mathbb{N} \), \( 1 \leq i \leq m \), then \( l(p(s_1, ... , s_m)) = n \).
Example 3.1. Suppose $\mathcal{P}$ is the following program:

\[
\begin{align*}
  a & \approx b \\
  p(a) & \leftarrow q(c) \\
  p(b) & \leftarrow r(d)
\end{align*}
\]

We assume that

\[
\begin{align*}
  l(q(c)) = 1, l(r(d)) = 2, \text{ and } l(p(a)) = 3 \\
  \text{If } l(p(a)) = 3 \text{ and } a \approx b \text{ then } l(p(b)) = 3
\end{align*}
\]

$\mathcal{P}$ is acyclic with extended level mapping:

\[
\begin{align*}
  l(p(a)) > l(q(c)) \\
  l(p(b)) > l(r(d))
\end{align*}
\]

Consider this example taken from [1]:

A simple example about an ill man in an apartment who needed to be helped to go to the hospital. Let’s consider the following story:

Suppose there was a man who was severely ill living in an apartment. He could not go by himself to see a doctor. An ambulance car was asked to bring him to the hospital. He was carried by the ambulance men to the ambulance car. The ambulance car is driven to the hospital.

- The fluents are: ill man ($\text{ill}$), the apartment ($\text{apt}$), the ambulance ($\text{amb}$), and the hospital ($\text{hos}$).
- The possible actions are described by carrying the patient to the ambulance car ($\text{carry}$), and driving to the hospital ($\text{drive}$).
- The initial state of the program is $\{\text{ill}, \text{apt}\}$
- The final state of program is $\{\text{ill}, \text{hos}\}$

The logic program $\mathcal{P}_1$:

\[
\begin{align*}
  \text{action(ill }\circ\text{ apt, carry, ill }\circ\text{ amb)} \\
  \text{action(ill }\circ\text{ amb, drive, ill }\circ\text{ hos)} \\
  \text{causes(X, [], X)} \\
  \text{causes(I, [A|P], G) } \leftarrow \text{applicable(I, A, Q) } \land \text{causes(Q, P, G)} \\
  \text{applicable(C }\circ\text{ Z, V, E }\circ\text{ Z) } \leftarrow \text{action(C, V, E)}
\end{align*}
\]

with AC1 properties.
The fluents which occur in states are \{ill, apt, amb, hos\}. We restrict
the number of ground instances of the program instead of infinite one by
restriction the multiplicity of fluents in ground term to |multiplicity| ≤ 1.
We conclude that the number of ground terms is \(2^{\text{fluent}} - 1\). The max-
imum number of ground terms in the example is restricted to 31. The
ground terms as multisets of fluents are \{\{ill\},\{apt\},\{amb\},\{hos\},\{ill,apt\},
...,\{ill,apt,amb,hos\}\}.

The acyclicity of the program is guaranteed by the existence of a level map-
ping. Let \(\theta\) is a substitution which is an assignment of variables to ground
terms, and \(\text{length}\) is the length of a list.
At first, we define a level for the ground atoms of the program as follows:

- The level for facts is 0 and for applicable/3 is 2:
  \[
  L_1 = l(\text{action}(\text{ill} \circ \text{apt}, \text{carry}, \text{ill} \circ \text{amb})) = 0 \\
  L_2 = l(\text{applicable}(\text{ill} \circ \text{apt}, \text{carry}, \text{ill} \circ \text{amb})) = 2
  \]

- For causes/3:
  \[
  l(\text{causes}(I\theta, [A|P]\theta, G\theta)) = 3 + \text{length}([A|P]\theta) \\
  L_3 = l(\text{causes}(\text{ill} \circ \text{hos}, [], \text{ill} \circ \text{hos})) = 3 \\
  L_4 = l(\text{causes}(\text{ill} \circ \text{amb, [drive], ill} \circ \text{hos})) = 4 \\
  L_5 = l(\text{causes}(\text{ill} \circ \text{apt, [carry,drive], ill} \circ \text{hos})) = 5
  \]

Now, we check the acyclicity of the program. We choose the clause where
the head is applicable/3 predicate

\[
L_2 > L_1
\]

And for the causes rule, we have

\[
L_5 > L_2 \\
L_5 > L_4
\]

Now, it is not difficult to see that the example above is acyclic with respect
to \(l\), so we conclude that there is a least fixpoint of \(T_P\).
The least fixpoint for \(P_1\) with restricted number of ground terms is obtained
as follows:
\[
T_{FC,P1} \uparrow 0 = \emptyset
\]
\[
T_{FC,P1} \uparrow 1 = T_{FC,P1} \uparrow 0 \cup \{\text{[action}(\text{ill o apt, carry, ill o amb})],
\text{[action}(\text{ill o amb, drive, ill o hos})],
\text{causes}(\text{ill, []}, \text{ill}), \text{[causes}(\text{ill o hos, []}, \text{ill o hos})],...,
\text{causes}(\text{ill o apt o amb o hos, []}, \text{ill o apt o amb o hos})]\}
\]
\[
T_{FC,P1} \uparrow 2 = T_{FC,P1} \uparrow 1 \cup \{\text{[applicable}(\text{ill o apt, carry, ill o amb})],
\text{[applicable}(\text{ill o...o apt, [carry]}, \text{ill o...o amb})],
\text{[applicable}(\text{ill o amb, [drive]}, \text{ill o hos})],
\text{[applicable}(\text{ill o...o amb, [drive]}, \text{ill o...o hos})]\}
\]
\[
T_{FC,P1} \uparrow 3 = T_{FC,P1} \uparrow 2 \cup \{\text{[causes}(\text{ill o amb, [drive]}, []), \text{ill o hos})]\}
\]
\[
T_{FC,P1} \uparrow 4 = T_{FC,P1} \uparrow 3 \cup \{\text{[causes}(\text{ill o apt, [carry, drive]}, \text{ill o hos})]\}
\]
\[
T_{FC,P1} \uparrow 5 = T_{FC,P1} \uparrow 4
\]

Our aim is to generalize the concept of acyclicity to general program which occur a negative literal in a body of a clause from \( \mathcal{P} \). First, we extend a level mapping to mapping from ground literals to natural number by putting \( l(\neg A) = l(A) \) for all \( A \in \mathcal{P} \).

Let \( \mathcal{P} \) be acyclic program with respect to \( l \). The declarative semantics of \( \mathcal{P} \) is defined as a specific Herbrand interpretation \( M \) for \( \mathcal{P} \). \( M(n) \) contains all atoms in level \( n \).

\[
M(n) = T_{\mathcal{P}}(\bigcup_{i=0}^{n} M(i)) \cap \mathcal{B}_{\mathcal{P}}
\]

We denote on the declarative semantics of an acyclic program \( \mathcal{P} \) by \( M_{\mathcal{P}} \).

**Theorem 3.1.** Let \( \mathcal{P} \) be an equational logic program with restriction to acyclic program, then \( T_{\mathcal{P}} \) has a unique fixpoint \( M_{\mathcal{P}} \). We have to show:

1. \( M_{\mathcal{P}} \) is a fixpoint of \( T_{\mathcal{P}} \), and
2. \( T_{\mathcal{P}} \) operator has at most one fixpoint.

**Proof.**

1. Apt and Bezem have shown in Lemma 2.3 [9] for every acyclic program the immediate consequence operator has a fixpoint.

2. They have also shown in Lemma 2.4 [9] that acyclic program has at most one fixpoint. They have also shown that a unique fixpoint and the unique Herbrand model for \( \text{comp}(\mathcal{P}) \) are equivalent which is equivalent to \( M_{\mathcal{P}} \).

In the next subsection, we consider acyclic program with equational theory which contains a negative literals in their clauses.
3.2 Advanced conjunctive planning problems:

In conjunctive planning problems, we deal with simple actions where actions are only equipped with conditions and effects, and do not distinguish between conditions that are consumed and others that must be present without being consumed.

Let’s continue the story by adding the following information [1]:

He was really fat such that he could not fit through his apartment’s door. A helicopter was sent to help him. A crane was used to carry him out of his windows to the helicopter. The helicopter brought him to the hospital.

• The fluents are the same as previous, but we add a fluent the helicopter (hel).

• The possible action are described by carrying the patient to the ambulance car (carry), driving to the hospital (drive), carrying him out of his windows by a crane (crane), and piloting the helicopter to the hospital (piloting).

• The action (carry) has an obstacle \{fat\}

• The actions \{carry, drive, crane, piloting\} have a precondition \{ill\}

We introduced what so-called obstacles to an action. Obstacles are resources which can hinder an action from being taken. Unlike conditions and effects of an action, obstacles are neither consumed nor produced. Moreover, we also introduced so-called preconditions. Preconditions are conditions that must be present whenever an action is being taken.

We add to the program a particular clause hinder which is of the form

\[\text{hinder}(O \circ S, A) \leftarrow \text{inhabit}(O, A)\]

Or, in the words, if the obstacle \(O\) of an action \(A\) are part of the state \(O \circ S\) then action \(A\) is hindered in this state. The preconditions of an action are encoded with a binary predicate \text{pre}(R,A)\) where \(R\) encodes the precondition of an action \(A\). We modify applicable clause as follows

\[\text{applicable}(C \circ S, A, E \circ S) \leftarrow \text{action}(C, A, E)\]
\[\quad \land \neg\text{hinder}(C \circ S, A)\]
\[\quad \land \text{pre}(R, A)\]
\[\quad \land R \circ R' \approx C \circ S\]
It means that an action \( A \) is applicable at state \( C \circ S \) and its application yields a state \( E \circ S \) if condition \( C \) of \( A \) is a part of \( C \circ S \) and \( R \) is a precondition of \( A \) and there is no obstacle with action \( A \) at state \( C \circ S \).

The logic program \( P_2 \) is as follows:

\[
\begin{align*}
\text{pre}(\text{ill}, \text{carry}) \\
\text{pre}(\text{ill}, \text{drive}) \\
\text{pre}(\text{ill}, \text{crane}) \\
\text{pre}(\text{ill}, \text{piloting}) \\
\text{inhabit}(\text{fat}, \text{carry}) \\
\text{action}(\text{apt}, \text{carry}, \text{amb}) \\
\text{action}(\text{amb}, \text{drive}, \text{hos}) \\
\text{action}(\text{apt}, \text{crane}, \text{hel}) \\
\text{action}(\text{hel}, \text{piloting}, \text{hos}) \\
\text{causes}(S, [], S) \\
\text{hinder}(O \circ S, A) \leftarrow \text{inhabit}(O, A) \\
\text{causes}(I, [A|P], G) \leftarrow \text{applicable}(I, A, Q) \\
& \quad \land \text{causes}(Q, P, G) \\
\text{applicable}(C \circ S, A, E \circ S) \leftarrow \text{action}(C, A, E) \\
& \quad \land \neg \text{hinder}(C \circ S, A) \\
& \quad \land \text{pre}(R, A) \\
& \quad \land R \circ R' \approx C \circ S \\
S \approx S
\end{align*}
\]

We have here a predicate \( \text{hinder}/2 \) which is a negative literal in the body of the clause \( \text{applicable} \). To check the acyclicity of the program we show that the level mapping of all clauses of the program are satisfied. It is the same as before, but we have here a negative literal. For example, we assign a level for ground instances of \( \text{hinder} \) predicate. For instance

\[ l(\text{hinder}(\text{fat}, \text{carry})) = 1 \]

Now, we check the acyclicity for the clause where \( \text{hinder}/2 \) predicate occur in its body

\[
\begin{align*}
l(\text{applicable}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})) &> l(\text{hinder}(\text{fat}, \text{carry})), \\
l(\text{applicable}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})) &> l(\text{action}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})) \\
l(\text{applicable}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})) &> l(\text{pre}(\text{ill}, \text{drive})) \\
l(\text{applicable}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})) &> l(\text{ill} \circ \text{amb} \approx \text{ill} \circ \text{amb})
\end{align*}
\]

Thus, there exists a least fixpoint.
The least fixpoint $\text{lfp}(T_p)$ is obtained thus:

\[
T_{FC,P_2}↑0 = \emptyset
\]
\[
T_{FC,P_2}↑1 = T_{FC,P_2}↑0 \cup \{[\text{action}(\text{apt}, \text{carry}, \text{amb})],
\text{action}(\text{amb}, \text{drive}, \text{hos}), [\text{action}(\text{apt}, \text{crane}, \text{hel})],
\text{pre}(\text{ill}, \text{carry}), [\text{pre}(\text{ill}, \text{drive})], [\text{pre}(\text{ill}, \text{crane})],
[\text{pre}(\text{ill}, \text{piloting})],
\text{inhabit}(\text{fat}, \text{carry})], [\text{action}(\text{hel}, \text{piloting}, \text{hos})],
[\text{causes}(\text{ill}, [], \text{ill})], [\text{causes}(\text{ill} \circ \text{hos}, [], \text{ill} \circ \text{hos})], ...,
[\text{causes}(\text{ill} \circ ... \circ \text{apt}, [], \text{ill} \circ ... \circ \text{apt})],
[\text{ill}] = [\text{ill}], ..., [\text{ill} \circ \text{apt} \circ ... \circ \text{hos}] = [\text{ill} \circ \text{apt} \circ ... \circ \text{hos}]\}
\]
\[
T_{FC,P_2}↑2 = T_{FC,P_2}↑1 \cup \{[\text{hinder}(\text{fat}, \text{carry})], [\text{hinder}(\text{fat} \circ \text{ill} \circ ..., \text{carry})]\}
\]
\[
T_{FC,P_2}↑3 = T_{FC,P_2}↑2 \cup \{[\text{applicable}(\text{ill} \circ \text{amb}, \text{drive}, \text{ill} \circ \text{hos})],
[\text{applicable}(\text{ill} \circ ... \circ \text{amb}, \text{drive}, \text{ill} \circ ... \circ \text{hos})],
[\text{applicable}(\text{ill} \circ \text{amb}, \text{crane}, \text{ill} \circ \text{hel})],
[\text{applicable}(\text{ill} \circ ... \circ \text{amb}, \text{crane}, \text{ill} \circ ... \circ \text{hel})],
[\text{applicable}(\text{ill} \circ \text{hel}, [\text{piloting}], \text{ill} \circ \text{hos})],
[\text{applicable}(\text{ill} \circ ... \circ \text{hel}, [\text{piloting}], \text{ill} \circ ... \circ \text{hos})]\}
\]
\[
T_{FC,P_2}↑4 = T_{FC,P_2}↑3 \cup \{[\text{causes}(\text{ill} \circ \text{hel} \circ \text{fat}, [\text{piloting}, []], \text{ill} \circ \text{hos} \circ \text{fat})]\}
\]
\[
T_{FC,P_2}↑5 = T_{FC,P_2}↑4 \cup \{[\text{causes}(\text{ill} \circ \text{apt} \circ \text{fat}, [\text{crane}, \text{piloting}], \text{ill} \circ \text{hos} \circ \text{fat})]\}
\]
\[
T_{FC,P_2}↑6 = T_{FC,P_2}↑5
\]

4 Conclusion

We consider general program which is restricted to acyclic program. The acyclicity of the program is guaranteed by the existence of a level mapping. We extend the level mapping to meet the equational theory with the logic programs. Thus, we have a least fixpoint for general programs with equational theories.

Acknowledgments

I thank Prof. Steffen Hölldobler and Emmanuelle-Anna Dietz very much for their supervision and academic supports.
I am supported by the European Master’s Program in Computational Logic (EMCL).
References


Default Rules in Functional Logic Programs
– Extended Abstract –

Sergio Antoy
1 Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

Michael Hanus
2 Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. In functional logic programs, rules are applicable independently of textual order, i.e., any rule can potentially be used to evaluate an expression. This is similar to logic languages and opposite to functional languages, e.g., Haskell enforces a strict sequential interpretation of rules. However, in some situations it is convenient to express alternatives by means of compact default rules. Although default rules are often used in functional programs, the non-deterministic nature of functional logic programs does not allow to directly transfer this concept from functional to functional logic languages in a meaningful way. In this paper we propose a new concept of default rules for Curry that supports a programming style similar to functional programming while preserving the core properties of functional logic programming, i.e., completeness, non-determinism, and logic-oriented uses of functions. We discuss the basic concept and sketch an initial implementation of it which exploits advanced features of functional logic languages.

1 Motivation

Functional logic languages combine the most important features of functional and logic programming in a single language (see [3, 6] for recent surveys). In particular, the functional logic language Curry [7] extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Moreover, the amalgamated features of Curry supports new programming techniques, like deep pattern matching through the use of functional patterns, i.e., evaluable functions at pattern positions [1]. As a simple example, consider an operation isSet intended to check whether a given list represents a set, i.e., does not contain duplicates. In Curry, we might implement it as follows (“++” denotes the concatenation of lists):

\[
\begin{align*}
\text{isSet } (_++[x]+++_++[x]++_+) & = \text{False} \\
\text{isSet } _ & = \text{True}
\end{align*}
\]

The first rule uses a functional pattern: it returns False if the argument matches a list where two identical elements occur. If this is not the case, the second rule

* This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.
returns True. However, according to the Curry’s semantics, all rules are tried to evaluate an expression. Therefore, the second rule is always applicable to calls of isSet so that the expression isSet [1,1] will be evaluated to False and True.

The unindented application of the second rule can be avoided by the additional requirement that this rule should be applied only if no other rule is applicable. We call such a rule a default rule and mark it by adding the suffix ‘default’ to the function’s name. Thus, if we define isSet with the rules

\[
\text{isSet} \left( \_++[x]++[x]++ \right) = \text{False} \\
\text{isSet} \text{'default } \_ = \text{True}
\]

then isSet [1,1] evaluates only to False and isSet [0,1] only to True.

In the following, we sketch an implementation of default rules in Curry where we assume familiarity with the basic concepts of functional logic programming and Curry (see [3, 6, 7]).

2 Default Rules

Default rules are often used in both functional and logic programming. For instance, the following Haskell function reverses a two-element list and leaves all other lists unchanged:

\[
\text{rev2} \left( [x,y] \right) = [y,x] \\
\text{rev2} \left( xs \right) = xs
\]

The second rule is applied only if the first rule is not applicable, which yields the intended semantics. We can avoid the consideration of rule orderings by replacing the second rule with rules for the patterns not matching the first rule:

\[
\text{rev2} \left( [x,y] \right) = [y,x] \\
\text{rev2} \left( [] \right) = [] \\
\text{rev2} \left( [x] \right) = [x] \\
\text{rev2} \left( x:y:z:xs \right) = x:y:z:xs
\]

This coding is cumbersome in general and impossible in conjunction with functional patterns, as used in the first rule of isSet above, since a functional pattern conceptually may denote an infinite set of standard patterns (e.g., \([x,x]\), \([x,\_x]\), \([\_x,\_x]\), \([\_x,\_x,\_x]\), \ldots). Thus, there is no finite complement of some functional patterns.

In Prolog, one often uses the cut operator to implement the behavior of default rules. For instance, rev2 can be defined as a Prolog predicate as follows:

\[
\text{rev2}([X,Y],[Y,X]) :- !. \\
\text{rev2}(Xs,Xs).
\]

Although this behaves as intended for instantiated lists, the completeness of logic programming is destroyed by the cut operator. For instance, the goal \(\text{rev2}([],[])\) is provable, but Prolog does not compute the answer \(\{Xs=[],Ys=[]\}\) for the goal \(\text{rev2}(Xs,Ys)\).

These examples show that a new concept of default rules is required for functional logic programming if we want to keep the strong properties of the base language, in particular, the completeness of logic-oriented evaluations. To avoid developing a new logic foundation of functional logic programming with
default rules, we try to reuse existing features of functional logic languages. We
describe our approach explaining the translation of the default rule for \texttt{rev2}. The
extension to functional patterns and conditional rules can be done in a similar way.

An operation is defined by a set of “standard” rules and one optional default
rule that is applied only if no standard rule is applicable because it do not match
or its condition is not satisfiable. For this reason, we translate a default rule into
a standard rule by adding the condition that no other rule is applicable. For this
purpose, we translate the original non-default rules into “test applicability only”
rules where the right-hand side is replaced by a constant (here: the unit value
“()”):

\texttt{rev2'TEST} \ [x,y] = ()

Now we add to the default rule the condition that \texttt{rev2'TEST} is not applicable.
Since we are interested in the failure of attempts to apply \texttt{rev2'TEST}, we use a
primitive for encapsulating search to check whether \texttt{rev2'TEST} has no value. In
functional logic programming, set functions \cite{2} or an operator \texttt{allValues} \cite{5}
have been proposed for this purpose, which behave similarly to Prolog’s \texttt{findall}
but can be used in a declarative manner. Using these primitives, one could translate
the default rule into

\texttt{rev2'DEFAULT} \ xs | isEmpty \ (allValues \ (rev2'TEST \ xs)) = xs

Hence, this rule can be applied only if all attempts to apply a non-default rule
fail. To complete our example, we add this translated default rule as a further
alternative to the non-default rule so that we obtain the definition

\texttt{rev2} \ [x,y] = [y,x]
\texttt{rev2} \ xs | isEmpty \ (allValues \ (rev2'TEST \ xs)) = xs

Thanks to the logic features of Curry, one can use this definition also to generate
appropriate argument values for \texttt{rev2}. For instance, if we evaluate the expression
\texttt{rev2} \ xs with the Curry implementation KiCS2 \cite{4}, the search space is finite and
computes, among others, the binding \{\texttt{xs=}[]}\. This shows that our concept of
default rules is more powerful than existing concepts in functional or logic pro-
gramming. The actual transformation scheme for default rules is more advanced
than sketched above in order to accommodate also functional patterns and condi-
tionals rules and to ensure the optimality of functional logic computations even
in the presence of default rules.

3 Examples

To show the advantages of default rules for functional logic programming, we
sketch a few more examples. In the classical \textit{n}-queens puzzle, one must place
\textit{n} queens on a chess board so that no queen can attack another queen. This
can be solved by computing some permutation of the list \{1..\textit{n}\}, where the \textit{i}-
th element denotes the row of the queen placed in column \textit{i}, and check whether
this permutation is a safe placement. The latter property can easily be expressed
with functional patterns and default rules where the non-default rule fails on a
non-safe placement:
safe (_++[x]++y++[z]++_) | abs (x-z) == length y + 1 = failed
safe'default xs = xs

Hence, a solution can be obtained by computing a safe permutation:

queens n = safe (permute [1..n])

This example shows that default rules are a convenient way to express negation-as-failure from logic programming. This programming pattern can also be applied to solve the map coloring problem. Our map consists of the states of the Pacific Northwest and a list of adjacent states:

data State = WA | OR | ID | BC

adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]

Furthermore, we define the available colors and an operation that associates (non-deterministically) some color to a state (the infix operator "?" denotes a non-deterministic choice between its arguments):

data Color = Red | Green | Blue

color x = (x, Red ? Green ? Blue)

A map coloring can be computed by an operation solve that takes the information about potential colorings and adjacent states as arguments, i.e., we compute correct colorings by evaluating the initial expression

solve (map color [WA,OR,ID,BC]) adjacent

The operation solve fails on a coloring where two states have an identical color and are adjacent, otherwise it returns the coloring:

solve (_++[(s1,c)]++_++[(s2,c)]++_) (_++[(s1,s2)]++_) = failed
solve'default cs _ = cs

References

Bottom-Up Evaluation of Datalog: Preliminary Report

Stefan Brass and Heike Stephan

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, Von-Seekendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de, stephan@informatik.uni-halle.de

Abstract. Bottom-up evaluation of Datalog has been studied for a long time, and is standard material in textbooks. However, if one actually wants to develop a deductive database system, it turns out that there are many implementation options. For instance, the sequence in which rule instances are applied is not given. In this paper, we study a method that immediately uses a derived tuple to derive more tuples. In this way, storage space for intermediate results can be reduced. The main contribution of our method is the way in which we minimize the copying of values at runtime, and do much work already at compile-time.

1 Introduction

The efficient evaluation of queries to logic programs remains an everlasting problem. Of course, big achievements have been made, but at the same time problem size and complexity grows. Any further progress can increase the practical applicability of logic-based, declarative programming.

Our long-term goal is to develop a new deductive database system. This has many aspects, for instance, language design. However, in the current paper, we exclude all special language features, including negation, and focus on efficient query evaluation for basic Datalog.

The magic set method is the standard textbook method for making bottom-up evaluation goal-directed. Many optimizations have been proposed, including our own SLDMagic method [1] and a method based on Earley deduction [3]. We assume in the current paper that such a rewriting of the program has been done, so we can concentrate on pure bottom-up evaluation.

As we understand it, bottom-up evaluation is an implementation of the $T_P$-operator that computes the minimal model of the program. However, an implementation is free in the order in which it applies the rule instances, while the $T_P$-operator first derives all facts that are derivable with a given set of known facts, before the derived facts are used (in the next iteration). Furthermore, facts do not have to be stored until the end of query evaluation, but can be deleted as soon as all possible derivations using them have been done, except for the facts that are relevant for the query. Therefore, the sequence of rule instance application becomes important. If one computes predicate by predicate as the
standard textbook method, one of course needs to store the entire extension of the predicates. However, if one uses derived tuples immediately, it might be possible to store only one tuple of the predicate during the evaluation. Of course, for duplicate elimination and termination, it might still be necessary to store extensions of a few selected predicates. It is also not given that tuples (facts) must be represented explicitly as records or objects in the program. It suffices if one knows where the values of single columns (predicate arguments) can be found. In this way, a lot of copying can be saved because tuples for the rule head are typically constructed from values bound in the rule body. Of course, one must ensure that the values are not changed before all usages are finished.

Our plan is to translate Datalog to C++, and to generate executable code from the resulting program. This permits to use existing compilers for low-level optimizations and gives an interface for defining built-in predicates. In [2], we already discussed implementation alternatives for bottom-up evaluation and did performance comparisons for a few example programs. Now we will improve the “push method” from that paper by changing the set of variables used to represent intermediate facts. This is essential for reducing the amount of copying. It also enables us to do more precomputation at “compile time”.

The idea of immediately using derived facts to derive more facts is not new. For instance, variants of semi-naive evaluation have been studied which work in this way [10, 12]. It also seems to be related to the propagation of updates to materialized views. However, the representation of tuples at runtime and the code structure is different from [10] (and this is essential for the reduction of copying values). The paper [12] translates from a temporal Datalog extension to Prolog, which makes any efficiency comparison dependend on implementation details of the used Prolog compiler. We also believe that the rule application graph introduced in our paper is a useful concept. Further literature about the implementation of deductive database systems is, for instance, [8, 4, 9, 11, 7, 13]. A current commercial deductive DB system is Logicblox [5]. A benchmark collection is OpenRuleBench [6].

2 Basic Definitions

In this paper, we consider basic Datalog, i.e. pure Prolog without negation and without function symbols (i.e. terms can only be variables or constants). We also assume without loss of generality that all rules have at most two body literals. The output of our rewriting methods [1, 3] has this property. (But in any case, it is no restriction since one can introduce intermediate predicates.) Finally, we require range-restriction (allowedness), i.e. all variables in the head of the rule must also appear in a body literal. For technical purposes, we assume that each rule has a unique rule number.

As usual in deductive databases, we assume that EDB and IDB predicates are distinguished (“extensional” and “intensional database”). EDB predicates are defined by facts only, e.g. stored in a relational database or specially formatted files. Also program input is represented in this way. IDB predicates are defined
by rules. There is a special IDB-predicate \texttt{answer} that only appears in the head of one or more rules. The task is to compute the extension of this predicate in the minimal model of the program.

We assume that the logic program for the IDB predicates as well as the query (i.e. the \texttt{answer}-rules) are given at “compile time”, whereas the database for the EDB predicates is only known at “runtime”. Since the same program can be executed several times with different database states, any optimization or precomputation we can do at compile time will pay off in most cases. It might even be advantageous in a single execution because the database is large.

Since we want to generate C++ code, we assume that there is a data type known for every argument of an EDB predicate. The method does not need type information for IDB predicates (this is implicitly computed).

As mentioned above, our rewriting methods \cite{1, 3} produce rules that have at most two body literals. Furthermore the case of two IDB-literals is rare—it is only used in special cases for translating complex recursions. Most rules have one body literal with IDB-predicate and one with EDB-predicate. Of course, there are also rules with only one body literal (EDB or IDB).

\section{Accessing Database Relations}

The approach we want to follow is to translate Datalog into C++, which can then be compiled to machine code. Of course, we need an interface to access relations for the EDB predicates. These relations can be stored in a standard relational database, but it is also possible to program this part oneself (at the moment, we do not consider concurrent updates and multi-user access).

We assume that it is possible to open a cursor (scan, iterator) over the relation, which permits to loop over all tuples. We assume that for every EDB predicate \( p \) there is a class \( p\_\text{cursor} \) with the following methods:

\begin{itemize}
  \item \texttt{void open()}: Open a scan over the relation, i.e. place the cursor before the first tuple.
  \item \texttt{bool fetch()}: Move the cursor to the next tuple. This function must also be called to access the first tuple. It returns \texttt{true} if there is a first/next tuple, or \texttt{false} if the cursor is at the end of the relation.
  \item \( T \ col\_i() \): Get the value of the \( i \)-th column (attribute) in the current tuple. Here \( T \) is the type of the \( i \)-th column.
  \item \texttt{close()}: Close the cursor.
\end{itemize}

For recursive rules, we will also need

\begin{itemize}
  \item \texttt{push()}: Save the state of the cursor on a global stack.
  \item \texttt{pop()}: Restore the state of the cursor.
\end{itemize}

A relation may have special access structures (e.g. it might be stored in a B-tree or an array). Then not only a full scan (corresponding to binding pattern \( ff\ldots f \)) is possible, but also scans only over tuples with given values for certain arguments. We assume that in such cases there are additional cursor classes called
\[p_{\text{cursor}}_\beta, \text{ with a binding pattern } \beta. \] These classes have the same methods as the other cursor classes, only the \texttt{open}-method has parameters for the bound arguments. E.g. if \(p\) is a predicate of arity 3 that permits particularly fast access to tuples with a given value of the first argument, and if this argument has type \texttt{int}, the class \(p_{\text{cursor}}_{\text{bff}}\) would have the method \texttt{open(int x)}.

4 Duplicate Elimination and Termination

The main contribution of this paper is the way in which copying and materialization of tuples is avoided. Our method basically pushes newly derived facts to body literals where they can be used to derive further facts.

However, in the presence of recursion, we must be able to notice whether a derived tuple is new or not. Therefore, in each recursive cycle, at least one predicate must be materialized ("tabled") to ensure termination. A simple solution is to create hash tables for the predicates in question.

This solution means that we materialize the extensions of some IDB predicates (hopefully, only a few) and copy all data values for the tuples of these predicates. In some cases, information about order or acyclicity might help to avoid this. Information about keys and data distribution could be used to make sensible optimization decisions. Furthermore, if tuples are produced in a sort order, the duplicate check can be done very efficiently and without storing the predicate extension. All this is subject of our future work.

It is also interesting that the data values in a derived tuple are stored at different times in program variables. For instance, we might know that when \(p(X, Y)\) is generated, \(X\) only seldom changes, and \(Y\) changes much more often. Then a nested relation might be best for tabling the predicate for the purpose of duplicate detection.

Of course, breaking each recursive cycle with a duplicate detection is only the minimum we have to do to ensure termination. Also non-recursive rules can generate duplicates, and in some cases it might be more efficient to detect these duplicates early in order to avoid duplicate computations (since the price for duplicate detection is quite high, in other cases it might be more efficient to simply do the duplicate work).

5 Code Generation: Overall Structure

The result of the translation looks basically as shown in Figure 1. So there are many small code pieces, each with a label that is suitable for a \texttt{goto}. Furthermore, when there are several things to do, e.g. a generated fact can be used in more than one rule, a backtrack point is set up for the second rule, and then a \texttt{goto} is done for the first. When an execution path reaches an end, the \texttt{switch} is left with \texttt{break}, and one of the delayed tasks is taken from the stack. Therefore, each code piece also has a unique number, which can be stored on the backtrack stack, and used in the \texttt{switch} to reach the code piece.
Fig. 1. Overall structure of the generated code

Optimizations are possible, e.g., one can order the code pieces such that some jumps can be eliminated, because the target is immediately following. Some backtrack points can be avoided by finding a suitable code sequence.

5.1 Declaration Section

Data not known at compile time always originates from the database. In order to minimize copying, we (usually) introduce a C++ variable only for Datalog variables which

- occur in an EDB body literal,
- but do not occur in an IDB body literal of that rule (because then the value comes from another rule, where a variable has been created, if the value is not known at compile time),
- and occur in the head of that rule (because otherwise the value does not really have to be processed in the program).

For instance, consider the following rule:

\[ p(X, Y, a) \leftarrow q(Y) \land r(X, Y, Z, Z). \]

If \( q \) is an IDB predicate and \( r \) an EDB predicate, we create a C++ variable only for \( X \). A variable or constant for \( Y \) exists already when the rule is activated.

In seldom cases of recursive rule applications (see Section 5.4 below) we create C++ variables for all variables of the rule.

If the above condition shows the we must create a C++ variable for variable \( X \) in rule \( \rho \), we generate the following code line in the declaration section:

\[ T \text{\_\_}_X; \]

We use the prefix with the rule number so that there can be no name conflicts between variables of different rules. \( T \) is the C++ data type for the database column in which \( X \) occurs.
5.2 Symbolic Facts

A symbolic fact consists of an IDB predicate \( p \) and a tuple \((t_1, \ldots, t_n)\) of C++ variables (i.e. their identifiers) and constants, where \( n \) is the arity of \( p \). So a symbolic fact represents what is known at compile time about a fact that will be derived at runtime. For some arguments, we might know the exact value (a constant), for other arguments, we know the C++ variable which will contain the value.

An initial set of symbolic facts is derived by rules without IDB body literals. Then our task is to pass each derived symbolic fact to matching IDB body literals and to derive a symbolic fact for the rule head. For each such rule application, a code piece is generated which does the remaining computation at runtime.

“Matching” between a symbolic fact and a body literal means that they are unifiable. In general a full unification must be done (at compile time). Consider e.g. the body literal \( p(X, X, a) \) and the symbolic fact \( p(b, v_1Y, v_1Y) \). The rule cannot be applied to the symbolic fact, so no code is generated for this case.

5.3 Rule Application Graph

A “Symbolic Rule Application” is

- a rule from the logic program with one IDB body literal, together with a symbolic fact matching this body literal,
- a rule without IDB body literals,
- a rule with two IDB body literals, together with a symbolic fact matching one of them. (In the rare case of two IDB body literals, we use temporary tables for facts matching each body literal. The symbolic fact in this rule application describes the situation that we just computed a new fact for one of the IDB body literals. For the other body literal we use the table with previously computed facts.)

The result of a symbolic rule application is a symbolic fact. Let \( p(t_1, \ldots, t_n) \) be the head of the rule, and \( \rho \) be its rule number. If the rule has an IDB body literal, let \( \theta \) be a most general unifier with the input symbolic fact. We require that variable-to-variable bindings are done such that logic variables are replaced by C++ variables. Then the derived symbolic fact is \( p(u_1, \ldots, u_n) \), where \( u_i \) is

- \( t_i \) if \( t_i \) is a constant.
- \( t_i \theta \) if \( t_i \) is a variable which appears in the IDB body literal (if there is one).
- \( v_{\rho.X} \) if \( t_i \) is a variable \( X \) which does not appear in the IDB body literal.

Now we can do a standard fixpoint computation to compute all symbolic facts which are derivable from the program. This process will come to an end, because the number of symbolic facts is bounded: There is only a finite number of C++ variables (at most the number of variables in the given logic program, where variables with the same name in different rules count as distinct). Furthermore, only a finite number of constants occurs in the given logic program (constants which appear only in the database are not known at “compile time” and not used for computing symbolic facts).
The structure of the computation can be shown in a “rule application graph”. It has two types of nodes, namely symbolic facts (“fact nodes”), and symbolic rule applications (“rule nodes”). There is an edge from every symbolic fact to every symbolic rule application which uses the symbolic fact. Furthermore, there is an edge from every symbolic rule application to the symbolic fact it generates.

Of course, it is possible to show only the rule in nodes for symbolic rule applications (since the symbolic fact is identified by the incoming edge, except in the case of two IDB body literals). However, then there can be several nodes marked with the same rule: It is possible that a single rule is compiled several times for different symbolic facts matching its IDB body literal.

Note also that not every application of a recursive rule to a symbolic fact is actually recursive: Only if the same symbolic fact can be generated by applying this rule (maybe indirectly via other rules), we have to be prepared for recursive invocations of the code piece for the symbolic rule application. This can be seen from cycles in the graph.

Finally, nodes in the graph from which there is no path to an answer-node can be eliminated: They do not contribute to the computation of the answer. If the program is the result of a program transformation like magic sets, this path will not be followed at runtime, but it is better not to generate code for it. An example of such a program is

\[
\begin{align*}
\text{answer}(X) & \leftarrow q(X, a).
q(X, Y) & \leftarrow p(Y, X).
p(a, X) & \leftarrow r(X).
p(b, X) & \leftarrow s(X).
\end{align*}
\]

The rule application graph is shown in Figure 2. The right path is useless. In

\[
\begin{align*}
\text{answer}(v3, X) & \\
\text{answer}(X) & \leftarrow q(X, a).
q(v3, X, a) & \\
q(X, Y) & \leftarrow p(Y, X).
p(a, v3, X) & \\
p(a, X) & \leftarrow r(X).
q(v4, X, b) & \\
q(X, Y) & \leftarrow p(Y, X).
p(b, v4, X) & \\
p(b, X) & \leftarrow s(X).
\end{align*}
\]

**Fig. 2.** Rule Application Graph with Useless Part (to be eliminated).
the code generation below, we assume that this has been removed, i.e. every fact node with a predicate different from “answer” has an outgoing edge.

5.4 Variable Conflicts

In rare cases of recursive rule applications, it is possible that a rule is applied to a symbolic fact which contains already a variable generated for that rule. An example is

\[ p(X,Y) \leftarrow r(X,Y). \]
\[ p(Y,Z) \leftarrow p(X,Y) \land r(Y,Z). \]

The first rule generates the symbolic fact \( p(v_1,X,v_1,Y) \). When we insert this into the second rule, we get \( p(v_1,Y,v_2,Z) \). Now we have to insert this again into the second rule: \( v_2,Z \) contains the input value for \( Y \), but must also be set with a new data value from \( r \). In this case, some copying seems unavoidable. While there are optimizations possible, the simplest solution is to create a C++ variable for each logical variable of the rule, and to copy first the values from the input fact to the right variable (which might need temporary variables, e.g. for swapping the values of two variables). For recursive rule applications, the previous variable values are also stored on a stack (see Section 5.6 below).

5.5 Labels for Code Pieces

We need a goto label and/or a case selector value (a unique number) for each code piece implementing a symbolic rule application. We write

\[ 1_{\text{start}}(p(t_1,\ldots,t_n), \rho, p(u_1,\ldots,u_n)) \]

for the goto-label of the code piece for application of rule \( \rho \) with body literal \( p(u_1,\ldots,u_n) \) to the symbolic fact \( p(t_1,\ldots,t_n) \). The implementation will replace this by \( 1_{\text{start}}\_n \) with some unique number \( n \). The symbolic constant for the case-value is written as \( L_{\text{START}}(\ldots) \) (and also made a legal C++ identifier by using the same unique number). Sometimes there are continuations or other code pieces, therefore the label is marked as “start”.

5.6 Protection of Variable Values

Of course, when a code piece corresponding to a symbolic rule application is executed, the C++ variables in the symbolic fact \( p(t_1,\ldots,t_n) \) must still have the same value as when this task was generated. It is possible that the ID/label of the code piece was pushed on the backtrack stack and it is executed only later.

However, for every C++ variable, a new value is assigned only in code pieces for the single rule for which the variable was introduced (to hold a data value for an EDB literal in that rule).

Furthermore, it is important that the backtrack points are kept on a stack. So we will return to that rule only after all backtrack points which use the value
(and are thus generated later) have been processed—unless the rule is recursive. In this case, the variable value must be saved (on another stack suitable for the data type), and we put the ID of a code piece on the backtrack stack which restores the variable value. This is done whenever we enter a recursive rule, and only for variables set in this rule (the derived fact might contain also variables passed from elsewhere and not changed in the rule).

If the backtrack stack shrinks below this point, all usages of the new variable value are done, and the old value is restored, so that older backtrack points find the value which was current when the backtrack point was created.

6 Code Pieces

In this section, we define a number of code pieces which are translations of different types of rules. Each code piece corresponds to a symbolic rule application. For simplicity, we do not consider variable conflicts (Section 5.4) here.

6.1 IDB-Facts

Suppose the program contains an IDB-fact \( p(c_1, \ldots, c_n) \). Whenever this matches a body literal \( p(t_1, \ldots, t_n) \) of a rule \( \rho \), the case selector value

\[
\text{L_START}(p(c_1, \ldots, c_n), \rho, p(t_1, \ldots, t_n))
\]

is pushed on the backtrack stack during initialization.

6.2 One EDB-Body Literal

Consider the rule \( p(t_1, \ldots, t_n) \leftarrow r(u_1, \ldots, u_m) \) where \( r \) is an EDB predicate. Let \( \rho \) be the rule number. Let \( p(t_1, \ldots, t_n) \) be the symbolic fact generated by the rule \( (t_i := t_i \text{ if } t_i \text{ is a constant, and } t_i := \nu_{\rho_{-}X} \text{ if } t_i \text{ is the variable } X) \).

Among all possible cursors \( \text{cursor}_{r_{-}\beta} \) for \( r \) choose one such that for all bound argument positions \( i \) (i.e. \( \beta_i = b \)), \( u_i \) is a constant. This is always possible because every relation supports a full table scan, i.e. an access path with all argument positions “free”. But obviously, if there are constants among the \( u_i \), and there are available indexes, it is best to choose one with the smallest estimated result size. In the declaration section, generate

\[
\text{cursor}_{r_{-}\beta} c\rho;
\]

Define symbolic constants \( \text{L_INIT}_{-}\rho \) and \( \text{L_CONT}_{-}\rho \) as unique numbers for cases in the switch. Generate the following code in the initialization section:

\[
\text{backtrack_stack.push(L_INIT}_{-}\rho);
\]

All following code is generated in the switch:

1. Generate

\[
\text{case L_INIT}_{-}\rho:
\]
2. Let \( i_1, \ldots, i_k \) be the bound argument positions in \( \beta \). Generate:
\[
c_\rho . \text{open}(u_{i_1}, \ldots, u_{i_k});
\]
(Note that although another case follows, execution simply continues.).

3. Generate:
\[
\text{case L\_CONT\_}\rho: \\
\]
The following loop (item 4) is left with goto when the first fact is generated. But before the jump, this case label is pushed on the backtrack stack, so that the loop is continued later.

4. Generate
\[
\text{while}(c_\rho . \text{fetch}) \{
\]
5. Let \( u_{i_1}, \ldots, u_{i_k} \) be the constants among the \( u_1, \ldots, u_m \) which correspond to free argument positions in \( \beta \). If \( k \geq 1 \), generate
\[
\text{if}(c_\rho . \text{col}_{i_1}() \neq u_{i_1} \mid \cdots \mid c_\rho . \text{col}_{i_k}() \neq u_{i_k})
\]
\[
\text{continue};
\]
I.e. if the current tuple of the EDB-predicate does not have the required values for the constant arguments, we immediately start the next iteration of the while-loop (i.e. fetch the next tuple).

6. For every variable \( Y \), which appears more than once among the \( u_1, \ldots, u_m \):
Let \( u_{i_1}, \ldots, u_{i_k} \) be all equal to \( Y \) (note that \( k \geq 2 \)). Generate:
\[
\text{if}(c_\rho . \text{col}_{i_1}() \neq c_\rho . \text{col}_{i_2} \mid \cdots \mid c_\rho . \text{col}_{i_{k-1}}() \neq c_\rho . \text{col}_{i_k}())
\]
\[
\text{continue};
\]
7. For every variable \( X_i \) in the head let \( u_j \) be any occurrence of this variable among the \( u_1, \ldots, u_m \). Because of the range restriction (allowedness) condition on the rules, it must occur in the body. Generate for each \( X_i \):
\[
v_{\rho X_i} = c_\rho . \text{col}_j();
\]
8. In case the predicate \( p \) was selected for a duplicate check, the following must be done here: The result tuple \( p(\bar{t}_1, \ldots, \bar{t}_n) \) with the current values of the C++ variables is entered into a hash table or other data structure. If the tuple was already present, one simply does “continue;” to skip it.

9. Generate:
\[
\text{backtrack_stack.push(L\_CONT}\_\rho); \\
\]
Since this code piece will change the values of the variables introduced in the rule, it must be on the stack below every task using the generated tuple.

10. Let \( \rho_1, \ldots, \rho_k \) be all rules with an IDB body literal \( B_i, i := 1, \ldots, k \), which matches the generated symbolic fact \( p(\bar{t}_1, \ldots, \bar{t}_n) \). For \( i := 2, \ldots, k \), generate
\[
\text{backtrack_stack.push(L\_START}(p(\bar{t}_1, \ldots, \bar{t}_n), \rho_i, B_i));
\]
Finally, generate
\[
goto \text{l\_start}((\bar{p}(\bar{t}_1, \ldots, \bar{t}_n), \rho_1, B_1));
\]
11. Generate
\[
}\ // \text{End of while-loop}
\]
break;
The break; is important if the while-loop ends because no (further) matching fact is found in the relation \( r \). Otherwise, the loop is left with goto when the first/next matching fact is found.
6.3 Two EDB-Body Literals

In the output of SLDMagic, this case does not occur. However, it is easy to extend the above program code. One uses two cursors, one for each body literal, and two nested while-loops. For simplicity, we implement all joins as “nested loop join”. Later, sort orders might be used, so that also a “merge join” can be generated (which is faster than the nested loop join).

6.4 One IDB-Body Literal

Consider the rule

\[ p(t_1, \ldots, t_n) \leftarrow q(u_1, \ldots, u_m), \]

where \( q \) is an IDB-predicate. Let \( \rho \) be the number of this rule. There is one code piece per symbolic fact \( q(\bar{u}_1, \ldots, \bar{u}_m) \) which matches the body literal. Let \( \theta \) be a most general unifier, where variable-to-variable bindings are done such that logic variables are replaced by C++ variables i.e. \( u_i \) is replaced by \( \bar{u}_i \), if both are variables. The generated symbolic fact is \( p(t_1\theta, \ldots, t_n\theta) \). Note that because of the range restriction requirement, every variable among the \( t_i \) also appears as an \( u_j \), and then it is unified with a constant or a C++ variable.

1. Generate

\[
\text{case L\_START}(q(\bar{u}_1, \ldots, \bar{u}_m), \rho, q(u_1, \ldots, u_m)):
\]

2. Now the part of the unification which can only be done at runtime must be generated. Let \( V_1, \ldots, V_k \) be all C++ variables which \( \theta \) replaces by constants or a different variable (i.e. \( V_i \theta \neq V_i \)). If \( k > 0 \), generate:

\[
\text{if}(V_1 \neq V_1\theta | | \cdots | | V_k \neq V_k\theta)
\]

\[
\text{break};
\]

So we simply stop executing this code piece if the current fact for \( q \) does not unify with the body literal.

3. In case predicate \( p \) was selected for a duplicate check, the code to enter the result tuple \( p(t_1\theta, \ldots, t_n\theta) \) with the current values of the C++ variables into a hash table would go here. If the tuple was already present (so we just computed a duplicate), one simply does “break;” to end the code piece. Then another task will be taken from the backtrack stack.

4. Let \( \rho_1, \ldots, \rho_k \) be all rules with an IDB body literal \( B_i, i := 1, \ldots, k \), which matches the generated symbolic fact \( p(t_1\theta, \ldots, t_n\theta) \). For \( i := 2, \ldots, k \), generate

\[
\text{backtrack\_stack.push}(\text{L\_START}(p(t_1\theta, \ldots, t_n\theta), \rho_i, B_i));
\]

Finally, generate

\[
\text{goto l\_start}(p(t_1\theta, \ldots, t_n\theta), \rho_1, B_1);
\]
6.5 One IDB- and one EDB-Body Literal

Consider the rule
\[ p(t_1, \ldots, t_n) \leftarrow q(u_1, \ldots, u_m) \land r(v_1, \ldots, v_l), \]
where \( q \) is an IDB-predicate and \( r \) is an EDB-predicate. Let \( \rho \) be the number of this rule. There is one code piece per symbolic fact \( q(\bar{u}_1, \ldots, \bar{u}_m) \) which matches the IDB body literal. Let \( \theta \) be a most general unifier and \( p(\bar{t}_1, \ldots, \bar{t}_n) \) be the generated symbolic fact as defined in Section 5.3. As in Section 6.2, select a binding pattern \( \beta \) for accessing the EDB-relation \( r \). A value for \( v_i \) is known (i.e. \( \beta_i \) can be “bound”) if \( v_i \) is a constant or a variable which also appears in \( q(u_1, \ldots, u_m) \) (when execution reaches this code piece, a concrete fact is given for the IDB body literal). In the declaration section, generate
\[
\text{cursor_r_}\beta \ c\rho;
\]
All following code is generated in the `switch`:

1. Generate
   ```
   \text{case L\_START}(q(\bar{u}_1, \ldots, \bar{u}_m), \rho, q(u_1, \ldots, u_m));
   \text{l\_start}(q(\bar{u}_1, \ldots, \bar{u}_m), \rho, q(u_1, \ldots, u_m));
   ```
2. Now the part of the unification of the given fact with the IDB body literal, which can only be done at runtime, must be generated. Let \( V_1, \ldots, V_k \) be all \( C++ \) variables with \( V_i \theta \neq V_i \). If \( k > 0 \), generate:
   ```
   \text{if}(V_1 \neq V_1 \theta | | \cdots | | V_k \neq V_k \theta)
   \text{break;}
   ```
   This ends the execution of this code piece if the rule is not applicable.
3. Now we must use a cursor to access the tuples for the EDB body literal \( r(v_1, \ldots, v_l) \). In case this rule is recursive, it might be possible that the state of the cursor and the values of the variables \( v_{\rho_\beta}X \) set in this rule are still needed by backtrack points on the stack (unless we know that there are no such backtrack points, e.g. because the recursive rule application is the last use of the fact). Therefore, we generate
   ```
   \text{c\rho.push();}
   ```
   And for each variable \( v_{\rho_\beta}X \) we generate
   ```
   \text{value_stack.push}(v_{\rho_\beta}X);
   ```
   (There are probably several value stacks for different data types.) Finally we generate a backtrack point:
   ```
   \text{backtrack_stack.push}(L\_RESTORE_\rho);
   ```
   The code for this `case` in the `switch` simply restores the variable values and the cursor state by popping them (in the inverse order). In this way, all earlier backtrack points (below the one just generated) find the old cursor state and variable values.
4. Let \( i_1, \ldots, i_k \) be the bound argument positions in \( \beta \). Generate:
   ```
   \text{c\rho.open}(\bar{v}_{i_1}, \ldots, \bar{v}_{i_k});
   ```
   where \( \bar{v}_{ij} \) is
   - \( \bar{v}_{ij} \) if this is a constant,
   - \( \bar{v}_{ij} \theta \) if \( \bar{v}_{ij} \) is a variable which appears in the IDB body literal \( q(\ldots) \).
5. Generate:
   
   ```
   case L_CONT_ρ:
   
   When we are finished with using the computed fact, backtracking returns here to continue the following loop.
   ```
   
6. Generate
   ```
   while(cp.fetch()) {
   ```
7. Let \( i_1, \ldots, i_k \) be the free argument positions in \( \beta \) such that \( v_{i_j} \) is a constant or a variable which appears the the IDB body literal \( q(...) \). Let \( \bar{v}_{i_j} \) be defined as in 4 above. If \( k \geq 1 \), generate
   ```
   if((cp.col_{i_1}() != \bar{v}_{i_1} || \cdots || cp.col_{i_k}() != \bar{v}_{i_k})
   ```
   ```
   continue;
   ```
   I.e. if the current tuple in the EDB relation does not have the required values, we continue with the next iteration of the while-loop under 6.
8. For every variable \( Y \), which appears more than once among the \( u_1, \ldots, u_m \), but not in the IDB literal \( q(...) \): Let \( u_{i_1}, \ldots, u_{i_k} \) be all equal to \( Y \) (note that \( k \geq 2 \)). Generate:
   ```
   if((cp.col_{i_1}() != cp.col_{i_2} || \cdots ||
   ```
   ```
   cp.col_{i_{k-1}}() != cp.col_{i_k}())
   ```
   ```
   continue;
   ```
9. For every variable \( X_i \) in the head, which does not appear in the IDB body literal \( q(...) \), let \( u_j \) be any occurrence of this variable among the \( u_1, \ldots, u_m \). Because of the range restriction (allowedness) condition on the rules, it must occur there. Generate for each \( X_i \):
   ```
   v_ρ_{X_i} = cp.col_{j}();
   ```
10. In case predicate \( p \) was selected for a duplicate check, we again enter the result tuple \( p(\bar{t}_1, \ldots, \bar{t}_n) \) with the current values of the C++ variables into a hash table. If the tuple was already present, one simply does “continue;” to compute the next tuple.
11. Generate:
   ```
   backtrack_stack.push(L_CONT_ρ);
   ```
12. Let \( ρ_1, \ldots, ρ_k \) be all rules with an IDB body literal \( B_{i_1}, i := 1, \ldots, k \), which matches the generated symbolic fact \( p(\bar{t}_1, \ldots, \bar{t}_n) \). For \( i := 2, \ldots, k \), generate
   ```
   backtrack_stack.push(L_START(p(\bar{t}_1, \ldots, \bar{t}_n), ρ_i, B_i));
   ```
   Then generate
   ```
   goto l_start(p(\bar{t}_1, \ldots, \bar{t}_n), ρ_1, B_1);
   ```
13. Finally, we must close the open while-loop (6. above) and finish the code piece in case the loop does not find any (further) matching tuple. Generate:
   ```
   }
   ```
   ```
   break;
   ```

In addition, there is a code piece for case L_RESTORE_ρ as explained under item 3 above. It pops everything pushed there.
6.6 Two IDB-Body Literals

This is a complicated case and needs intermediate storage of tuples generated for the body literals. Fortunately, this occurs rarely in the output of the SLDMagic method (only when translating recursions that are not tail recursions).

A general solution, which does not need information about the order of generated tuples, is to have one set of tuples for each body literal. In the code piece for the case that a new tuple has been derived for the left body literal, this tuple is joined with all tuples in the current set for the right body literal. In the same way, when a new tuple is generated for the right body literal, it is joined with all existing tuples for the left body literal.

This means that we now need cursors also for the intermediate storage of generated IDB facts, and these cursors must keep information about the last fact when they were created (since new facts can be appended to the list while the cursor is active—these facts must not be returned by the cursor).

Recursion can be handled in the same way as before: When a new fact is generated for a body literal, we save the state of the cursor and all variables of that code piece, continue with derivations using the new fact, and later return to the old fact. However, since we anyway have intermediate storage now, it is also possible to create a queue of facts for each body literal, which must still be used in derivations.

If it is possible to generate all facts for the left body literal before the first fact for the right body literal, one obviously needs intermediate storage only for the left body literal. In this case it can later be treated like an EDB literal.

If one can generate facts for both literals in the sort order of the common variables (the join attributes), we would need intermediate storage only for a single tuple for each body literal (we would basically do a merge join).

6.7 Generated answer-Facts

For rules about the predicate answer, one can print the generated tuple or insert it into a result relation whenever the above code would jump to a body literal which uses the generated fact (there are no body literals with predicate answer). One can also offer the cursor interface of Section 3.

7 Conclusion

We have presented a detailed description of the push method, an efficient bottom-up evaluation algorithm for pure Datalog programs. In the push method, derived facts are immediately used to derive new facts without generally materializing immediate results. A specific feature is the representation of derived tuples which significantly reduces the amount of copying. The rule application graph introduced here is useful for planning the evaluation. First performance tests show some improvement over the previous version of the push method from [2]. We plan to develop a more complete implementation and to investigate further optimizations. The current state of the project will be reported at http://www.informatik.uni-halle.de/~brass/botup/
References


A practical view on renaming

Marija Kulaš
FernUniversität Hagen, Praktische Informatik VIII, 58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract. For logic program analysis or formal semantics, the issue of renaming terms and generally handling substitutions is inevitable. In logic programming literature, there is a common sentiment against substitutions, grounded basically on the fact that by composing substitutions, properties like idempotence, equivalence, or restriction are not always preserved. Also, some equivalences are felt to be counter-intuitive. For an own work, we needed another missing property, extensibility: starting from a pair of variant queries, we move along their respective derivations and need to be able to add new variables at each step, thus incrementally constructing the variance. Since traditional renamings are not extensible, in this paper we introduce a slight generalization of renaming, which solves the problem of extensibility. Building on this concept, a propagation claim for logic programming systems has been proved. As a corollary, a variant lemma for Prolog is obtained. Underway, we revise an embedding lemma and touch on the discrepancy between the rather abundant theory of logic programming and a scarcity of mathematical claims for implemented logic programming systems.

1 Introduction

For anyone embarking on a journey of logic program analysis or formal semantics, sooner or later the issue of renaming terms and generally handling substitutions becomes inevitable. In logic programming literature, there is a common sentiment against substitutions, as being "very tricky" ([She94]) or "quite hard matter to deal with" ([Pal90]). The sentiment is grounded basically on two facts:

– by composing substitutions, properties like idempotence, equivalence, or restriction are not preserved
– due to the group structure of renamings, permuting any number of variables amounts to "doing nothing", as in \( \left( x \ y \right) \sim \varepsilon \); such equivalences are felt to be counter-intuitive.

To these complaints another one can be added. In [Kul05] we introduced an operational semantics for pure Prolog, S1:PP, which needs just two linear lists to represent the state of computation. To prove its correctness and completeness, we needed a particular property of renamings, let us call it extensibility: starting from a pair of variant queries, we move along their respective formal derivations...
and observe new variables being added at each step, maintaining the status of being variant. This setup is known from the classical variant lemma. The novelty here is that we wish to collect the variables, obtaining at each step the temporary variance between the derivations.

Renamings cannot be used for this task, being non-extensible: Assume the first query is \( p(z,u) \) and the second \( p(y,z) \). The variance might be \( \rho = (z \ u \ y \ z \ u) \) and if we want a relevant renaming, in this example there is only one. Now assume in the next step the first derivation acquires the variable \( y \), and the second \( x \). The relevant renaming this time would be \( \rho' = (z \ u \ y \ z \ u \ x) \). Clearly, \( \rho' \) is not an extension of \( \rho \), as can be seen from \( y \), which makes it unsafe to proceed: are the properties of the previous step now in danger?

In this paper we introduce a slight generalization of renaming, called prenaming, which solves the above problem of extensibility (Theorem 20). As a bonus, it is a mathematical underpinning of the intuitive practice of “renaming” terms by just considering the necessary bindings, and not worrying whether the result is a permutation. In the above example that would be \( z \mapsto y, u \mapsto z \).

Prenamings relate to previous work as follows: A safe prenaming is more general than renaming for a term from [Apt97], and it maximizes \( W \) in the notion of \( W \)-renaming from [Ede85] (pages 7–8).

Application on a nontrivial example is shown in Lemma 31, where a propagation claim for logic programming systems has been proved, in a constructive way. This was enabled by substitution prenaming, which is compositional (for constant prenaming). Substitution prenaming generalizes substitution renaming from [AS09]. As a corollary, a variant lemma (Lemma 33) for Prolog is obtained.

Underway, we provide some auxiliary concepts and results, revise an embedding lemma (Theorem 12) and touch on the discrepancy between the rather abundant theory of logic programming and a scarcity of mathematical claims for implemented logic programming systems.

A final remark, concerning the original motivation above: propagation claims for our formal semantics have been proved in a similar way as in section 5, enabling a completeness proof (forthcoming publication).

## 2 Substitution

First we need a bit of notation. Assume an infinite set of variables \( V \). If \( W \subseteq V \), any mapping \( F \) with \( F(W) \subseteq V \) shall be called variable-pure on \( W \). A mapping variable-pure on the whole set of variables \( V \) shall be called all-vars mapping. If \( V \setminus W \) is finite, \( W \) is said to be co-finite A mapping \( F \) is injective on \( W \), if whenever \( F(x) = F(y) \) for \( x,y \in W \) we have \( x = y \).

A recurrent theme in this paper shall be relevance, meaning "no extraneous variables" (relative to some term). For example, we say a mapping \( F \) is relevant for \( t_1 \) to \( t_2 \), if \( \text{Dom}(F) \subseteq \text{Vars}(t_1) \) and \( \text{Range}(F) \subseteq \text{Vars}(t_2) \).

If the terms \( s \) and \( t \) share a variable, that shall be written \( s \bowtie t \). Otherwise, we say \( s, t \) are variable-disjunct, written as \( s \not\bowtie t \). If \( s \) is a subterm of \( t \), this shall be written as \( s \in t \).
Definition 1 (substitution). A substitution $\theta$ is a function mapping variables to terms, which is identity almost everywhere. In other words, a function $\theta$ with domain $\text{Dom}(\theta) = V$ such that the following requirement holds:

finite action$^1$ The set $\{x \in V \mid \theta(x) \neq x\}$ is finite.

The set $\text{Core}(\theta) := \{x \in V \mid \theta(x) \neq x\}$ shall be called the active domain$^2$ or core of $\theta$, and its elements active variables$^3$ of $\theta$. The set $\theta(\text{Core}(\theta))$ is called the active range of $\theta$. For completeness, a variable $x$ such that $\theta(x) = x$ shall be called a passive variable, or a fixpoint, for $\theta$. Also, we say that $\theta$ is active on the variables from $\text{Core}(\theta)$, and passive on all the other variables.

If $\text{Core}(\theta) = \{x_1, ..., x_k\}$, where $x_1, ..., x_k$ are pairwise distinct variables, and $\theta$ maps each $x_i$ to $t_i$, then $\theta$ shall have the following core representation: $\{x_1/t_1, ..., x_k/t_k\}$, or the perhaps more visual $\left(\frac{x_1}{t_1}, ..., \frac{x_k}{t_k}\right)$. Hence, the above requirement shall also be called finite core. Each pair $x_i/t_i$ is called the binding for $x_i$ via $\theta$, denoted by $x_i/t_i \in \theta$.

Often we identify a substitution with its core representation, and thus regard it as a syntactical object, a term. So the set of all variables of a substitution is defined as $\text{Vars}(\theta) := \text{Core}(\theta) \cup \text{Vars}(\theta(\text{Core}(\theta)))$.

The notions of restriction and extension of a mapping shall also be transported to core representation: the core of a restriction $\theta$ is a subset of the core of its extension $\sigma$, for simplicity we write $\theta \subseteq \sigma$.

The substitution $\theta$ is extended to terms in a structure-preserving way by $\theta(f(t_1, ..., t_n)) := f(\theta(t_1), ..., \theta(t_n))$. If $s$ is a term, then $\theta(s)$ is an instance of $s$ via $\theta$. If $M$ is a set of terms, we define $\theta(M) := \{\theta(t) \mid t \in M\}$.

The composition $\theta \circ \sigma$ of substitutions $\theta$ and $\sigma$ is defined by $(\theta \circ \sigma)(t) := \theta(\sigma(t))$. Composition may be iterated, written as $\sigma^n := \sigma \circ \sigma^{n-1}$ for $n \geq 1$, and $\sigma^0 := \varepsilon$. Here $\varepsilon := ()$ is the identity function on $V$. In case an all-vars substitution $\rho$ is bijective, its inverse shall be denoted as $\rho^{-1}$. A substitution $\theta$ satisfying the equality $\theta \circ \theta = \theta$ is called idempotent.

Example 2. $\left(\frac{x w u v}{u v x w}\right) \cdot \left(\frac{u v x y z w}{x w y u v z}\right) = \left(\frac{y y' x y z w w' y y' y z w}{y y' y x z w w' y y' y z w}\right) = \left(\frac{x y z w}{y x z w}\right)$.

3 Renaming

Definition 3 (renaming). A renaming of variables is a bijective all-vars substitution.

$^1$ [Gal86] uses the name finite support.

$^2$ Literature traditionally uses the name domain. However, in the usual mathematical sense it is always the whole $V$ which is the domain of any substitution. It may be less confusing to have both the domain, which is uniformly $V$, and the core or active domain, making it clear that, while every variable can be mapped, only active variables are of interest.

$^3$ The name active variable appears in [JL92].

$^4$ We do not omit braces. The reason is to prevent ambiguity between a composition of substitutions, and an application of a substitution on a substitution (subsection 4.4).
In [Ede85], it is synonymously called "permutation". We shall reserve the word for the general case where infinite movements like translation are possible. Here we shall synonymously speak of finite permutation due to the fact that, being a substitution, any renaming has a finite core.

From the definition of substitution, we know: if \( s \in t \), then \( \sigma(s) \in \sigma(t) \). For bijective substitutions (i.e. renamings), a complementary property holds as well:

**Corollary 4 (renaming stability of ",", ":", "\( \neq \")\).** Let \( \rho \) be a renaming and \( s,t \) be terms. Then \( s = t \) \iff \( \rho(s) = \rho(t) \), and also \( s \in t \) \iff \( \rho(s) \in \rho(t) \). As a consequence, \( s \neq t \) \iff \( \rho(s) \neq \rho(t) \).

**Lemma 5 (fixpoint).** A substitution \( \rho \) is a renaming \iff \( \rho(\text{Core}(\rho)) = \text{Core}(\rho) \).

**Proof.** Let \( \rho \) be a renaming. If there is \( x \in \rho(\text{Core}(\rho)) \setminus \text{Core}(\rho) \), then for some \( y \in \text{Core}(\rho) \), \( \rho(y) = x \), and also \( \rho(x) = x \), although clearly \( x \neq y \), hence \( \rho \) wouldn’t be injective. If there is \( x \in \text{Core}(\rho) \setminus \rho(\text{Core}(\rho)) \), then \( \rho(V) \neq V \), i.e. \( \rho \) wouldn’t be surjective.

For the other direction, the finiteness of \( \text{Core}(\rho) \) means that \( \rho \) has to be bijective on \( \text{Core}(\rho) \). On the same set it is variable-pure. Outside of \( \text{Core}(\rho) \), it is identity. Hence, \( \rho \) is a bijection on \( V \), and all-vars.

**Lemma 6 ([Ede85]).** Every injective all-vars substitution is a renaming.

We retell the key observation of Eder in proving this property. If \( x \in \text{Core}(\rho) \), then \( \rho(x) \in \text{Core}(\rho) \). Otherwise \( \rho(\rho(x)) = \rho(x) \), so due to injectivity \( \rho(x) = x \). By injectivity and finiteness, that means \( \text{Core}(\rho) = \rho(\text{Core}(\rho)) \), so by Lemma 5 the claim holds.

So composition of renamings is a renaming. The next property is about cycle decomposition of a finite permutation.

**Lemma 7 (cycles).** Let \( \sigma \) be an all-vars substitution. It is injective \iff for every \( x \in V \), there is \( n \in N \) such that \( \sigma^n(x) = x \).

**Proof.** Assume \( \sigma \) injective, and choose \( x_0 \in V \). If \( \sigma(x_0) = x_0 \), we are done. Otherwise, \( \sigma^i(x_0) \neq \sigma^{i-1}(x_0) \) for all \( i \geq 1 \), due to injectivity. Hence, \( \sigma^{i-1}(x_0) \in \text{Core}(\sigma) \) for every \( i \geq 1 \). Because of the finiteness of \( \text{Core}(\sigma) \), there is \( m \geq k \geq 1 \) such that \( \sigma^k(x_0) = \sigma^k(x_0) \). Due to injectivity, \( \sigma^{m-1}(x_0) = \sigma^{k-1}(x_0) \). By iteration we get \( n := m - k \).

For the other direction, assume \( \sigma(x) = \sigma(y) \), and minimal \( m,n \) such that \( \sigma^m(x) = x \), \( \sigma^m(y) = y \). Consider the case \( m \neq n \), say \( m > n \). Then \( \sigma^{m-n}(y) = \sigma^m(x) = \sigma^{m-n}(\sigma^n(x)) = \sigma^{m-n}(\sigma^n(y)) = \sigma^m(y) = y \), contradicting minimality of \( m \). Hence \( m = n \), and so \( x = \sigma^n(x) = \sigma^n(y) = y \).

### 4 Prenaming

In practice, one would like to change the variables in a term, without bothering to check whether this change is a permutation or not. For example, the term
\(p(z, u, x)\) can be mapped on \(p(y, z, x)\) via \(z \mapsto y, u \mapsto z, x \mapsto x\). Let us call such a mapping \textit{prenaming}\(^5\). Obviously, injectivity is important for such a mapping, since \(p(z, u, x)\) cannot be mapped on \(p(y, y, x)\) without losing a variable. Hence,

\textbf{Definition 8 (prenaming).} A \textit{prenaming} \(\alpha\) is an all-vars substitution injective on a finite set of variables \(C(\alpha) \supseteq \text{Core}(\alpha)\).

The associated range \(\alpha(C(\alpha))\) we denote as \(R(\alpha)\). As a relaxation of "activeness" of a substitution, we say that \(\alpha\) is complete for \(t\) if \(\text{Vars}(t) \subseteq C(\alpha)\).

Motivation: in Lemma 31, we shall have to deal with mappings of variables between two terms. There, it is possible that a variable stays the same, so \((x, x)\) would have to be tolerated as a "binding", since we need our mapping to cover all variables in the two terms. Therefore, we allow the set \(C\) to contain some passive variables, raising those above the rest, as it were.

Like any substitution, a prenaming shall also be represented finitely, akin to the core representation, but now we shall use \(C\) instead of \(\text{Core}\), in order to allow \(x/x\).

\textbf{Definition 9 (relaxed core representation).} If \(C(\alpha) = \{x_1, ..., x_n\}\) and \(\alpha(C(\alpha)) = \{y_1, ..., y_n\}\), where \(x_1, ..., x_n\) as well as \(y_1, ..., y_n\) are pairwise distinct, then \((x_1 \ldots x_n, y_1 \ldots y_n)\) shall be called \textit{relaxed core representation} for \(\alpha\).

The set of variables of \(\alpha\) is \(V(\alpha) := \{x_1, ..., x_n, y_1, ..., y_n\}\).

Clearly, any renaming \(\rho\) is a prenaming, where \(C(\rho)\) can be any finite superset of \(\text{Core}(\rho)\).

4.1 Extension

For Lemma 33, we need a possibility to increment a given prenaming by new bindings.

\textbf{Corollary 10 (extending a prenaming).} Let \(\alpha = (x_1 \ldots x_n, y_1 \ldots y_n)\) and \(\beta = (u_1 \ldots u_k, v_1 \ldots v_k)\) be prenaming such that \(\{u_1, ..., u_k\} \not\sqsubset \{x_1, ..., x_n\}\) and \(\{v_1, ..., v_k\} \not\sqsubset \{y_1, ..., y_n\}\).

Then the disjunct union \(\alpha \uplus \beta = (x_1 \ldots x_n, u_1 \ldots u_k, y_1 \ldots y_n, v_1 \ldots v_k)\) is also a prenaming. Conversely, if a prenaming \(\alpha'\) is an extension of \(\alpha\), then \(\alpha'\) can be split as above.

\textbf{Definition 11 (sum of prenaming).} The prenaming \(\alpha \uplus \beta\) from the previous claim shall be called the \textit{sum of prenamings} \(\alpha\) and \(\beta\).

Clearly, \(C(\alpha \uplus \beta) = C(\alpha) \cup C(\beta)\) and \(R(\alpha \uplus \beta) = R(\alpha) \cup R(\beta)\).

\(^5\) Finding an appropriate name can be a struggle. Shortlisted were \textit{pre-renaming} and \textit{proto-renaming}.
4.2 The question of inverse

In practice, a prenaming is more natural, but a "full" renaming is better mathematically tractable (inverse exists). Hence we want to know whether each prenaming can be embedded in a renaming.

The next property shows how to extend a prenaming \( \alpha \) to obtain a renaming, and a relevant one at that, i.e. acting only on the variables from \( V(\alpha) \). The claim is essentially given in [LS91], [Apt97] and [AS09] with the emphasis on the existence\(^6\) of such an extension. In [Ede85], the emphasis is on the actual reach\(^7\) of the extension. The latter is our concern as well. We formulate the claim around the notion of prenaming, and provide a constructive proof based on Lemma 7.

**Theorem 12 (embedding).** Let \( \alpha \) be a prenaming. Then there is a renaming \( \rho \) which coincides with \( \alpha \) on \( \text{InDom}(\alpha) \), such that \( \text{Vars}(\rho) \subseteq V(\alpha) \).

Additionally, if \( \alpha(x) \neq x \) on \( C(\alpha) \), then \( \rho(x) \neq x \) on \( V(\alpha) \).

**Proof.** If \( \alpha \) is a prenaming, then \( C(\alpha) = D \) and \( \alpha(C(\alpha)) = R \) are sets of \( n \) distinct variables each. We shall construct the wanted renaming in Algorithm 4.1, where it is named \( \overline{\alpha} \). Let us see if for every \( x \) there is a \( j \) such that \( \overline{\alpha}^j(x) = x \).

\[
\overline{\alpha}(x) := \begin{cases} 
\alpha(x), & \text{if } x \in D \\
z, & \text{if } x \in R \setminus D \text{ and } \alpha^m(z) = x \text{ for maximal } m \leq n \\
x, & \text{outside of } D \cup R
\end{cases}
\]

Algorithm 4.1: Closure: the natural relevant embedding

If \( x \in D \), we start as in the proof of Lemma 7, and consider the sequence \( x, \alpha(x), \alpha^2(x), \ldots \). Since \( D \) is finite, either we get two equals (and proceed as there), or we get \( \alpha^k(x) \notin D \) and are stuck. For \( y := \alpha^k(x) \) we know \( \overline{\alpha}(y) = z \) such that \( \alpha^m(z) = y \) with maximal \( m \), so \( m \geq k \). Therefore, \( \alpha^m(\overline{\alpha}(y)) = y = \alpha^k(x) \). Due to injectivity of \( \alpha \) on \( C(\alpha) \) we get \( \alpha^{m-k}(\overline{\alpha}(\alpha^k(x))) = x \), and hence \( \overline{\alpha}^{m+1}(x) = x \).

The cases \( x \in R \setminus D \) or \( x \notin D \cup R \) are easy. By Lemma 7, \( \overline{\alpha} \) is injective. By Lemma 6, \( \overline{\alpha} \) is a renaming. The discussion of the case \( \alpha(x) \neq x \) on \( C(\alpha) \) is straightforward. \( \square \)

**Definition 13 (closure of a prenaming).** The renaming \( \overline{\alpha} \) constructed in Algorithm 4.1 shall be called the closure of the prenaming \( \alpha \).

---

\(^6\) [Apt97], p. 23: "Every finite 1-1 mapping \( f \) from \( A \) onto \( B \) can be extended to a permutation \( g \) of \( A \cup B \). Moreover, if \( f \) has no fixpoints, then it can be extended to a \( g \) with no fixpoints."

\(^7\) [Ede85], p. 35: "Let \( W \) be a co-finite set of variables (…) and let \( \sigma \) be a \( W \)-renaming. Then there is a permutation \( \pi \) which coincides with \( \sigma \) on the set \( W \)."
Remark 14 (relevant embedding is not unique). Given prenaming $\alpha = (z \ u \ y \ w_1, y \ z \ u \ w_2)$, let us embed it in a relevant renaming. The Algorithm 4.1 gives $\pi = (z \ u \ y \ w_1 \ y \ z \ w_2, u \ w_2)$. But $\rho = (z \ u \ y \ w_1 \ x \ w_2, y \ z \ x \ w_2, u \ w_1)$ is also a relevant renaming which is embedding $\alpha$.

In the usual notation for cycle decomposition, $\rho = \{(x, w_1, w_2, u, z, y)\}$ and $\pi = \{(x, u, z, y), (w_1, w_2)\}$.

The problem with embedding a prenaming is that there would be no monotonicity: if $\alpha' \subseteq \alpha$, then not always $\overline{\alpha'} \subseteq \overline{\alpha}$. Example: if $\alpha' = (z \ u)$ and $\alpha = (z \ u \ y \ z \ x)$, then $\overline{\alpha'} = (z \ u \ y)$ and $\overline{\alpha} = (z \ u \ y \ z \ x \ u)$.

4.3 Step back: injectivity

Let us look more closely into the problem. In the above example $\alpha(y) = x$ and $\alpha(x) = x$, so $y$ and $x$ may not simultaneously occur in the candidate term. Otherwise, a variable shall be lost, which we call aliasing, like in $(y) (p(x, f(y))) = p(x, f(x))$.

Definition 15 (aliasing). Let $\alpha$ be a prenaming. If $x \neq y$ but $\alpha(x) = \alpha(y)$, then we say $\alpha$ is aliasing $x$ and $y$.

The problem here comes down to: if we want to use $\alpha$ on a larger set than $C(\alpha)$, then the set $\alpha(C(\alpha)) \setminus C(\alpha)$ is compromising injectivity. But, luckily, its complement is not:

Lemma 16 (larger set). Let $\bar{\alpha} = \alpha(C(\alpha)) \setminus C(\alpha)$. A prenaming $\alpha$ is injective on the co-finite set $V \setminus \bar{\alpha}$. The set is maximal.

Proof. Let $x, y \in V \setminus \bar{\alpha}$. Is it possible that $\alpha(x) = \alpha(y)$?

Possible cases: If $x, y \in C(\alpha)$, then by definition of prenaming $\alpha(x) \neq \alpha(y)$. If $x, y \notin C(\alpha)$, then $\alpha(x) = x \neq y = \alpha(y)$. It remains to consider the mixed case $x \in C(\alpha), y \notin C(\alpha)$. We have $\alpha(x) \in \alpha(C(\alpha))$ and $\alpha(y) = y$. So is $\alpha(x) = y$ possible? If yes, then $y \in \alpha(C(\alpha))$, but since $y \notin C(\alpha)$, that would mean $y \in \bar{\alpha}$.

Contradiction.

The set cannot be made larger: if $y \in \bar{\alpha}$, then there is $x$ with $x \neq y$ and $\alpha(x) = y = \alpha(y)$, so injectivity is compromised. $\square$

Definition 17 (injectivity domain). For a prenaming $\alpha$, let $\text{InDom}(\alpha) := V \setminus \bar{\alpha}$. Since $\text{InDom}(\alpha)$ is the largest co-finite set containing $C(\alpha)$ on which $\alpha$ is injective, it shall be called the injectivity domain of $\alpha$.

The injectivity domain of a prenaming is clearly the only safe place for it to be mapping terms from. Hence,

Definition 18 (safety of prenaming). A prenaming safe\(^8\) for a term $t$ is a prenaming $\alpha$ with $\text{Vars}(t) \subseteq \text{InDom}(\alpha)$.

\(^8\)Our definition of safe prenaming is more general than the definition of renaming for a term in [Apt97], p. 24, since we do not require $\text{Core}(\alpha) \subseteq \text{Vars}(t)$. 

42
Clearly, $\text{InDom}(\alpha) = C(\alpha) \cup (V \setminus \alpha(C(\alpha)))$, so any extended core is safe.

For a prenaming $\alpha$ with the quality $\alpha(C(\alpha)) = C(\alpha)$, i.e. a renaming, it is no surprise that $\text{InDom}(\alpha) = V$ and safety guaranteed.

**Lemma 19 (monotonicity).** Assume $\alpha \uplus \beta$ is defined. Then

1. $\text{InDom}(\alpha) \cup \text{InDom}(\beta) = V$
2. $\text{InDom}(\alpha) \cap \text{InDom}(\beta) \subseteq \text{InDom}(\alpha \uplus \beta)$

**Proof.** Since $(V \setminus A) \cup (V \setminus B) = V \setminus (A \cap B)$, and $\text{Pit}(\alpha) \not\subseteq \text{Pit}(\beta)$, we obtain $\text{InDom}(\alpha) \cup \text{InDom}(\beta) = V$.

Further, $(V \setminus A) \cap (V \setminus B) = V \setminus (A \cup B)$ and so $\text{Pit}(\alpha \uplus \beta) = (R(\alpha) \uplus R(\beta)) \setminus (C(\alpha) \uplus C(\beta)) \subseteq (R(\alpha) \setminus C(\alpha)) \cup (R(\beta) \setminus C(\beta)) = \text{Pit}(\alpha) \cup \text{Pit}(\beta)$. □

In the example above, $\text{Pit}(\alpha') = \{y\}$, $\text{Pit}(\{y\}) = \{x\}$, and $\text{Pit}(\alpha) = \{x\}$, hence $\text{InDom}(\alpha') = V \setminus \{y\}$, $\text{InDom}(\{y\}) = V \setminus \{x\}$ and $\text{InDom}(\alpha) = V \setminus \{x\}$.

By the last claim, staying within $\text{InDom}(\alpha)$ and $\text{InDom}(\beta)$ ensures staying within $\text{InDom}(\alpha \uplus \beta)$, which is a kind of monotonicity useful for working with prenamings.

By assuming a bit more about $\alpha$ than just safety, we may ignore the nature of extension $\beta$, and still ensure a kind of backward compatibility of the extension $\alpha \uplus \beta$. This shall be used in section 5.

Observe the importance of relaxed core for this to work: otherwise, passive bindings $x/x$ would not be accounted for.

**Theorem 20 (extensibility).** Assume $\alpha \uplus \beta$ is defined.

1. If $\alpha$ is safe for $t$ and $\beta$ is safe for $t$, then $\alpha \uplus \beta$ is safe for $t$.
2. If $\alpha$ is complete for $t$, then $\alpha \uplus \beta$ is safe for $t$ and $(\alpha \uplus \beta)(t) = \alpha(t)$.

The first part follows from Lemma 19 and the second from $\text{Vars}(t) \subseteq C(\alpha)$.

A prenaming behaves like a renaming on its injectivity domain, since it coincides with its closure there. This follows immediately from Theorem 12:

**Theorem 21 (injectivity domain).** Let $x \in \text{InDom}(\alpha)$. Then

$$\alpha(x) = \overline{\alpha}(x).$$

**Corollary 22 (prenaming stability).** A generalization of Corollary 4 holds:

Let $s, t$ be terms and $\alpha$ be a prenaming safe for $s, t$. Then $s = t$ iff $\alpha(s) = \alpha(t)$ and also $s \in t$ iff $\alpha(s) \in \alpha(t)$. As a consequence, $s \not\in t$ iff $\alpha(s) \not\in \alpha(t)$.

Our definition of prenaming was inspired by the following more general notion from [Ede85].

**Definition 23 (W-renaming).** Let $W \subseteq V$. A substitution $\sigma$ is a $W$-renaming if $\sigma$ is variable-pure on $W$, and $\sigma$ is injective on $W$.

With this notion, Lemma 16 can be summarized as: $\text{InDom}(\alpha)$ is a co-finite set of variables, and the largest set $W$ such that $\alpha$ is a $W$-renaming.
4.4 Variant of term and substitution

The traditional notion of term variance, which is term renaming, shall be generalized to prenaming. As a special case, substitution variance is defined, inspired by substitution renaming from [AS09]. For this, substitution shall be regarded as a special case of term. The term is of course the extended core representation. This concept shall come in handy for proving properties of renamed derivations, as in subsection 5.2.

Definition 24 (term variant). If \( \rho \) is a renaming, we call \( \rho(t) \) a variant of \( t \), denoted by \( \rho \circ t \sim t \). In case of prenaming, there is an added requirement of safety: if \( \rho \) is a prenaming safe for \( t \), then we also call \( \rho(t) \) a variant of \( t \), and write \( \rho \circ t \sim t \).

Sometimes the particular variance and the direction of its application shall be explicated: \( s \sim \rho \circ t \iff s = \rho(t) \).

If \( s \sim \rho \circ t \), then there is a unique \( \alpha \) mapping \( s \) to \( t \) in a relevant and complete manner, i.e. mapping each variable pair and nothing else, as computed by Algorithm 4.2. This prenaming shall be simply called the prenaming of \( s \) to \( t \), and denoted \( \text{Pren}(s, t) \).

Start from the set \( E := \{ s = t \} \) and transform according to the following rules. The transformation is bound to stop. If the stop was not due to failure, then the current set \( E \) is the prenaming of \( s \) to \( t \), \( E = \text{Pren}(s, t) \).

**elimination** \( E \cup \{ x = y \} \leadsto E \), if \( x/y \in E \)

**failure: alias** \( E \cup \{ x = y \} \leadsto \text{failure} \), if \( (x/z \in E, z \neq y) \) or \( (z/y \in E, z \neq x) \)

**binding** \( E \cup \{ x = y \} \leadsto E \cup \{ x/y \} \), if \( (x/\_ \notin E) \) and \( (\_/y \notin E) \)

**failure: instance** \( E \cup \{ x = t \} \leadsto \text{failure} \), if \( t \notin \mathbf{V} \);

\( E \cup \{ t = x \} \leadsto \text{failure} \), if \( t \notin \mathbf{V} \)

**decomposition** \( E \cup \{ f(s_1, \ldots, s_n) = f(t_1, \ldots, t_m) \} \leadsto E \cup \{ s_1 = t_1, \ldots, s_n = t_n \} \)

**failure: clash** \( E \cup \{ f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m) \} \leadsto \text{failure} \), if \( f \neq g \) or \( m \neq n \)

Algorithm 4.2: Computing the prenaming of \( s \) to \( t \)

The algorithm makes do with only one set for equations and bindings, thanks to different types. Termination can be seen from the tuple \((\text{lfun}_E(E), \text{card}_E(E))\) decreasing in lexicographic order with each rule application, where \(\text{lfun}_E(E)\) is the number of function symbols in equations in \( E \), and \(\text{card}_E(E)\) is the number of equations in \( E \).

Even substitutions themselves can be renamed. To rename a substitution, one regards it as a syntactical object, a set of bindings, and renames those bindings.

\(^9\) for prenaming, we naturally use \( C \) for \( \text{Dom} \) and \( R \) for \( \text{Range} \).
If $\rho$ is a renaming and $\sigma$ is a substitution, [AS09] define substitution renaming by $\rho(\sigma) := \{\rho(x)/\rho(\sigma(x)) \mid x \in \text{Core}(\sigma)\}$. It is easy to see that $\rho(\sigma)$ is a substitution in core representation. For this we only need two properties of $\rho$: variable-pure on $\text{Vars}(\sigma)$ and injective on $\text{Vars}(\sigma)$. These requirements are clearly fulfilled by prenaminings safe on $\sigma$ as well. Hence,

**Definition 25 (substitution variant).** Let $\sigma$ be a substitution and let $\alpha$ be a prenaming safe for $\sigma$, i.e. $\text{Vars}(\sigma) \subseteq \text{InDom}(\alpha)$. Then a variant of $\sigma$ by $\alpha$ is

$$\alpha(\sigma) := \{\alpha(x)/\alpha(\sigma(x)) \mid x \in \text{Core}(\sigma)\} \quad (1)$$

As in the case of ”real renaming”, the concept of variance by prenaming is well-defined.

**Lemma 26.** Substitution variant is well-defined, i.e. (1) is a core representation of a substitution, and $\alpha$ does not introduce aliasing.

**Proof.** Let $\text{Core}(\sigma) = \{x_1, ..., x_n\}$; if $\alpha(x_i) = \alpha(x_j)$, then $x_i = x_j$. Next, by Corollary 22, if $\alpha(\sigma(x_i)) \bowtie \alpha(\sigma(x_j))$, then $\sigma(x_i) \bowtie \sigma(x_j)$, meaning that $\alpha$ does not introduce aliasing.

To prove (1) a core representation, observe $x \in \text{Core}(\sigma)$ iff $x \neq \sigma(x)$ iff $\alpha(x) \neq \alpha(\sigma(x))$, due to injectivity of $\alpha$. Therefore, (1) is well-defined. \(\Box\)

By Theorem 21, $\overline{\sigma}(\alpha) = \alpha(\sigma)$. If $(\alpha \cdot \beta)(\sigma)$ is defined, then $(\alpha \cdot \beta)(\sigma) = \alpha(\beta(\sigma))$.

For the case of ”full” renaming, an interesting property is shown in [AS09]:

$$\rho(\sigma) = \rho \cdot \sigma \cdot \rho^{-1} \quad (2)$$

Would such a claim hold for our weakened case, prenaminings?

**Theorem 27 (substitution variant).** Let $\sigma$ be a substitution and $\alpha$ be a prenaming safe for $\sigma$. Then

$$\alpha(\sigma) \cdot \alpha = \alpha \cdot \sigma$$

which further gives

$$\alpha(\sigma) = \overline{\alpha} \cdot \sigma \cdot \overline{\alpha}^{-1}$$

**Proof.** According to Definition 25, for every $x \in V$ holds $(\alpha(\sigma) \cdot \alpha)(x) = \alpha(\sigma(x))$. Since any substitution is structure-preserving, the claim holds for any term $t$ as well. If $\alpha$ is even a renaming, we obtain Amato and Scozzari’s (2).

For the second part, by Theorem 21 holds $\alpha(\sigma) = \overline{\alpha}(\sigma) = \overline{\alpha} \cdot \sigma \cdot \overline{\alpha}^{-1}$. \(\Box\)

It is known that idempotence and equivalence of substitutions are not compatible with composition ([Ede85]). Luckily, the concept of variance, with constant prenaming, does not share this handicap:

10 Therefore, $\rho(\sigma) \neq \rho \cdot \sigma$. A good reason for not omitting braces.
Theorem 28 (compositionality). Let \( \sigma, \theta \) be substitutions and \( \alpha \) be their safe prenaming. Then

\[
\alpha(\sigma \cdot \theta) = \alpha(\sigma) \cdot \alpha(\theta)
\]

Proof. Since \( \text{Vars}(\sigma \cdot \theta) \subseteq \text{Vars}(\sigma) \cup \text{Vars}(\theta) \), clearly \( \text{Vars}(\sigma \cdot \theta) \subseteq \text{InDom}(\alpha) \).

Let \( \sigma' \equiv_\alpha \sigma \) and \( \theta' \equiv_\alpha \theta \). Then by Theorem 27 \( \sigma' \cdot \theta' = \alpha(\sigma) \cdot \alpha(\theta) = \overline{\alpha} \cdot \sigma \cdot \overline{\alpha}^{-1} \cdot \overline{\alpha} \cdot \theta \cdot \overline{\alpha}^{-1} = \overline{\alpha} \cdot \sigma \cdot \theta \cdot \overline{\alpha}^{-1} = \alpha(\sigma \cdot \theta) \), hence \( \sigma' \cdot \theta' \equiv_\alpha \sigma \cdot \theta \). \( \square \)

5 Application

Implementing logic programming means that the freedom of Horn clause logic must be restrained:

- most general unifier (mgu) is provided by a fixed algorithm,
- standardization-apart is provided by a fixed algorithm.

Now, from the literature (variant lemma) we know that this is not a serious loss, but a loss it is: if we have an SLD-derivation, we may not any more just rename it wholesale (the resolvents, the mgus, the input clauses), based on Corollary 4, as was possible in Horn clause logic. This is because the two fixed algorithms do not have to be renaming-compatible. In fact, the second one cannot be.

Let us cast a look at the first restriction.

5.1 Renaming compatibility for unification

For any two unifiable terms \( s, t \) holds that the set of their mgus, written as \( \text{Mgus}(s, t) \), is infinite. On the other hand, any particular unification algorithm \( A \) produces, for the given two unifiable terms, just one deterministic value as their mgu. We shall denote this particular mgu of \( s \) and \( t \) as \( \text{Mgu}_A(s, t) \), the algorithmic (or concrete) mgu of \( s \) and \( t \), relative to the given algorithm \( A \).

The abundance of mgus is not only good, it also stands in the way of proofs. The simplest unification problem \( p(x) = p(y) \) has among others two equally attractive candidate mgus, \( \{x/y\} \) and \( \{y/x\} \). Assume our unification algorithm decided upon \( \{x/y\} \). Assume further that we rename the protagonists and obtain the unification problem \( p(x) = p(z) \). What mgu shall be chosen this time?

To ensure some dependability in this issue, we shall require of any unification algorithm the following simple requirement, postulated as an axiom:

Axiom 1 (renaming compatibility of \( \text{Mgu}_A \)). Let \( A \) be a unification algorithm. For any renaming \( \rho \) and any equation \( E \), it has to hold

\[
\text{Mgu}_A(\rho(E)) = \rho(\text{Mgu}_A(E)).
\]

Since classical unification algorithms like Robinson’s and Martelli-Montanari’s do not depend upon the actual names of variables, this requirement is in praxis always satisfied.
Remark 29 (renaming compatibility of Mgus). For every \( \rho \) and \( E \),
\[
\text{Mgus}(\rho(E)) = \rho(\text{Mgus}(E)).
\]
This follows from Theorem 27 and Corollary 4. Assume \( \sigma \in \text{Mgus}(s, t) \), then \( \rho(\sigma)(\rho(s)) = \rho(\sigma(t)) = \rho(\rho(\sigma(s))) \). Further, if \( \theta \) is a unifier of \( \rho(s), \rho(t) \), then \( \theta \cdot \rho \) is a unifier of \( s, t \), hence there is a renaming \( \delta \) with \( \theta \cdot \rho = \delta \cdot \sigma \), giving \( \theta = \delta \cdot \sigma \cdot \rho^{-1} = \delta \cdot \rho^{-1} \cdot \rho \cdot \sigma \cdot \rho^{-1} = (\delta \cdot \rho^{-1}) \cdot \rho(\sigma) \), meaning \( \rho(\sigma) \in \text{Mgus}(\rho(E)) \).

For the other direction, observe \( \theta = \rho \cdot \rho^{-1} \cdot \delta \cdot \sigma \cdot \rho^{-1} = \rho(\rho^{-1} \cdot \delta \cdot \sigma) \).

5.2 Variant lemma for Prolog

For logic programming implementations complying with Axiom 1 and yielding relevant mgus, that is to say for all of them, a propagation result can be proved, which leads to a constructive and incremental version of the variant lemma.

But first we need some auxiliary definitions. Regarding SLD-derivations, we shall assume traditional concepts as given in [Apt97], with one slight change: In denoting an SLD-resolution we employ the actually used variant of a program clause (i.e. not the program clause itself).

Definition 30 (variables of a derivation). Assume \( \mathcal{D} \) to be an SLD-derivation \( G_0 \leftrightarrow_{K_1, \sigma_1} G_1 \ldots \leftrightarrow_{K_n, \sigma_n} G_n \). We shall define the set of variables of \( \mathcal{D} \) as would be natural for a term, i.e. we regard the annotations \( K_i, \sigma_i \) as part of the derivation: \( \text{Vars}(\mathcal{D}) := (\text{Vars}(G_0) \cup \ldots \cup \text{Vars}(G_n)) \cup (\text{Vars}(\sigma_1) \cup \ldots \cup \text{Vars}(\sigma_n)) \cup (\text{Vars}(K_1) \cup \ldots \cup \text{Vars}(K_n)) \).

Lemma 31 (propagation of variance). Assume a unification algorithm \( \mathcal{A} \) satisfying Axiom 1. Assume an SLD-derivation \( \mathcal{D} \) ending with \( G \) and an SLD-derivation \( \mathcal{D}' \) ending with \( G' \) such that \( \alpha(G) = \alpha(G') \) for some prenaming \( \alpha \) which is complete for \( G \) and relevant for \( \mathcal{D} \) to \( \mathcal{D}' \).

Further assume that \( G \leftrightarrow_{K, \sigma} H \) and \( G' \leftrightarrow_{K', \sigma'} H' \) such that in \( G \) and \( G' \) atoms in the same positions were selected and \( K, K' \) are variants of the same program clause. Lastly assume that \( \sigma \) is a relevant mgu. Then for \( \lambda := \text{Pren}(K, K') \) holds

1. \( \alpha \cup \lambda \) is complete for \( H \)
2. \( \alpha \cup \lambda \) is relevant for \( \mathcal{D} \leftrightarrow_{K, \sigma} H \) to \( \mathcal{D}' \leftrightarrow_{K', \sigma'} H' \)
3. \( \sigma' = (\alpha \cup \lambda)(\sigma) \) and \( H' = (\alpha \cup \lambda)(H) \)

Proof. First let us establish that \( \alpha \cup \lambda \) is defined. Due to relevance of \( \alpha \) for \( \mathcal{D}, \mathcal{D}' \),
\[
\text{C}(\alpha) \subseteq \text{Vars}(\mathcal{D}) \quad \text{and} \quad \alpha(\text{C}(\alpha)) \subseteq \text{Vars}(\mathcal{D}')
\]
Due to standardization-apart, \( K \not\Delta \mathcal{D} \) and \( K' \not\Delta \mathcal{D}' \), hence
\[
\text{C}(\lambda) \not\Delta \mathcal{D} \quad \text{and} \quad \lambda(C(\lambda)) \not\Delta \mathcal{D}'
\]

\(^{11}\) Classical unification algorithms not only satisfy Axiom 1 but also yield idempotent mgus. Idempotent mgus are always relevant.
Thus \( C(\alpha) \not\subseteq C(\lambda) \) and \( R(\alpha) \not\subseteq R(\lambda) \), so \( \alpha \uplus \lambda \) is defined. Also, (4) proves that \( \lambda \) is passive on old variables, i.e., \( \lambda(D) = \cdots \) \( (\alpha \uplus \lambda)(\sigma)((\alpha \uplus \lambda)(M), (\alpha \uplus \lambda)(B), (\alpha \uplus \lambda)(N)) \), by (5) and Theorem 20

\[
\begin{align*}
\sigma' &= Mgu_\Delta(A', A_2), \quad B = \sigma(B_1) \\
\sigma' &= Mgu_\Delta(A', A_2), \quad B' = \sigma'(B_2)
\end{align*}
\]

According to the assumption of completeness of \( \alpha \) for \( G \),

\[
\begin{align*}
\text{Vars}(G) &= \text{Vars}(M, A, N) \subseteq C(\alpha) \\
\text{Vars}(K) &= \text{Vars}(A_1, B_1) = C(\lambda) \\
\text{Vars}(\sigma) &\subseteq \text{Vars}(A) \cup \text{Vars}(A_1)
\end{align*}
\]

Having thus fielded all the assumptions, we obtain

\[
\begin{align*}
\text{Vars}(M, A, N) &\subseteq \text{InDom}(\alpha \uplus \lambda), \text{ by (5) and Theorem 20} \\
\text{Vars}(A_1, B_1) &\subseteq \text{InDom}(\alpha \uplus \lambda), \text{ by (6) and Theorem 20} \\
\text{Vars}(\sigma) &\subseteq \text{InDom}(\alpha \uplus \lambda), \text{ by (7), (8) and (9)} \\
\overline{(\alpha \uplus \lambda)}(\sigma) &= (\alpha \uplus \lambda)(\sigma), \text{ by (10)}
\end{align*}
\]

**Proof of 3:**

\[
\begin{align*}
\sigma' &= Mgu_\Delta(A', A_2) = Mgu_\Delta(\alpha(A), \lambda(A_1)) \\
\sigma' &= Mgu_\Delta((\alpha \uplus \lambda)(A), (\alpha \uplus \lambda)(A_1)), \text{ by (5), (6) and Theorem 20} \\
\sigma' &= Mgu_\Delta((\alpha \uplus \lambda)(A), (\alpha \uplus \lambda)(A_1)), \text{ by (8) and (9)} \\
\sigma' &= (\alpha \uplus \lambda)(Mgu_\Delta(A, A_1)), \text{ by Axiom 1} \\
\sigma' &= (\alpha \uplus \lambda)(\sigma)(\alpha \uplus \lambda)(\sigma), \text{ by (11)} \\
B' &= \sigma'(B_2) = (\alpha \uplus \lambda)(\sigma)(\lambda(B_1)) \\
B' &= (\alpha \uplus \lambda)(\sigma)((\alpha \uplus \lambda)(B_1)), \text{ by (6) and Theorem 20} \\
B' &= (\alpha \uplus \lambda)(\sigma(B_1)), \text{ by Theorem 27} \\
B' &= (\alpha \uplus \lambda)(B) \\
H' &= \sigma'(\alpha(M), B', \alpha(N)) \\
H' &= (\alpha \uplus \lambda)(\sigma)(\alpha(M), (\alpha \uplus \lambda)(B), \alpha(N)) \\
H' &= (\alpha \uplus \lambda)(\sigma)((\alpha \uplus \lambda)(M), (\alpha \uplus \lambda)(B), (\alpha \uplus \lambda)(N)), \text{ by (5) and Theorem 20} \\
H' &= (\alpha \uplus \lambda) \cdot \sigma(M, B, N), \text{ by Theorem 27} \\
H' &= (\alpha \uplus \lambda)(H)
\end{align*}
\]

48
Proof of 2: By definition, \( C(\lambda) = Vars(K) \) and \( \lambda(C(\lambda)) = Vars(K') \). Hence, and due to relevance of \( \alpha \),
\[
C(\alpha \uplus \lambda) = C(\alpha) \]
Similarly, \( R(\alpha \uplus \lambda) \subseteq Vars(D' \hookrightarrow_{K', \sigma'} H') \), therefore \( \alpha \uplus \lambda \) is relevant for \( D \hookrightarrow_{K, \sigma} H \) to \( D' \hookrightarrow_{K', \sigma'} H' \).

Proof of 1: By (5) and (6), \( Vars(H) \subseteq Vars(G) \cup Vars(K) \subseteq C(\alpha) \cup C(\lambda) = C(\alpha \uplus \lambda) \), meaning that \( \alpha \uplus \lambda \) is complete for \( H \).

\( \Box \)

**Definition 32 (similarity).** Two SLD-derivations of the same length

\[
G_0 \hookrightarrow_{K_1, \sigma_1} G_1 \hookrightarrow_{K_2, \sigma_2} \ldots \hookrightarrow_{K_n, \sigma_n} G_n
\]
\[
G'_0 \hookrightarrow_{K'_1, \sigma'_1} G'_1 \hookrightarrow_{K'_2, \sigma'_2} \ldots \hookrightarrow_{K'_n, \sigma'_n} G'_n
\]
are similar if \( G_0 \) and \( G'_0 \) are variants and additionally at each step \( i \) holds: atoms in the same position are selected, and the input clauses \( K_i \) and \( K'_i \) are variants of the same program clause.

That the name "similarity" is justified, follows from the claim known as variant lemma ([Llo87], [LS91], [Apt97]), here in the formulation from [Doe93], with our additions for implemented systems.

The added conclusion imparts that renaming a query takes a degree of freedom out: if we treat the variants of program clauses as free to choose, then the mgu is fixed, as could be expected from the added assumptions, Axiom 1 and relevance.

**Lemma 33 (variant claim for logic programming systems).** Assume a unification algorithm \( A \) satisfying Axiom 1 and yielding relevant mgus. Then:

- finite derivations which are similar and start from variant queries have variant resultants,
- the variance depends only on the starting queries and input clauses.

**Proof.** Let the two similar derivations be denoted as in (12). By assumption, \( G_0 \) and \( G'_0 \) are variants, so
\[
\alpha := \text{Pren}(G_0, G'_0)
\]
extists. Clearly, \( \alpha \) is complete for \( G_0 \), since \( Vars(G_0) = C(\alpha) \). By construction, \( \alpha \) is also relevant for \( D_0 := G_0 \) to \( D'_0 := G'_0 \).

We may iterate Lemma 31, obtaining for every \( i = 1, \ldots, n \)
\[
\sigma'_i = (\alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_i)(\sigma_i)
\]
\[
G'_i = (\alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_i)(G_i)
\]
where \( \lambda_i := \text{Pren}(K_i, K'_i) \). Therefore, \( \sigma'_1 \cdot \sigma'_{i-1} \cdot \ldots \cdot \sigma'_1 = (\alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_i)(\sigma_i) \cdot (\alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_{i-1})(\sigma_{i-1}) \cdot \ldots \cdot (\alpha \uplus \lambda_1)(\sigma_1) \).
Let \( k < i \). Since \( Vars(\sigma_k) \subseteq Vars(G_0) \cup Vars(K_1) \cup \ldots \cup Vars(K_k) \subseteq C(\alpha) \cup C(\lambda_1) \cup \ldots \cup C(\lambda_k) = C(\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_k) \), by Theorem 20 \( \alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_k \) is safe for \( \sigma_k \) and \( (\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_k)(\sigma_k) = (\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_k)(\sigma_k) \). Hence,

\[
(\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_{i-1})(\sigma_{i-1}) = (\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_i)(\sigma_{i-1}) \\
\vdots \\
(\alpha \oplus \lambda_1)(\sigma_1) = (\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_i)(\sigma_1) \\
\alpha(G_0) = (\alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_i)(G_0)
\]

(16)

(17)

Let us abbreviate \( \beta_i := \alpha \oplus \lambda_1 \oplus \ldots \oplus \lambda_i \). Then from (14) and (16) by Theorem 28

\[
\sigma_i' \cdot \sigma_{i-1}' \cdot \ldots \cdot \sigma_1' = \beta_i(\sigma_1) \cdot \beta_i(\sigma_{i-1}) \cdot \ldots \cdot \beta_i(\sigma_1) = \beta_i(\sigma_i \cdot \sigma_{i-1} \ldots \cdot \sigma_1)
\]

(18)

Knowing that the resultant of step \( i \) is \( R_i := (\sigma_i \cdot \ldots \cdot \sigma_1(G_0) \leftarrow G_i) \), we obtain

\[
R_i' = (\sigma_i' \cdot \ldots \cdot \sigma_1'(G_0) \leftarrow G_i')
\]

\[
= \beta_i(\sigma_i) \cdot \beta_i(\sigma_{i-1}) \cdot \ldots \cdot \beta_i(\sigma_1)(G_0) \leftarrow \beta_i(G_i), \text{ by (18), (13) and (15)}
\]

\[
= \beta_i(\sigma_i \cdot \ldots \cdot \sigma_1)(\beta_i(G_0)) \leftarrow \beta_i(G_i), \text{ by (17)}
\]

\[
= \beta_i((\sigma_i \cdot \ldots \cdot \sigma_1)(G_0)) \leftarrow \beta_i(G_i), \text{ by Theorem 27}
\]

\[
= \beta_i(R_i)
\]

Summarily, we have \( \sigma_i' = \beta_i(\sigma_i) \), \( G_i' = \beta_i(G_i) \), \( R_i' = \beta_i(R_i) \). □

References


Refutationally Complete Hierarchic Theorem
Proving with Definitions

Adrián Rebola-Pardo
Dresden University of Technology, Germany

Abstract. Hierarchic Superposition is an automated theorem proving system for first-order logic with “background” sorts such as the integers or reals. Refutational completeness of Hierarchic Superposition is guaranteed in the sufficiently complete fragment. Baumgartner et al. developed a method for finitely quantified theorem proving that reduces satisfiability of an input to satisfiability of a number of problems called Finite Domain Transformations. It was stated as a theorem that Finite Domain Transformations lie in the sufficiently complete fragment, thus showing refutational completeness for finitely quantified problems. However, we show that in general this is not the case, by providing an example of a Finite Domain Transformation that is not sufficiently complete. We furthermore identify a new refutationally complete fragment for Hierarchic Superposition, called BSFG-safe. This new fragment contains all Finite Domain Transformations, which shows that Hierarchic Superposition is refutationally complete for finitely quantified problems.

1 Introduction

While automated theorem proving systems for first-order logic have become reasonably powerful in the last years, many industrial applications of theorem proving require reasoning with respect to logics that are not expressible in first-order logic (“background (BG) logics”, e.g. integer or real arithmetic), combined with uninterpreted (“foreground”, FG) operators. Different approaches have been proposed to tackle refutational theorem proving on such logics.

Separately, both features are easy to handle. Decision procedures for many background logics have been developed (e.g. Fourier-Motzkin Elimination [7] or Cooper’s Algorithm [6] for linear arithmetic); uninterpreted operators are natural to first-order logic, thus covered by superposition [1].

The Hierarchic Superposition (HSP) calculus [2, 4] proposes the use of specialized provers for the background logic as off-the-shelf solvers. Background problems are first inferred from the foreground input by first-order superposition, and the background solver is then called to check their satisfiability. The HSP calculus has been implemented in the test-bed solver Beagle [5].

HSP is somewhat similar to Satisfiability Modulo Theories (SMT) [10] insofar as both run background solvers as black-box decision procedures. Whereas background clauses in HSP are derived by superposition inferences, SMT uses a variant of the DPLL algorithm. Although some complete fragments for SMT exist [9],
these results are limited by the heuristic nature of instantiation methods [8] used in SMT.

Refutational completeness results [4] for the HSP calculus require the background logic to be compact and the input clause set to satisfy a condition called sufficient completeness related to the ability to eliminate BG-sorted FG (BSFG) terms. In the context of finitely quantified theorem proving [3], Finite Domain Transformations (FDT) are generated from a clause set by adding definitions of the form \( f(x) \approx \alpha \) where \( \alpha \) is a BG term. FDTs were stated to introduce sufficient completeness into finitely quantified clause sets, but no proof was provided for this claim. Were this true, refutational completeness of HSP for FDTs would follow, thus allowing a refutationally complete search procedure for finitely quantified problems.

In [11], it was shown that this claim is actually false: sufficient completeness is not achieved for general FDTs. However, this does not mean that HSP is not refutationally complete for FDTs, since sufficient completeness is only a sufficient condition for refutational completeness. In that work, the fragment of FG-ground, BSFG-safe problems was found to be refutationally complete for HSP. Since this fragment contains all FDTs generated from clause sets without FG-sorted variables, this proves refutational completeness of the procedure from [3] for finitely quantified theorem proving, as long as FG-sorted variables do not occur in the input clause set.

Whether this condition could or could not be relaxed was stated as an open problem, although no counterexample is given in the original work [11]. In this paper, we show that it can be safely dropped without any further conditions. To do so, in addition to the rewiring technique described there, we present a new technique called augmentation, which allows to extend the results to the general case where FG-sorted variables may occur in the input clause set.

This work is divided as follows. Section 2 briefly introduces the HSP calculus. Section 3 discusses the relation between sufficient completeness and Finite Domain Transformations, with a counterexample to the aforementioned statement. In Section 4 we introduce the BSFG-safe fragment, and discuss how it is preserved by operations performed by HSP solvers. Section 5 shows how can refutational completeness be recovered for this fragment, by describing the augmentation and rewiring techniques in Sections 5.1 and 5.2 respectively; the proof is then explained in Section 5.3. Section 6 outlines the main results.

2 Hierarchic Superposition

Hierarchic theorem proving works in the framework of many-sorted first-order logic with equality under a signature \( \Sigma \) given by a countably infinite set of operators together with their sort. Terms, equations and disequations over the signature \( \Sigma \) are defined as usual. Equations and disequations together are referred to as literals. A clause is a finite disjunction of literals \( L_1 \lor \cdots \lor L_n \); the empty clause is denoted by \( \Box \). Given terms \( t, s, r \) and a position \( \pi \) within \( t \), we denote by \( t[\pi] \) the subterm of \( t \) in position \( \pi \). In an abuse of notation, we use as well
the notation \( t[s] \) to indicate that \( s \) occurs in \( t \) as a subterm; and whenever \( t[p] \) is used after any of the former expressions, it refers to term \( t \) after the subterm \( s \) or the subterm in position \( \pi \) has been replaced by \( r \).

For the sake of simplicity, we do not consider predicates other than equality; every predicate \( p \) with sort \( \xi_1 \ldots \xi_n \) can be regarded as an operator with sort \( \xi_1 \ldots \xi_n \rightarrow \xi_{\text{Bool}} \) where \( \xi_{\text{Bool}} \) is a distinguished sort and every atom \( p(t_1, \ldots, t_n) \) is rewritten as the equation \( p(t_1, \ldots, t_n) \approx \top \), for a distinguished operator \( \top \) of sort \( \xi_{\text{Bool}} \).

Any of the former objects is said to be ground if no variable occurs in it. Substitutions are defined as usual. A ground instance of a clause \( C \) is a ground clause obtained by the application of a substitution to \( C \). We denote the set of ground instances of a clause set \( N \) by \( g(N) \). An interpretation \( I \) is defined by a family of disjoint, non-empty sets \( I^\xi \) for every sort \( \xi \), called carrier sets, together with a mapping \( f^I : I^{\xi_1} \times I^{\xi_n} \rightarrow I^{\xi_0} \) for every operator \( f \) of sort \( \xi_1 \ldots \xi_n \rightarrow \xi_0 \).

The interpretation \( t^I \) of a ground term \( t \) is defined recursively as usual. \( I \) is said to be term-generated if all elements in \( I^\xi \) are the interpretation of some ground term of sort \( \xi \).

\( I \) is said to satisfy a ground equation \( s \approx t \) if \( s^I = t^I \); similarly, \( I \) satisfies \( s \neq t \) if \( s^I \neq t^I \). \( I \) satisfies a clause \( C \) if every ground instance of \( C \) contains a literal satisfied by \( I \). Finally, \( I \) satisfies a set of clauses \( N \) if \( I \) satisfies every clause in \( N \). We abbreviate the expression “\( I \) satisfies \( N \)” as \( I \models N \), and similarly for clauses and literals. Note that these semantics do not correspond with the usual semantics given by variable assignments whenever the interpretation is not term-generated. However, by Herbrand’s Theorem, satisfiability and entailment relations are equivalent for both semantics.

A specification \( S = (\Sigma, \mathcal{M}) \) is a pair where \( \Sigma \) is a signature and \( \mathcal{M} \) is a class of term-generated interpretations closed under isomorphism. Consider a pair of specifications \( S_B = (\Sigma_B, \mathcal{M}_B) \) and \( S_F = (\Sigma_F, \mathcal{M}_F) \) such that \( \Sigma_B \) is a subsignature of \( \Sigma_F \) (i.e. all sorts and operators of \( \Sigma_B \) exist in \( \Sigma_F \) as well). Given a \( \Sigma_B \)-interpretation \( I \) and a \( \Sigma_F \)-interpretation \( J \), we say that \( J \) is a conservative extension of \( I \) if \( I \) is the restriction of \( J \) to operators and sorts in \( \Sigma_B \). The pair \( (S_B, S_F) \) is a hierarchic specification if \( \mathcal{M}_F \) contains exactly the conservative extensions of interpretations in \( \mathcal{M}_B \).

We refer to terms over the signature \( \Sigma_B \) as background (BG) terms; all other terms are called foreground (FG), and similarly for literals and clauses. Note that FG operators with a BG target sort are allowed; these are said to be background-sorted foreground (BSFG) operators. Similarly, interpretations in \( \mathcal{M}_B \) or \( \mathcal{M}_F \) are called BG models or FG models respectively. We assume BG specifications to fulfill the following properties:

- \( \Sigma_B \) includes a countably infinite set of Skolem BG-sorted constants, called parameters. We denote parameters by greek letters.
- \( \Sigma_B \) includes a set of BG-sorted constants called domain elements, such that for every \( I \in \mathcal{M}_B \) and every pair of distinct domain elements \( d_1 \neq d_2 \), we have \( d_1^I \neq d_2^I \).
– $\Sigma_B$ includes a boolean sort denoted by $\xi_{\text{Bool}}$ and a boolean-sorted top constant $\top$.
– $\Sigma_B$ includes a domain predicate operator $p_D$ of sort $\xi \rightarrow \xi_{\text{Bool}}$ for every finite set of domain elements $D$ and every BG sort $\xi$, such that $p_D^I(o) = \top$ if and only if $o \in \{d^I \mid d \in D\}$ for all BG models $I$. For readability, we denote the equation $p_D(t) \approx \top$ as $t \in D$.

We say that a clause set $N$ is satisfiable (with respect to hierarchic semantics) if there is an FG model $J \in \mathcal{M}_F$ with $J \models N$. Furthermore, the hierarchic signature is said to be compact if, for every unsatisfiable set of BG clauses $N$, a finite, unsatisfiable subset $N' \subseteq N$ exists.

A substitution $\sigma$ is said to be simple if it maps BG-sorted variables to BG terms. The set of simple ground instances of a clause set $N$, i.e. ground instances of $N$ obtained by the application of a simple substitution, is denoted by $\text{sgi}(N)$.

In the following we sketch several central notions to the Hierarchic Superposition calculus. A precise definition of such concepts is not essential to present our main results and is not included here due to space constraints. The technical report [12] includes a description in full detail of our framework and findings.

The goal of Hierarchic Superposition is to detect when a clause set is unsatisfiable. Upon an input clause set, HSP works by iteratively appending all possible HSP inferences to the input. Every time a new BG clause is inferred, an off-the-shelf BG solver is called on the set of BG clauses; unsatisfiability of the BG clauses would then prove unsatisfiability of the input. Abstraction is used to split literals into their BG and FG components by replacing a clause $C[t]$ by $C[x] \lor x \neq t$ where $x$ is a fresh variable. Weak abstraction is a restriction of abstraction so that the transformation above is applied only to target terms $t$ that satisfy some conditions. We denote the weakly abstracted version of a clause $C$ by $\text{abs}(C)$, that is, the resulting clause after all terms targeted by weak abstraction have been abstracted out.

HSP is parameterized by a hierarchic reduction ordering, that is, a well-founded reduction ordering $\prec$ over terms that satisfies some compatibility properties with respect to substitution, context and the FG-BG separation (see [4, 12] for details). A hierarchic reduction ordering can be extended to a well-founded ordering over literals and clauses. In particular, we say that a literal $L$ is strictly maximal in $C$ whenever $L$ is maximal among the literals in $C$ w.r.t. $\prec$ and there is only one occurrence of $L$ in $C$.

A full-detail description of HSP inferences is presented in [4, 12]; we present a simplified version thereof. HSP inferences are modifications of the original superposition calculus [1]. In particular, we have inference rules such as Positive Superposition:

\[
\begin{align*}
& \text{Positive Superposition: } \quad l \approx r \lor C \quad s[u] \approx t \lor D \\
& \quad \text{abs}(s[r] \approx t \lor C \lor D \sigma)
\end{align*}
\]

where:

– Neither $l$ nor $u$ are BG terms, nor is $u$ a variable.
– $σ$ is a mgu of $l$ and $u$ that maps BG-sorted variables to either domain elements or variables.
– $lσ \not< rσ$ and $sσ \not< tσ$.
– $(l \approx r)σ$ is strictly maximal in $(l \approx r \lor C)σ$.
– $(s \approx t)σ$ is strictly maximal in $(s \approx t \lor D)σ$.

**Example 1.**

Calls to the BG solver are modeled by the Close inference rule:

\[
\text{Close} \quad C_1 \quad \ldots \quad C_n \quad \square
\]

where \{\(C_1, \ldots, C_n\}\} is an unsatisfiable BG clause set (with respect to the BG models of the specification).

Furthermore, a simplification rule allows to replace clauses in a satisfiability-preserving way. The current implementation of HSP allows the application of several *ad hoc* techniques within the simplification rule, namely demodulation by unit clauses, tautology deletion, elimination of subsumed clauses, arithmetic simplification, unabstraction of domain elements and negative unit simplification [5].

HSP is endowed as well with *redundancy criteria* that detect inferences which are not needed to infer the empty clause. A clause set \(N\) is *saturated* if all possible HSP inferences over \(N\) are redundant. We say that HSP is *refutationally complete* for a class of clause sets \(\mathcal{P}\) if for all saturated unsatisfiable clause sets \(N \in \mathcal{P}\) we have \(\square \in N\).

### 3 Sufficient Completeness and Definitions

An interpretation \(I\) is said to be *BG-complete* if for every ground BG-sorted term \(t\) there exists a ground BG term \(s\) such that \(I \models s \approx t\). The intuition behind BG-completeness is the following: whenever a property holds for a ground BG-sorted term under an interpretation, then the same property holds for some ground BG term. In particular, such a property expressed by a clause set can be translated into a BG clause set. Were this property “non-modelic” (i.e. not holding in the BG models of the specification) the Close rule would detect a contradiction.

**Example 2.** Let \(N\) be any clause set using as a BG specification the Linear Integer Arithmetic (LIA) specification. \(\mathbb{Z}\), together with the usual interpretation for number constants, + and < is the standard BG model of LIA. However, we may consider the interpretation with carrier set \(\mathbb{Z} \cup \{\infty\}\) that interpretes + and < as usual in the extended real line. Assume there is a ground BG term (e.g. a parameter) \(\alpha\) mapped by the second interpretation to \(\infty\). In this case, \(x \leq \alpha\) is satisfied by the latter interpretation, but not by the former.

Define GndTh(\(\mathcal{M}_B\)) as the set of ground BG clauses satisfied by all \(I \in \mathcal{M}_B\). A clause set \(N\) is said to be *sufficiently complete* if every interpretation \(I\) (not necessarily in \(\mathcal{M}_B\)) with \(I \models sgi(N) \cup \text{GndTh}(\mathcal{M}_B)\) is BG-complete. The following completeness result for HSP was proved in [4].
Theorem 1. Let $C$ be the class of weakly abstracted, sufficiently complete clause sets. If the hierarchic specification is compact, then HSP is refutationally complete for $C$.

In practice, sufficient completeness is a rather restrictive condition. Different techniques have been developed in order to introduce sufficient completeness in a problem, such as the Define rule in [4] or Finite Domain Transformations (FDT) in [3]. Such methods work by selecting every occurrence of a BSFG operator $f$ and replacing the term $f(t_1, \ldots, t_n)$ in that position by a parameter $\alpha$. Furthermore, a definition of the form $f(t_1, \ldots, t_n) \approx \alpha$ is added to the clause set. This forces $f(t_1, \ldots, t_n)$ to be interpreted constantly over the variables occurring in $t_1, \ldots, t_n$. The method from [3] in particular reduces satisfiability of finitely quantified clauses, i.e. clauses where BG-sorted variables are quantified over a bounded domain, to satisfiability of some of its FDTs. For simplicity, we introduce here a simplified version of FDTs; a definition in full detail is given in the original article [3].

A clause $C$ is said to be finitely quantified if it is of the form $C^\dual \lor \bigvee_{i \in I} x_i /\in \Delta_i$ where:

1. The $x_i$ are pairwise distinct and comprise exactly the variables below a BSFG operator in the clause $C^\dual$.
2. The $\Delta_i$ are finite sets of same-sorted domain elements.
3. $C^\dual$ does not contain further predicate domains ($x \notin \Delta$).

Consider a set of finitely quantified clauses $N$ with variables $x_1, \ldots, x_n$ quantified over $\Delta_1, \ldots, \Delta_n$. A domain partition $P$ of $N$ is defined by a partition of each $\Delta_i$ into subdomains $\Delta^1_i, \ldots, \Delta^{k_i}_i$. We now build the Finite Domain Transformation of $N$ for the domain partition $P$. For every choice $j = (j_1, \ldots, j_n)$ where $1 \leq j_i \leq k_i$ and every clause $C = C^\dual \lor \bigvee_{i \in I} x_i /\in \Delta_i$, the clause set $\text{FDT}(C, j)$ is given by the following procedure:

1. Initialize $D$ as the clause $C^\dual \lor \bigvee_{i \in I} x_i /\in \Delta^1_i$ and $M := \emptyset$.
2. While $D$ contains some BSFG operator, choose a fresh parameter $\alpha$ and a position $\pi$ minimal among those containing a BSFG operator. Add the clause $D[\pi] \approx \alpha \lor \bigvee_{i \in I} x_i /\in \Delta^1_i$ to the clause set $M$, and set $D := D[\alpha]$.
3. Let $\text{FDT}(C, j) = \{D\} \cup M$.

The FDTs of $N$ are defined for every domain partition $P$ as:

$$\text{FDT}(N, P) = \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j_i \leq k_i} \text{FDT}(C, (j_1, \ldots, j_n))$$

Example 3. Consider the clause $C$ given by $f(x) < 0 \lor x /\in \{-2, \ldots, 2\}$ and a domain partition given by the partition $\{-2, \ldots, -1\} \cup \{0, \ldots, 2\} = \{-2, \ldots, 2\}$
for $x$. Then $\text{FDT}(C, P)$ contains the clauses:

\[
\alpha < 0 \lor x \not\in \{-2, \ldots, -1\}
\]

\[
f(x) \approx \alpha \lor x \not\in \{-2, \ldots, -1\}
\]

\[
\beta < 0 \lor x \not\in \{0, \ldots, 2\}
\]

\[
f(x) \approx \beta \lor x \not\in \{0, \ldots, 2\}
\]

FDTs are used in [3] to build a search procedure over the domains $D_1, \ldots, D_n$ that reduces satisfiability of $N$ to satisfiability of $\text{FDT}(N, P)$ for some domain partitions $P$. Refutational completeness of FDTs would thus yield a refutationally complete procedure for finitely quantified problems. In it was stated as a theorem that FDT produces sufficiently complete clause sets, which implies refutational completeness of HSP for FDTs by Theorem 1. However, this claim is actually false.

Example 4. Consider a hierarchic specification using the linear integer arithmetic (LIA) BG specification, i.e. the integer logic where only addition and inequalities are allowed. Let $\xi_B$ be the integer BG sort and $\xi_F$ a freely interpreted FG sort. Consider $\xi_F$-sorted constants $a, b$, and operators $f, g$ of sort $\xi_B \to \xi_F$ and $\xi_F \to \xi_B$ respectively. Let $N = \{a \neq b, \ g(f(x)) \neq x \lor x \not\in \{1, \ldots, 2\}\}$. For a domain partition $P$ taking $\{1, \ldots, 2\}$ as the only subdomain for $\{1, \ldots, 2\}$, we obtain the following clauses in $N' = \text{FDT}(N, P)$:

\[
a \neq b
\]

\[
\alpha \neq x \lor x \not\in \{1, \ldots, 2\}
\]

\[
g(f(x)) \approx \alpha \lor x \not\in \{1, \ldots, 2\}
\]

Let $J$ be an extension of the standard model for LIA to a domain $\mathbb{Z} \cup \{\infty\}$ for the integer sort $\xi_B$, completed with a carrier set $J^{\xi_F} = \{a^*, b^*\}$ and the following interpreted operators: $\alpha^J = 0$, $a^J = a^*$, $b^J = b^*$ and:

\[
- \ f^J(x) = a^* \text{ for all } x \in \mathbb{Z} \cup \{\infty\}. \\
- \ g^J(a^*) = 0 \text{ and } g^J(b^*) = \infty.
\]

Then, $J \models N'$ (in particular $J \models \text{sg}(N')$), and $J \models \text{GndTh}(M_B)$. Nevertheless, $\infty$ can only be obtained as the interpretation of a BSFG term (e.g. $g(b)$), but no BG term is interpreted to $\infty$ under $J$. Thus, $N'$ is not sufficiently complete. This contradicts the aforementioned claim.

4 The BSFG-safe fragment

The previous example shows that sufficient completeness is not achieved in general for FDTs, so refutational completeness for FDTs is not guaranteed by Theorem 1. However, we can modify the proof for this theorem to extend it to a new class of problems including the ones produced by FDT.

57
A term is said to be BSFG-free if it contains no BSFG operators; and BSFG-headed if it is of the form $f(t_1, \ldots, t_n)$ where $f$ is a BSFG operator and the $t_i$ are all BSFG-free. BSFG-free literals and clauses are defined analogously. A literal is a BSFG-definition if it is of the form $f(t_1, \ldots, t_n) \equiv \alpha$, where $f(t_1, \ldots, t_n)$ is a BSFG-headed term and $\alpha$ is a parameter. A clause is said to be BSFG-safe if it only contains BSFG-free literals and BSFG-definitions.

The following result follows straightforward from the definition of FDTs, which are generated by exhaustively “defining apart” BSFG operators in the input clause set.

**Proposition 1.** Let $N$ be a finitely quantified clause set, and $P$ a domain partition for the finite domains in $N$. Then, FDT($N, P$) is BSFG-safe.

We obtain the following results showing that the BSFG-safe fragment is stable under the operations of the HSP calculus. This means that from an input, the clause set obtained by saturation by HSP from its FDTs belong to the BSFG-safe fragment.

**Proposition 2.** Let $N$ be a BSFG-safe clause set. Then,
1. If $N'$ is obtained from $N$ by weak abstraction and HSP inferences, then $N'$ is BSFG-safe.
2. If $N'$ is obtained from $N$ by the simplification rule applying demodulation by unit clauses, tautology deletion, elimination of subsumed clauses, arithmetic simplification, unabstraction of domain elements or negative unit simplification, then $N'$ is BSFG-safe.

We have not included the proof for this result due to lack of space and interest, since it is rather long and purely syntactical; as usual it is included in our technical report [12]. The BSFG-safe fragment is not stable under the general simplification rule described in [4]. Nevertheless, this rule is much more general than the actual application cases. In practice, HSP-based solvers only apply the techniques mentioned in Proposition 2.

5 Refutational completeness

Our approach is based on a modification of the proof of Theorem 1 found in [4]. The following definitions are introduced there for an arbitrary BG model $I \in \mathcal{M}_B$ and a saturated clause set $N; \rightarrow$ is used below to denote rules of a term-rewriting system.

\[
m(t) = \min_{\prec} \{ s \mid s \text{ is a ground BG term and } I \models t \approx s \}
\]

\[
E'_t = \{ t \rightarrow m(t) \mid t \text{ is a ground BG term and } t \neq m(t) \}
\]

\[
E''_t = \{ l \rightarrow r \in E'_t \mid l \text{ is not reducible by } E' \setminus \{ l \rightarrow r \} \}
\]

\[
E_t = \{ l \approx r \mid l \rightarrow r \in E''_t \}
\]

\[
D_t = \{ l \neq r \mid l, r \text{ are ground BG terms with different } E_t\text{-normal forms} \}
\]

\[
A_I = \{ C\sigma \in gi(N) \mid C \in N \text{ and } \sigma \text{ is a simple, } E''_t\text{-reduced substitution} \}
\]

\[
N_I = A_I \cup E_t \cup D_t
\]

58
Both the canonical term rewriting system $E''_I$ and the clause set $E_I \cup D_I$ force equality over ground BG terms to behave as in $I$; they can be thought of as encodings of $I$ into ground first-order clauses. We denote the $E''_I$-normal form of a term $t$ by $t^*$. Most importantly for these definitions, note that BSFG-safety of $N$ does not imply BSFG-safety of $N_I$ when FG-sorted variables occur in $N$.

**Example 5.** Consider the signature from Example 4 and the clause set $N = \{g(y) \approx a\}$ is BSFG-safe. However, the non-BSFG-safe clause $g(f(1)) \approx a$ belongs to $N_I$. □

The following is an abstraction of the steps in the proof of Theorem 1 in [4].

1. Assume that $N$ does not contain $\square$. It is shown that a BG model $I$ satisfying the BG clauses in $N$ exists. The existence of one further term-generated interpretation $J$ satisfying $N_I$ is then proved.
2. Since $N$ is sufficiently complete and satisfies $\text{sgi}(N) \cup \text{GndTh}(M_B)$, BG-completeness of $J$ is shown.
3. Since $J$ is BG-complete and term-generated, from $J \models C_I$ follows that $J \models N$.
4. Since $J$ is BG-complete and term generated, from $J \models E_I \cup D_I$ follows that $J$ is a conservative extension of $I$.
5. Thus, $J$ an FG model of $N$, so $N$ is satisfiable with respect to the hierarchic semantics.

Note that the notion of BG-completeness is not explicitly defined in [4], so steps 3 and 4 in the original proof seem to rely on sufficient completeness of $N$; nevertheless, it suffices to consider that $J$ is BG-complete. Our modified proof for the BSFG-safe fragment exploits this abstraction. In particular, in this proof sufficient completeness is exclusively required to show that $J$ is BG-complete in step 2.

The idea behind our proof is to modify step 2 by using two techniques. First, the interpretation $J$ is “augmented” into a term-generated, FG-complete interpretation $K$. Augmentation is performed in a satisfiability preserving way, by changing the interpretation of FG-sorted “augmentation constants”, which do not occur in $N$, nor therefore in $N_I$. $K$ is further refined by “rewiring” it into a term-generated, BG- and FG-complete interpretation $K_0$. Rewiring only preserves satisfiability for ground BSFG-safe clause sets.

However, as shown above, $N_I$ is not necessarily BSFG-safe whenever $N$ contains FG-sorted variables. In [11], where no occurrences of FG-sorted variables are allowed, rewiring without augmentation yields a BG-complete interpretation that satisfies $N_I$, and thus the proof above can be resumed to show that $N$ is satisfiable with respect to the hierarchic semantics. However, in the general case $N_I$ may not be BSFG-safe.

This is circumvented by showing instead that $K_0$ satisfies a different clause set $V_I$ defined as follows. Consider a new augmented signature where a countably infinite number of $\xi$-sorted augmentation constants have been added to every FG sort $\xi$. Terms in the signature previous to augmentation are referred to
as original terms; terms possibly containing augmentation constants are called augmented terms. Most importantly, since in the proof we consider augmentation after definition of $N_I$, simple substitutions will be considered as substitutions over the original signature. An ultrasimple substitution is a substitution over the augmented signature that maps BG variables to BG terms and FG variables to augmentation constants. Then, $V_I$ is defined by:

$$U_I = \{ C\sigma \in \text{gi}(N) \mid C \in N \text{ and } \sigma \text{ is an ultrasimple, } E_I''\text{-reduced substitution} \}$$

$$V_I = U_I \cup E_I \cup D_I$$

$V_I$ is BSFG-safe and therefore we can show that $K_0$ satisfies $V_I$. Step 3 can then be substituted by proving that, if $K_0$ satisfies $V_I$, then it satisfies $N_I$, which can be shown by the additional fact that $K_0$ is FG-complete. The proof can then be resumed from step 4. Our new proof looks as follows:

1. Assume that $N$ does not contain $\Box$. It is shown that a BG model $I$ satisfying the BG clauses in $N$ exists. The existence of one further term-generated interpretation $J$ satisfying $N_I$ is then proved.
2. Obtain a term-generated, FG-complete interpretation $K$ from $J$ by augmentation, and show that $K$ satisfies $V_I$.
3. Obtain a term-generated, BG- and FG-complete interpretation $K_0$ from $K$ by rewiring.
4. Show that $V_I$ is BSFG-safe. Then, $K_0 \models V_I$.
5. Since $K_0$ is BG- and FG-complete and term-generated, from $K_0 \models V_I$ follows that $K_0 \models N_I$.
6. Since $K_0$ is BG-complete and term generated, from $K_0 \models E_I \cup D_I$ follows that $K_0$ is a conservative extension of $I$.
7. Thus, $K_0$ an FG model of $N$, so $N$ is satisfiable with respect to the hierarchic semantics.

5.1 Augmentation of Interpretations

Consider the interpretation $J$ obtained in step 1. Since $J$ is term-generated and there are only countably many terms, carrier sets in $J$ contain only countably many elements. Then, for every FG sort $\xi$ and every element $o \in J^\xi$ we can choose a distinct augmentation constant $a_o$. Then, we define the augmentation of $J$ as an extension $K$ of $J$ with the same carrier sets as $J$ where, for every augmentation constant $a_o$, $a_o^K = o$, and all other operators interpreted as in $J$. The following result is trivial from the definition:

**Lemma 1.** $K$ is a term-generated, FG-complete interpretation.

To prove step 2 in the modified proof, we just need to show that $K$ satisfies $U_I$, since augmentation only changes the interpretation of operators not occurring in $N_I$.

**Lemma 2.** If $J$ satisfies $N_I$, then $K$ satisfies $U_I$.
Proof. Consider an arbitrary clause $C \in N$ and an ultrasimple, $E''_I$-reduced substitution $\sigma$ such that $C\sigma$ is ground. Since $K$ is FG-complete and $J$ is term-generated, we can find a mapping $c$ that maps every ground FG-sorted augmented term $t$ to a FG-sorted original term $c(t)$ such that $c(t)^J = t^K$. Then, the substitution $\tau$ with the same domain as $\sigma$ given by

$$x\tau = \begin{cases} 
  x\sigma & \text{if } x \text{ is BG-sorted} \\
  c(x\sigma) & \text{if } x \text{ is FG-sorted}
\end{cases}$$

is an simple, $E''_I$-reduced substitution such that $C\tau$ is ground. Furthermore, since $c(t)^J = t^K$ for all FG-sorted augmented terms, it can be shown by induction on the structure of terms that, if $t$ is an arbitrary term with both $t\sigma$ and $t\tau$ ground terms, $(t\sigma)^K = (t\tau)^J$. Then, we conclude that, $K \models C\sigma$ if and only if $J \models C\tau$. Since $C\tau \in N_I$ and $J$ satisfies $N_I$, we obtain $K \models C\sigma$ for an arbitrary clause $C\sigma \in U_I$, and the claim follows. $\square$

5.2 Rewiring of Interpretations

We have obtained the augmentation $K$ of $J$, and we have proved the claims in step 2 in the modified proof. From there on, the proof uses instead the rewiring of $K$, which is an interpretation $K_0$ with carrier sets defined by:

$$K_0^\xi = \{ o \in K^\xi \mid \exists \text{ a ground } BG \text{ term } t \text{ with } t^K = o \}$$

if $\xi$ is a BG sort, and $K_0^\xi = K^\xi$ if $\xi$ is an FG sort. This is: we drop elements from BG carrier sets that prevent $K$ from being BG-complete. This requires some modification on the interpretation of operators, since now they might not be well-defined. In this case, there are no constraints on how interpreted operators for $K$ are remapped in undefined values. In particular, we can choose an element $q_\xi \in K_0^\xi$ for every sort $\xi$, and define for every operator $f$ with sort $\xi_1 \cdots \xi_n \to \xi_0$:

$$f^K_0(o_1, \ldots, o_n) = \begin{cases} 
  f^K(o_1, \ldots, o_n) & \text{if } f^K(o_1, \ldots, o_n) \in K_0^\xi_0 \\
  q_{o_0} & \text{if } f^K(o_1, \ldots, o_n) \notin K_0^\xi_0
\end{cases}$$

This definition justifies the name “rewiring” for this process: after problematic terms have been dropped from the BG carrier sets, functions defining the interpretation of operators need to be redirected in a minimal way. The next result follows trivially from the definition of $K_0$ and the fact that $K$ is term-generated and FG-complete, and gives step 3 in the modified proof.

Example 6. Consider the clause set $N'$ and the interpretation $I$ given in Example 4. The only element in the $\xi_B$ sort carrier set violating BG-completeness is $\infty$. Thus we can drop $\infty$ from $J^\xi_B$ and remap $g^I(b)$ to, for example, 7. Rewiring then yields the interpretation $K_0$ with carrier sets $K_0^\xi_B = \mathbb{Z}$ and $K_0^\xi_F = \{ a^*, b^* \}$ and interpreted operators: $\alpha^{K_0} = 0$, $\alpha^{K_0} = \alpha^*$, $\beta^{K_0} = \beta^*$ and

$$- f^{K_0}(x) = a^* \text{ for all } x \in \mathbb{Z}.$$
– \( g^{K_0}(a^*) = 0 \) and \( g^{K_0}(b^*) = 7 \).

Then, \( K_0 \) is a BG-complete, term-generated interpretation that satisfies \( N' \). Furthermore, \( K_0 \) is a conservative extension of the standard model of the LIA specification, which shows satisfiability of \( N' \) with respect to the hierarchic semantics.

**Lemma 3.** \( K_0 \) is a term-generated, FG- and BG-complete interpretation.

To obtain step 4 in the modified proof, we must show that \( V_I \) is a ground BSFG-safe clause set and rewiring preserves satisfiability of such clause sets.

**Lemma 4.** If \( N \) is BSFG-safe, then so is \( V_I \).

**Proof.** Clauses in \( E_I \cup D_I \) are trivially BSFG-safe, since they are BG clauses. For ultrasimple ground instances \( C\sigma \in U_I \) of \( N \), it suffices to note that a substitution \( \tau \) can turn a BSFG-safe clause \( D \) into a non-BSFG-safe one only if there is a variable \( x \) in \( D \) such that \( x\tau \) contains BSFG operators. However, this is not the case for \( C \) and \( \sigma \), because \( \sigma \) is ultrasimple. This means that BG-sorted variables are mapped by \( \sigma \) to ground BG terms and FG-sorted variables are to augmentation constants, hence instantiation by \( \sigma \) preserves BSFG-safety.

**Lemma 5.** Let \( C \) be a ground BSFG-safe clause. If \( K \models C \), then \( K_0 \models C \).

**Proof.** To show this, we use the following result.

**Result:** If \( t \) is a ground BSFG-free term, then \( t^K = t^{K_0} \).

This can be shown by induction by first proving it for the case when \( t \) is a ground BG term, and then for the general case.

Now assume that \( K \models C \), so \( K \) satisfies some literal \( L \) in \( C \). Since \( C \) is BSFG-safe, one of the following holds:

– \( L \) is a BSFG-free term. In this case, \( K_0 \models L \) follows from the result above.
– \( L \) is a BSFG definition of the form \( f(t_1, \ldots , t_n) \approx \alpha \), where \( \alpha \) is a \( \xi \)-sorted parameter, \( f \) is a BSFG operator and the \( t_i \) are BSFG-free terms. By the result above, we have \( (f(t_1, \ldots , t_n))^{K_0} = f^{K_0}(t_1^K, \ldots , t_n^K) \) and \( \alpha^{K_0} = \alpha^K \).

Now, since \( K \models L \), we have that

\[
 f^K(t_1^K, \ldots , t_n^K) = (f(t_1, \ldots , t_n))^K = \alpha^K \in K_0^\xi
\]

Hence, by the definition of interpreted operators for \( K_0 \), we obtain

\[
(f(t_1, \ldots , t_n))^{K_0} = f^{K_0}(t_1^K, \ldots , t_n^K) = f^K(t_1^K, \ldots , t_n^K) = (f(t_1, \ldots , t_n))^K
\]

and \( K_0 \models L \) follows.

In both cases, \( L \) is satisfied by \( K_0 \). Therefore, \( K_0 \models C \).
5.3 The full modified proof

So far, our modified proof is as follows:

1. Assume that $N$ does not contain $\Box$. It is shown that a BG model $I$ satisfying the BG clauses in $N$ exists. The existence of one further term-generated interpretation $J$ satisfying $N_I$ is then proved. This step is the same as in the original proof without modification.

2. Obtain a term-generated, FG-complete interpretation $K$ from $J$ by augmentation, and show that $K$ satisfies $V_I$. This is shown by Lemmas 1 and 2.

3. Obtain a term-generated, BG- and FG-complete interpretation $K_0$ from $K$ by rewiring. This is given by Lemma 3.

4. Show that $V_I$ is BSFG-safe. Then, $K_0 \models V_I$. This was shown as Lemmas 4 and 5.

5. Since $K_0$ is BG- and FG-complete and term-generated, from $K_0 \models V_I$ follows that $K_0 \models N$. This step remains to be shown.

6. Since $K_0$ is BG-complete and term-generated, from $K_0 \models E_I \cup D_I$ follows that $K_0$ is a conservative extension of $I$. Since we already know that $K_0$ is BG-complete and term-generated, this step can be obtained from step 4 in the original proof by replacing $K$ by $K_0$.

7. Thus, $K_0$ an FG model of $N$, so $N$ is satisfiable with respect to the hierarchic semantics. This follows by definition of FG models in the hierarchic semantics from steps 5 and 6.

Thus, only step 5 remains to be shown. That is, we need to show that, if $K_0 \models V_I$, then $K_0 \models N$. To prove this, we will exploit the properties we have introduced in $K_0$ by rewiring and augmentation, namely BG- and FG-completeness. The main idea is the following: any ground instance of a clause $C \in N$ obtained through a substitution $\sigma$ can be rewritten into an ultrasimple ground instance $C_\sigma$ of $N$ in such a way that terms in $C_\sigma$ are interpreted by $K_0$ in the same way as corresponding terms in $C_\tau$. To do so, for every variable $x$, we find a BG term or an augmentation constant (depending on $x$ being BG- or FG-sorted) $x_\tau$ such that $(x_\sigma)^{K_0} = (x_\tau)^{K_0}$. Such a term exists because of BG- and FG-completeness.

Lemma 6. If $K_0 \models V_I$, then $K_0 \models N$.

Proof. Before providing a full proof, note that BG- and FG-completeness can be rephrased as the existence of:

- a mapping $a$ from ground BG-sorted terms to ground BG terms such that $a(t)^{K_0} = t^{K_0}$,
- a mapping $b$ from ground (original) FG-sorted terms to augmentation constants such that $b(t)^{K_0} = t^{K_0}$.

Let $C$ be any clause in $N$. We need to show that any arbitrary ground instance $C_\sigma$ is satisfied by $K_0$. Then, consider the substitution $\tau$ with the same domain as $\sigma$ defined by

$$x_\tau = \begin{cases} a(x_\sigma) & \text{if } x \text{ is BG-sorted} \\ b(x_\sigma) & \text{if } x \text{ is FG-sorted} \end{cases}$$
τ is an ultrasimple $E''_I$-reduced substitution with the property that $(x\sigma)^{K_0} = (x\tau)^{K_0}$ for all variables $x$. It is then easy to show by induction that $(t\sigma)^{K_0} = (t\tau)^{K_0}$ for all original ground terms $t$, and thus $K_0 \models C\sigma$ if and only if $K_0 \models C\tau$. Now, $C\tau$ is an ultrasimple ground instance of $C$, and since $K_0$ satisfies $V_I$, we can find a literal $L\tau$ in $C\tau$ satisfied by $K_0$. Then, $L\sigma$ is satisfied by $K_0$, and therefore so is $C\sigma$, as we wanted.

This finished the proof of our main result:

**Theorem 2.** Let $S$ be the class of weakly abstracted, BSFG-safe clause sets. If the hierarchic specification is compact, then HSP is refutationally complete for $S$.

In particular, by Propositions 1 and 2 we obtain the following corollary that fixes the problem posed by discovering that in general FDTs are not sufficiently complete. This is done by showing that they belong nevertheless to a refutationally complete fragment for HSP, namely the BSFG-safe fragment.

**Corollary 1.** HSP is refutationally complete for FDTs.

The result above extends the result in [11], to the extent that the requirement there for FG-groundness of the input clause set, i.e. the input not containing FG-sorted variables, has been lifted.

**6 Conclusions and future work**

Hierarchic Superposition is a powerful calculus for many-sorted first-order logic with background theories. Applications for Hierarchic Superposition include formal verification of software and processes, which often requires to detect unsatisfiability of quantified formulae.

Completeness results for Hierarchic Superposition guarantee refutational completeness in the sufficiently complete fragment. However, sufficient completeness is a rather complex property that is often missing in input clause sets. Several techniques based on definition introduction have been developed to circumvent this problem. In particular, an application of Hierarchic Superposition to finitely quantified theorem proving relies on iteratively solving problems where definitions are introduced for background-sorted foreground operators. The generated problems are called Finite Domain Transformations, and were claimed to be sufficiently complete.

We found a counterexample to this claim, this is, we provide problems whose Finite Domain Transformations do not lie in the sufficiently complete case. Thus, refutational completeness was not guaranteed in principle for Finite Domain Transformations. Without a proof for refutational completeness of Hierarchic Superposition for such problems, recent advances in the field relying on this feature would be ineffective.

Such counterexamples only show that Finite Domain Transformations do not belong to a given refutationally complete fragment, but this does not mean that Hierarchic Superposition is not refutationally complete for them. We define
a the BSFG-safe fragment, which contains all Finite Domain Transformations. We show this fragment to be stable under the operations applied by current implementations of Hierarchic Superposition.

Finally, we show that Hierarchic Superposition is refutationally complete for the BSFG-safe fragment. This is, for saturated clause sets in this fragment and compact background theories, unsatisfiability detection can be reduced to detect whether the empty clause belongs to the clause set. In particular, we obtain refutational completeness for Finite Domain Transformations. This represents a full solution to the problem we found with our counterexample.

Our solution does not only work for Finite Domain Transformations, but for any BSFG-safe input. In particular, finite quantification is not needed at all for refutational completeness. However, introduced definitions have a very rigid form, where terms with a background-sorted foreground operator in the root position are equated to a background constant. Further work is needed to find out if this condition can be relaxed, possibly allowing non-ground definitions, how can this be used to develop complete search procedures similar to the ones using Finite Domain Transformations.

References

Knowledge Engineering for Hybrid Deductive Databases

Dietmar Seipel

University of Würzburg, Institute for Computer Science, Germany
seipel@informatik.uni-wuerzburg.de

Abstract. Modern knowledge base systems frequently need to combine a collection of databases in different formats: e.g., relational databases, XML databases, rule bases, ontologies, etc. In the deductive database system DDBASE, we can manage these different formats of knowledge and reason about them. Even the file systems on different computers can be part of the knowledge base. Often, it is necessary to handle different versions of a knowledge base. E.g., we might want to find out common parts or differences of two versions of a relational database. We will examine the use of abstractions of rule bases by predicate dependency and rule predicate graphs. Also the proof trees of derived atoms can help to compare different versions of a rule base. Moreover, it might be possible to have derivations joining rules with other formalisms of knowledge representation. Ontologies have shown their benefits in many applications of intelligent systems, and there have been many proposals for rule languages compatible with the semantic web stack, e.g., SWRL, the semantic web rule language. Recently, ontologies are used in hybrid systems for specifying the provenance of the different components.

Keywords. hybrid knowledge bases, deductive databases, rules, ontologies

1 Introduction

Relational databases and deductive databases with rule bases have been prominent formalisms of knowledge representation for a long time [3]. A strong research interest in deductive database technology and its applications has reemerged in the recent years, leading to what has been called a resurgence [9] or even a renaissance [1] for DATALOG. This revival is propelled by important new applications areas, such as distributed computations and big–data applications, the success of a first commercial DATALOG System, and progress in semantics extensions to support non–monotonic constructs such as default negation and aggregates – a theoretical thread that had actually continued through the years [14].

In the last years, the use of ontologies has shown its benefits in many applications of intelligent systems. There have been many proposals for rule languages compatible with the semantic web stack, e.g., the definition of SWRL (semantic web rule language) originating from RULEML and similar approaches [11]. It is well agreed that the combination of ontologies with rule–based knowledge is essential for many interesting semantic web tasks, e.g., the realization of semantic web agents and services. SWRL allows for the combination of a high–level abstract syntax for Horn–like rules with OWL, and a
model theoretic semantics is given for the combination of OWL with SWRL rules. An XML syntax derived from RULEML allows for a syntactical compatibility with OWL. However, with the increased expressiveness of such ontologies new demands for the development and for maintenance guidelines arise.

Thus, approaches for evaluating and maintaining hybrid knowledge bases need to be extended and revised to work with different kinds of knowledge including rules, relational databases, XML documents, and ontologies. In this paper, we are interested in various types of knowledge bases including a collection of relational databases or ontologies, or the hierarchy in a file system. Here, the interaction between the knowledge bases can be very important. E.g., there could be a call from a logical rule to a bayesian network. The abstraction of knowledge bases is related to the schemas of relational and XML databases, the predicate dependency and rule predicate graphs of deductive databases and rule–based systems. Obviously, w.r.t. versioning, there is a relationship to synchronisation and diff for programs and files. For file systems, well–known characteristics are the size parameters, such as words, lines, characters.

Organization of the Paper. The rest of this paper is organized as follows: Section 2 shows how rule bases can be abstracted and visualized in DDBASE using different types of dependency graphs. Derivations can be explained by proof trees. Section 3 investigates ontologies with rules in SWRL, and it shows how the provenance of ontologies can be modelled and reasoned about. Hybrid knowledge bases can also be queried in combined PROLOG statements in DDBASE, see Section 4. The paper is concluded with some final remarks.

2 Graphs for Rule Bases

In a deductive database, a logic program can be abstracted by a predicate dependency or a rule predicate graph, and a derivation can be visualized by its proof tree, cf. [3]. It should also be possible to have bottom–up and top–down evaluation in one system. DATALOG* can evaluate logic programs with PROLOG syntax (extended DATALOG programs) in a bottom–up style; it is designed to evaluate embedded PROLOG calls in a top–down manner [20].

Dependency Graphs

We can define two sorts of dependency graphs for a normal logic program $P$. Both reflect, which predicates call other predicates in $P$. Let $p_L$ be the predicate symbol of a literal $L$. The *predicate dependency graph* $G^d_P = \langle V^d_P, E^d_P \rangle$ of $P$ is given by:

- the node set $V^d_P$ is the set of predicate symbols in $P$,
- $E^d_P$ contains an edge $\langle p_A, p_L_i \rangle$ for every rule $A \leftarrow L_1 \land \ldots \land L_m$ of $P$ and $1 \leq i \leq m$.

The *rule predicate graph* $G^r_P = \langle V^r_P, E^r_P \rangle$ of $P$ is given by:

- $V^r_P$ contains a node for every predicate symbol $p_L$ in $P$ and every rule $r$ in $P$,
- $E_P^r$ contains an edge $\langle p_A, r \rangle$ and an edge $\langle r, p_{L_i} \rangle$ for every rule $r = A \leftarrow L_1 \land \ldots \land L_m$ of $P$ and $1 \leq i \leq m$.

In both graphs, the edges $\langle p_A, p_{L_i} \rangle \in E_P$ and $\langle r, p_{L_i} \rangle \in E_{G_P^r}$, respectively, which come from default negated body literals $L_i = \text{not } C_i$, are marked by “not”. We get the following dependency graphs for a normal rule $r = A \leftarrow B_1 \land \ldots \land B_m \land \text{not } C_1 \land \ldots \land \text{not } C_n$.

The rule predicate graph $G_{G_P}^r$ is more refined than the predicate dependency graph $G_P^d$, see Figure 1.

The following definite logic programs have the same predicate dependency graph $G_P^d$ but different rule predicate graphs $G_{G_P}^r$ and $G_{G_P}^r$:

$P_1 = \{ p \leftarrow q_1, p \leftarrow q_2 \}$,
$P_2 = \{ p \leftarrow q_1 \land q_2 \}$

The predicate dependency graph can be obtained from the rule predicate graph by transitively connecting the head and body predicate symbols and omitting the rule nodes between. Restricted to the predicate symbols, both graphs have the same paths. They are typically used for analysing logic programs of software packages and for refactoring and slicing.

**Dependency Graphs with Helper Rules**

Dependency graphs can help to detect that two logic programs are equivalent apart from helper rules:

$r_1 = A \leftarrow B_1 \land \ldots \land B_{i-1} \land B_i \land B_{i+1} \land \ldots \land B_n$
$r_2 = B \leftarrow C_1 \land \ldots \land C_m$

With the most general unifier $\theta = \text{MGU}(B_i, B)$ we can obtain the following resolvent of the two rules:
Apart from the helper predicate symbol \( p_B = p_{B_i} \), the set of predicate symbols that is reachable from \( p_A \) in the predicate dependency or rule predicate graph is the same for the two programs \( \{ r_1, r_2 \} \) and \( \{ r_3 \} \). For the former, we get \( \Pi = \{ p_{B_1}, \ldots, p_{B_n} \} \cup \{ p_{C_1}, \ldots, p_{C_m} \} \), for the latter we get \( \Pi \setminus \{ p_B \} \).

**Dependency Graphs with Meta–Predicates**

In DDBASE, we use an extra node for every call of a meta–predicate to correctly reflect a predicate. The specific difference between normal predicates and meta–predicates in our extension of the rule predicate graph is that there can be several nodes labelled with the same meta–predicate.

The following example contains two such calls, i.e. to the meta–predicates `not/1` and `findall/3`. The predicate `ancestor_list/2` derives the list \( Xs \) of ancestors of a person \( X \).

```prolog
ancestor_list(X, [X]) :-
    not(parent(X, _)),
    !.
ancestor_list(X, Xs) :-
    findall( Ys,
        { parent(X, Y),
          ancestor_list(Y, Ys) },
        Yss ),
    append(Yss, Xs).
```

The PROLOG program is recursive, since `ancestor_list/2` calls itself through `findall/2`, see Figure 2. After that, the predicate `append/2` appends the list \( Yss \) of derived lists to a regular list \( Xs \).

Obviously, the program above only terminates for top–down evaluation on acyclic predicates `parent/2`. The nodes for the meta–predicates are necessary to show that `ancestor_list/2` depends on `parent/2` and `ancestor_list/2` itself. If we would use ordinary predicate dependency or rule predicate graphs, then we would lose this information.

**Derivations and Proof Trees**

We are developing a deductive database system DDBASE, which can manage hybrid rule bases and embed PROLOG calls into DATALOG rules. The underlying language DATALOG* has PROLOG syntax; its mixing of bottom–up and top–down evaluation is described in [20]. The following logic program deals with a well–known extension of the route finding problem from deductive databases. In addition to the lengths of derived routes (3rd argument of `route/4`), we can construct a proof tree (4th argument).

An atom `prolog:A` leads to an embedded top–down call of the goal \( A \) in PROLOG. The goal `(L is N+M)` computes the sum \( L \) of the path lengths \( N \) and \( M \) in PROLOG. A goal of the form `pt(T, ...)` constructs a proof tree \( T \).
route(X, Y, L, T) :-
  street(X, Y, L, T1),
  prolog:pt(T, t(route(X, Y, L), e, T1)).
route(X, Y, L, T) :-
  street(X, Z, N, T1), route(Z, Y, M, T2),
  prolog:(L is N+M),
  prolog:pt(T, 
    t(route(X, Y, L), r, T1, T2, (L is N+M))).

street(‘KT’, ’Wue’, 15, T) :-
  prolog:pt(T, t(street(‘KT’, ’Wue’, 15), f1)).
street(’Wue’, ’Mue’, 280, T) :-
  prolog:pt(T, t(street(’Wue’, ’Mue’, 280), f2)).

In DDBASE, we have implemented a generalized $\mathcal{T}_p$-operator, which can derive the following atom by a bottom–up evaluation. It can be seen, that the evaluation derives a proof tree in the last argument of the predicates route/4 and street/4. This tree serves as an explanation of the derived atoms, a very useful concept known from expert systems. E.g., this program will derive the atom shown in the following. The last argument of the atom contains the proof tree, which was automatically layouted and visualized in DDBASE, see Figure 3.

route(KT, Mue, 295,
  t(route(KT, Mue, 295), r,
    t(street(KT, Wue, 15), f1),
    t(route(Wue, Mue, 280), e,
      t(street(Wue, Mue, 280), f2))))
3 Ontologies with Rules in SWRL

A hybrid information system can include ontologies of various origins. Before working with and – and for designing such ontologies – the knowledge engineer has to analyse them and check them for anomalies. In DDBASE, we use methods for detecting anomalies in ontologies with rules, such as SWRL ontologies, which we have investigated in [2]. For handling different versions of an ontology, it is also possible to use well-known alignment methods to find out common parts and differences.

3.1 Schema Graph for XML

SWRL ontologies can be represented in XML notation:

```xml
<swrlx:Ontology swrlx:name="people">
  <swrlx:classAtom>
    <owlx:Class owlx:name="person"/>
    <ruleml:var>X</ruleml:var>
  </swrlx:classAtom>

  <swrlx:classAtom>
    <owlx:IntersectionOf>
      <owlx:Class owlx:name="person"/>
      <owlx:ObjectRestriction owlx:property="parent">
        <owlx:someValuesFrom owlx:class="Physician"/>
      </owlx:ObjectRestriction>
    </owlx:IntersectionOf>
    <ruleml:var>Y</ruleml:var>
  </swrlx:classAtom>
</swrlx:Ontology>
```

Fig. 3. Proof Tree.
In DDBase, a SWRL ontology can be visualized by the schema graph of its XML representation. Since ontologies – or XML files in general – can be very large, it is helpful to get a short overview in advance, see Figure 4.

**Fig. 4.** Schema Graph for XML.

### 3.2 Rules in SWRL

SWRL ontologies can have rules in RULEML. E.g., the following rule, which states that the brother of a parent is an uncle, can be represented in RULEML:

\[
\text{uncle}(X, Z) \leftarrow \text{parent}(X, Y) \land \text{brother}(Y, Z).
\]

In the XML representation, this could look like follows:

```xml
<ruleml:imp>
  <ruleml:_body>
    <swrlx:individualPropertyAtom
      swrlx:property="parent">
      <ruleml:var>X</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom
      swrlx:property="brother">
      <ruleml:var>Y</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
</ruleml:imp>
```
3.3 Syntax and Semantics

The syntax for SWRL in this section abstracts from any exchange syntax for OWL and thus facilitates access to and evaluation of the language. This syntax extends the abstract syntax of OWL described in the OWL Semantics and Abstract Syntax document [15]. This abstract syntax is not particularly readable for rules. Thus, examples will thus often be given in an informal syntax, which will neither be given an exact syntax nor a mapping to any of the fully-specified syntaxes for SWRL.

The abstract syntax is specified here by means of a version of Extended BNF, very similar to the EBNF notation used for XML. Terminals are quoted; non-terminals are bold and not quoted. Alternatives are either separated by vertical bars (|) or are given in different productions. Components that can occur at most once are enclosed in square brackets; components that can occur any number of times (including zero) are enclosed in curly braces. Whitespace is ignored in the productions here. Names in the abstract syntax are RDF URI references. The meaning of some constructs in the abstract syntax will be informally described. The formal meaning of these constructs can be defined via an extension of the OWL DL model-theoretic semantics [15].

An OWL ontology in the abstract syntax contains a sequence of axioms and facts. Axioms may be of various kinds, e.g., subClass axioms and equivalentClass axioms. It is proposed to extend this with rule axioms. Similar to what is usual in logic programming, a rule axiom consists of an antecedent (body) and a consequent (head), each of which consists of a (possibly empty) set of atoms. Antecedents and consequents are treated as the conjunctions of their atoms.

\[
\text{rule ::= 'Implies(' annotation antecedent consequent ')')'}
\]  
\[
\text{antecedent ::= 'Antecedent(' atom ')')'}
\]  
\[
\text{consequent ::= 'Consequent(' atom ')')'}
\]

Rules with an empty antecedent can be used to provide unconditional facts; however such unconditional facts are better stated in OWL itself, i.e., without the use of the rule construct. Rules with conjunctive consequents could easily be transformed – via the Lloyd–Topor transformations [12] – into multiple rules each with an atomic consequent. Atoms can be of the form \(c(X)\), \(p(X,Y)\), \(same_as(X,Y)\) or \(different_from(X,Y)\), where
c is an OWL description, p is an OWL property, and X,Y are either variables, OWL individuals or OWL data values. In the context of OWL Lite, descriptions in atoms of the form c(X) may be restricted to class names. Informally,

- an atom c(X) holds, if X is an instance of the class description c,
- an atom p(X,Y) holds, if X is related to Y by property p,
- an atom same_as(X,Y) holds, if X is interpreted as the same object as Y, and
- an atom different_from(X,Y) holds if X and Y are interpreted as different objects.

The latter two forms can be seen as syntactic sugar: they are convenient, but do not increase the expressive power of the language. Atoms may refer to individuals, data literals, individual variables or data variables. As in PROLOG or DATALOG, variables are treated as universally quantified, with their scope limited to a given rule. Only variables occurring in the antecedent of a rule may occur in the consequent (range-restrictedness). This condition does not, in fact, restrict the expressive power of the language, because existentials can already be captured in OWL using someValuesFrom restrictions.

While the abstract EBNF syntax is consistent with the OWL specification, and is useful for defining XML and RDF serialisations, it is rather verbose and not particularly easy to read. Deductive databases therefore often use a relatively informal human readable form. In this syntax, a rule has the form \( \alpha \leftarrow \beta \), where the antecedent \( \beta \) is a conjunction of atoms and the consequent \( \alpha \) is a single atom. In standard convention, variables are indicated by prefixing them with a question mark; in this paper, however, we represent them in PROLOG convention with strings starting with a capital letter. Then, a rule asserting that the composition of parent and brother properties implies the uncle property would be written as

\[
\text{uncle}(X, Z) \leftarrow \text{parent}(X, Y) \land \text{brother}(Y, Z).
\]

An even simpler rule would be to assert that students are persons:

\[
\text{person}(X) \leftarrow \text{student}(X).
\]

However, this kind of use for rules in OWL just duplicates the OWL subclass facility. It is logically equivalent to write instead Class(Student partial Person) or SubClassOf(Student Person) which would make the information directly available to an OWL reasoner. A very common use for rules is to move property values from one individual to a related individual.

### 3.4 Hybrid Information Systems

For collaborations across disciplines, hybrid information systems using data and techniques from many different sources with no preexisting agreement about the semantics of the processes or data is important. The infrastructure must provide general purpose mechanisms for annotating (i.e., making assertions about), discovering, and reasoning about processes and data. Some of the inferences require additional reasoning beyond that supported by OWL and SWRL. Also graphs such as the provenance graph are very useful here for representing causal relationships. The Open Provenance Model (OPM)
defines logical constraints on the provenance graph [13]. Some constraints cannot be expressed in OWL, but can be expressed based on SWRL rules.

PROV is a specification that provides a vocabulary to interchange provenance information. It defines a core data model for the interchange of provenance on the web; it allows for building representations of the entities, people and processes involved in producing a piece of data in the world. The provenance of digital objects represents their origins; the records of a PROV specification can describe the entities and activities involved in producing and delivering or otherwise influencing a given object. Provenance can be used for many purposes, such as understanding how data was collected so it can be meaningfully used, determining ownership and rights over an object, making judgements about information to determine whether to trust it, verifying that the process and steps used to obtain a result complies with given requirements, and reproducing how something was generated.

The Open Provenance Model (OPM) provides a case study for how to use Semantic Web technology and rules to implement semantic metadata. [13] discusses a binding of the OPM written in OWL with rules written in SWRL. This allows for the development of hybrid systems that use OWL, SWRL, and other semantic software, interoperating through a shared space of RDF triples. E.g., four derived relations between two artifacts (e.g., input and output datasets) can be inferred from used and generated_by to the same process:

\[
\begin{align*}
&\text{derived_sink}(B, H) \land \text{derived_source}(B, Y) \land \\
&\text{derived_account}(B, D) \land \text{derived_account}(B, G) \leftarrow \\
&\text{artifact}(Y) \land \text{generated_by_artifact}(C, Y) \land \\
&\text{process}(X) \land \text{generated_by_process}(C, X) \land \\
&\text{account}(D) \land \text{generated_by_account}(C, D) \land \\
&\text{relation}(E) \land \text{used_process}(E, X) \land \\
&\text{artifact}(H) \land \text{used_artifact}(E, H) \land \\
&\text{account}(G) \land \text{used_account}(E, G) \land \\
&\text{swrlx:create_owl_thing}(B, X, C, E)
\end{align*}
\]

But several of the key constraints and inferences of the OPM cannot be expressed in OWL and SWRL, due to fundamental limits of the semantics of these languages. E.g., it is not possible to modify the value of an asserted property, or to write a rule discovering the number of times an artifact is used, or to detect a cycle in the provenance graph. Storing the OPM records in triples makes it possible to use other reasoning engines or languages such as PROLOG or DATALOG to implement queries or inferences.

OWL and SWRL’s RDF representations provide a simple and well-understood means of exchanging provenance information with other tools, such as RDF databases and declarative programming languages. This hybrid system shows that Semantic Web technologies are not only useful for provenance information but also provide a base level of interoperability that can enable loosely-coupled tools with varying levels of capability and expressiveness.
4 Querying Hybrid Knowledge Bases in DDBASE

Hybrid knowledge bases can be managed and queried using logic programming techniques. In the deductive database system DDBASE, various representations of knowledge can be accessed, see Figure 5. The database query language DATALOG* is an extension of DATALOG [20], where logic programs in PROLOG syntax are evaluated bottom-up, and embedded calls to PROLOG are evaluated top-down. XML data from XML databases or documents can be stored in a term representation; calls to XML data based on path expressions are evaluated in PROLOG using the query, transformation and update language FQUERY [19].

![Diagram of Data Sources, Programming Languages, DB Query Languages]

**Fig. 5. Hybrid Knowledge Bases in DDBASE.**

In DDBASE, we can compute joins of relational databases and XML documents in PROLOG. The following example is a modified version of the well-known example from the book of Elmasri and Navathe [4]. The atoms for employee/8 could, e.g., be derived using ODBC from a relational database; they could also stem from an ontology; SWI PROLOG [21] offers a loader producing RDF triples, which can be transformed to PROLOG facts easily.

```prolog
% employee(Name, SSN, Date, SEX, Salary, Super(SSN), DNO)
employee('Borg', 11, date(1927,11,10), 'M', 55000, null, 1).
employee('Wong', 22, date(1945,12,08), 'M', 40000, 11, 5).
employee('Wallace', 33, date(1931,6,20), 'F', 43000, 11, 4).
employee('Smith', 44, date(1955,1,09), 'M', 30000, 22, 5).
...
```

Additionally, we work with the following XML version of the table works_on/3; a row represents an employee ESSN working on a project PNO a number of HOURS.
The following query joins the atoms for employee/8 with the rows in the XML document works_on.xml in DDbase. The attribute value H of the attribute ‘HOURS’ of Row is an atom that has to be converted to a number HOURS. Clearly, this predicate fails for HOURS = null, which is desired to ignore null values in aggregations in SQL. The handling of path expressions applied to XML documents has been described in [19]. The template [DNO, sum(HOURS)] leads to a grouping on the department numbers. For every DNO, the list Xs of all corresponding HOURS is computed, and the sum by \( \text{sum}(Xs, \text{Sum}) \); thus, we obtain a standard result tuple [DNO, Sum].

\[
\text{?- ddbase_aggregate( [DNO, sum(HOURS)],}
\text{\{
employee(_, SSN, _,_,_,_,_, DNO),
Row := doc('works_on.xml')/row::[@'ESSN'=SSN],
H := Row@'HOURS', atom_number(H, HOURS) ),
Tuples \}.
\]

A query optimizer of DDbase could rearrange the Goal in the second argument of the atom for ddbase_aggregate/3 by changing the order of the calls to the predicate employee/8 provided by ODBC and the XML document works_on.xml. It might be the best to first completely load the table EMPLOYEE from the relational database to PROLOG using ODBC and to index it on the second argument position, which holds the SSN. Then, in a single pass through the XML document, the working hours can be obtained using a path expression in FQuery from the XML rows, and the corresponding department numbers of the employees can be obtained using the index.

## 5 Final Remarks

We have developed a PROLOG-based deductive database system DDbase for hybrid knowledge bases. Knowledge sources with different types of knowledge representation – including relational, XML, SWRL – can be managed in DDbase.

The transition from relational databases to deductive databases brings in recursive rules, and thus generalizes the concept of views. SWRL ontologies add further ideas from artificial intelligence; ontologies can be augmented by rules to enhance expressiveness. PROV uses SWRL to model the provenance of digital data.

In DDbase, we are building a system for handling hybrid queries in a deductive database. A query optimizer should extend relational systems, and it should be able to handle relational data and rules together with XML data and ontologies.
References

Learning Instance-Level Constraints in Folksonomies for Semi-supervised Clustering using CHR

Maged Shalaby, Slim Abdennadher, Nada Sharaf, and Ghada Fakhry

Computer Science and Engineering Department
German University in Cairo
maged.shalaby@student.guc.edu.eg, {slim.abdennadher, nada.hamed, ghada.fakhry}@guc.edu.eg

Abstract. In our modern days, huge amount of information is available to be searched and browsed by simple users. However, due to the sheer volume of information available, the task of browsing this information becomes increasingly difficult. Clustering algorithms have emerged as an automated tool to organize information for easier browsing. In this paper, we employ constraint reasoning to reason about the different pairwise constraints present, such as must-link and cannot-link constraints. These constraints are used to aid in semi-supervised clustering algorithms, and they are used to guide the algorithm whether a pair of items should or should not be placed in the same cluster. This implementation was done using CHR, a declarative rule-based language. We prove via experimental evaluation that inferring instance-level constraints in the domain of folksonomies yields an improvement in clustering performance.

Keywords: constraint reasoning, semi-supervised clustering, CHR

1 Introduction

The modern rise of the World Wide Web has led to an explosion in the amounts of data available to users. It is now a tedious task to find items of interest in this growing web of data. Folksonomies have risen to be a possible solution to the recommendation problem. Folksonomies are websites where users can tag items (artists, pictures, etc.) with relevant tags. An example for tags for a photo of a sandy beach are sea and beach. Users can view, add and tag items of interest, and get personalized recommendations out of those systems.

Clustering techniques have emerged as one possible means of recommending items for easier searching and browsing. Among those clustering techniques, there is a new generation of algorithms known as semi-supervised algorithms.

Semi-supervised algorithms utilize pairwise and group constraints that aid in the clustering process. An example of pairwise constraints are must-link and cannot-link constraints that enforce that items must be in the same cluster or different clusters, respectively [14]. These constraints can be added manually by domain experts, or inferred from the available data. Inferring these constraints using procedural languages is more difficult than using rule-based languages.
In this paper, we choose to use Constraint Handling Rules (CHR) to implement a constraint handler tuned for this problem. We found that CHR rules are easier to be written and implemented than procedural approaches due to their declarativity. CHR also gives a naturally incremental approach to adding constraints to semi-supervised algorithms. Therefore, we can add even more instance-level constraints than must-link and cannot-link constraints.

We employ constraint inference from folksonomies in our work to improve upon unsupervised clustering performance. We use tagging information about items to extract constraints. An example is generating must-link constraints for a pair of items if this pair has been tagged by many users. Another example is generating cannot-link constraints if the pair of items has less than a specific number of tags in common.

This paper is organized as follows: Section 2 reviews the relevant background about folksonomy systems, semi-supervised clustering, and CHR (our programming language of choice). Section 3 discusses the constraint inference module implemented. Finally, we conclude with a summary and discussion of future work in Section 4.

2 Background

In our work, we employ constraint inference to learn rules for folksonomies, to aid in semi-supervised clustering algorithms. We do this using CHR, a rule-based declarative language. Therefore we shall introduce first the notions of folksonomies, semi-supervised clustering and CHR.

2.1 Folksonomies

One arising application of Web 2.0, the new look for websites and webpages that started in the early 2000’s, is that of folksonomies. Folksonomies are the collection of users, items, tags in social tagging websites and applications, such as Flickr and del.icio.ous. Users can create user profiles, add or show interest in certain items (such as pictures, songs or bookmarks), and tag these items. Formally, a folksonomy is a tuple $F := (U, T, R, Y)$, where $U$, $T$, and $R$ are finite sets, whose elements are called users, tags and resources, respectively, and $Y$ is a ternary relation between them, i. e., $Y \subseteq U \times T \times R$ [8].

We will focus in our work on clustering items in folksonomies. An example of this is grouping photos together in a social photo-sharing website, such as Flickr. This approach is opposite to tag clustering, where tags are grouped into semantically meaningful groups, often used for tag disambiguation. Both are used in the process of recommendation, either recommending new items previously unseen to the users [11], or in recommending new tags to tag resources with.
2.2 Semi-supervised Clustering

In more recent times, there has been an interest in what is called constrained, or semi-supervised clustering. However, we ought to define supervised and unsupervised learning first.

In the field of machine learning, it is well known that there are two broad classes of problems, namely supervised and unsupervised learning. In supervised learning, given a set of training data, each item in the training data being comprised of $x$ features and a label. Given this training data, a model should be trained, that given a new previously unseen instance that has the $x$ features, will predict correctly its label. In unsupervised learning, the luxury of the labeled data does not exist. Instead, the goal is to find hidden patterns in the data from instances of only $x$ features.

As mentioned above, classical clustering is viewed as an unsupervised learning problem. However, with the use of some extra data inferred from domain knowledge or crowdsourced data, as we will do in our work, the problem of clustering can be categorized as a new class of problems: semi-supervised learning.

So what are the extra pieces of information that will transform our problem? The most commonly used are what are known as instance-level constraints. These can be categorized into either must-link constraints or cannot-link constraints between pairs of items. The presence of a must-link constraint forces, or at least advises, an algorithm to place the two items in the same cluster. The presence of a cannot-link constraint is the opposite, it advises an algorithm to place the two items in different clusters. The use of instance-level constraints can tremendously help any unsupervised clustering algorithm to produce better results, if only this information is somehow available.

There is an important difference between hard constraints and soft constraints. The difference between the two is that hard constraints force the algorithm to satisfy the constraint, placing the items in the same cluster for must-link constraints and placing the items in different clusters for cannot-link constraints. On the other hand, soft constraints are merely a guide to the algorithm that it is simply better to follow the constraints, with some sort of penalty for breaking the constraints.

It is possible to infer the instance link constraints, using some sort of domain knowledge. One example in folksonomies is generating must-link constraints for items that share at least $t$ tags in common. This sort of inference is possible in the domain of folksonomies, and is rather done using soft constraints, not hard constraints [3]. Another example in the fields of information retrieval and natural language processing is inferring must-link constraints from text documents that share $x$ n-grams [2].

There are more examples related to natural life. A third example is using GPS data for automatic lane detection, where there are two heuristics used in [14]. The first is for trace contiguity, where a must-link constraint is generated for all data points originating from the same vehicle in the absence of lane changes. The second is for maximum separation, which limits how far apart two points can be (perpendicular to the centerline) while still being in the same lane. If
two points are separated by at least four meters, then a cannot-link constraint is generated that will prevent those two points from being placed in the same cluster.

2.3 CHR

Constraint Handling Rules (CHR) is a high level language that was introduced for writing constraint solvers. CHR is a committed choice language based on multi-headed and guarded rules. CHR programs works by transforming constraints into simpler ones until they are solved. Over the last decades, CHR has matured into a general purpose language. CHR usually does not exist on its own, but is built on top of a host language. The most popular host language, and the one we shall use in our work, is Prolog, but there have also been CHR implementations for other languages, such as Haskell and Java. While originally intended for constraint solvers, CHR has extended to other problems and fields such as semantic web (description logics) [6], and natural language processing [5].

In CHR, two types of constraints are available. Built-in constraints provided through the host language (Prolog) and user-defined constraints that are defined through the rules of a CHR program [7]. Each rule consists of a head, which contains a set of constraints, an optional guard and a body. A rule is fired when constraints from the constraints store match the constraints in the rule head and the guard is satisfied. In general, there are three types of rules in CHR. The first type is simplification rule, such rules replace constraints by simpler ones. In simplification rules the head of the rule $H^r$, which comes before the ($\iff$) are removed on executing the rule. ($G$) is the optional guard that consists of built-in constraints. The body ($B$) could contain both CHR and built-in constraints. A simplification rule thus has the following format:

$$H^r \iff G \mid B.$$  

The second rule type is propagation rules. In a propagation rule, all head-constraints $H^k$ are kept after the rule is executed adding the constraints in the body to the constraint store. This may cause further simplification afterwards. Propagation rules have the following format:

$$H^k \Rightarrow G \mid B.$$  

The third type of rules, defined as simpagation rules, is a hybrid between the simplification and propagation rules. The elements of $H^k$ are the constraints that are kept after the rule is executed. On the other hand, the constraints in $H^r$ are removed after executing the rule. Simpagation rules take the following format:

$$H^k \setminus H^r \iff G \mid B.$$  

An illustrative example can be viewed in the less-than-or-equal simple constraint solver The. The constraint $\text{leq}(X,Y)$ represent the relation between the
two numbers \(X\) and \(Y\) respectively. The first rule adds a new inequality constraint between items \(X\) and \(Z\) given that \(X \leq Y\) and \(Y \leq Z\). Thus, the first rule is a propagation rule. The second rule denotes that if \(X \leq Y\) and \(Y \leq X\), then \(X\) and \(Y\) must be equal. Thus, the second rule is a simplification rule.

\[
\text{leq}(X,Y), \text{leq}(Y,Z) \implies \text{leq}(X,Z).
\]

\[
\text{leq}(X,Y), \text{leq}(Y,X) \iff X=Y.
\]

### 2.4 Tag Reasoning in CHR

A notable effort done previously is the work done by Sharaf et al [12] where a CHR tag reasoning system was provided. Due to its declarative nature, with CHR, different properties were easily encoded. The system was able to capture different properties including application specific features such as “absence of a tag in the presence of another”. This encodes the fact that in some of the cases if a tag is absent (fixed for example) in the presence of another tag (bug for example), a new tag (todo) should be added. Some, more general, rules were included to deal with inconsistent items (annotated with contradicting tags), and inconsistent users (annotating the same item with contradicting tags). With CHR, it was also possible to easily generate the co-occurrence graph of tags using the number of times the tags appear together.

### 3 Learning Instance-Level Constraints in Folksonomies

In this section we will discuss our constraint reasoning techniques, to infer must-link and cannot-link constraints in the field of folksonomies, and to reason about those constraints. We generate soft constraints here, where each constraint is a triple \(I_1, I_2, V\) where \(V\) is the value associated to the strength of the constraint. It is then easier to use them as soft constraints, or simply transform them to hard constraints by ignoring \(V\).

To represent the tagging information, we use the CHR constraint \(\text{annotation}(U, I, T)\), which means that user \(U\) has tagged item \(I\) with tag \(T\).

#### 3.1 Inferring Must-Link Constraints

The technique we apply to generate must-link constraints is to link pairs of items that have been co-tagged by \(X\) users. To do this, we need to count how many users have commonly tagged this pair of items, and generate a must-link constraint only once this reaches a certain threshold. We show an example in the following rules:

\[
\text{annotation}(U, I_1, \_), \text{annotation}(U, I_2, \_), \text{possiblePair}(I_1, I_2) \implies I_1 \leq I_2 ~|~ \text{userInCommon}(I_1, I_2, U).
\]

\[
\text{userInCommon}(I_1, I_2, U) \setminus \text{userInCommon}(I_1, I_2, U) \iff \text{true}.
\]
In the first rule, we generate all users that have commonly tagged a pair of items. In the second rule, we remove duplicate instances of the same constraint. In the third and fourth rules, we count how many users in common a pair of items have. In the fifth rule, we generate a must-link constraint when the users in common has exceeded a certain threshold (here we set it to 10).

In the first rule, using a "." in the place of the tag variable indicates that we don’t care whether the same tag is used or not. Also, the guard @< is used between I₁ and I₂ to force that I₁ must be lexicographically smaller than I₂. This is done to prevent the rule to fire for each pair of items twice, and to prevent it to fire for two annotations of the same item.

We use the constraint possiblePair(I₁, I₂) to stop the constraint generation procedure early if found that this pair of items has exceeded the threshold. This speeds up computation. The generation of the constraint possiblePair(I₁, I₂) is simple to do using either CHR or a procedural language. It can be generated by parsing an input file containing all the items, or from the constraint annotation(I, U, T).

To compare against a procedural approach, it would’ve needed a double for-loop for each pair of items, then computing the set intersection for the users that have tagged each item, which is a non trivial task. Instead, we need only five rules in the rule-based approach.

### 3.2 Inferring Cannot-Link Constraints

For generating cannot-link constraints between pairs of items, we count how many tags they have in common, and if it is less than a certain threshold, we generate a cannot-link constraint. Here are the rules for this task:

\[
\text{annotation(\_, I₁, T), annotation(\_, I₂, T), possiblePair(I₁, I₂) \implies I₁ \text{ @< I₂ | tagInCommon(I₁, I₂, T)}.}
\]

\[
\text{tagInCommon(I₁, I₂, T) \text{ \textbackslash tagInCommon(I₁, I₂, T) \implies true}.}
\]

\[
\text{tagInCommon(I₁, I₂, \_), possiblePair(I₁, I₂) \implies I₁ \text{ @< I₂ | tagsInCommon(I₁, I₂,1)}.}
\]

\[
\text{tagsInCommon(I₁, I₂, X₁), tagsInCommon(I₁, I₂, X₂) \implies X \text{ is } X₁ + X₂ | tagsInCommon(I₁, I₂, X)}.\]
tagsInCommon(I1, I2, X), possiblePair(I1, I2) \=> X < 9 |
cannotLink(I1, I2, 1).

\( \Rightarrow \) tagsInCommon(I1, I2, X) \ \setminus \ \text{possiblePair}(I1, I2) \iff X \geq 9 |
okLink(I1, I2, 1).

okLink(I1, I2, 1) \ \setminus \ \text{cannotLink}(I1, I2, 1) \iff \text{true}.

The first rule is used to generate all tags in common between all pairs of items. This is equivalent to building the item-tag bipartite graph. The second rule is used to remove duplicate constraints. The third and fourth rules count how many tags in common are between this pair of items. The fifth rule generates a cannot-link constraint if the tags in common are less than a certain threshold. The sixth rule generates an ok-link constraint if the tags in common is greater than a certain threshold. The undesired cannot-link constraints are removed by the seventh rule.

To compare against a procedural approach, it would’ve needed two for loops to enumerate all pairs of items, and then computing the set intersection of the tags for each item. The rule based approach is again much simpler.

### 3.3 Generating Transitive Closure

One task often done before the start of a semi-supervised clustering algorithm is to find the transitive closure. The transitive closure is defined on a graph \((V, E)\), and it is finding the reachability of all nodes in \(V\). Therefore, for must-link and cannot-link constraints, we try to generate all possible constraints that can be inferred from the original constraint. For example, if \(I_1, I_2\) must be linked together, and \(I_2, I_3\) must be linked together, then it follows that \(I_1, I_3\) must be linked together. We present now the two rules we used to generate the transitive closure:

\[
\text{mustLink}(A, B, Cnt1), \text{mustLink}(B, C, Cnt2) \Rightarrow \\
\text{allDifferent}(A, B, C), \text{minimum}(Cnt1, Cnt2, Cnt) \mid \text{mustLink}(A, C, Cnt).
\]

\[
\text{mustLink}(A, B, Cnt1), \text{cannotLink}(B, C, Cnt2) \Rightarrow \\
\text{allDifferent}(A, B, C), \text{minimum}(Cnt1, Cnt2, Cnt) \mid \text{cannotLink}(A, C, Cnt).
\]

The predicate \text{allDifferent}(A, B, C) is a Prolog predicate we implemented which simply checks that the three inputs are distinct.

To compare against a procedural approach, we would’ve needed to generate a graph of the constraints, and traverse it using any \(O(n + m)\) traversal such as breadth-first search or depth-first search, and find the connected components in this graph, then generate must-link constraints for all pairs of items in a connected component. The rule based approach is much simpler and needed only two rules.
3.4 Evaluation

In our work, we use a publicly available Last.fm [1] dataset [4]. This dataset contains listening and tagging information from the music website Last.fm for 1892 users. We work with the tagging, not the listening information. It contains 17632 artists, 186479 tag assignments, i.e. tuples [user, tag, artist]. Since some artists are sparsely tagged, we consider only the 100 most annotated items in our evaluation.

To evaluate whether inferred instance-level constraints using the rules above improve clustering performance, we use the constraints in a semi-supervised clustering algorithm, namely COP K-Means [14]. We use a simple variation of the algorithm, which employs soft constraints rather than hard constraints. If there is a must-link constraint between an item $I_1$ and an already assigned item $I_2$ in cluster $C$, we increase distance of $I_1$ to $C$ by 1. Similarly, we decrease the distance by 1 for cannot-link constraints.

We represent items by their tag vectors. Each item has a tag vector of length equal to the number of unique tags, and with values indicating how many times this item has been tagged with this tag. Our distance measure is the cosine similarity [13].

We compare the silhouette coefficient (SC) [9] and mean-square-error (MSE) for regular K-Means (without the inferred constraints) and COP K-Means, to evaluate clustering performance. Both are "internal indices", which means they don’t use a labeled dataset to assess the clustering performance [10]. We resorted to using internal indices because after extensive research, we could not find a labeled dataset for item-clustering in folksonomies.

The silhouette coefficient is always between [-1, 1]: higher values closer to 1 indicate a better clustering performance. The mean-square error is the opposite, however, where lower values indicate a better clustering performance.

To account for the possible difference in results due to picking random seeds in the clustering process, we choose a set of seed points using the following strategy. We find the $k$ most used tags, where $k$ is the desired number of clusters for items, and pick the most tagged artist by each tag as the seed points.

There are two variables that influence the results: the must-link threshold and the cannot-link threshold. The must-link threshold is the number of users $u$, where a pair of items must have more than $u$ users in common to generate a must-link constraint between them. The cannot-link threshold is the number of tags $t$, where a pair of items must have less than $t$ tags in common to generate a cannot-link constraint between them. We note that we did not apply the transitive closure on the constraints as this would have taken a large amount of time.

Table 1 shows the results for different runs. The first run is regular K-Means, without constraints. The second and third run are the ones that optimize the SC for must-link and cannot-link only, respectively. The last run is the optimal run, both in terms of the MSE and the SC (highlighted in bold), and it incorporates both must-link and cannot-link constraints.

This indicates that inferring instance-level constraints can improve item clustering in folksonomies.
### Table 1. Clustering Performance

<table>
<thead>
<tr>
<th>ML Threshold</th>
<th>CL Threshold</th>
<th># ML Constraints</th>
<th># CL Constraints</th>
<th>MSE</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1754.09</td>
<td>0.21</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>0</td>
<td>325</td>
<td>1772.56</td>
<td>0.26</td>
</tr>
<tr>
<td>68</td>
<td>-</td>
<td>9</td>
<td>0</td>
<td>1779.69</td>
<td>0.24</td>
</tr>
<tr>
<td>68</td>
<td>8</td>
<td>9</td>
<td>325</td>
<td><strong>1713.64</strong></td>
<td><strong>0.29</strong></td>
</tr>
</tbody>
</table>

We note, however, the reason behind why the optimal must-link threshold was relatively high, giving a small number of constraints. This is because our distance measure for clustering does not include user information; it only relies on tag information. Therefore, it is natural that generating must-link constraints from user information could be detrimental, or in this case, giving a slight improvement but only when a small number of constraints is used.

### 4 Conclusion and Future Work

In this paper we used the declarative language Constraint Handling rules for improving semi-supervised clustering by generating must-link and cannot-link constraints. As a result of the work, it was found that rule based approaches are much simpler than their procedural counterparts, and are more natural to write. Our approach focused on the domain of folksonomies, where annotations are the primitive constraints that we used to infer new constraints. We found that inferring instance-level constraints could help improve item-clustering performance in folksonomies.

In the future, we intend to apply the approach over different domains, and show that rule based constraint reasoning is generally superior to procedural approaches. In addition, we intend to further incorporate the notion of popularity in our rules to produce better constraints. For example, a popular tag that is in common between a pair of items should have less impact on the cannot-link constraint than an unpopular tag in common.

We also would like to evaluate the clustering performance using a labeled dataset, which we would create or use a combination of datasets from folksonomies and clustering datasets. Optimally, we should test using different semi-supervised clustering algorithms. Using other distance measures (incorporating user information), as well as using dimensionality reduction techniques should be investigated. Another related problem is investigating more advanced techniques for seed-point initialization in folksonomies and their effect on clustering performance.

### References

6. Thom Frühwirth. Description logic and rules the chr way. 2007.